How to design your own CPU on FPGAs with VHDL

A Lecture by Dr. Juergen Sauermann

Copyright (C) 2009, 2010 by Dr. Juergen Sauermann

Note: Due to scaling, the figures (and some other things) in this PDF file look pretty bad. Please use the HTML version of this document instead. You can get the HTML version by one of the following:

- Go to http://www.opencores.org and log in.
- Go to the Projects page and select project "CPU Lecture" in the Soc section.
- Click Latest version: download
- Unpack the tar file (tar -xvzf)
- Open index.html with your browser

or:

- Check out the project: **svn co http://opencores.org/ocsvn/cpu_lecture/cpu_lecture**
- Open index.html with your browser

Table of Contents

1 INTRODUCTION AND OVERVIEW....1
- 1.1 Prerequisites....1
- 1.2 Other useful links....1
- 1.3 Structure of this Lecture....2
- 1.4 Naming Conventions....2
- 1.5 Directory Structure....2
- 1.6 Other Useful Tools....3

2 TOP LEVEL....4
- 2.1 Design Purpose....4
- 2.2 Top Level Design....4
- 2.3 General Structure of our VHDL Files....6
  - 2.3.1 Header and Library Declarations....6
  - 2.3.2 Entity Declaration....6
  - 2.3.3 Entity Architecture....6
- 2.4 avr_fpga.vhd....6
  - 2.4.1 Header and Library Declarations....6
  - 2.4.2 Entity Declaration....7
  - 2.4.3 Architecture of the Top Level Entity....8
- 2.5 Component Tree....14

3 DIGRESSION: PIPELINING....15

4 THE CPU CORE....18

5 OPCODE FETCH....23
- 5.1 Program Memory....25
  - 5.2.1 Dual Port Memory....25
  - 5.2.2 Look-ahead for two word instructions....25
  - 5.2.3 Memory block instantiation and initialization....27
  - 5.2.3 Delayed PC....29
- 5.3 Two Cycle Opcodes....29
- 5.4 Interrupts....30

6 DATA PATH....32
- 6.1 Register File....34
  - 6.1.1 Register Pair....34
  - 6.1.2 The Status Register....35
  - 6.1.3 Register File Components....35
  - 6.1.4 Addressing of General Purpose Registers....37
  - 6.1.5 Addressing of General Purpose Register Pairs....37
  - 6.1.6 Requirements on the Register File....37
  - 6.1.7 Reading Registers or Register Pairs....37
  - 6.1.8 Writing Registers or Register Pairs....40
  - 6.1.9 Addressing Modes....43
- 6.2 Data memory....47
- 6.3 Arithmetic/Logic Unit (ALU)....47
  - 6.3.3 Arithmetic and Logic Functions....48
  - 6.3.4 Output and Flag Multiplexing....49
  - 6.3.5 Individual ALU Operations....50

Table of Contents

6 DATA PATH
6.3.5 Temporary Z and T Flags....52
6.4 Other Functions....52

7 OPCODE DECODER....55
7.1 Inputs of the Opcode Decoder....55
7.2 Outputs of the Opcode Decoder....55
7.3 Structure of the Opcode Decoder....56
7.4 Default Values for the Outputs....57
7.5 Checklist for the Design of an Opcode.....57
7.6 Opcode Implementations....57
7.6.1 The NOP instruction....57
7.6.2 8-bit Monadic Instructions....57
7.6.3 8-bit Dyadic Instructions, Register/Register....58
7.6.4 8-bit Dyadic Instructions, Register/Immediate....60
7.6.5 16-bit Dyadic Instructions....61
7.6.6 Bit Instructions....61
7.6.7 Multiplication Instructions....62
7.6.8 Instructions Writing To Memory or I/O....63
7.6.9 Instructions Reading From Memory or I/O....63
7.6.10 Jump and Call Instructions....65
7.6.11 Instructions Not Implemented....69
7.7 Index of all Instructions....69

8 INPUT/OUTPUT....72
8.1 Interface to the CPU....72
8.2 CLR Signal....72
8.3 Connection the FPGA Pins....72
8.4 I/O Read....73
8.5 I/O Write....74
8.6 Interrupts....75
8.7 The UART....76
8.7.1 The UART Baud Rate Generator....76
8.7.2 The UART Transmitter....77
8.7.3 The UART Receiver....78

9 TOOLCHAIN SETUP....80
8.1 Functional Simulation....82
8.2 Timing Simulation and FPGA Configuration....84
8.3 Downloading and Building the Tools....84
8.4 Preparing the Memory Content....89
8.5 Performing the Functional Simulation....91
8.5.1 Preparing a Testbench....91
8.5.2 Defining Memory Modules....92
8.5.3 Creating the testbench executable....95
8.6 Building the Design....96
8.6.1 Creating an UCF file....97
8.6.2 Synthesis and Implementation....98
8.7 Creating a Programming File....99
8.8 Configuring the FPGA....99

Table of Contents

9 TOOLCHAIN SETUP
8.8.1 Configuring the FPGA via JTAG Boundary Scan....99
8.8.2 Flashing PROMs....99

10 LISTING OF alu.vhd....100

11 LISTING OF avr_fpga.vhd....107

12 LISTING OF baudgen.vhd....111

13 LISTING OF common.vhd....113

14 LISTING OF cpu_core.vhd....116

15 LISTING OF data_mem.vhd....121

16 LISTING OF data_path.vhd....125

17 LISTING OF io.vhd....130

18 Listing of opc_deco.vhd....134

19 LISTING OF opc_fetch.vhd....145

20 LISTING OF prog_mem_content.vhd....148

21 LISTING OF prog_mem.vhd....151

22 LISTING OF reg_16.vhd....156

23 LISTING OF register_file.vhd....158

24 LISTING OF segment7.vhd....165

25 LISTING OF status_reg.vhd....168

26 LISTING OF uart_rx.vhd....170

27 LISTING OF uart_tx.vhd....172

28 LISTING OF RAMB4_S4_S4.vhd....174

29 LISTING OF test_tb.vhd....177

30 LISTING OF avr_fpga.ucf....179

31 LISTING OF Makefile....180

Table of Contents

32 LISTING OF hello.c 181

33 LISTING OF make_mem.cc 182

34 LISTING OF end_conv.cc 185

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.

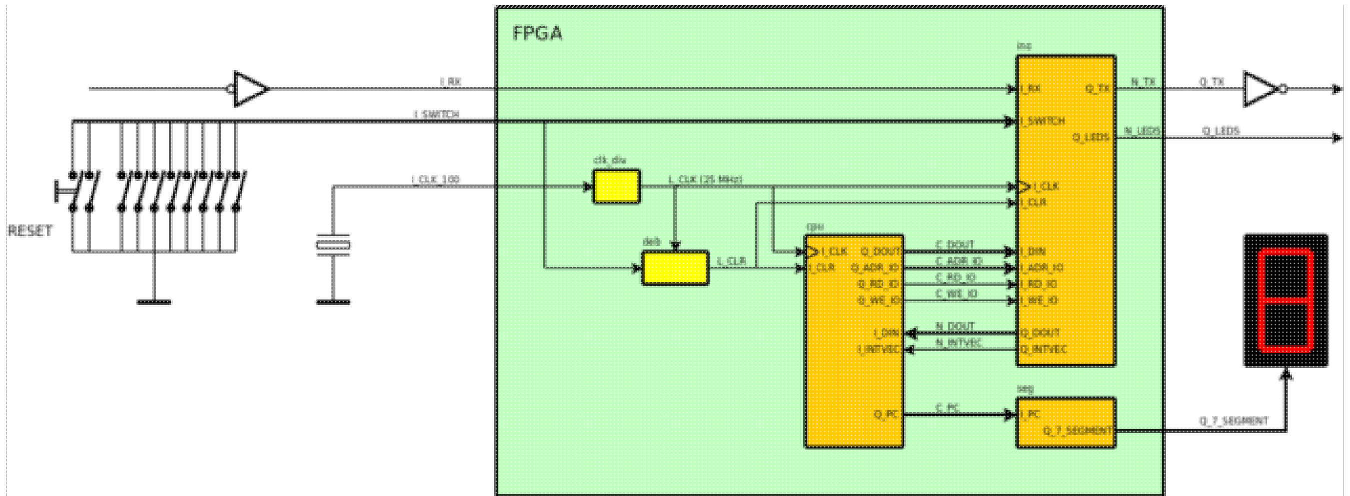
2.2 Top Level Design

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:

2 TOP LEVEL



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.

```
  1     -------------------------------------------------------------------------------
  2     --
  3     -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
  4     --
  5     --  This code is free software: you can redistribute it and/or modify
  6     --  it under the terms of the GNU General Public License as published by
  7     --  the Free Software Foundation, either version 3 of the License, or
  8     --  (at your option) any later version.
  9     --
 10     --  This code is distributed in the hope that it will be useful,
 11     --  but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 12     --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 13     --  GNU General Public License for more details.
 14     --
 15     --  You should have received a copy of the GNU General Public License
 16     --  along with this code (see the file named COPYING).
 17     --  If not, see http://www.gnu.org/licenses/.
 18     --
 19     -------------------------------------------------------------------------------
 20     -------------------------------------------------------------------------------
 21     --
 22     -- Module Name:     avr_fpga - Behavioral
 23     -- Create Date:     13:51:24 11/07/2009
 24     -- Description:     top level of a CPU
 25     --
 26     -------------------------------------------------------------------------------
 27
```

src/avr_fpga.vhd

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

```
 28     library IEEE;
 29     use IEEE.STD_LOGIC_1164.ALL;
 30     use IEEE.STD_LOGIC_ARITH.ALL;
 31     use IEEE.STD_LOGIC_UNSIGNED.ALL;
 32
```

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:

```
 33     entity avr_fpga is
 34         port (  I_CLK_100   : in  std_logic;
 35                 I_SWITCH    : in  std_logic_vector(9 downto 0);
 36                 I_RX        : in  std_logic;
 37
 38                 Q_7_SEGMENT : out std_logic_vector(6 downto 0);
 39                 Q_LEDS      : out std_logic_vector(3 downto 0);
 40                 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:

```
 43     architecture Behavioral of avr_fpga is
```

src/avr_fpga.vhd

```
107     begin
```

src/avr_fpga.vhd

```
184     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:

```
 45     component cpu_core
 46         port (  I_CLK       : in  std_logic;
 47                 I_CLR       : in  std_logic;
 48                 I_INTVEC    : in  std_logic_vector( 5 downto 0);
 49                 I_DIN       : in  std_logic_vector( 7 downto 0);
 50
 51                 Q_OPC       : out std_logic_vector(15 downto 0);
 52                 Q_PC        : out std_logic_vector(15 downto 0);
 53                 Q_DOUT      : out std_logic_vector( 7 downto 0);
 54                 Q_ADR_IO    : out std_logic_vector( 7 downto 0);
 55                 Q_RD_IO     : out std_logic;
 56                 Q_WE_IO     : out std_logic);
 57     end component;
 58
 59     signal  C_PC            : std_logic_vector(15 downto 0);
 60     signal  C_OPC           : std_logic_vector(15 downto 0);
 61     signal  C_ADR_IO        : std_logic_vector( 7 downto 0);
 62     signal  C_DOUT          : std_logic_vector( 7 downto 0);
 63     signal  C_RD_IO         : std_logic;
 64     signal  C_WE_IO         : std_logic;
```

src/avr_fpga.vhd

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

```
 66     component io
 67         port (  I_CLK       : in  std_logic;
 68                 I_CLR       : in  std_logic;
 69                 I_ADR_IO    : in  std_logic_vector( 7 downto 0);
 70                 I_DIN       : in  std_logic_vector( 7 downto 0);
```

```
 71                 I_RD_IO     : in  std_logic;
 72                 I_WE_IO     : in  std_logic;
 73                 I_SWITCH    : in  std_logic_vector( 7 downto 0);
 74                 I_RX        : in  std_logic;
 75
 76                 Q_7_SEGMENT : out std_logic_vector( 6 downto 0);
 77                 Q_DOUT      : out std_logic_vector( 7 downto 0);
 78                 Q_INTVEC    : out std_logic_vector(5 downto 0);
 79                 Q_LEDS      : out std_logic_vector( 1 downto 0);
 80                 Q_TX        : out std_logic);
 81     end component;
 82
 83     signal N_INTVEC         : std_logic_vector( 5 downto 0);
 84     signal N_DOUT           : std_logic_vector( 7 downto 0);
 85     signal N_TX             : std_logic;
 86     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**:

```
 88     component segment7
 89         port ( I_CLK        : in  std_logic;
 90
 91                I_CLR        : in  std_logic;
 92                I_OPC        : in  std_logic_vector(15 downto 0);
 93                I_PC         : in  std_logic_vector(15 downto 0);
 94
 95                Q_7_SEGMENT  : out std_logic_vector( 6 downto 0));
 96     end component;
 97
 98     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:

```
100     signal L_CLK            : std_logic := '0';
101     signal L_CLK_CNT        : std_logic_vector( 2 downto 0) := "000";
102     signal L_CLR            : std_logic;            -- reset,  active low
103     signal L_CLR_N          : std_logic := '0';     -- reset,  active low
104     signal L_C1_N           : std_logic := '0';     -- switch debounce, active low
105     signal L_C2_N           : std_logic := '0';     -- switch debounce, active low
106
107     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:

```
109         cpu : cpu_core
110         port map(   I_CLK       => L_CLK,
111                     I_CLR       => L_CLR,
112                     I_DIN       => N_DOUT,
113                     I_INTVEC    => N_INTVEC,
114
115                     Q_ADR_IO    => C_ADR_IO,
116                     Q_DOUT      => C_DOUT,
117                     Q_OPC       => C_OPC,
118                     Q_PC        => C_PC,
119                     Q_RD_IO     => C_RD_IO,
120                     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**:

```
122         ino : io
123         port map(   I_CLK       => L_CLK,
124                     I_CLR       => L_CLR,
125                     I_ADR_IO    => C_ADR_IO,
126                     I_DIN       => C_DOUT,
127                     I_RD_IO     => C_RD_IO,
128                     I_RX        => I_RX,
129                     I_SWITCH    => I_SWITCH(7 downto 0),
130                     I_WE_IO     => C_WE_IO,
131
132                     Q_7_SEGMENT => N_7_SEGMENT,
133                     Q_DOUT      => N_DOUT,
134                     Q_INTVEC    => N_INTVEC,
135                     Q_LEDS      => Q_LEDS(1 downto 0),
136                     Q_TX        => N_TX);
137
138         seg : segment7
139         port map(   I_CLK       => L_CLK,
140                     I_CLR       => L_CLR,
141                     I_OPC       => C_OPC,
142                     I_PC        => C_PC,
143
144                     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.

```
148         clk_div : process(I_CLK_100)
149         begin
150             if (rising_edge(I_CLK_100)) then
151                 L_CLK_CNT      <= L_CLK_CNT + "001";
152                 if (L_CLK_CNT = "001") then
153                     L_CLK_CNT      <= "000";
154                     L_CLK      <= not L_CLK;
```

```
155                 end if;
156             end if;
157         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.

```
161         deb : process(L_CLK)
162         begin
163             if (rising_edge(L_CLK)) then
164                 -- switch debounce
165                 if ((I_SWITCH(8) = '0') or (I_SWITCH(9) = '0')) then    -- pushed
166                     L_CLR_N      <= '0';
167                     L_C2_N       <= '0';
168                     L_C1_N       <= '0';
169                 else                                                      -- released
170                     L_CLR_N      <= L_C2_N;
171                     L_C2_N       <= L_C1_N;
172                     L_C1_N       <= '1';
173                 end if;
174             end if;
175         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.

```
177         L_CLR       <= not L_CLR_N;
178
179         Q_LEDS(2)       <= I_RX;
180         Q_LEDS(3)       <= N_TX;
181         Q_7_SEGMENT        <= N_7_SEGMENT when (I_SWITCH(7) = '1') else S_7_SEGMENT;
182         Q_TX       <= N_TX;
```

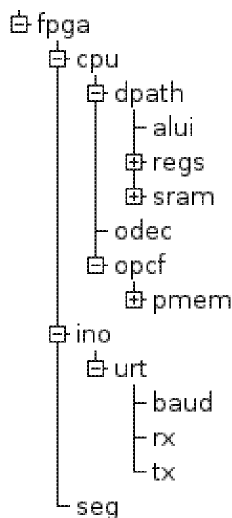
src/avr_fpga.vhd

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**.

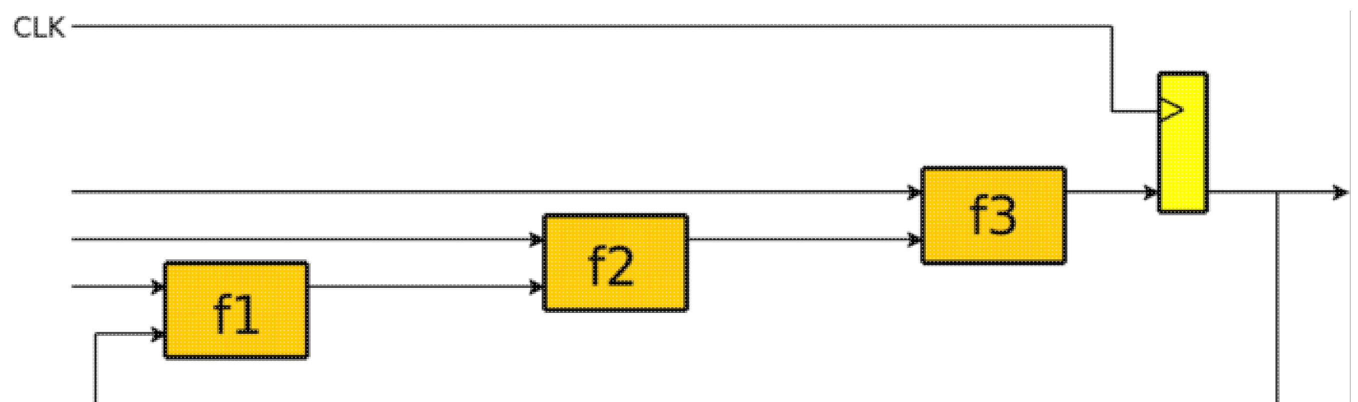
The **ino** contains a **uart** that is further broken down into a baud rate generator **baud**, a serial receiver **rx**, and a serial transmitter **tx**.

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.

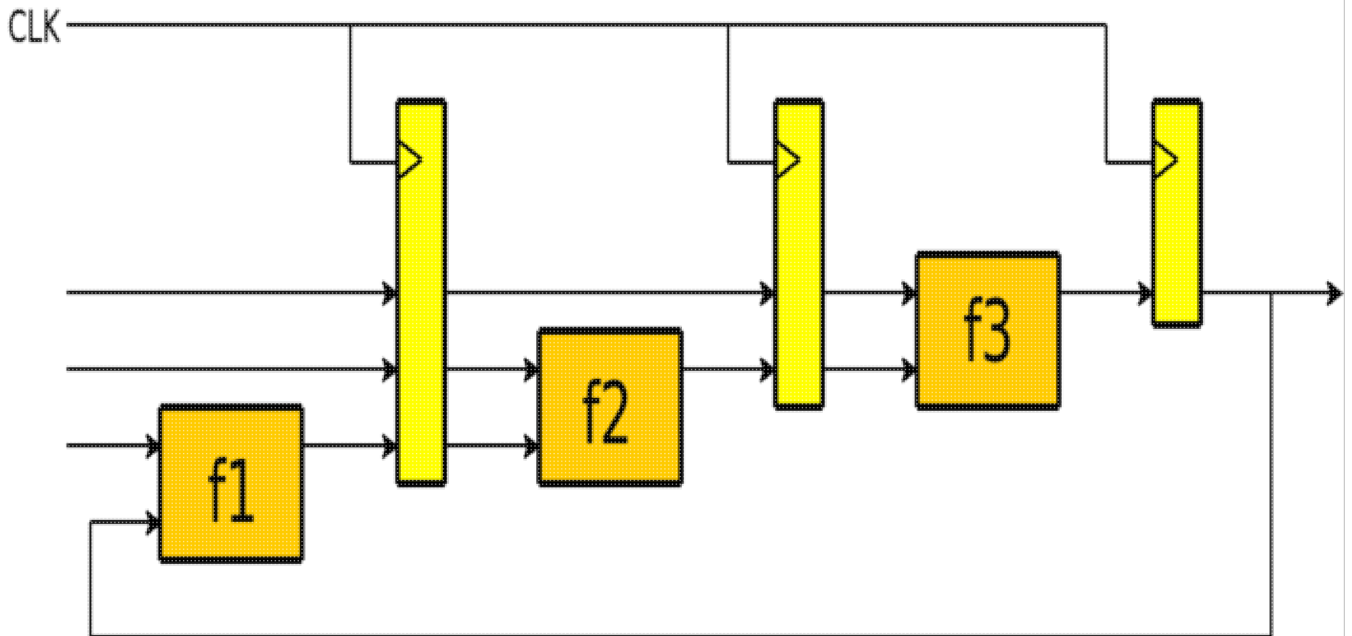


The circuit is a sequential logic which consists of 3 combinational functions f1, f2, and f3 and a flip-flop at the output of f3.

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.

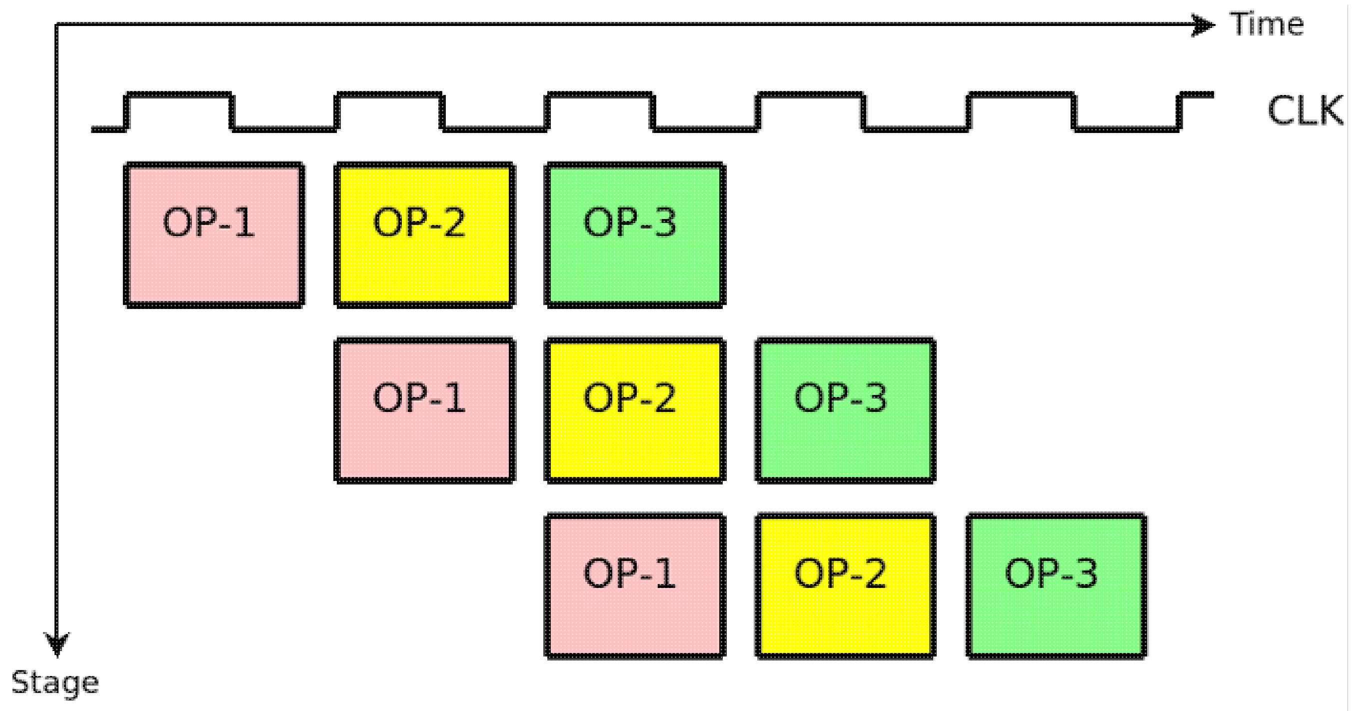


The slowest path is now max(t1, t2, t3) and the new circuit can be clocked with frequency 1/max(t1, t2, t3) instead of 1/(t1 + t2 + t3). If the functions f1, f2, and f3 had equal propagation delays, then the max. frequency of the new circuit would have tripled compared to the old circuit.

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

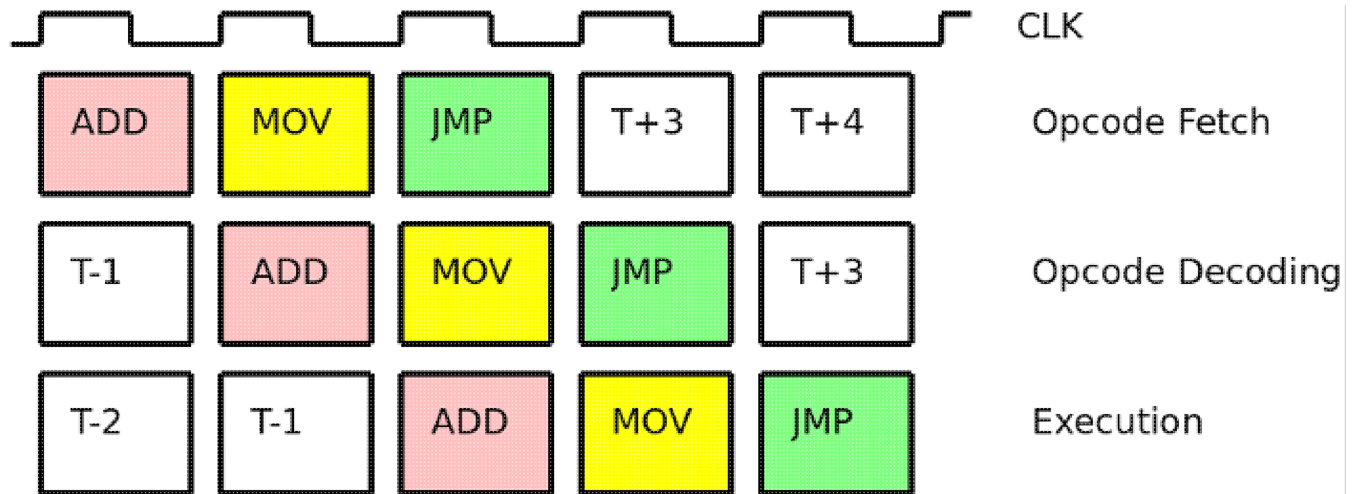
Opcode fetch means that for a given value of the program counter **PC**, the instruction (opcode) stored at location PC is read from the program memory and that the PC is advanced to the next instruction.

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

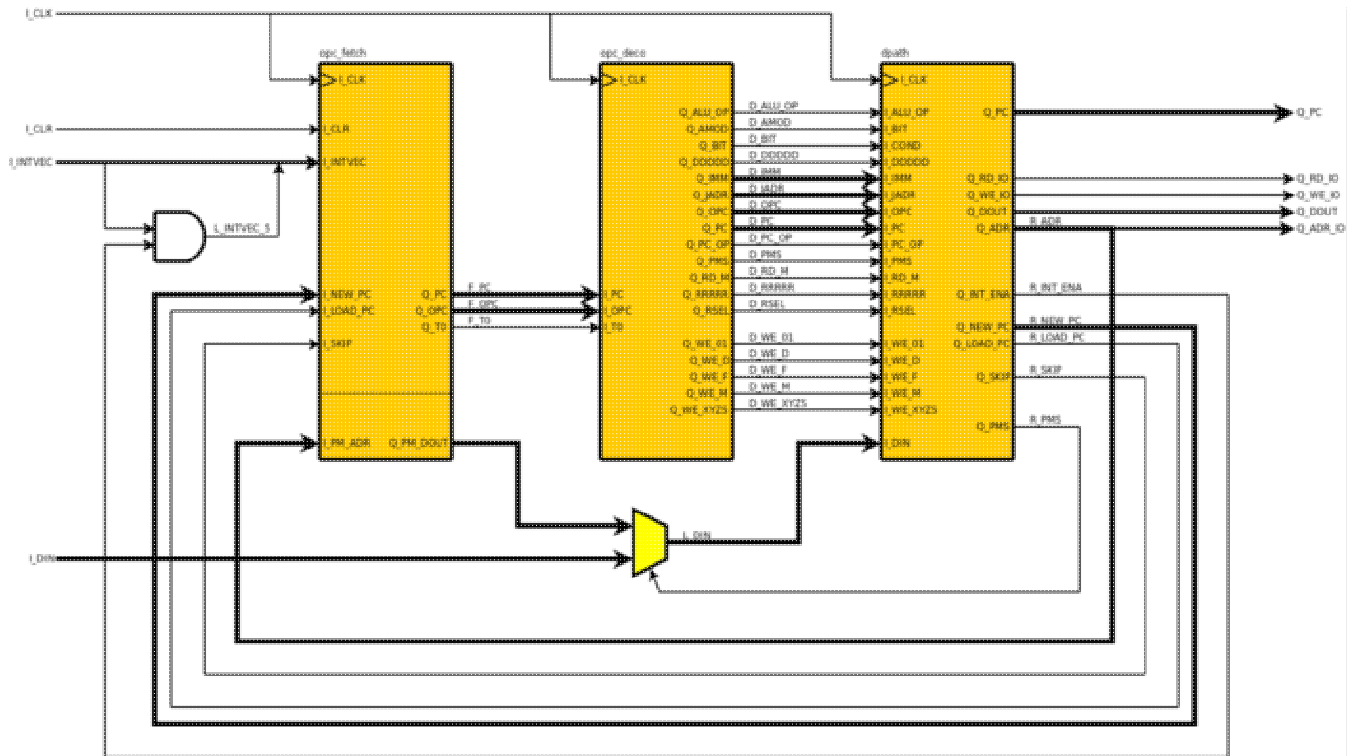
Execution then executes the opcode which means that a small number of registers or memory locations is read and/or written.

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
I_CLR
I_INTVEC
L_INTVEC_5
I_DIN
opc_fetch
I_CLK
I_CLR
I_INTVEC
I_NEW_PC
I_LOAD_PC
I_SKIP
I_PM_ADR
Q_PC
Q_OPC
Q_T0
Q_PM_DOUT
F_PC
F_OPC
F_T0
opc_deco
I_CLK
I_PC
I_OPC
I_T0
Q_ALU_OP
Q_AMOD
Q_BIT
Q_DDDDD
Q_IMM
Q_JADR
Q_OPC
Q_PC
Q_PC_OP
Q_PMS
Q_RD_M
Q_RRRRR
Q_RSEL
Q_WE_01
Q_WE_D
Q_WE_F
Q_WE_M
Q_WE_XYZS
D_ALU_OP
D_AMOD
D_BIT
D_DDDDD
D_IMM
D_JADR
D_OPC
D_PC
D_PC_OP
D_PMS
D_RD_M
D_RRRRR
D_RSEL
D_WE_01
D_WE_D
D_WE_F
D_WE_M
D_WE_XYZS
dpath
I_CLK
I_ALU_OP
I_BIT
I_COND
I_DDDDD
I_IMM
I_JADR
I_OPC
I_PC
I_PC_OP
I_PMS
I_RD_M
I_RRRRR
I_RSEL
I_WE_01
I_WE_D
I_WE_F
I_WE_M
I_WE_XYZS
I_DIN
Q_PC
Q_RD_IO
Q_WE_IO
Q_DOUT
Q_ADR
Q_INT_ENA
Q_NEW_PC
Q_LOAD_PC
Q_SKIP
Q_PMS
R_ADR
R_INT_ENA
R_NEW_PC
R_LOAD_PC
R_SKIP
R_PMS
L_DIN
Q_PC
Q_RD_IO
Q_WE_IO
Q_DOUT
Q_ADR_IO

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:

```
 33     entity cpu_core is
 34         port (  I_CLK       : in  std_logic;
 35                 I_CLR       : in  std_logic;
 36                 I_INTVEC    : in  std_logic_vector( 5 downto 0);
 37                 I_DIN       : in  std_logic_vector( 7 downto 0);
 38
 39                 Q_OPC       : out std_logic_vector(15 downto 0);
 40                 Q_PC        : out std_logic_vector(15 downto 0);
 41                 Q_DOUT      : out std_logic_vector( 7 downto 0);
 42                 Q_ADR_IO    : out std_logic_vector( 7 downto 0);
 43                 Q_RD_IO     : out std_logic;
 44                 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**):

```
240         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):

```
241         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:

```
 69     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:

```
 86         lpc: process(I_CLK)
 87          begin
 88              if (rising_edge(I_CLK)) then
 89                  L_PC      <= L_NEXT_PC;
 90                  L_T0      <= not L_WAIT;
 91              end if;
 92          end process;
```

src/opc_fetch.vhd

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

```
 94          L_NEXT_PC      <= X"0000"        when (I_CLR     = '1')
 95                  else L_PC           when (L_WAIT    = '1')
 96                  else I_NEW_PC       when (I_LOAD_PC = '1')
 97                  else L_PC + X"0002" when (L_LONG_OP = '1')
 98                  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:

```
100         -- Two word opcodes:
101         --
102         --        9       3210
103         -- 1001 000d dddd 0000 kkkk kkkk kkkk kkkk - LDS
104         -- 1001 001d dddd 0000 kkkk kkkk kkkk kkkk - SDS
105         -- 1001 010k kkkk 110k kkkk kkkk kkkk kkkk - JMP
106         -- 1001 010k kkkk 111k kkkk kkkk kkkk kkkk - CALL
107         --
108         L_LONG_OP       <= '1' when (((P_OPC(15 downto  9) = "1001010") and
109                                 (P_OPC( 3 downto  2) = "11"))        -- JMP, CALL
110                            or  ((P_OPC(15 downto 10) = "100100") and
111                                 (P_OPC( 3 downto  0) = "0000")))    -- LDS, STS
112                 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.

```
133         L_INVALIDATE       <= I_CLR or I_SKIP;
134
135         Q_OPC       <= X"00000000" when (L_INVALIDATE = '1')
136             else P_OPC       when (I_INTVEC(5) = '0')
137             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:

```
36     entity prog_mem is
37         port (  I_CLK       : in  std_logic;
38
39                 I_WAIT      : in  std_logic;
40                 I_PC        : in  std_logic_vector(15 downto 0); -- word address
41                 I_PM_ADR    : in  std_logic_vector(11 downto 0); -- byte address
42
43                 Q_OPC       : out std_logic_vector(31 downto 0);
44                 Q_PC        : out std_logic_vector(15 downto 0);
45                 Q_PM_DOUT   : out std_logic_vector( 7 downto 0));
46     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

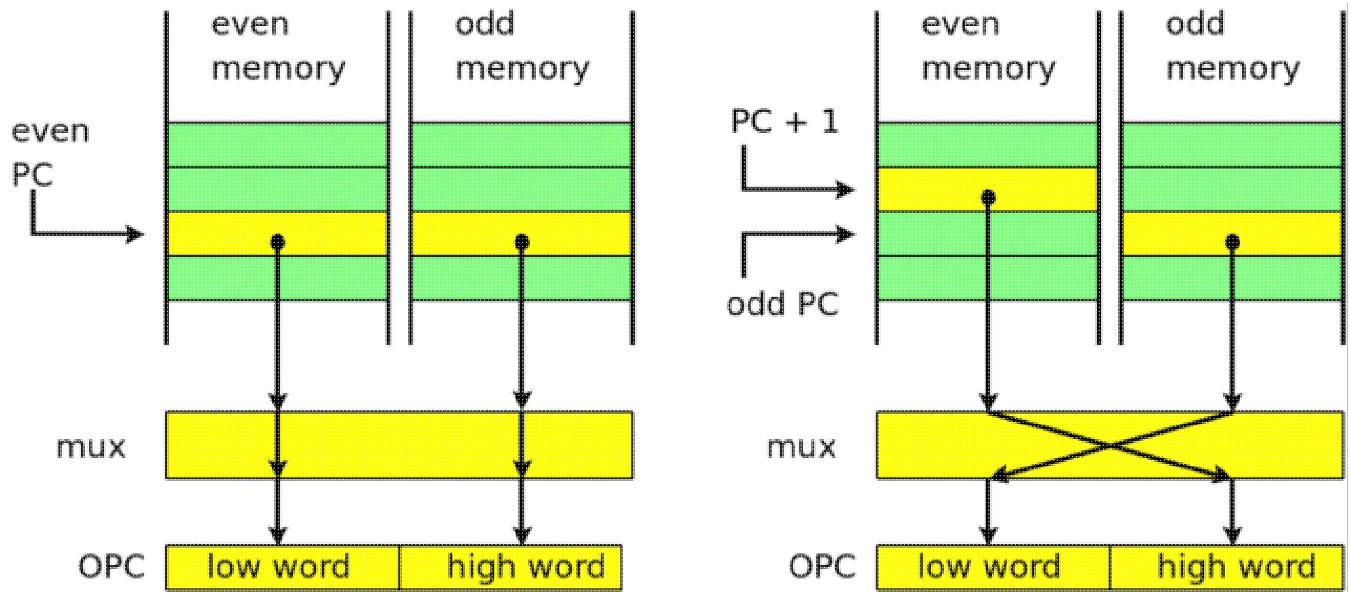
The vast majority of AVR instructions are single-word (16-bit) instructions. There are 4 exceptions, which are **CALL**, **JMP**, **LDS**, and **STS**. These instructions have addresses (the target address for **CALL** and **JMP** and data memory address for $LDS# and **STS**) in the word following the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

```
252         L_PC_O      <= I_PC(10 downto 1);
253         L_PC_E      <= I_PC(10 downto 1) + ("000000000" & I_PC(0));
254         Q_OPC(15 downto  0)      <= M_OPC_E when L_PC_0 = '0' else M_OPC_O;
255         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**:

```
224         pc0: process(I_CLK)
225         begin
226             if (rising_edge(I_CLK)) then
227                 Q_PC      <= I_PC;
228                 L_PM_ADR_1_0      <= I_PM_ADR(1 downto 0);
229                 if ((I_WAIT = '0')) then
230                     L_PC_0      <= I_PC(0);
231                 end if;
232             end if;
233         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:

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

```
109         port map(ADDRA => L_PC_E,                ADDRB => I_PM_ADR(11 downto 2),
110                  CLKA  => I_CLK,                 CLKB  => I_CLK,
111                  DIA   => "0000",                DIB   => "0000",
112                  ENA   => L_WAIT_N,              ENB   => '1',
113                  RSTA  => '0',                   RSTB  => '0',
114                  WEA   => '0',                   WEB   => '0',
115                  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:

```
 34     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).

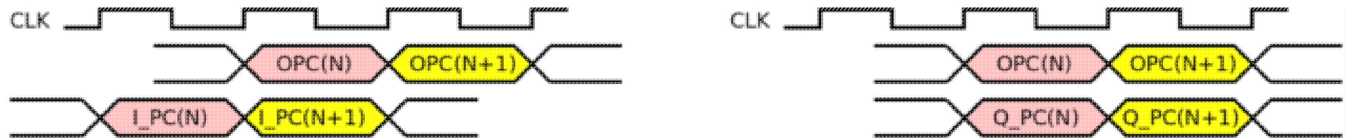
There are other ways of initializing the memory modules, such as updating sections of the bitmap file, but we found the above sequence easier to use.

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:

```
227                 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:

```
114         -- Two cycle opcodes:
115         --
116         -- 1001 000d dddd .... - LDS etc.
117         -- 1001 0101 0000 1000 - RET
118         -- 1001 0101 0001 1000 - RETI
119         -- 1001 1001 AAAA Abbb - SBIC
120         -- 1001 1011 AAAA Abbb - SBIS
121         -- 1111 110r rrrr 0bbb - SBRC
122         -- 1111 111r rrrr 0bbb - SBRS
123         --
124         L_WAIT      <= '0'  when (L_INVALIDATE = '1')
125              else '0'  when (I_INTVEC(5)   = '1')
126              else L_T0 when ((P_OPC(15 downto   9) = "1001000" )     -- LDS etc.
127                          or   (P_OPC(15 downto   8) = "10010101")     -- RET etc.
128                          or   ((P_OPC(15 downto 10) = "100110")      -- SBIC, SBIS
129                            and P_OPC(8) = '1')
```

```
130                          or  (P_OPC(15 downto  10) = "111111"))     -- SBRC, SBRS
131             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:

```
169                             L_INTVEC      <= "101011";       -- _VECTOR(11)
170                         end if;
171                     elsif (L_TX_INT_ENABLED and not U_TX_BUSY) = '1' then
172                         if (L_INTVEC(5) = '0') then     -- no interrupt pending
173                             L_INTVEC      <= "101100";       -- _VECTOR(12)
174                         end if;
175                     else                                -- no interrupt
176                         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):

```
241         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":

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

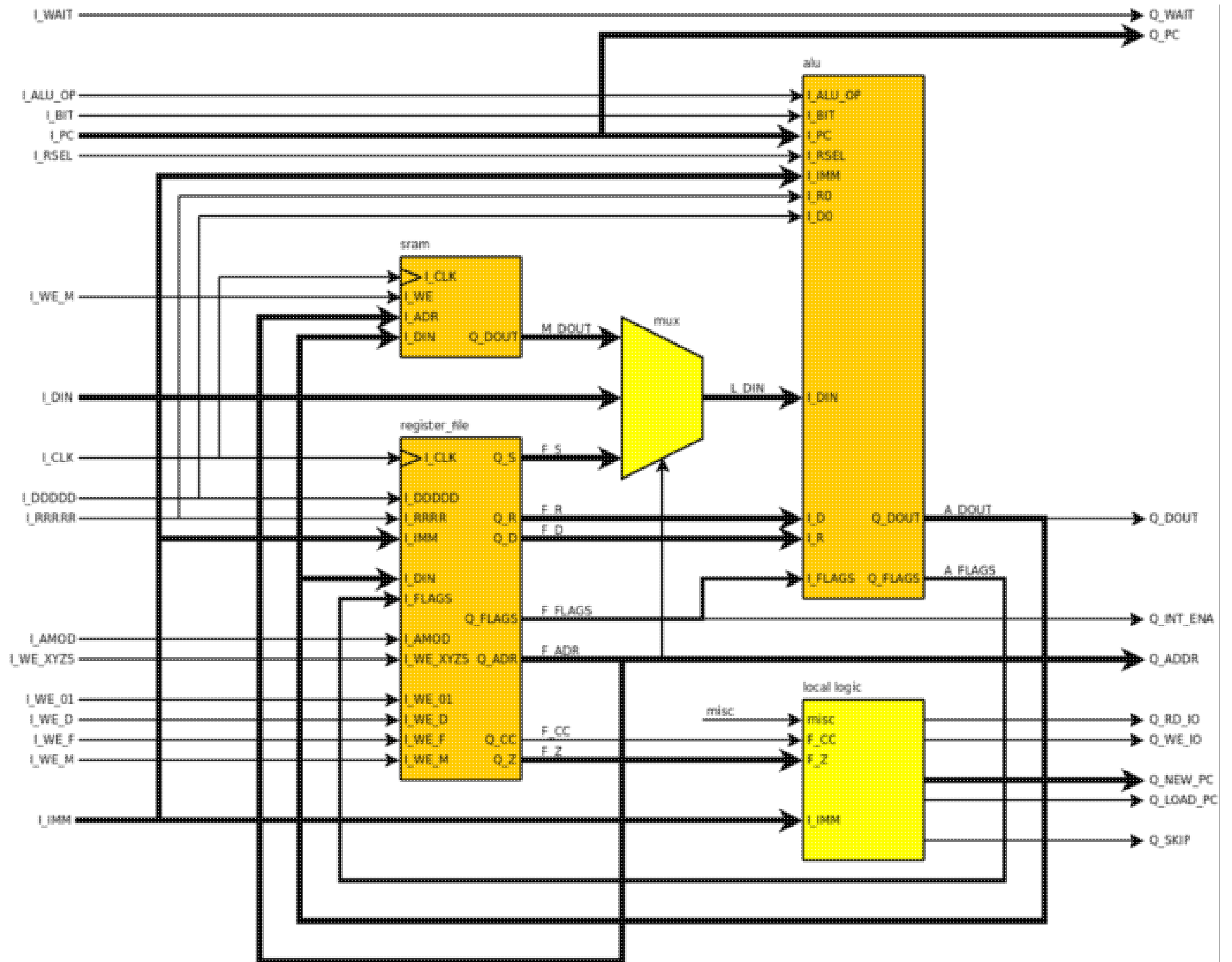
src/opc_fetch.vhd

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
Q_WAIT
Q_PC
alu
I_ALU_OP
I_BIT
I_PC
I_RSEL
I_IMM
I_R0
I_D0
sram
I_CLK
I_WE_M
I_WE
I_ADR
I_DIN
Q_DOUT
M_DOUT
mux
L_DIN
I_DIN
register_file
I_CLK
Q_S
F_S
I_DDDDD
I_RRRRR
Q_R
F_R
I_IMM
Q_D
F_D
I_D
I_R
Q_DOUT
A_DOUT
Q_DOUT
I_FLAGS
Q_FLAGS
A_FLAGS
F_FLAGS
Q_INT_ENA
I_AMOD
I_WE_XYZS
Q_ADR
F_ADR
Q_ADDR
local logic
I_WE_01
I_WE_D
I_WE_F
I_WE_M
Q_CC
Q_Z
F_CC
F_Z
misc
Q_RD_IO
Q_WE_IO
Q_NEW_PC
Q_LOAD_PC
I_IMM
Q_SKIP

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:

```
32     entity reg_16 is
33         port (  I_CLK       : in  std_logic;
34
35                 I_D         : in  std_logic_vector (15 downto 0);
36                 I_WE        : in  std_logic_vector ( 1 downto 0);
37
38                 Q           : out std_logic_vector (15 downto 0));
39     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**:

```
46         process(I_CLK)
47         begin
48             if (rising_edge(I_CLK)) then
49                 if (I_WE(1) = '1') then
50                     L(15 downto 8)      <= I_D(15 downto 8);
51                 end if;
52                 if (I_WE(0) = '1') then
53                     L( 7 downto 0)      <= I_D( 7 downto 0);
54                 end if;
55             end if;
56         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:

```
 32     entity status_reg is
 33         port (  I_CLK       : in  std_logic;
 34
 35                 I_COND      : in  std_logic_vector ( 3 downto 0);
 36                 I_DIN       : in  std_logic_vector ( 7 downto 0);
 37                 I_FLAGS     : in  std_logic_vector ( 7 downto 0);
 38                 I_WE_F      : in  std_logic;
 39                 I_WE_SR     : in  std_logic;
 40
 41                 Q           : out std_logic_vector ( 7 downto 0);
 42                 Q_CC        : out std_logic);
 43     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**:

```
131         r00: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 1 downto  0), I_D => I_DIN, Q => R
132         r02: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 3 downto  2), I_D => I_DIN, Q => R
133         r04: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 5 downto  4), I_D => I_DIN, Q => R
134         r06: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 7 downto  6), I_D => I_DIN, Q => R
135         r08: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 9 downto  8), I_D => I_DIN, Q => R
136         r10: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(11 downto 10), I_D => I_DIN, Q => R
```

```
137         r12: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(13 downto 12), I_D => I_DIN, Q => R
138         r14: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(15 downto 14), I_D => I_DIN, Q => R
139         r16: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(17 downto 16), I_D => I_DIN, Q => R
140         r18: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(19 downto 18), I_D => I_DIN, Q => R
141         r20: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(21 downto 20), I_D => I_DIN, Q => R
142         r22: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(23 downto 22), I_D => I_DIN, Q => R
143         r24: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(25 downto 24), I_D => I_DIN, Q => R
144         r26: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(27 downto 26), I_D => L_DX,  Q => R
145         r28: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(29 downto 28), I_D => L_DY,  Q => R
146         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

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

src/register_file.vhd

```
149         sr: status_reg
150         port map(   I_CLK       => I_CLK,
151                     I_COND      => I_COND,
152                     I_DIN       => I_DIN(7 downto 0),
153                     I_FLAGS     => I_FLAGS,
154                     I_WE_F      => I_WE_F,
155                     I_WE_SR     => L_WE_SR,
156                     Q           => S_FLAGS,
157                     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:

```
 71     signal R_R00            : std_logic_vector(15 downto 0);
 72     signal R_R02            : std_logic_vector(15 downto 0);
 73     signal R_R04            : std_logic_vector(15 downto 0);
 74     signal R_R06            : std_logic_vector(15 downto 0);
 75     signal R_R08            : std_logic_vector(15 downto 0);
 76     signal R_R10            : std_logic_vector(15 downto 0);
 77     signal R_R12            : std_logic_vector(15 downto 0);
 78     signal R_R14            : std_logic_vector(15 downto 0);
 79     signal R_R16            : std_logic_vector(15 downto 0);
 80     signal R_R18            : std_logic_vector(15 downto 0);
 81     signal R_R20            : std_logic_vector(15 downto 0);
 82     signal R_R22            : std_logic_vector(15 downto 0);
 83     signal R_R24            : std_logic_vector(15 downto 0);
 84     signal R_R26            : std_logic_vector(15 downto 0);
 85     signal R_R28            : std_logic_vector(15 downto 0);
 86     signal R_R30            : std_logic_vector(15 downto 0);
 87     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)**:

```
189         process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
190                 R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
191                 I_DDDDD(4 downto 1))
192         begin
193             case I_DDDDD(4 downto 1) is
194                 when "0000" => Q_D      <= R_R00;
195                 when "0001" => Q_D      <= R_R02;
196                 when "0010" => Q_D      <= R_R04;
197                 when "0011" => Q_D      <= R_R06;
198                 when "0100" => Q_D      <= R_R08;
199                 when "0101" => Q_D      <= R_R10;
200                 when "0110" => Q_D      <= R_R12;
201                 when "0111" => Q_D      <= R_R14;
202                 when "1000" => Q_D      <= R_R16;
203                 when "1001" => Q_D      <= R_R18;
204                 when "1010" => Q_D      <= R_R20;
205                 when "1011" => Q_D      <= R_R22;
206                 when "1100" => Q_D      <= R_R24;
207                 when "1101" => Q_D      <= R_R26;
208                 when "1110" => Q_D      <= R_R28;
209                 when others => Q_D      <= R_R30;
210             end case;
211         end process;
```

src/register_file.vhd

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

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

```
218             case I_RRRR is
219                 when "0000" => Q_R      <= R_R00;
220                 when "0001" => Q_R      <= R_R02;
221                 when "0010" => Q_R      <= R_R04;
222                 when "0011" => Q_R      <= R_R06;
223                 when "0100" => Q_R      <= R_R08;
224                 when "0101" => Q_R      <= R_R10;
225                 when "0110" => Q_R      <= R_R12;
226                 when "0111" => Q_R      <= R_R14;
227                 when "1000" => Q_R      <= R_R16;
228                 when "1001" => Q_R      <= R_R18;
229                 when "1010" => Q_R      <= R_R20;
230                 when "1011" => Q_R      <= R_R22;
231                 when "1100" => Q_R      <= R_R24;
232                 when "1101" => Q_R      <= R_R26;
233                 when "1110" => Q_R      <= R_R28;
234                 when others => Q_R      <= R_R30;
235             end case;
236         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**:

```
161         process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
162                 R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
163                 R_SP, S_FLAGS, L_ADR(6 downto 1))
164         begin
165             case L_ADR(6 downto 1) is
166                 when "000000" => L_S      <= R_R00;
167                 when "000001" => L_S      <= R_R02;
168                 when "000010" => L_S      <= R_R04;
169                 when "000011" => L_S      <= R_R06;
170                 when "000100" => L_S      <= R_R08;
171                 when "000101" => L_S      <= R_R10;
172                 when "000110" => L_S      <= R_R12;
173                 when "000111" => L_S      <= R_R14;
```

```
174                 when "001000" => L_S      <= R_R16;
175                 when "001001" => L_S      <= R_R18;
176                 when "001010" => L_S      <= R_R20;
177                 when "001011" => L_S      <= R_R22;
178                 when "001100" => L_S      <= R_R24;
179                 when "001101" => L_S      <= R_R26;
180                 when "001110" => L_S      <= R_R28;
181                 when "001111" => L_S      <= R_R30;
182                 when "101111" => L_S      <= R_SP ( 7 downto 0) & X"00";     -- SPL
183                 when others   => L_S      <= S_FLAGS & R_SP (15 downto 8);   -- SR/SPH
184             end case;
185         end process;
186
```

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**:

```
288         L_WE_D( 0)      <= I_WE_D(0) when (I_DDDDD = "00000") else '0';
289         L_WE_D( 1)      <= I_WE_D(0) when (I_DDDDD = "00001") else '0';
290         L_WE_D( 2)      <= I_WE_D(0) when (I_DDDDD = "00010") else '0';
291         L_WE_D( 3)      <= I_WE_D(0) when (I_DDDDD = "00011") else '0';
292         L_WE_D( 4)      <= I_WE_D(0) when (I_DDDDD = "00100") else '0';
293         L_WE_D( 5)      <= I_WE_D(0) when (I_DDDDD = "00101") else '0';
294         L_WE_D( 6)      <= I_WE_D(0) when (I_DDDDD = "00110") else '0';
295         L_WE_D( 7)      <= I_WE_D(0) when (I_DDDDD = "00111") else '0';
296         L_WE_D( 8)      <= I_WE_D(0) when (I_DDDDD = "01000") else '0';
297         L_WE_D( 9)      <= I_WE_D(0) when (I_DDDDD = "01001") else '0';
298         L_WE_D(10)      <= I_WE_D(0) when (I_DDDDD = "01010") else '0';
299         L_WE_D(11)      <= I_WE_D(0) when (I_DDDDD = "01011") else '0';
300         L_WE_D(12)      <= I_WE_D(0) when (I_DDDDD = "01100") else '0';
301         L_WE_D(13)      <= I_WE_D(0) when (I_DDDDD = "01101") else '0';
302         L_WE_D(14)      <= I_WE_D(0) when (I_DDDDD = "01110") else '0';
303         L_WE_D(15)      <= I_WE_D(0) when (I_DDDDD = "01111") else '0';
304         L_WE_D(16)      <= I_WE_D(0) when (I_DDDDD = "10000") else '0';
305         L_WE_D(17)      <= I_WE_D(0) when (I_DDDDD = "10001") else '0';
306         L_WE_D(18)      <= I_WE_D(0) when (I_DDDDD = "10010") else '0';
```

```
307         L_WE_D(19)      <= I_WE_D(0) when (I_DDDDD = "10011") else '0';
308         L_WE_D(20)      <= I_WE_D(0) when (I_DDDDD = "10100") else '0';
309         L_WE_D(21)      <= I_WE_D(0) when (I_DDDDD = "10101") else '0';
310         L_WE_D(22)      <= I_WE_D(0) when (I_DDDDD = "10110") else '0';
311         L_WE_D(23)      <= I_WE_D(0) when (I_DDDDD = "10111") else '0';
312         L_WE_D(24)      <= I_WE_D(0) when (I_DDDDD = "11000") else '0';
313         L_WE_D(25)      <= I_WE_D(0) when (I_DDDDD = "11001") else '0';
314         L_WE_D(26)      <= I_WE_D(0) when (I_DDDDD = "11010") else '0';
315         L_WE_D(27)      <= I_WE_D(0) when (I_DDDDD = "11011") else '0';
316         L_WE_D(28)      <= I_WE_D(0) when (I_DDDDD = "11100") else '0';
317         L_WE_D(29)      <= I_WE_D(0) when (I_DDDDD = "11101") else '0';
318         L_WE_D(30)      <= I_WE_D(0) when (I_DDDDD = "11110") else '0';
319         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**:

```
326         L_DDDD      <= I_DDDDD(4 downto 1);
327         L_WE_D2      <= I_WE_D(1) & I_WE_D(1);
328         L_WE_DD( 1 downto  0)      <= L_WE_D2 when (L_DDDD = "0000") else "00";
329         L_WE_DD( 3 downto  2)      <= L_WE_D2 when (L_DDDD = "0001") else "00";
330         L_WE_DD( 5 downto  4)      <= L_WE_D2 when (L_DDDD = "0010") else "00";
331         L_WE_DD( 7 downto  6)      <= L_WE_D2 when (L_DDDD = "0011") else "00";
332         L_WE_DD( 9 downto  8)      <= L_WE_D2 when (L_DDDD = "0100") else "00";
333         L_WE_DD(11 downto 10)      <= L_WE_D2 when (L_DDDD = "0101") else "00";
334         L_WE_DD(13 downto 12)      <= L_WE_D2 when (L_DDDD = "0110") else "00";
335         L_WE_DD(15 downto 14)      <= L_WE_D2 when (L_DDDD = "0111") else "00";
336         L_WE_DD(17 downto 16)      <= L_WE_D2 when (L_DDDD = "1000") else "00";
337         L_WE_DD(19 downto 18)      <= L_WE_D2 when (L_DDDD = "1001") else "00";
338         L_WE_DD(21 downto 20)      <= L_WE_D2 when (L_DDDD = "1010") else "00";
339         L_WE_DD(23 downto 22)      <= L_WE_D2 when (L_DDDD = "1011") else "00";
340         L_WE_DD(25 downto 24)      <= L_WE_D2 when (L_DDDD = "1100") else "00";
341         L_WE_DD(27 downto 26)      <= L_WE_D2 when (L_DDDD = "1101") else "00";
342         L_WE_DD(29 downto 28)      <= L_WE_D2 when (L_DDDD = "1110") else "00";
343         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**:

```
350         L_WE_IO( 0)       <= L_WE_A when (L_ADR(4 downto 0) = "00000") else '0';
351         L_WE_IO( 1)       <= L_WE_A when (L_ADR(4 downto 0) = "00001") else '0';
352         L_WE_IO( 2)       <= L_WE_A when (L_ADR(4 downto 0) = "00010") else '0';
353         L_WE_IO( 3)       <= L_WE_A when (L_ADR(4 downto 0) = "00011") else '0';
354         L_WE_IO( 4)       <= L_WE_A when (L_ADR(4 downto 0) = "00100") else '0';
355         L_WE_IO( 5)       <= L_WE_A when (L_ADR(4 downto 0) = "00101") else '0';
356         L_WE_IO( 6)       <= L_WE_A when (L_ADR(4 downto 0) = "00110") else '0';
357         L_WE_IO( 7)       <= L_WE_A when (L_ADR(4 downto 0) = "00111") else '0';
358         L_WE_IO( 8)       <= L_WE_A when (L_ADR(4 downto 0) = "01000") else '0';
359         L_WE_IO( 9)       <= L_WE_A when (L_ADR(4 downto 0) = "01001") else '0';
360         L_WE_IO(10)       <= L_WE_A when (L_ADR(4 downto 0) = "01010") else '0';
361         L_WE_IO(11)       <= L_WE_A when (L_ADR(4 downto 0) = "01011") else '0';
362         L_WE_IO(12)       <= L_WE_A when (L_ADR(4 downto 0) = "01100") else '0';
363         L_WE_IO(13)       <= L_WE_A when (L_ADR(4 downto 0) = "01101") else '0';
364         L_WE_IO(14)       <= L_WE_A when (L_ADR(4 downto 0) = "01110") else '0';
365         L_WE_IO(15)       <= L_WE_A when (L_ADR(4 downto 0) = "01111") else '0';
366         L_WE_IO(16)       <= L_WE_A when (L_ADR(4 downto 0) = "10000") else '0';
367         L_WE_IO(17)       <= L_WE_A when (L_ADR(4 downto 0) = "10001") else '0';
368         L_WE_IO(18)       <= L_WE_A when (L_ADR(4 downto 0) = "10010") else '0';
369         L_WE_IO(19)       <= L_WE_A when (L_ADR(4 downto 0) = "10011") else '0';
370         L_WE_IO(20)       <= L_WE_A when (L_ADR(4 downto 0) = "10100") else '0';
371         L_WE_IO(21)       <= L_WE_A when (L_ADR(4 downto 0) = "10101") else '0';
372         L_WE_IO(22)       <= L_WE_A when (L_ADR(4 downto 0) = "10110") else '0';
373         L_WE_IO(23)       <= L_WE_A when (L_ADR(4 downto 0) = "10111") else '0';
374         L_WE_IO(24)       <= L_WE_A when (L_ADR(4 downto 0) = "11000") else '0';
375         L_WE_IO(25)       <= L_WE_A when (L_ADR(4 downto 0) = "11001") else '0';
376         L_WE_IO(26)       <= L_WE_A when (L_ADR(4 downto 0) = "11010") else '0';
377         L_WE_IO(27)       <= L_WE_A when (L_ADR(4 downto 0) = "11011") else '0';
378         L_WE_IO(28)       <= L_WE_A when (L_ADR(4 downto 0) = "11100") else '0';
379         L_WE_IO(29)       <= L_WE_A when (L_ADR(4 downto 0) = "11101") else '0';
380         L_WE_IO(30)       <= L_WE_A when (L_ADR(4 downto 0) = "11110") else '0';
381         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:

```
389         L_WE_X      <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WX) else '0';
390         L_WE_Y      <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WY) else '0';
391         L_WE_Z      <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WZ) else '0';
392         L_WE_MISC      <= L_WE_Z & L_WE_Z &      -- -Z and Z+ address modes  r30
393                      L_WE_Y & L_WE_Y &      -- -Y and Y+ address modes  r28
394                      L_WE_X & L_WE_X &      -- -X and X+ address modes  r26
395                      X"000000" &            -- never                    r24 - r02
396                      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:

```
398         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**).

```
280         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:

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

src/register_file.vhd

```
152                     I_DIN       => I_DIN(7 downto 0),
153                     I_FLAGS     => I_FLAGS,
154                     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:

```
 84         constant AS_SP  : std_logic_vector(2 downto 0) := "000";     -- SP
 85         constant AS_Z   : std_logic_vector(2 downto 0) := "001";     -- Z
 86         constant AS_Y   : std_logic_vector(2 downto 0) := "010";     -- Y
 87         constant AS_X   : std_logic_vector(2 downto 0) := "011";     -- X
 88         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:

```
 91         constant AO_0   : std_logic_vector(5 downto 3) := "000";     -- as is
 92         constant AO_Q   : std_logic_vector(5 downto 3) := "010";     -- +q
 93         constant AO_i   : std_logic_vector(5 downto 3) := "001";     -- +1
 94         constant AO_ii  : std_logic_vector(5 downto 3) := "011";     -- +2
 95         constant AO_d   : std_logic_vector(5 downto 3) := "101";     -- -1
 96         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:

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

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

src/register_file.vhd

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

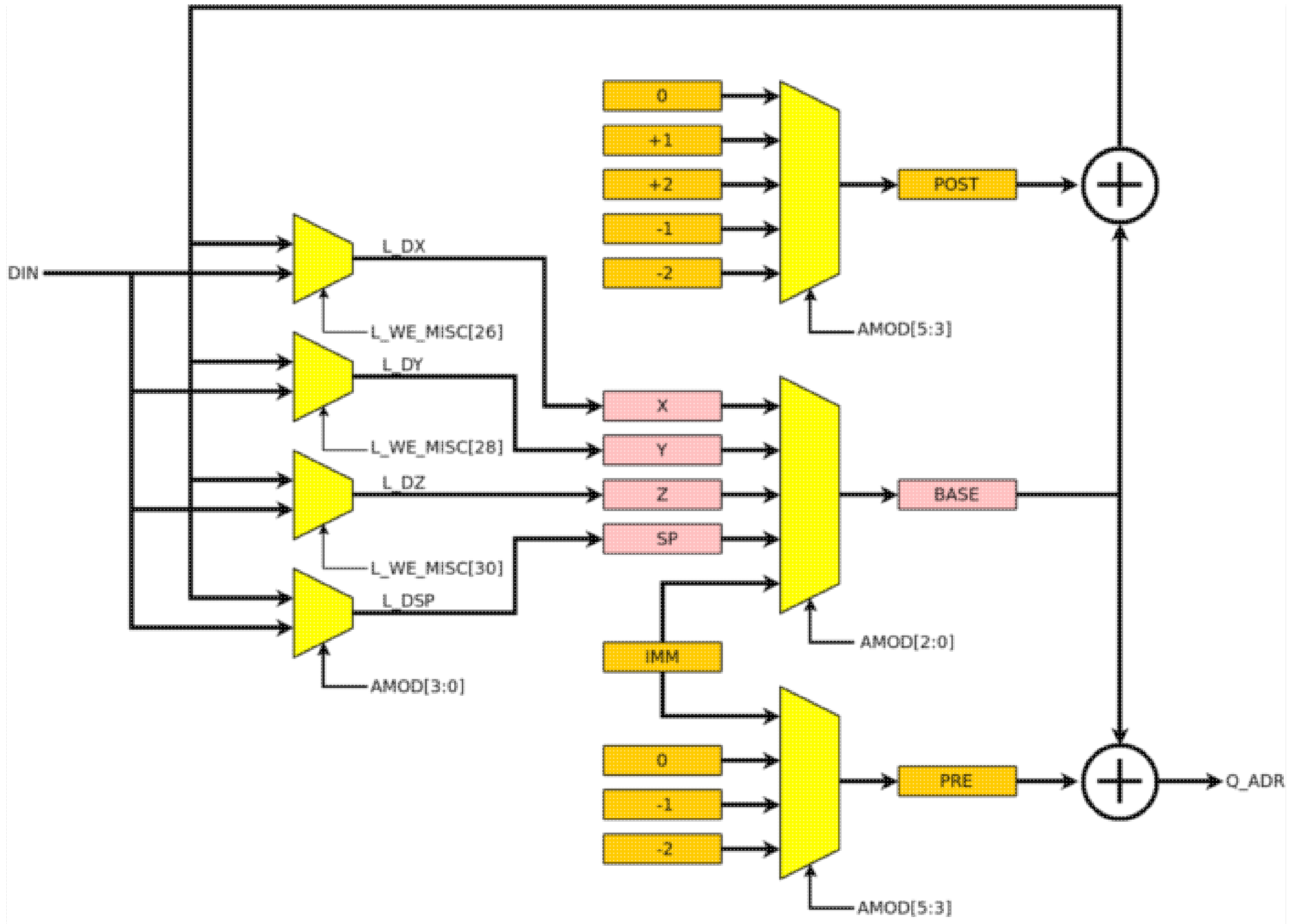
src/register_file.vhd

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

```
108         constant AMOD_ABS : std_logic_vector(5 downto 0) := AO_0  & AS_IMM; -- IMM
109         constant AMOD_X   : std_logic_vector(5 downto 0) := AO_0  & AS_X;   -- (X)
110         constant AMOD_Xq  : std_logic_vector(5 downto 0) := AO_Q  & AS_X;   -- (X+q)
111         constant AMOD_Xi  : std_logic_vector(5 downto 0) := AO_i  & AS_X;   -- (X++)
112         constant AMOD_dX  : std_logic_vector(5 downto 0) := AO_d  & AS_X;   -- (--X)
113         constant AMOD_Y   : std_logic_vector(5 downto 0) := AO_0  & AS_Y;   -- (Y)
114         constant AMOD_Yq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Y;   -- (Y+q)
115         constant AMOD_Yi  : std_logic_vector(5 downto 0) := AO_i  & AS_Y;   -- (Y++)
116         constant AMOD_dY  : std_logic_vector(5 downto 0) := AO_d  & AS_Y;   -- (--Y)
117         constant AMOD_Z   : std_logic_vector(5 downto 0) := AO_0  & AS_Z;   -- (Z)
118         constant AMOD_Zq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Z;   -- (Z+q)
119         constant AMOD_Zi  : std_logic_vector(5 downto 0) := AO_i  & AS_Z;   -- (Z++)
120         constant AMOD_dZ  : std_logic_vector(5 downto 0) := AO_d  & AS_Z;   -- (--Z)
121         constant AMOD_SPi : std_logic_vector(5 downto 0) := AO_i  & AS_SP;  -- (SP++)
122         constant AMOD_SPii: std_logic_vector(5 downto 0) := AO_ii & AS_SP;  -- (SP++)
123         constant AMOD_dSP : std_logic_vector(5 downto 0) := AO_d  & AS_SP;  -- (--SP)
124         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]
0
+1
+2
-1
-2
POST
AMOD[5:3]
X
Y
Z
SP
BASE
IMM
AMOD[2:0]
0
-1
-2
PRE
AMOD[5:3]
Q_ADR

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:

```
179         L_DIN_E       <= I_DIN( 7 downto 0) when (I_ADR(0) = '0') else I_DIN(15 downto 8);
180         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]**:

```
178                     Q_D         => F_D,
```

src/data_path.vhd

```
146                      I_D          => F_D,
147                      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**:

```
356         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.

```
357         L_R8       <= I_R(15 downto 8) when (I_R0 = '1') else I_R(7 downto 0);
358         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.

```
360         L_ADIW_D        <= I_D + ("0000000000" & I_IMM(5 downto 0));
361         L_SBIW_D        <= I_D - ("0000000000" & I_IMM(5 downto 0));
362         L_ADD_DR        <= L_D8 + L_RI8;
363         L_ADC_DR        <= L_ADD_DR + ("0000000" & I_FLAGS(0));
364         L_ASR_D         <= L_D8(7) & L_D8(7 downto 1);
365         L_AND_DR        <= L_D8 and L_RI8;
366         L_DEC_D         <= L_D8 - X"01";
367         L_INC_D         <= L_D8 + X"01";
368         L_LSR_D         <= '0' & L_D8(7 downto 1);
369         L_NEG_D         <= X"00" - L_D8;
370         L_NOT_D         <= not L_D8;
371         L_OR_DR         <= L_D8 or L_RI8;
372         L_PROD          <= (L_SIGN_D & L_D8) * (L_SIGN_R & L_R8);
373         L_ROR_D         <= I_FLAGS(0) &  L_D8(7 downto 1);
374         L_SUB_DR        <= L_D8 - L_RI8;
375         L_SBC_DR        <= L_SUB_DR - ("0000000" & I_FLAGS(0));
376         L_SIGN_D        <= L_D8(7) and I_IMM(6);
377         L_SIGN_R        <= L_R8(7) and I_IMM(5);
378         L_SWAP_D        <= L_D8(3 downto 0) & L_D8(7 downto 4);
379         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.

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

```
124             Q_FLAGS(9)      <= L_RBIT xor not I_BIT(3);      -- DIN[BIT] = BIT[3]
125             Q_FLAGS(8)      <= ze(L_SUB_DR);                 -- D == R for CPSE
126             Q_FLAGS(7 downto 0)      <= I_FLAGS;
127             L_DOUT      <= X"0000";
128
129             case I_ALU_OP is
130                 when ALU_ADC =>
131                     L_DOUT      <= L_ADC_DR & L_ADC_DR;
132                     Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Carry
133                     Q_FLAGS(1)      <= ze(L_ADC_DR);                         -- Zero
134                     Q_FLAGS(2)      <= L_ADC_DR(7);                          -- Negative
135                     Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Overflow
136                     Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Signed
137                     Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADC_DR(3));   -- Halfcarry
138
139                 when ALU_ADD =>
140                     L_DOUT      <= L_ADD_DR & L_ADD_DR;
141                     Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Carry
142                     Q_FLAGS(1)      <= ze(L_ADD_DR);                         -- Zero
143                     Q_FLAGS(2)      <= L_ADD_DR(7);                          -- Negative
144                     Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Overflow
145                     Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Signed
146                     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:

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

src/alu.vhd

The result is stored in the data path:

```
195         flg98: process(I_CLK)
196         begin
197             if (rising_edge(I_CLK)) then
198                 L_FLAGS_98      <= A_FLAGS(9 downto 8);
199             end if;
200         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:

```
231         Q_ADR           <= F_ADR;
232         Q_DOUT          <= A_DOUT(7 downto 0);
233         Q_INT_ENA       <= A_FLAGS(7);
234         Q_OPC           <= I_OPC;
235         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:

```
237         Q_RD_IO        <= '0'                    when (F_ADR <X"20")
238                 else (I_RD_M and not I_PMS) when (F_ADR       <X"5D")
239                 else '0';
240         Q_WE_IO         <= '0'                    when (F_ADR <X"20")
241                 else I_WE_M(0)             when (F_ADR       <X"5D")
242                 else '0';
243         L_WE_SRAM       <= "00"   when  (F_ADR <X"0060") else I_WE_M;
244         L_DIN           <= I_DIN  when (I_PMS = '1')
245                 else F_S    when  (F_ADR      <X"0020")
246                 else I_DIN  when  (F_ADR       <X"005D")
247                 else F_S    when  (F_ADR       <X"0060")
248                 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:

```
252         Q_NEW_PC      <= F_Z    when I_PC_OP = PC_LD_Z       -- IJMP, ICALL
253                else M_DOUT when I_PC_OP = PC_LD_S       -- RET, RETI
254                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:

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

```
206         begin
207             case I_PC_OP is
208                 when PC_BCC  => Q_LOAD_PC      <= F_CC;       -- maybe (PC on I_JADR)
209                 when PC_LD_I => Q_LOAD_PC      <= '1';        -- yes: new PC on I_JADR
210                 when PC_LD_Z => Q_LOAD_PC      <= '1';        -- yes: new PC in Z
211                 when PC_LD_S => Q_LOAD_PC      <= '1';        -- yes: new PC on stack
212                 when others  => Q_LOAD_PC      <= '0';        -- no.
213             end case;
214         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:

```
218         process(I_PC_OP, L_FLAGS_98, F_CC)
219         begin
220             case I_PC_OP is
221                 when PC_BCC    => Q_SKIP      <= F_CC;            -- if cond met
222                 when PC_LD_I   => Q_SKIP      <= '1';             -- yes
223                 when PC_LD_Z   => Q_SKIP      <= '1';             -- yes
224                 when PC_LD_S   => Q_SKIP      <= '1';             -- yes
225                 when PC_SKIP_Z => Q_SKIP      <= L_FLAGS_98(8);  -- if Z set
226                 when PC_SKIP_T => Q_SKIP      <= L_FLAGS_98(9);  -- if T set
227                 when others    => Q_SKIP      <= '0';             -- no.
228             end case;
229         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:

```
 66         process(I_CLK)
 67         begin
 68         if (rising_edge(I_CLK)) then
 69             --
 70             -- set the most common settings as default.
 71             --
 72             Q_ALU_OP        <= ALU_D_MV_Q;
 73             Q_AMOD          <= AMOD_ABS;
 74             Q_BIT           <= I_OPC(10) & I_OPC(2 downto 0);
 75             Q_DDDDD         <= I_OPC(8 downto 4);
 76             Q_IMM           <= X"0000";
 77             Q_JADR          <= I_OPC(31 downto 16);
 78             Q_OPC           <= I_OPC(15 downto  0);
 79             Q_PC            <= I_PC;
 80             Q_PC_OP         <= PC_NEXT;
 81             Q_PMS           <= '0';
 82             Q_RD_M          <= '0';
 83             Q_RRRRR         <= I_OPC(9) & I_OPC(3 downto 0);
 84             Q_RSEL          <= RS_REG;
 85             Q_WE_D          <= "00";
 86             Q_WE_01         <= '0';
 87             Q_WE_F          <= '0';
 88             Q_WE_M          <= "00";
 89             Q_WE_XYZS       <= '0';
 90
 91             case I_OPC(15 downto 10) is
 92                 when "000000" =>
```

src/opc_deco.vhd

...

```
653                 when others =>
654             end case;
655         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:

```
465                                        --
466                                        --  1001 010d dddd 1010 - DEC
467                                        --
468                                        Q_ALU_OP      <= ALU_DEC;
469                                        Q_WE_D      <= "01";
470                                        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.

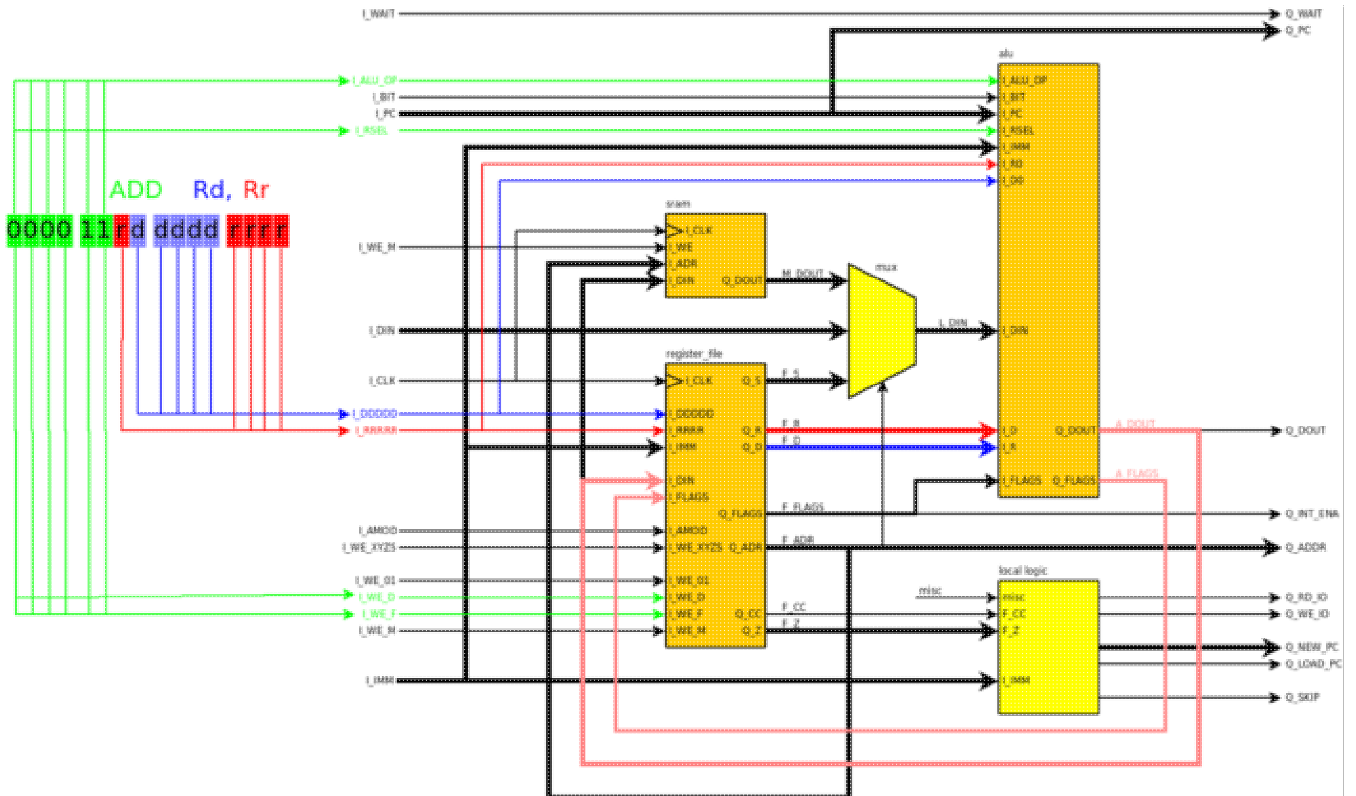
Compared to the monadic functions now item 3. in the checklist applies as well. This would mean we have to set **RRRRR** but by chance the default value is already correct. Therefore:

```
165                        --
166                        -- 0000 11rd dddd rrrr - ADD
167                        --
168                        Q_ALU_OP      <= ALU_ADD;
169                        Q_WE_D      <= "01";
170                        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:

7 OPCODE DECODER



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.

```
263                         --
264                         -- 0111 KKKK dddd KKKK - ANDI
265                         --
266                         Q_ALU_OP       <= ALU_AND;
267                         Q_IMM(7 downto 0)       <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
268                         Q_RSEL       <= RS_IMM;
269                         Q_DDDDD(4)       <= '1';    -- Rd = 16...31
270                         Q_WE_D       <= "01";
271                         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.

```
499                             --
500                             --  1001 0110 KKdd KKKK - ADIW
501                             --  1001 0111 KKdd KKKK - SBIW
502                             --
503                             if (I_OPC(8) = '0') then    Q_ALU_OP       <= ALU_ADIW;
504                             else                        Q_ALU_OP       <= ALU_SBIW;
505                             end if;
506                             Q_IMM(5 downto 4)       <= I_OPC(7 downto 6);
507                             Q_IMM(3 downto 0)       <= I_OPC(3 downto 0);
508                             Q_RSEL      <= RS_IMM;
509                             Q_DDDDD      <= "11" & I_OPC(5 downto 4) & "0";
510
511                             Q_WE_D      <= "11";
512                             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.

```
129                              --
130                              -- 0000 0011 0ddd 0rrr - _MULSU  SU "010"
131                              -- 0000 0011 0ddd 1rrr - FMUL    UU "100"
132                              -- 0000 0011 1ddd 0rrr - FMULS   SS "111"
133                              -- 0000 0011 1ddd 1rrr - FMULSU  SU "110"
134                              --
135                              Q_DDDDD(4 downto 3)      <= "10";    -- regs 16 to 23
136                              Q_RRRRR(4 downto 3)      <= "10";    -- regs 16 to 23
137                              Q_ALU_OP       <= ALU_MULT;
138                              if I_OPC(7) = '0' then
139                                  if I_OPC(3) = '0' then
140                                      Q_IMM(7 downto 5)      <= MULT_SU;
141                                  else
142                                      Q_IMM(7 downto 5)      <= MULT_FUU;
143                                  end if;
144                              else
145                                  if I_OPC(3) = '0' then
146                                      Q_IMM(7 downto 5)      <= MULT_FSS;
147                                  else
148                                      Q_IMM(7 downto 5)      <= MULT_FSU;
149                                  end if;
150                              end if;
151                              Q_WE_01       <= '1';
152                              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:

```
333                          --
334                          -- 1001 00-1r rrrr 0000 - STS
335                          -- 1001 00-1r rrrr 0001 - ST Z+. Rr
336                          -- 1001 00-1r rrrr 0010 - ST -Z. Rr
337                          -- 1001 00-1r rrrr 1000 - ST Y. Rr
338                          -- 1001 00-1r rrrr 1001 - ST Y+. Rr
339                          -- 1001 00-1r rrrr 1010 - ST -Y. Rr
340                          -- 1001 00-1r rrrr 1100 - ST X. Rr
341                          -- 1001 00-1r rrrr 1101 - ST X+. Rr
342                          -- 1001 00-1r rrrr 1110 - ST -X. Rr
343                          -- 1001 00-1r rrrr 1111 - PUSH Rr
344                          --
345                          Q_ALU_OP       <= ALU_D_MV_Q;
346                          Q_WE_M      <= "01";
347                          Q_WE_XYZS       <= '1';
348                          case I_OPC(3 downto 0) is
349                              when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
350                              when "0001" => Q_AMOD      <= AMOD_Zi;
351                              when "0010" => Q_AMOD      <= AMOD_dZ;
352                              when "1001" => Q_AMOD      <= AMOD_Yi;
353                              when "1010" => Q_AMOD      <= AMOD_dY;
354                              when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
355                              when "1101" => Q_AMOD      <= AMOD_Xi;
356                              when "1110" => Q_AMOD      <= AMOD_dX;
357                              when "1111" => Q_AMOD      <= AMOD_dSP;
358                              when others =>
359                          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:

```
297                     Q_IMM      <= I_OPC(31 downto 16);   -- absolute address for LDS/STS
298                     if (I_OPC(9) = '0') then        -- LDD / POP
299                         --
300                         -- 1001 00-0d dddd 0000 - LDS
301                         -- 1001 00-0d dddd 0001 - LD Rd, Z+
302                         -- 1001 00-0d dddd 0010 - LD Rd, -Z
303                         -- 1001 00-0d dddd 0100 - (ii)  LPM Rd, (Z)
304                         -- 1001 00-0d dddd 0101 - (iii) LPM Rd, (Z+)
305                         -- 1001 00-0d dddd 0110 - ELPM Z        --- not mega8
306                         -- 1001 00-0d dddd 0111 - ELPM Z+       --- not mega8
307                         -- 1001 00-0d dddd 1001 - LD Rd, Y+
308                         -- 1001 00-0d dddd 1010 - LD Rd, -Y
309                         -- 1001 00-0d dddd 1100 - LD Rd, X
310                         -- 1001 00-0d dddd 1101 - LD Rd, X+
311                         -- 1001 00-0d dddd 1110 - LD Rd, -X
312                         -- 1001 00-0d dddd 1111 - POP Rd
313                         --
314                         Q_RSEL      <= RS_DIN;
315                         Q_RD_M      <= I_T0;
316                         Q_WE_D      <= '0' & not I_T0;
317                         Q_WE_XYZS      <= not I_T0;
318                         Q_PMS      <= (not I_OPC(3)) and I_OPC(2) and (not I_OPC(1));
319                         case I_OPC(3 downto 0) is
320                             when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
321                             when "0001" => Q_AMOD      <= AMOD_Zi;
322                             when "0100" => Q_AMOD      <= AMOD_Z;    Q_WE_XYZS <= '0';
323                             when "0101" => Q_AMOD      <= AMOD_Zi;
324                             when "1001" => Q_AMOD      <= AMOD_Yi;
325                             when "1010" => Q_AMOD      <= AMOD_dY;
326                             when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
327                             when "1101" => Q_AMOD      <= AMOD_Xi;
328                             when "1110" => Q_AMOD      <= AMOD_dX;
329                             when "1111" => Q_AMOD      <= AMOD_SPi;
330                             when others =>                      Q_WE_XYZS      <= '0';
331                         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**:

```
478                                         --
479                                         --  1001 010k kkkk 110k - JMP (k = 0 for 16 bit)
480                                         --  kkkk kkkk kkkk kkkk
481                                         --
482                                         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:

```
209                 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:

```
222                 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**:

```
450                                     --
451                                     --  1001 0100 0000 1001 IJMP
452                                     --  1001 0100 0001 1001 EIJMP   -- not mega8
453                                     --  1001 0101 0000 1001 ICALL
454                                     --  1001 0101 0001 1001 EICALL   -- not mega8
455                                     --
456                                     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**.

```
580                     --
581                     -- 1100 kkkk kkkk kkkk - RJMP
582                     --
583                     Q_JADR       <= I_PC + (I_OPC(11) & I_OPC(11) & I_OPC(11) & I_OPC(11)
584                                         & I_OPC(11 downto 0)) + X"0001";
585                     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**.

```
610                         --
611                         -- 1111 00kk kkkk kbbb - BRBS
612                         -- 1111 01kk kkkk kbbb - BRBC
613                         --       v
614                         -- bbb: status register bit
615                         -- v: value (set/cleared) of status register bit
616                         --
617                         Q_JADR      <= I_PC + (I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
618                                         & I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
619                                         & I_OPC(9) & I_OPC(9 downto 3)) + X"0001";
620                         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:

```
485                                         --
486                                         --  1001 010k kkkk 111k - CALL (k = 0)
487                                         --  kkkk kkkk kkkk kkkk
488                                         --
489                                         Q_ALU_OP      <= ALU_PC_2;
490                                         Q_AMOD       <= AMOD_ddSP;
491                                         Q_PC_OP       <= PC_LD_I;
492                                         Q_WE_M       <= "11";     -- both PC bytes
493                                         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:

```
516                     --
517                     --  1001 1000 AAAA Abbb - CBI
518                     --  1001 1001 AAAA Abbb - SBIC
519                     --  1001 1010 AAAA Abbb - SBI
520                     --  1001 1011 AAAA Abbb - SBIS
521                     --
522                     Q_ALU_OP      <= ALU_BIT_CS;
523                     Q_AMOD      <= AMOD_ABS;
524                     Q_BIT(3)      <= I_OPC(9);   -- set/clear
525
526                     -- IMM = AAAAAA + 0x20
527                     --
528                     Q_IMM(4 downto 0)      <= I_OPC(7 downto 3);
529                     Q_IMM(6 downto 5)      <= "01";
530
531                     Q_RD_M      <= I_T0;
532                     if ((I_OPC(8) = '0') ) then     -- CBI or SBI
533                         Q_WE_M(0)      <= '1';
534                     else                            -- SBIC or SBIS
535                         if (I_T0 = '0') then        -- second cycle.
536                             Q_PC_OP      <= PC_SKIP_T;
537                         end if;
538                     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:

```
226                 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:

```
 95                                --
 96                                -- 0000 0000 0000 0000 - NOP
 97                                -- 0000 0000 001v vvvv - INTERRUPT
 98                                --
 99                                if (I_OPC(5)) = '1' then   -- interrupt
100                                    Q_ALU_OP       <= ALU_INTR;
101                                    Q_AMOD       <= AMOD_ddSP;
102                                    Q_JADR       <= "0000000000" & I_OPC(4 downto 0) & "0";
103                                    Q_PC_OP       <= PC_LD_I;
104                                    Q_WE_F       <= '1';
105                                    Q_WE_M       <= "11";
106                                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:

```
 89                        I_RX        => I_RX,
 90
 91                        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).

```
132                   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.

```
147                           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:

```
183
```

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:

```
 98         iord: process(I_ADR_IO, I_SWITCH,
 99                       U_RX_DATA, U_RX_READY, L_RX_INT_ENABLED,
100                       U_TX_BUSY, L_TX_INT_ENABLED)
101         begin
102             -- addresses for mega8 device (use iom8.h or #define __AVR_ATmega8__).
103             --
104             case I_ADR_IO is
105                 when X"2A"  => Q_DOUT      <=             -- UCSRB:
106                                    L_RX_INT_ENABLED  -- Rx complete int enabled.
107                                  & L_TX_INT_ENABLED  -- Tx complete int enabled.
108                                  & L_TX_INT_ENABLED  -- Tx empty int enabled.
109                                  & '1'               -- Rx enabled
110                                  & '1'               -- Tx enabled
111                                  & '0'               -- 8 bits/char
112                                  & '0'               -- Rx bit 8
113                                  & '0';              -- Tx bit 8
114                 when X"2B"  => Q_DOUT      <=             -- UCSRA:
115                                    U_RX_READY       -- Rx complete
116                                  & not U_TX_BUSY    -- Tx complete
117                                  & not U_TX_BUSY    -- Tx ready
118                                  & '0'              -- frame error
119                                  & '0'              -- data overrun
120                                  & '0'              -- parity error
121                                  & '0'              -- double dpeed
122                                  & '0';             -- multiproc mode
123                 when X"2C"  => Q_DOUT      <= U_RX_DATA; -- UDR
124                 when X"40"  => Q_DOUT      <=            -- UCSRC
125                                    '1'              -- URSEL
126                                  & '0'              -- asynchronous
127                                  & "00"             -- no parity
128                                  & '1'              -- two stop bits
129                                  & "11"             -- 8 bits/char
130                                  & '0';             -- rising clock edge
131
132                 when X"36"  => Q_DOUT      <= I_SWITCH;  -- PINB
133                 when others => Q_DOUT      <= X"AA";
134             end case;
135         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:

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

```
141             if (rising_edge(I_CLK)) then
142                 if (I_CLR = '1') then
143                     L_RX_INT_ENABLED       <= '0';
144                     L_TX_INT_ENABLED       <= '0';
145                 elsif (I_WE_IO = '1') then
146                     case I_ADR_IO is
147                         when X"38"  => Q_7_SEGMENT      <= I_DIN(6 downto 0);    -- PORTB
148                                        L_LEDS      <= not L_LEDS;
149                         when X"40"  =>  -- handled by uart
150                         when X"41"  =>  -- handled by uart
151                         when X"43"  => L_RX_INT_ENABLED      <= I_DIN(0);
152                                        L_TX_INT_ENABLED      <= I_DIN(1);
153                         when others =>
154                     end case;
155                 end if;
156             end if;
157         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:

```
182         L_WE_UART      <= I_WE_IO when (I_ADR_IO = X"2C") else '0'; -- write UART UDR
183         L_RD_UART      <= I_RD_IO when (I_ADR_IO = X"2C") else '0'; -- read  UART UDR
184
```

src/io.vhd

8.6 Interrupts

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

```
161         ioint: process(I_CLK)
162         begin
163             if (rising_edge(I_CLK)) then
164                 if (I_CLR = '1') then
165                     L_INTVEC      <= "000000";
166                 else
167                     if (L_RX_INT_ENABLED and U_RX_READY) = '1' then
168                         if (L_INTVEC(5) = '0') then     -- no interrupt pending
169                             L_INTVEC      <= "101011";       -- _VECTOR(11)
170                         end if;
171                     elsif (L_TX_INT_ENABLED and not U_TX_BUSY) = '1' then
```

```
172                         if (L_INTVEC(5) = '0') then     -- no interrupt pending
173                             L_INTVEC       <= "101100";        -- _VECTOR(12)
174                         end if;
175                     else                                -- no interrupt
176                         L_INTVEC       <= "000000";
177                     end if;
178                 end if;
179             end if;
180         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:

```
 54
 55         baud16: process(I_CLK)
 56         begin
 57             if (rising_edge(I_CLK)) then
 58                 if (I_CLR = '1') then
 59                     L_COUNTER      <= X"00000000";
 60                 elsif (L_COUNTER >= LIMIT) then
 61                     L_COUNTER      <= L_COUNTER - LIMIT;
 62                 else
 63                     L_COUNTER      <= L_COUNTER + BAUD_16;
 64                 end if;
 65             end if;
 66         end process;
 67
 68         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:

```
 70             if (rising_edge(I_CLK)) then
 71                 if (I_CLR = '1') then
 72                     L_CNT_16      <= "0000";
 73                 elsif (L_CE_16 = '1') then
 74                     L_CNT_16      <= L_CNT_16 + "0001";
 75                 end if;
 76             end if;
 77         end process;
 78
 79         L_CE_16      <= '1' when (L_COUNTER >= LIMIT) else '0';
 80         Q_CE_16      <= L_CE_16;
 81         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):

```
 67                     elsif (L_FLAG /= I_FLAG) then       -- new byte
 68                         Q_TX      <= '0';                    -- start bit
 69                         L_BUF      <= I_DATA;                -- data bits
 70                         L_TODO      <= "1001";
 71                     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:

```
 59                 elsif (I_CE_1 = '1') then
 60                     if (L_TODO /= "0000") then          -- transmitting
 61                         Q_TX      <= L_BUF(0);               -- next bit
 62                         L_BUF          <= '1' & L_BUF(7 downto 1);
 63                         if (L_TODO = "0001") then
 64                             L_FLAG       <= I_FLAG;
 65                         end if;
 66                         L_TODO      <= L_TODO - "0001";
 67                     elsif (L_FLAG /= I_FLAG) then       -- new byte
 68                         Q_TX      <= '0';                    -- start bit
 69                         L_BUF      <= I_DATA;                -- data bits
 70                         L_TODO      <= "1001";
 71                     end if;
 72                 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:

```
 56         process(I_CLK)
 57         begin
 58             if (rising_edge(I_CLK)) then
 59                 if (I_CLR = '1') then
 60                     L_SERIN      <= '1';
 61                     L_SER_HOT      <= '1';
 62                 else
 63                     L_SERIN        <= I_RX;
 64                     L_SER_HOT      <= L_SERIN;
 65                 end if;
 66             end if;
 67         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:

```
 86                     if (L_POSITION = X"00") then            -- uart idle
 87                         L_BUF       <= "1111111111";
 88                         if (L_SER_HOT = '0')  then          -- start bit received
 89                             L_POSITION      <= X"01";
 90                         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:

```
 93                         if (L_POSITION(3 downto 0) = "0111") then       -- 1/2 bit
 94                             L_BUF      <= L_SER_HOT & L_BUF(9 downto 1);     -- sample data
 95                             --
 96                             -- validate start bit
 97                             --
 98                             if (START_BIT and L_SER_HOT = '1') then     -- 1/2 start bit
 99                                 L_POSITION      <= X"00";
100                             end if;
101
102                             if (STOP_BIT) then                          -- 1/2 stop bit
103                                 Q_DATA      <= L_BUF(9 downto 2);
104                             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**:

```
105                         elsif (STOP_POS) then                       -- 3/4 stop bit
106                             L_FLAG      <= L_FLAG xor (L_BUF(9) and not L_BUF(0));
107                             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.

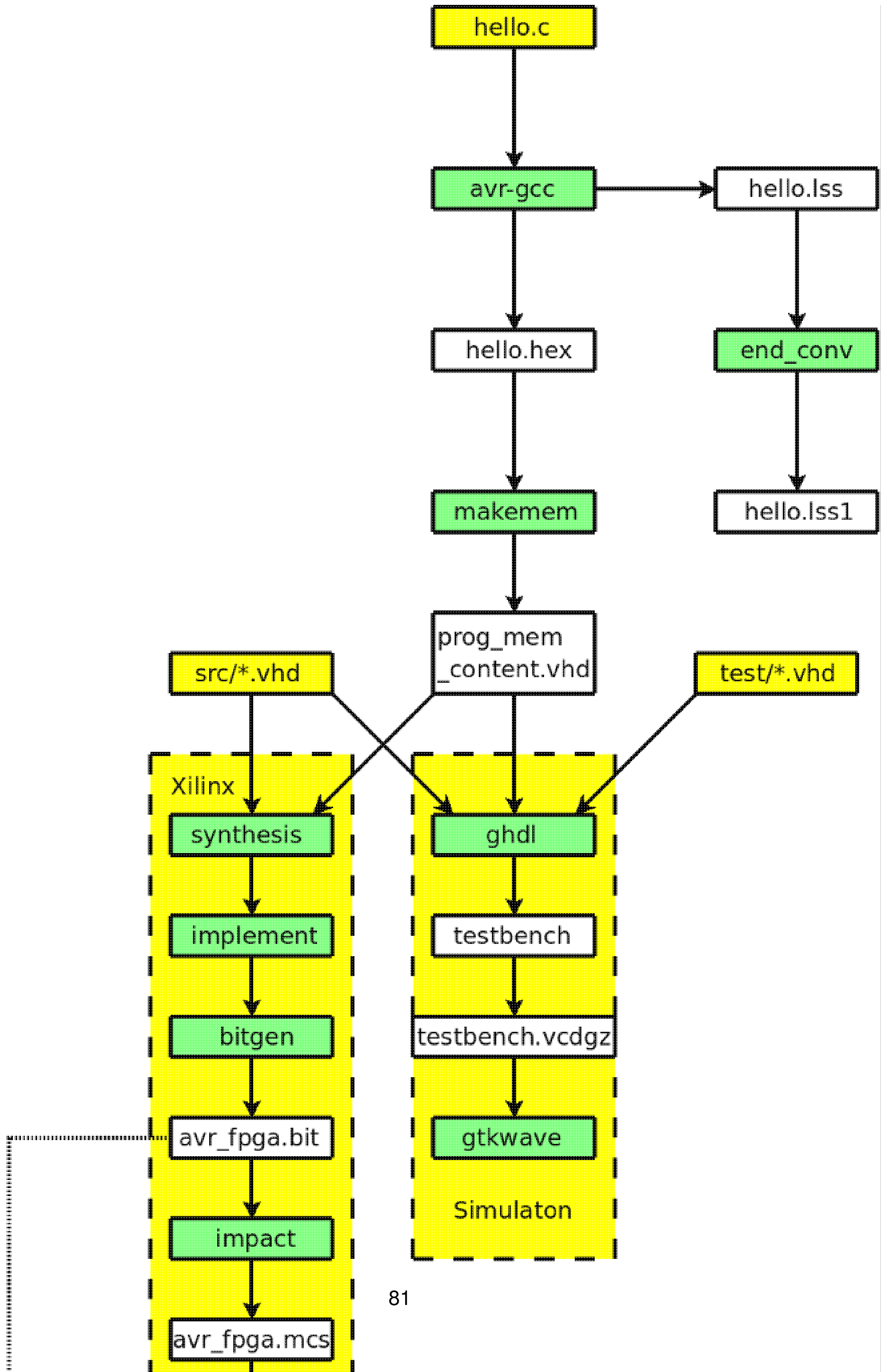
This finalizes the description of the FPGA. We will proceed with the design flow in the next lesson.

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.lss
hello.hex
end_conv
makemem
hello.lss1
prog_mem
_content.vhd
src/*.vhd
test/*.vhd
Xilinx
synthesis
ghdl
implement
testbench
bitgen
testbench.vcdgz
avr_fpga.bit
gtkwave
impact
Simulaton
avr_fpga.mcs

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

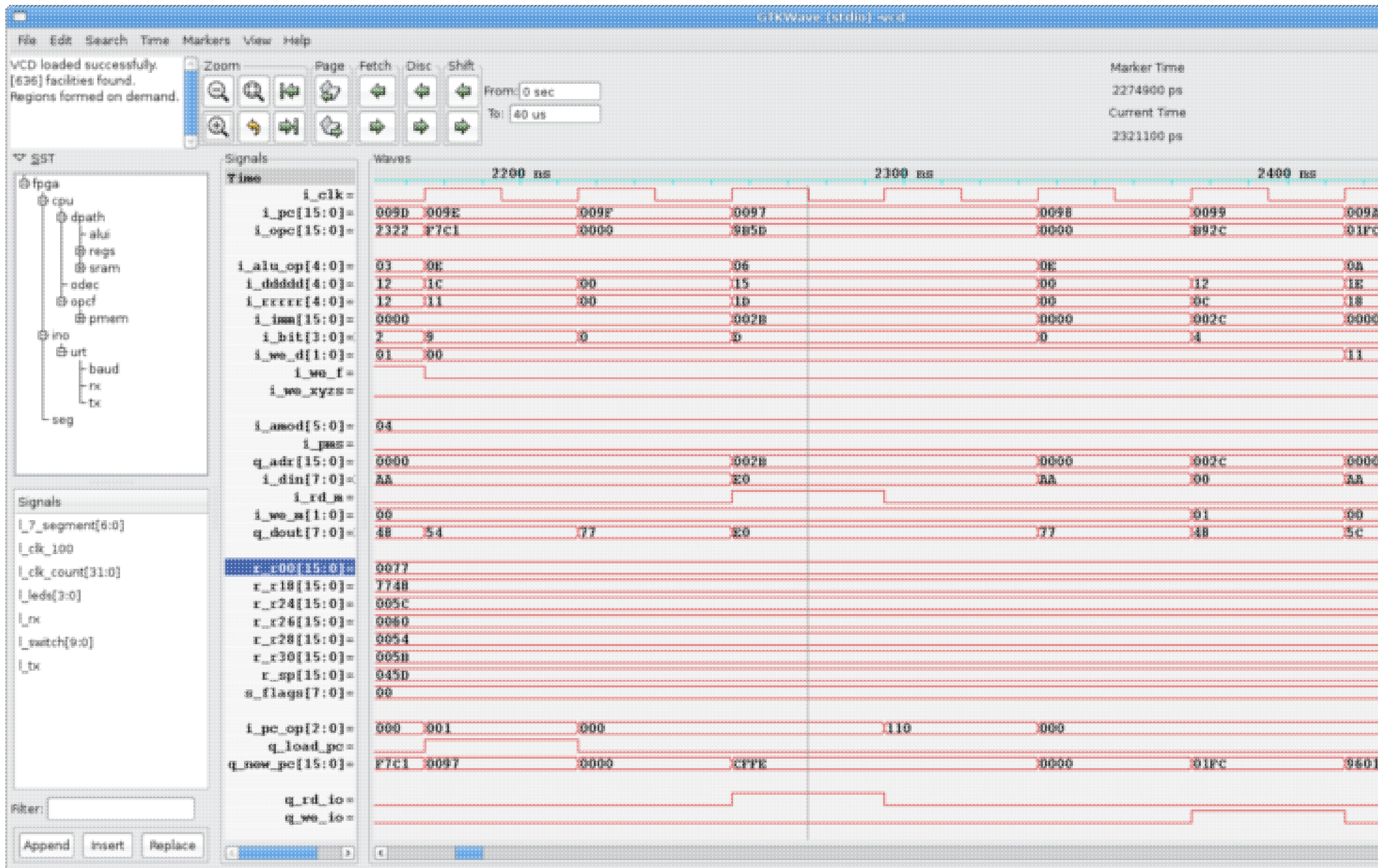
Initially you will be concerned mostly with the functional simulation. On this branch you debug the VHDL code until it looks functionally OK. In order to perform the functional simulation, you need 3 sorts of VHDL files:

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:


File Edit Search Time Markers View Help
VCD loaded successfully.
[636] facilities found.
Regions formed on demand.
Zoom
Page
Fetch
Disc
Shift
From: 0 sec
To: 40 us
Marker Time
2274900 ps
Current Time
2321100 ps
SST
fpga
cpu
dpath
alui
regs
sram
odec
opcf
pmem
ino
urt
baud
rx
tx
seg
Signals
I_7_segment[6:0]
I_clk_100
I_clk_count[31:0]
I_leds[3:0]
I_rx
I_switch[9:0]
I_tx
Filter:
Append
Insert
Replace
Waves
2200 ns
2300 ns
2400 ns
i_clk
i_pc[15:0]
i_opc[15:0]
i_alu_op[4:0]
i_ddddd[4:0]
i_rrrrr[4:0]
i_imm[15:0]
i_bit[3:0]
i_we_d[1:0]
i_we_f
i_we_xyzs
i_amod[5:0]
i_pms
q_adr[15:0]
i_din[7:0]
i_rd_m
i_we_m[1:0]
q_dout[7:0]
r_r18[15:0]
r_r24[15:0]
r_r26[15:0]
r_r28[15:0]
r_r30[15:0]
r_sp[15:0]
s_flags[7:0]
i_pc_op[2:0]
q_load_pc
q_new_pc[15:0]
q_rd_io
q_we_io

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

```
156     00000095: (uart_puts):
157       95:   01AC            movw    r20, r24
158       96:   C003            rjmp     0x9A           ; 0x134
159       97:   9B5D            sbis    0x0b, 5 ; 11
160       98:   CFFE            rjmp     0x97           ; 0x12e
161       99:   B92C            out     0x0c, r18       ; 12
162       9A:   01FC            movw    r30, r24
163       9B:   9601            adiw    r24, 0x01       ; 1
164       9C:   9124            lpm     r18, Z
165       9D:   2322            and     r18, r18
166       9E:   F7C1            brne     0x97           ; 0x12e
167       9F:   1B84            sub     r24, r20
168       A0:   0B95            sbc     r25, r21
169       A1:   9701            sbiw    r24, 0x01       ; 1
170       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:

```
1     #include "assert.h"
2     #include "stdio.h"
3     #include "stdint.h"
4     #include "string.h"
5
6     const char * hex_file = 0;
7     const char * vhdl_file = 0;
8
9     uint8_t buffer[0x10000];
10
11    //-----------------------------------------------------------------------------
12    uint32_t
13    get_byte(const char *  cp)
14    {
15    uint32_t value;
16    const char cc[3] = { cp[0], cp[1], 0 };
17    const int cnt = sscanf(cc, "%X", &value);
18       assert(cnt == 1);
19       return value;
20    }
21    //-----------------------------------------------------------------------------
22    void
23    read_file(FILE * in)
24    {
25       memset(buffer, 0xFF, sizeof(buffer));
26    char line[200];
27       for (;;)
28           {
29             const char * s = fgets(line, sizeof(line) - 2, in);
30             if (s == 0)   return;
31             assert(*s++ == ':');
32             const uint32_t len     = get_byte(s);
33             const uint32_t ah      = get_byte(s + 2);
34             const uint32_t al      = get_byte(s + 4);
35             const uint32_t rectype = get_byte(s + 6);
36             const char * d = s + 8;
37             const uint32_t addr = ah       <<8 | al;
38
39             uint32_t csum = len + ah + al + rectype;
40             assert((addr + len)       <= 0x10000);
41             for (uint32_t l = 0; l       <len; ++l)
42                 {
43                   const uint32_t byte = get_byte(d);
44                   d += 2;
45                   buffer[addr + l] = byte;
46                   csum += byte;
47                 }
48
49             csum = 0xFF & -csum;
50             const uint32_t sum = get_byte(d);
51             assert(sum == csum);
52           }
53    }
54    //-----------------------------------------------------------------------------
55    void
56    write_vector(FILE * out, bool odd, uint32_t mem, uint32_t v)
```

```
57     {
58     const uint8_t * base = buffer;
59
60        // total memory is 2 even bytes, 2 odd bytes, 2 even bytes, ...
61        //
62        if (odd)   base += 2;
63
64        // total memory is 4 kByte organized into 8 memories.
65        // thus each of the 16 vectors covers 256 bytes.
66        //
67        base += v*256;
68
69        // memories 0 and 1 are the low byte of the opcode while
70        // memories 2 and 3 are the high byte.
71        //
72        if (mem >= 2)   ++base;
73
74     const char * px = odd ? "po" : "pe";
75        fprintf(out, "constant %s_%u_%2.2X : BIT_VECTOR := X\"", px, mem, v);
76        for (int32_t d = 63; d >= 0; --d)
77            {
78              uint32_t q = base[4*d];
79              if (mem & 1)   q >>= 4;     // high nibble
80              else           q &= 0x0F;   // low nibble
81              fprintf(out, "%X", q);
82            }
83
84        fprintf(out, "\";\r\n");
85     }
86     //-----------------------------------------------------------------------------
87     void
88     write_mem(FILE * out, bool odd, uint32_t mem)
89     {
90     const char * px = odd ? "po" : "pe";
91
92        fprintf(out, "-- content of %s_%u --------------------------------------"
93                     "--------------------------------------------\r\n", px, mem);
94
95        for (uint32_t v = 0; v       <16; ++v)
96            write_vector(out, odd, mem, v);
97
98        fprintf(out, "\r\n");
99     }
100    //-----------------------------------------------------------------------------
101    void
102    write_file(FILE * out)
103    {
104       fprintf(out,
105    "\r\n"
106    "library IEEE;\r\n"
107    "use IEEE.STD_LOGIC_1164.all;\r\n"
108    "\r\n"
109    "package prog_mem_content is\r\n"
110    "\r\n");
111
112       for (uint32_t m = 0; m       <4; ++m)
113           write_mem(out, false, m);
114
115       for (uint32_t m = 0; m       <4; ++m)
116           write_mem(out, true,  m);
117
118       fprintf(out,
```

```
119     "end prog_mem_content;\r\n"
120     "\r\n");
121     }
122     //-----------------------------------------------------------------------------
123     int
124     main(int argc, char * argv[])
125     {
126        if (argc > 1)   hex_file = argv[1];
127        if (argc > 2)   vhdl_file = argv[2];
128
129     FILE * in = stdin;
130        if (hex_file)   in = fopen(hex_file, "r");
131        assert(in);
132        read_file(in);
133        fclose(in);
134
135     FILE * out = stdout;
136        if (vhdl_file)   out = fopen(vhdl_file, "w");
137        write_file(out);
138        assert(out);
139     }
140     //-----------------------------------------------------------------------------
```

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:

```
  1     #include "assert.h"
  2     #include "ctype.h"
  3     #include "stdio.h"
  4     #include "string.h"
  5
  6     //-----------------------------------------------------------------------------
  7     int
  8     main(int argc, const char * argv)
  9     {
 10     char buffer[2000];
 11     int pc, val, val2;
 12
 13        for (;;)
 14            {
 15              char * s = fgets(buffer, sizeof(buffer) - 2, stdin);
 16              if (s == 0)   return 0;
```

```
17
18              // map lines '  xx:' and 'xxxxxxxx; to 2* the hex value.
19              //
20              if (
21                  (isxdigit(s[0]) || s[0] == ' ') &&
22                  (isxdigit(s[1]) || s[1] == ' ') &&
23                  (isxdigit(s[2]) || s[2] == ' ') &&
24                   isxdigit(s[3]) && s[4] == ':')   // '  xx:'
25                 {
26                   assert(1 == sscanf(s, " %x:", &pc));
27                   if (pc & 1)       printf("%4X+:", pc/2);
28                   else              printf("%4X:", pc/2);
29                   s += 5;
30                 }
31              else if (isxdigit(s[0]) && isxdigit(s[1]) && isxdigit(s[2]) &&
32                       isxdigit(s[3]) && isxdigit(s[4]) && isxdigit(s[5]) &&
33                       isxdigit(s[6]) && isxdigit(s[7]))             // 'xxxxxxxx'
34                 {
35                   assert(1 == sscanf(s, "%x", &pc));
36                   if (pc & 1)   printf("%8.8X+:", pc/2);
37                   else          printf("%8.8X:", pc/2);
38                   s += 8;
39                 }
40              else                             // other: copy verbatim
41                 {
42                   printf("%s", s);
43                   continue;
44                 }
45
46               while (isblank(*s))   printf("%c", *s++);
47
48               // endian swap.
49               //
50               while (isxdigit(s[0]) &&
51                      isxdigit(s[1]) &&
52                               s[2] == ' ' &&
53                      isxdigit(s[3]) &&
54                      isxdigit(s[4]) &&
55                               s[5] == ' ')
56                  {
57                   assert(2 == sscanf(s, "%x %x ", &val, &val2));
58                   printf("%2.2X%2.2X  ", val2, val);
59                   s += 6;
60                  }
61
62              char * s1 = strstr(s, ".+");
63              char * s2 = strstr(s, ".-");
64              if (s1)
65                 {
66                   assert(1 == sscanf(s1 + 2, "%d", &val));
67                   assert((val & 1) == 0);
68                   sprintf(s1, " 0x%X", (pc + val)/2 + 1);
69                   printf(s);
70                   s = s1 + strlen(s1) + 1;
71                 }
72              else if (s2)
73                 {
74                   assert(1 == sscanf(s2 + 2, "%d", &val));
75                   assert((val & 1) == 0);
76                   sprintf(s2, " 0x%X", (pc - val)/2 + 1);
77                   printf(s);
78                   s = s2 + strlen(s2) + 1;
```

```
 79                }
 80
 81              printf("%s", s);
 82            }
 83     }
 84     //-----------------------------------------------------------------------------
```

```
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:

```
  1     #include "stdint.h"
  2     #include "avr/io.h"
  3     #include "avr/pgmspace.h"
  4
  5     #undef F_CPU
  6     #define F_CPU 25000000UL
  7     #include "util/delay.h"
  8
  9
 10          //-----------------------------------------------------------------------//
 11         //                                                                       //
 12        //   print char cc on UART.                                              //
 13       //    return number of chars printed (i.e. 1).                           //
 14      //                                                                     //
 15     //-----------------------------------------------------------------------//
 16     uint8_t
 17     uart_putc(uint8_t cc)
 18     {
 19         while ((UCSRA & (1      <<UDRE)) == 0)      ;
 20         UDR = cc;
 21         return 1;
 22     }
 23
 24          //-----------------------------------------------------------------------//
 25         //                                                                       //
 26        //   print char cc on 7 segment display.                                 //
```

```
 27       //    return number of chars printed (i.e. 1).                           //
 28      //                                                                        //
 29     //----------------------------------------------------------------------//
 30     // The segments of the display are encoded like this:
 31     //
 32     //
 33     //      segment     PORT B
 34     //      name        Bit number
 35     //      ----A----   ----0----
 36     //      |       |   |       |
 37     //      F       B   5       1
 38     //      |       |   |       |
 39     //      ----G----   ----6----
 40     //      |       |   |       |
 41     //      E       C   4       2
 42     //      |       |   |       |
 43     //      ----D----   ----3----
 44     //
 45     //-----------------------------------------------------------------------------
 46
 47     #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:

```
1    -------------------------------------------------------------------------------
2    --
3    -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4    --
5    --  This code is free software: you can redistribute it and/or modify
6    --  it under the terms of the GNU General Public License as published by
7    --  the Free Software Foundation, either version 3 of the License, or
8    --  (at your option) any later version.
9    --
10   --  This code is distributed in the hope that it will be useful,
11   --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12   --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13   --  GNU General Public License for more details.
14   --
15   --  You should have received a copy of the GNU General Public License
16   --  along with this code (see the file named COPYING).
17   --  If not, see http://www.gnu.org/licenses/.
18   --
19   -------------------------------------------------------------------------------
20   -------------------------------------------------------------------------------
21   --
22   -- Module Name:    alu - Behavioral
23   -- Create Date:    16:47:24 12/29/2009
24   -- Description:    arithmetic logic unit of a CPU
25   --
26   -------------------------------------------------------------------------------
27   --
28   library IEEE;
29   use IEEE.STD_LOGIC_1164.ALL;
30   use IEEE.STD_LOGIC_ARITH.ALL;
31   use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33   entity testbench is
34   end testbench;
35
36   architecture Behavioral of testbench is
37
38   component avr_fpga
39       port (  I_CLK_100   : in  std_logic;
40               I_SWITCH    : in  std_logic_vector(9 downto 0);
41               I_RX        : in  std_logic;
42
```

```
43                 Q_7_SEGMENT : out std_logic_vector(6 downto 0);
44                 Q_LEDS      : out std_logic_vector(3 downto 0);
45                 Q_TX        : out std_logic);
46     end component;
47
48     signal L_CLK_100            : std_logic;
49     signal L_LEDS               : std_logic_vector(3 downto 0);
50     signal L_7_SEGMENT          : std_logic_vector(6 downto 0);
51     signal L_RX                 : std_logic;
52     signal L_SWITCH             : std_logic_vector(9 downto 0);
53     signal L_TX                 : std_logic;
54
55     signal  L_CLK_COUNT         : integer := 0;
56
57     begin
58
59         fpga: avr_fpga
60         port map(   I_CLK_100   => L_CLK_100,
61                     I_SWITCH    => L_SWITCH,
62                     I_RX        => L_RX,
63
64                     Q_LEDS      => L_LEDS,
65                     Q_7_SEGMENT => L_7_SEGMENT,
66                     Q_TX        => L_TX);
67
68         process -- clock process for CLK_100,
69         begin
70             clock_loop : loop
71                 L_CLK_100      <= transport '0';
72                 wait for 5 ns;
73
74                 L_CLK_100      <= transport '1';
75                 wait for 5 ns;
76             end loop clock_loop;
77         end process;
78
79         process(L_CLK_100)
80         begin
81             if (rising_edge(L_CLK_100)) then
82                 case L_CLK_COUNT is
83                     when 0 => L_SWITCH      <= "0011100000";   L_RX <= '0';
84                     when 2 => L_SWITCH(9 downto 8)      <= "11";
85                     when others =>
86                 end case;
87                 L_CLK_COUNT      <= L_CLK_COUNT + 1;
88             end if;
89         end process;
90     end Behavioral;
91
```

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:

```
 1  -------------------------------------------------------------------------------
 2  --
 3  -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4  --
 5  --  This code is free software: you can redistribute it and/or modify
 6  --  it under the terms of the GNU General Public License as published by
 7  --  the Free Software Foundation, either version 3 of the License, or
 8  --  (at your option) any later version.
 9  --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  -------------------------------------------------------------------------------
20  -------------------------------------------------------------------------------
21  --
22  -- Module Name:    prog_mem - Behavioral
23  -- Create Date:    14:09:04 10/30/2009
24  -- Description:    a block memory module
25  --
26  -------------------------------------------------------------------------------
27
28  library IEEE;
29  use IEEE.STD_LOGIC_1164.ALL;
30  use IEEE.STD_LOGIC_ARITH.ALL;
31  use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33  entity RAMB4_S4_S4 is
34      generic(INIT_00 : bit_vector := X"00000000000000000000000000000000"
35                                   &  "00000000000000000000000000000000";
36              INIT_01 : bit_vector := X"00000000000000000000000000000000"
37                                   & X"00000000000000000000000000000000";
38              INIT_02 : bit_vector := X"00000000000000000000000000000000"
39                                   & X"00000000000000000000000000000000";
40              INIT_03 : bit_vector := X"00000000000000000000000000000000"
41                                   & X"00000000000000000000000000000000";
42              INIT_04 : bit_vector := X"00000000000000000000000000000000"
43                                   & X"00000000000000000000000000000000";
44              INIT_05 : bit_vector := X"00000000000000000000000000000000"
45                                   & X"00000000000000000000000000000000";
46              INIT_06 : bit_vector := X"00000000000000000000000000000000"
47                                   & X"00000000000000000000000000000000";
48              INIT_07 : bit_vector := X"00000000000000000000000000000000"
49                                   & X"00000000000000000000000000000000";
50              INIT_08 : bit_vector := X"00000000000000000000000000000000"
51                                   & X"00000000000000000000000000000000";
52              INIT_09 : bit_vector := X"00000000000000000000000000000000"
53                                   & X"00000000000000000000000000000000";
54              INIT_0A : bit_vector := X"00000000000000000000000000000000"
55                                   & X"00000000000000000000000000000000";
56              INIT_0B : bit_vector := X"00000000000000000000000000000000"
57                                   & X"00000000000000000000000000000000";
58              INIT_0C : bit_vector := X"00000000000000000000000000000000"
59                                   & X"00000000000000000000000000000000";
60              INIT_0D : bit_vector := X"00000000000000000000000000000000"
61                                   & X"00000000000000000000000000000000";
```

```
 62                INIT_0E : bit_vector := X"00000000000000000000000000000000"
 63                                      & X"00000000000000000000000000000000";
 64                INIT_0F : bit_vector := X"00000000000000000000000000000000"
 65                                      & X"00000000000000000000000000000000");
 66
 67        port(   ADDRA   : in  std_logic_vector(9 downto 0);
 68                ADDRB   : in  std_logic_vector(9 downto 0);
 69                CLKA    : in  std_ulogic;
 70                CLKB    : in  std_ulogic;
 71                DIA     : in  std_logic_vector(3 downto 0);
 72                DIB     : in  std_logic_vector(3 downto 0);
 73                ENA     : in  std_ulogic;
 74                ENB     : in  std_ulogic;
 75                RSTA    : in  std_ulogic;
 76                RSTB    : in  std_ulogic;
 77                WEA     : in  std_ulogic;
 78                WEB     : in  std_ulogic;
 79
 80                DOA     : out std_logic_vector(3 downto 0);
 81                DOB     : out std_logic_vector(3 downto 0));
 82    end RAMB4_S4_S4;
 83
 84    architecture Behavioral of RAMB4_S4_S4 is
 85
 86    function cv(A : bit) return std_logic is
 87    begin
 88       if (A = '1') then return '1';
 89       else              return '0';
 90       end if;
 91    end;
 92
 93    function cv1(A : std_logic) return bit is
 94    begin
 95       if (A = '1') then return '1';
 96       else              return '0';
 97       end if;
 98    end;
 99
100    signal DATA : bit_vector(4095 downto 0) :=
101        INIT_0F & INIT_0E & INIT_0D & INIT_0C & INIT_0B & INIT_0A & INIT_09 & INIT_08 &
102        INIT_07 & INIT_06 & INIT_05 & INIT_04 & INIT_03 & INIT_02 & INIT_01 & INIT_00;
103
104    begin
105
106        process(CLKA, CLKB)
107        begin
108            if (rising_edge(CLKA)) then
109                if (ENA = '1') then
110                    DOA(3)      <= cv(DATA(conv_integer(ADDRA & "11")));
111                    DOA(2)      <= cv(DATA(conv_integer(ADDRA & "10")));
112                    DOA(1)      <= cv(DATA(conv_integer(ADDRA & "01")));
113                    DOA(0)      <= cv(DATA(conv_integer(ADDRA & "00")));
114                    if (WEA = '1') then
115                        DATA(conv_integer(ADDRA & "11"))      <= cv1(DIA(3));
116                        DATA(conv_integer(ADDRA & "10"))      <= cv1(DIA(2));
117                        DATA(conv_integer(ADDRA & "01"))      <= cv1(DIA(1));
118                        DATA(conv_integer(ADDRA & "00"))      <= cv1(DIA(0));
119                    end if;
120               end if;
121            end if;
122
123            if (rising_edge(CLKB)) then
```

```
124                 if (ENB = '1') then
125                     DOB(3)      <= cv(DATA(conv_integer(ADDRB & "11")));
126                     DOB(2)      <= cv(DATA(conv_integer(ADDRB & "10")));
127                     DOB(1)      <= cv(DATA(conv_integer(ADDRB & "01")));
128                     DOB(0)      <= cv(DATA(conv_integer(ADDRB & "00")));
129                     if (WEB = '1') then
130                         DATA(conv_integer(ADDRB & "11"))      <= cv1(DIB(3));
131                         DATA(conv_integer(ADDRB & "10"))      <= cv1(DIB(2));
132                         DATA(conv_integer(ADDRB & "01"))      <= cv1(DIB(1));
133                         DATA(conv_integer(ADDRB & "00"))      <= cv1(DIB(0));
134                     end if;
135                 end if;
136             end if;
137         end process;
138
139     end Behavioral;
140
```

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:

```
 1     PROJECT=avr_core
 2
 3     # the vhdl source files (except testbench)
 4     #
 5     FILES           += src/*.vhd
 6
 7     # the testbench sources and binary.
 8     #
 9     SIMFILES        = test/test_tb.vhd test/RAMB4_S4_S4.vhd
10     SIMTOP          = testbench
11
12     # When to stop the simulation
13     #
14     # GHDL_SIM_OPT  = --assert-level=error
15     GHDL_SIM_OPT    = --stop-time=40us
16
17     SIMDIR          = simu
18
19     FLAGS           = --ieee=synopsys --warn-no-vital-generic -fexplicit --std=93c
20
21     all:
22             make compile
23             make run 2>& 1 | grep -v std_logic_arith
```

```
 24             make view
 25
 26     compile:
 27             @mkdir -p simu
 28             @echo -----------------------------------------------------------------
 29             ghdl -i $(FLAGS) --workdir=simu --work=work $(SIMFILES) $(FILES)
 30             @echo
 31             @echo -----------------------------------------------------------------
 32             ghdl -m $(FLAGS) --workdir=simu --work=work $(SIMTOP)
 33             @echo
 34             @mv $(SIMTOP) simu/$(SIMTOP)
 35
 36     run:
 37             @$(SIMDIR)/$(SIMTOP) $(GHDL_SIM_OPT) --vcdgz=$(SIMDIR)/$(SIMTOP).vcdgz
 38
 39     view:
 40             gunzip --stdout $(SIMDIR)/$(SIMTOP).vcdgz | gtkwave --vcd gtkwave.save
 41
 42     clean:
 43             ghdl --clean --workdir=simu
 44
```

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:

```
 1    NET    I_CLK_100      PERIOD = 10 ns;
 2    NET    L_CLK          PERIOD = 35 ns;
 3
 4    NET    I_CLK_100      TNM_NET = I_CLK_100;
 5    NET    L_CLK          TNM_NET = L_CLK;
 6
 7    NET    I_CLK_100      LOC = AA12;
 8    NET    I_RX           LOC = M3;
 9    NET    Q_TX           LOC = M4;
10
11    # 7 segment LED display
12    #
13    NET    Q_7_SEGMENT      LOC = V3;
14    NET    Q_7_SEGMENT      LOC = V4;
15    NET    Q_7_SEGMENT      LOC = W3;
16    NET    Q_7_SEGMENT      LOC = T4;
17    NET    Q_7_SEGMENT      LOC = T3;
18    NET    Q_7_SEGMENT      LOC = U3;
19    NET    Q_7_SEGMENT      LOC = U4;
20
21    # single LEDs
22    #
23    NET    Q_LEDS           LOC = N1;
24    NET    Q_LEDS           LOC = N2;
25    NET    Q_LEDS           LOC = P1;
26    NET    Q_LEDS           LOC = P2;
27
28    # DIP switch(0 ... 7) and two pushbuttons (8, 9)
29    #
30    NET    I_SWITCH         LOC = H2;
31    NET    I_SWITCH         LOC = H1;
32    NET    I_SWITCH         LOC = J2;
33    NET    I_SWITCH         LOC = J1;
34    NET    I_SWITCH         LOC = K2;
35    NET    I_SWITCH         LOC = K1;
36    NET    I_SWITCH         LOC = L2;
37    NET    I_SWITCH         LOC = L1;
38    NET    I_SWITCH         LOC = R1;
39    NET    I_SWITCH         LOC = R2;
40
41    NET    I_SWITCH         PULLUP;
42
```

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

```
1     -------------------------------------------------------------------------------
2     --
3     -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4     --
5     --  This code is free software: you can redistribute it and/or modify
6     --  it under the terms of the GNU General Public License as published by
7     --  the Free Software Foundation, either version 3 of the License, or
8     --  (at your option) any later version.
9     --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    alu - Behavioral
23    -- Create Date:    13:51:24 11/07/2009
24    -- Description:    arithmetic logic unit of a CPU
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.std_logic_1164.ALL;
30    use IEEE.std_logic_ARITH.ALL;
31    use IEEE.std_logic_UNSIGNED.ALL;
32
33    use work.common.ALL;
34
35    entity alu is
36        port (  I_ALU_OP    : in  std_logic_vector( 4 downto 0);
37                I_BIT       : in  std_logic_vector( 3 downto 0);
38                I_D         : in  std_logic_vector(15 downto 0);
39                I_D0        : in  std_logic;
40                I_DIN       : in  std_logic_vector( 7 downto 0);
41                I_FLAGS     : in  std_logic_vector( 7 downto 0);
42                I_IMM       : in  std_logic_vector( 7 downto 0);
43                I_PC        : in  std_logic_vector(15 downto 0);
44                I_R         : in  std_logic_vector(15 downto 0);
45                I_R0        : in  std_logic;
46                I_RSEL      : in  std_logic_vector( 1 downto 0);
47
48                Q_FLAGS     : out std_logic_vector( 9 downto 0);
49                Q_DOUT      : out std_logic_vector(15 downto 0));
50    end alu;
51
52    architecture Behavioral of alu is
53
54    function ze(A: std_logic_vector(7 downto 0)) return std_logic is
55    begin
56        return not (A(0) or A(1) or A(2) or A(3) or
57                    A(4) or A(5) or A(6) or A(7));
58    end;
```

```
 59
 60     function cy(D, R, S: std_logic) return std_logic is
 61     begin
 62         return (D and R) or (D and not S) or (R and not S);
 63     end;
 64
 65     function ov(D, R, S: std_logic) return std_logic is
 66     begin
 67         return (D and R and (not S)) or ((not D) and (not R) and S);
 68     end;
 69
 70     function si(D, R, S: std_logic) return std_logic is
 71     begin
 72         return S xor ov(D, R, S);
 73     end;
 74
 75     signal L_ADC_DR     : std_logic_vector( 7 downto 0);    -- D + R + Carry
 76     signal L_ADD_DR     : std_logic_vector( 7 downto 0);    -- D + R
 77     signal L_ADIW_D     : std_logic_vector(15 downto 0);    -- D + IMM
 78     signal L_AND_DR     : std_logic_vector( 7 downto 0);    -- D and R
 79     signal L_ASR_D      : std_logic_vector( 7 downto 0);    -- (signed D) >> 1
 80     signal L_D8         : std_logic_vector( 7 downto 0);    -- D(7 downto 0)
 81     signal L_DEC_D      : std_logic_vector( 7 downto 0);    -- D - 1
 82     signal L_DOUT       : std_logic_vector(15 downto 0);
 83     signal L_INC_D      : std_logic_vector( 7 downto 0);    -- D + 1
 84     signal L_LSR_D      : std_logic_vector( 7 downto 0);    -- (unsigned) D >> 1
 85     signal L_MASK_I     : std_logic_vector( 7 downto 0);    -- 1      <<IMM
 86     signal L_NEG_D      : std_logic_vector( 7 downto 0);    -- 0 - D
 87     signal L_NOT_D      : std_logic_vector( 7 downto 0);    -- 0 not D
 88     signal L_OR_DR      : std_logic_vector( 7 downto 0);    -- D or R
 89     signal L_PROD       : std_logic_vector(17 downto 0);    -- D * R
 90     signal L_R8         : std_logic_vector( 7 downto 0);    -- odd or even R
 91     signal L_RI8        : std_logic_vector( 7 downto 0);    -- R8 or IMM
 92     signal L_RBIT       : std_logic;
 93     signal L_SBIW_D     : std_logic_vector(15 downto 0);    -- D - IMM
 94     signal L_ROR_D      : std_logic_vector( 7 downto 0);    -- D rotated right
 95     signal L_SBC_DR     : std_logic_vector( 7 downto 0);    -- D - R - Carry
 96     signal L_SIGN_D     : std_logic;
 97     signal L_SIGN_R     : std_logic;
 98     signal L_SUB_DR     : std_logic_vector( 7 downto 0);    -- D - R
 99     signal L_SWAP_D     : std_logic_vector( 7 downto 0);    -- D swapped
100     signal L_XOR_DR     : std_logic_vector( 7 downto 0);    -- D xor R
101
102     begin
103
104         dinbit: process(I_DIN, I_BIT(2 downto 0))
105         begin
106             case I_BIT(2 downto 0) is
107                 when "000"  => L_RBIT       <= I_DIN(0);   L_MASK_I <= "00000001";
108                 when "001"  => L_RBIT      <= I_DIN(1);   L_MASK_I <= "00000010";
109                 when "010"  => L_RBIT      <= I_DIN(2);   L_MASK_I <= "00000100";
110                 when "011"  => L_RBIT      <= I_DIN(3);   L_MASK_I <= "00001000";
111                 when "100"  => L_RBIT      <= I_DIN(4);   L_MASK_I <= "00010000";
112                 when "101"  => L_RBIT      <= I_DIN(5);   L_MASK_I <= "00100000";
113                 when "110"  => L_RBIT      <= I_DIN(6);   L_MASK_I <= "01000000";
114                 when others => L_RBIT      <= I_DIN(7);   L_MASK_I <= "10000000";
115             end case;
116         end process;
117
118         process(L_ADC_DR, L_ADD_DR, L_ADIW_D, I_ALU_OP, L_AND_DR, L_ASR_D,
119                 I_BIT, I_D, L_D8, L_DEC_D, I_DIN, I_FLAGS, I_IMM, L_MASK_I,
120                 L_INC_D, L_LSR_D, L_NEG_D, L_NOT_D, L_OR_DR, I_PC, L_PROD,
```

```
121                 I_R, L_RI8, L_RBIT, L_ROR_D, L_SBIW_D, L_SUB_DR, L_SBC_DR,
122                 L_SIGN_D, L_SIGN_R, L_SWAP_D, L_XOR_DR)
123         begin
124             Q_FLAGS(9)      <= L_RBIT xor not I_BIT(3);      -- DIN[BIT] = BIT[3]
125             Q_FLAGS(8)      <= ze(L_SUB_DR);                 -- D == R for CPSE
126             Q_FLAGS(7 downto 0)      <= I_FLAGS;
127             L_DOUT      <= X"0000";
128
129             case I_ALU_OP is
130                 when ALU_ADC =>
131                     L_DOUT      <= L_ADC_DR & L_ADC_DR;
132                     Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Carry
133                     Q_FLAGS(1)      <= ze(L_ADC_DR);                         -- Zero
134                     Q_FLAGS(2)      <= L_ADC_DR(7);                          -- Negative
135                     Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Overflow
136                     Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Signed
137                     Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADC_DR(3));   -- Halfcarry
138
139                 when ALU_ADD =>
140                     L_DOUT      <= L_ADD_DR & L_ADD_DR;
141                     Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Carry
142                     Q_FLAGS(1)      <= ze(L_ADD_DR);                         -- Zero
143                     Q_FLAGS(2)      <= L_ADD_DR(7);                          -- Negative
144                     Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Overflow
145                     Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Signed
146                     Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADD_DR(3));   -- Halfcarry
147
148                 when ALU_ADIW =>
149                     L_DOUT      <= L_ADIW_D;
150                     Q_FLAGS(0)      <= L_ADIW_D(15) and not I_D(15);         -- Carry
151                     Q_FLAGS(1)      <= ze(L_ADIW_D(15 downto 8)) and
152                                     ze(L_ADIW_D(7 downto 0));            -- Zero
153                     Q_FLAGS(2)      <= L_ADIW_D(15);                         -- Negative
154                     Q_FLAGS(3)      <= I_D(15) and not L_ADIW_D(15);         -- Overflow
155                     Q_FLAGS(4)      <= (L_ADIW_D(15) and not I_D(15))
156                                 xor (I_D(15) and not L_ADIW_D(15));      -- Signed
157
158                 when ALU_AND =>
159                     L_DOUT      <= L_AND_DR & L_AND_DR;
160                     Q_FLAGS(1)      <= ze(L_AND_DR);                         -- Zero
161                     Q_FLAGS(2)      <= L_AND_DR(7);                          -- Negative
162                     Q_FLAGS(3)      <= '0';                                  -- Overflow
163                     Q_FLAGS(4)      <= L_AND_DR(7);                          -- Signed
164
165                 when ALU_ASR =>
166                     L_DOUT      <= L_ASR_D & L_ASR_D;
167                     Q_FLAGS(0)      <= L_D8(0);                              -- Carry
168                     Q_FLAGS(1)      <= ze(L_ASR_D);                          -- Zero
169                     Q_FLAGS(2)      <= L_D8(7);                              -- Negative
170                     Q_FLAGS(3)      <= L_D8(0) xor L_D8(7);                  -- Overflow
171                     Q_FLAGS(4)      <= L_D8(0);                              -- Signed
172
173                 when ALU_BLD =>     -- copy T flag to DOUT
174                     case I_BIT(2 downto 0) is
175                         when "000"  => L_DOUT( 0)      <= I_FLAGS(6);
176                                        L_DOUT( 8)      <= I_FLAGS(6);
177                         when "001"  => L_DOUT( 1)      <= I_FLAGS(6);
178                                        L_DOUT( 9)      <= I_FLAGS(6);
179                         when "010"  => L_DOUT( 2)      <= I_FLAGS(6);
180                                        L_DOUT(10)      <= I_FLAGS(6);
181                         when "011"  => L_DOUT( 3)      <= I_FLAGS(6);
182                                        L_DOUT(11)      <= I_FLAGS(6);
```

```
183                         when "100"  => L_DOUT( 4)       <= I_FLAGS(6);
184                                        L_DOUT(12)       <= I_FLAGS(6);
185                         when "101"  => L_DOUT( 5)       <= I_FLAGS(6);
186                                        L_DOUT(13)       <= I_FLAGS(6);
187                         when "110"  => L_DOUT( 6)       <= I_FLAGS(6);
188                                        L_DOUT(14)       <= I_FLAGS(6);
189                         when others => L_DOUT( 7)       <= I_FLAGS(6);
190                                        L_DOUT(15)       <= I_FLAGS(6);
191                     end case;
192
193                 when ALU_BIT_CS =>  -- copy I_DIN to T flag
194                     Q_FLAGS(6)       <= L_RBIT xor not I_BIT(3);
195                     if (I_BIT(3) = '0') then    -- clear
196                         L_DOUT(15 downto 8)       <= I_DIN and not L_MASK_I;
197                         L_DOUT( 7 downto 0)       <= I_DIN and not L_MASK_I;
198                     else                         -- set
199                         L_DOUT(15 downto 8)       <= I_DIN or L_MASK_I;
200                         L_DOUT( 7 downto 0)       <= I_DIN or L_MASK_I;
201                     end if;
202
203                 when ALU_COM =>
204                     L_DOUT      <= L_NOT_D & L_NOT_D;
205                     Q_FLAGS(0)       <= '1';                                 -- Carry
206                     Q_FLAGS(1)       <= ze(not L_D8);                        -- Zero
207                     Q_FLAGS(2)       <= not L_D8(7);                         -- Negative
208                     Q_FLAGS(3)       <= '0';                                 -- Overflow
209                     Q_FLAGS(4)       <= not L_D8(7);                         -- Signed
210
211                 when ALU_DEC =>
212                     L_DOUT      <= L_DEC_D & L_DEC_D;
213                     Q_FLAGS(1)       <= ze(L_DEC_D);                         -- Zero
214                     Q_FLAGS(2)       <= L_DEC_D(7);                          -- Negative
215                     if (L_D8 = X"80") then
216                         Q_FLAGS(3)       <= '1';                             -- Overflow
217                         Q_FLAGS(4)       <= not L_DEC_D(7);                  -- Signed
218                     else
219                         Q_FLAGS(3)       <= '0';                             -- Overflow
220                         Q_FLAGS(4)       <= L_DEC_D(7);                      -- Signed
221                     end if;
222
223                 when ALU_EOR =>
224                     L_DOUT      <= L_XOR_DR & L_XOR_DR;
225                     Q_FLAGS(1)       <= ze(L_XOR_DR);                        -- Zero
226                     Q_FLAGS(2)       <= L_XOR_DR(7);                         -- Negative
227                     Q_FLAGS(3)       <= '0';                                 -- Overflow
228                     Q_FLAGS(4)       <= L_XOR_DR(7);                         -- Signed
229
230                 when ALU_INC =>
231                     L_DOUT      <= L_INC_D & L_INC_D;
232                     Q_FLAGS(1)       <= ze(L_INC_D);                         -- Zero
233                     Q_FLAGS(2)       <= L_INC_D(7);                          -- Negative
234                     if (L_D8 = X"7F") then
235                         Q_FLAGS(3)       <= '1';                             -- Overflow
236                         Q_FLAGS(4)       <= not L_INC_D(7);                  -- Signed
237                     else
238                         Q_FLAGS(3)       <= '0';                             -- Overflow
239                         Q_FLAGS(4)       <= L_INC_D(7);                      -- Signed
240                     end if;
241
242                 when ALU_INTR =>
243                     L_DOUT      <= I_PC;
244                     Q_FLAGS(7)       <= I_IMM(6);    -- ena/disable interrupts
```

```
245
246                 when ALU_LSR  =>
247                     L_DOUT       <= L_LSR_D & L_LSR_D;
248                     Q_FLAGS(0)       <= L_D8(0);                             -- Carry
249                     Q_FLAGS(1)       <= ze(L_LSR_D);                         -- Zero
250                     Q_FLAGS(2)       <= '0';                                 -- Negative
251                     Q_FLAGS(3)       <= L_D8(0);                             -- Overflow
252                     Q_FLAGS(4)       <= L_D8(0);                             -- Signed
253
254                 when ALU_D_MV_Q =>
255                     L_DOUT       <= L_D8 & L_D8;
256
257                 when ALU_R_MV_Q =>
258                     L_DOUT       <= L_RI8 & L_RI8;
259
260                 when ALU_MV_16 =>
261                     L_DOUT       <= I_R(15 downto 8) & L_RI8;
262
263                 when ALU_MULT =>
264                     Q_FLAGS(0)       <= L_PROD(15);                          -- Carry
265                     if I_IMM(7) = '0' then              -- MUL
266                         L_DOUT       <= L_PROD(15 downto 0);
267                         Q_FLAGS(1)       <= ze(L_PROD(15 downto 8))          -- Zero
268                                 and ze(L_PROD( 7 downto 0));
269                     else                                -- FMUL
270                         L_DOUT       <= L_PROD(14 downto 0) & "0";
271                         Q_FLAGS(1)       <= ze(L_PROD(14 downto 7))          -- Zero
272                                 and ze(L_PROD( 6 downto 0) & "0");
273                     end if;
274
275                 when ALU_NEG =>
276                     L_DOUT       <= L_NEG_D & L_NEG_D;
277                     Q_FLAGS(0)       <= not ze(L_D8);                        -- Carry
278                     Q_FLAGS(1)       <= ze(L_NEG_D);                         -- Zero
279                     Q_FLAGS(2)       <= L_NEG_D(7);                          -- Negative
280                     if (L_D8 = X"80") then
281                         Q_FLAGS(3)       <= '1';                             -- Overflow
282                         Q_FLAGS(4)       <= not L_NEG_D(7);                  -- Signed
283                     else
284                         Q_FLAGS(3)       <= '0';                             -- Overflow
285                         Q_FLAGS(4)       <= L_NEG_D(7);                      -- Signed
286                     end if;
287                     Q_FLAGS(5)       <= L_D8(3) or L_NEG_D(3);               -- Halfcarry
288
289                 when ALU_OR =>
290                     L_DOUT       <= L_OR_DR & L_OR_DR;
291                     Q_FLAGS(1)       <= ze(L_OR_DR);                         -- Zero
292                     Q_FLAGS(2)       <= L_OR_DR(7);                          -- Negative
293                     Q_FLAGS(3)       <= '0';                                 -- Overflow
294                     Q_FLAGS(4)       <= L_OR_DR(7);                          -- Signed
295
296                 when ALU_PC_1 =>    -- ICALL, RCALL
297                     L_DOUT       <= I_PC + X"0001";
298
299                 when ALU_PC_2 =>    -- CALL
300                     L_DOUT       <= I_PC + X"0002";
301
302                 when ALU_ROR =>
303                     L_DOUT       <= L_ROR_D & L_ROR_D;
304                     Q_FLAGS(1)       <= ze(L_ROR_D);                         -- Zero
305                     Q_FLAGS(2)       <= I_FLAGS(0);                          -- Negative
306                     Q_FLAGS(3)       <= I_FLAGS(0) xor L_D8(0);              -- Overflow
```

```
307                     Q_FLAGS(4)       <= I_FLAGS(0);                           -- Signed
308
309                 when ALU_SBC =>
310                     L_DOUT      <= L_SBC_DR & L_SBC_DR;
311                     Q_FLAGS(0)       <= cy(L_SBC_DR(7), L_RI8(7), L_D8(7));   -- Carry
312                     Q_FLAGS(1)       <= ze(L_SBC_DR) and I_FLAGS(1);          -- Zero
313                     Q_FLAGS(2)       <= L_SBC_DR(7);                          -- Negative
314                     Q_FLAGS(3)       <= ov(L_SBC_DR(7), L_RI8(7), L_D8(7));   -- Overflow
315                     Q_FLAGS(4)       <= si(L_SBC_DR(7), L_RI8(7), L_D8(7));   -- Signed
316                     Q_FLAGS(5)       <= cy(L_SBC_DR(3), L_RI8(3), L_D8(3));   -- Halfcarry
317
318                 when ALU_SBIW =>
319                     L_DOUT      <= L_SBIW_D;
320                     Q_FLAGS(0)       <= L_SBIW_D(15) and not I_D(15);         -- Carry
321                     Q_FLAGS(1)       <= ze(L_SBIW_D(15 downto 8)) and
322                                    ze(L_SBIW_D(7 downto 0));              -- Zero
323                     Q_FLAGS(2)       <= L_SBIW_D(15);                         -- Negative
324                     Q_FLAGS(3)       <= I_D(15) and not L_SBIW_D(15);         -- Overflow
325                     Q_FLAGS(4)       <=  (L_SBIW_D(15) and not I_D(15))
326                                 xor (I_D(15) and not L_SBIW_D(15));       -- Signed
327
328                 when ALU_SREG =>
329                     case I_BIT(2 downto 0) is
330                         when "000"  => Q_FLAGS(0)       <= I_BIT(3);
331                         when "001"  => Q_FLAGS(1)       <= I_BIT(3);
332                         when "010"  => Q_FLAGS(2)       <= I_BIT(3);
333                         when "011"  => Q_FLAGS(3)       <= I_BIT(3);
334                         when "100"  => Q_FLAGS(4)       <= I_BIT(3);
335                         when "101"  => Q_FLAGS(5)       <= I_BIT(3);
336                         when "110"  => Q_FLAGS(6)       <= I_BIT(3);
337                         when others => Q_FLAGS(7)       <= I_BIT(3);
338                     end case;
339
340                 when ALU_SUB =>
341                     L_DOUT      <= L_SUB_DR & L_SUB_DR;
342                     Q_FLAGS(0)       <= cy(L_SUB_DR(7), L_RI8(7), L_D8(7));   -- Carry
343                     Q_FLAGS(1)       <= ze(L_SUB_DR);                         -- Zero
344                     Q_FLAGS(2)       <= L_SUB_DR(7);                          -- Negative
345                     Q_FLAGS(3)       <= ov(L_SUB_DR(7), L_RI8(7), L_D8(7));   -- Overflow
346                     Q_FLAGS(4)       <= si(L_SUB_DR(7), L_RI8(7), L_D8(7));   -- Signed
347                     Q_FLAGS(5)       <= cy(L_SUB_DR(3), L_RI8(3), L_D8(3));   -- Halfcarry
348
349                 when ALU_SWAP =>
350                     L_DOUT      <= L_SWAP_D & L_SWAP_D;
351
352                 when others =>
353             end case;
354         end process;
355
356         L_D8       <= I_D(15 downto 8) when (I_D0 = '1') else I_D(7 downto 0);
357         L_R8       <= I_R(15 downto 8) when (I_R0 = '1') else I_R(7 downto 0);
358         L_RI8       <= I_IMM           when (I_RSEL = RS_IMM) else L_R8;
359
360         L_ADIW_D        <= I_D + ("0000000000" & I_IMM(5 downto 0));
361         L_SBIW_D        <= I_D - ("0000000000" & I_IMM(5 downto 0));
362         L_ADD_DR        <= L_D8 + L_RI8;
363         L_ADC_DR        <= L_ADD_DR + ("0000000" & I_FLAGS(0));
364         L_ASR_D         <= L_D8(7) & L_D8(7 downto 1);
365         L_AND_DR        <= L_D8 and L_RI8;
366         L_DEC_D         <= L_D8 - X"01";
367         L_INC_D         <= L_D8 + X"01";
368         L_LSR_D         <= '0' & L_D8(7 downto 1);
```

```
369         L_NEG_D         <= X"00" - L_D8;
370         L_NOT_D         <= not L_D8;
371         L_OR_DR         <= L_D8 or L_RI8;
372         L_PROD          <= (L_SIGN_D & L_D8) * (L_SIGN_R & L_R8);
373         L_ROR_D         <= I_FLAGS(0) &  L_D8(7 downto 1);
374         L_SUB_DR        <= L_D8 - L_RI8;
375         L_SBC_DR        <= L_SUB_DR - ("0000000" & I_FLAGS(0));
376         L_SIGN_D        <= L_D8(7) and I_IMM(6);
377         L_SIGN_R        <= L_R8(7) and I_IMM(5);
378         L_SWAP_D        <= L_D8(3 downto 0) & L_D8(7 downto 4);
379         L_XOR_DR        <= L_D8 xor L_R8;
380
381         Q_DOUT      <= (I_DIN & I_DIN) when (I_RSEL = RS_DIN) else L_DOUT;
382
383     end Behavioral;
384
```

src/alu.vhd

11 LISTING OF avr_fpga.vhd

```
1    -------------------------------------------------------------------------------
2    --
3    -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4    --
5    --  This code is free software: you can redistribute it and/or modify
6    --  it under the terms of the GNU General Public License as published by
7    --  the Free Software Foundation, either version 3 of the License, or
8    --  (at your option) any later version.
9    --
10   --  This code is distributed in the hope that it will be useful,
11   --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12   --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13   --  GNU General Public License for more details.
14   --
15   --  You should have received a copy of the GNU General Public License
16   --  along with this code (see the file named COPYING).
17   --  If not, see http://www.gnu.org/licenses/.
18   --
19   -------------------------------------------------------------------------------
20   -------------------------------------------------------------------------------
21   --
22   -- Module Name:     avr_fpga - Behavioral
23   -- Create Date:     13:51:24 11/07/2009
24   -- Description:     top level of a CPU
25   --
26   -------------------------------------------------------------------------------
27
28   library IEEE;
29   use IEEE.STD_LOGIC_1164.ALL;
30   use IEEE.STD_LOGIC_ARITH.ALL;
31   use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33   entity avr_fpga is
34       port (  I_CLK_100   : in  std_logic;
35               I_SWITCH    : in  std_logic_vector(9 downto 0);
36               I_RX        : in  std_logic;
37
38               Q_7_SEGMENT : out std_logic_vector(6 downto 0);
39               Q_LEDS      : out std_logic_vector(3 downto 0);
40               Q_TX        : out std_logic);
41   end avr_fpga;
42
43   architecture Behavioral of avr_fpga is
44
45   component cpu_core
46       port (  I_CLK       : in  std_logic;
47               I_CLR       : in  std_logic;
48               I_INTVEC    : in  std_logic_vector( 5 downto 0);
49               I_DIN       : in  std_logic_vector( 7 downto 0);
50
51               Q_OPC       : out std_logic_vector(15 downto 0);
52               Q_PC        : out std_logic_vector(15 downto 0);
53               Q_DOUT      : out std_logic_vector( 7 downto 0);
54               Q_ADR_IO    : out std_logic_vector( 7 downto 0);
55               Q_RD_IO     : out std_logic;
56               Q_WE_IO     : out std_logic);
57   end component;
58
```

```
 59     signal  C_PC            : std_logic_vector(15 downto 0);
 60     signal  C_OPC           : std_logic_vector(15 downto 0);
 61     signal  C_ADR_IO        : std_logic_vector( 7 downto 0);
 62     signal  C_DOUT          : std_logic_vector( 7 downto 0);
 63     signal  C_RD_IO         : std_logic;
 64     signal  C_WE_IO         : std_logic;
 65
 66     component io
 67         port (  I_CLK       : in  std_logic;
 68                 I_CLR       : in  std_logic;
 69                 I_ADR_IO    : in  std_logic_vector( 7 downto 0);
 70                 I_DIN       : in  std_logic_vector( 7 downto 0);
 71                 I_RD_IO     : in  std_logic;
 72                 I_WE_IO     : in  std_logic;
 73                 I_SWITCH    : in  std_logic_vector( 7 downto 0);
 74                 I_RX        : in  std_logic;
 75
 76                 Q_7_SEGMENT : out std_logic_vector( 6 downto 0);
 77                 Q_DOUT      : out std_logic_vector( 7 downto 0);
 78                 Q_INTVEC    : out std_logic_vector(5 downto 0);
 79                 Q_LEDS      : out std_logic_vector( 1 downto 0);
 80                 Q_TX        : out std_logic);
 81     end component;
 82
 83     signal N_INTVEC         : std_logic_vector( 5 downto 0);
 84     signal N_DOUT           : std_logic_vector( 7 downto 0);
 85     signal N_TX             : std_logic;
 86     signal N_7_SEGMENT      : std_logic_vector( 6 downto 0);
 87
 88     component segment7
 89         port ( I_CLK        : in  std_logic;
 90
 91                I_CLR        : in  std_logic;
 92                I_OPC        : in  std_logic_vector(15 downto 0);
 93                I_PC         : in  std_logic_vector(15 downto 0);
 94
 95                Q_7_SEGMENT  : out std_logic_vector( 6 downto 0));
 96     end component;
 97
 98     signal S_7_SEGMENT      : std_logic_vector( 6 downto 0);
 99
100     signal L_CLK            : std_logic := '0';
101     signal L_CLK_CNT        : std_logic_vector( 2 downto 0) := "000";
102     signal L_CLR            : std_logic;            -- reset,  active low
103     signal L_CLR_N          : std_logic := '0';     -- reset,  active low
104     signal L_C1_N           : std_logic := '0';     -- switch debounce, active low
105     signal L_C2_N           : std_logic := '0';     -- switch debounce, active low
106
107     begin
108
109         cpu : cpu_core
110         port map(   I_CLK       => L_CLK,
111                     I_CLR       => L_CLR,
112                     I_DIN       => N_DOUT,
113                     I_INTVEC    => N_INTVEC,
114
115                     Q_ADR_IO    => C_ADR_IO,
116                     Q_DOUT      => C_DOUT,
117                     Q_OPC       => C_OPC,
118                     Q_PC        => C_PC,
119                     Q_RD_IO     => C_RD_IO,
120                     Q_WE_IO     => C_WE_IO);
```

```
121
122         ino : io
123         port map(   I_CLK       => L_CLK,
124                     I_CLR       => L_CLR,
125                     I_ADR_IO    => C_ADR_IO,
126                     I_DIN       => C_DOUT,
127                     I_RD_IO     => C_RD_IO,
128                     I_RX        => I_RX,
129                     I_SWITCH    => I_SWITCH(7 downto 0),
130                     I_WE_IO     => C_WE_IO,
131
132                     Q_7_SEGMENT => N_7_SEGMENT,
133                     Q_DOUT      => N_DOUT,
134                     Q_INTVEC    => N_INTVEC,
135                     Q_LEDS      => Q_LEDS(1 downto 0),
136                     Q_TX        => N_TX);
137
138         seg : segment7
139         port map(   I_CLK       => L_CLK,
140                     I_CLR       => L_CLR,
141                     I_OPC       => C_OPC,
142                     I_PC        => C_PC,
143
144                     Q_7_SEGMENT => S_7_SEGMENT);
145
146         -- input clock scaler
147         --
148         clk_div : process(I_CLK_100)
149         begin
150             if (rising_edge(I_CLK_100)) then
151                 L_CLK_CNT      <= L_CLK_CNT + "001";
152                 if (L_CLK_CNT = "001") then
153                     L_CLK_CNT      <= "000";
154                     L_CLK      <= not L_CLK;
155                 end if;
156             end if;
157         end process;
158
159         -- reset button debounce process
160         --
161         deb : process(L_CLK)
162         begin
163             if (rising_edge(L_CLK)) then
164                 -- switch debounce
165                 if ((I_SWITCH(8) = '0') or (I_SWITCH(9) = '0')) then    -- pushed
166                     L_CLR_N      <= '0';
167                     L_C2_N       <= '0';
168                     L_C1_N       <= '0';
169                 else                                                      -- released
170                     L_CLR_N      <= L_C2_N;
171                     L_C2_N       <= L_C1_N;
172                     L_C1_N       <= '1';
173                 end if;
174             end if;
175         end process;
176
177         L_CLR      <= not L_CLR_N;
178
179         Q_LEDS(2)      <= I_RX;
180         Q_LEDS(3)      <= N_TX;
181         Q_7_SEGMENT       <= N_7_SEGMENT when (I_SWITCH(7) = '1') else S_7_SEGMENT;
182         Q_TX      <= N_TX;
```

```
183
184     end Behavioral;
185
```

src/avr_fpga.vhd

12 LISTING OF baudgen.vhd

```
 1   -------------------------------------------------------------------------------
 2   --
 3   -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4   --
 5   --  This code is free software: you can redistribute it and/or modify
 6   --  it under the terms of the GNU General Public License as published by
 7   --  the Free Software Foundation, either version 3 of the License, or
 8   --  (at your option) any later version.
 9   --
10   --  This code is distributed in the hope that it will be useful,
11   --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12   --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13   --  GNU General Public License for more details.
14   --
15   --  You should have received a copy of the GNU General Public License
16   --  along with this code (see the file named COPYING).
17   --  If not, see http://www.gnu.org/licenses/.
18   --
19   -------------------------------------------------------------------------------
20   -------------------------------------------------------------------------------
21   --
22   -- Module Name:    baudgen - Behavioral
23   -- Create Date:    13:51:24 11/07/2009
24   -- Description:    fixed baud rate generator
25   --
26   -------------------------------------------------------------------------------
27   --
28   library IEEE;
29   use IEEE.STD_LOGIC_1164.ALL;
30   use IEEE.STD_LOGIC_ARITH.ALL;
31   use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33   entity baudgen is
34       generic(clock_freq  : std_logic_vector(31 downto 0);
35                   baud_rate   : std_logic_vector(27 downto 0));
36       port(   I_CLK       : in  std_logic;
37
38               I_CLR       : in  std_logic;
39               Q_CE_1      : out std_logic;    -- baud x  1 clock enable
40               Q_CE_16     : out std_logic);   -- baud x 16 clock enable
41   end baudgen;
42
43
44   architecture Behavioral of baudgen is
45
46   constant BAUD_16        : std_logic_vector(31 downto 0) := baud_rate & "0000";
47   constant LIMIT          : std_logic_vector(31 downto 0) := clock_freq - BAUD_16;
48
49   signal L_CE_16          : std_logic;
50   signal L_CNT_16         : std_logic_vector( 3 downto 0);
51   signal L_COUNTER        : std_logic_vector(31 downto 0);
52
53   begin
54
55       baud16: process(I_CLK)
56       begin
57           if (rising_edge(I_CLK)) then
58               if (I_CLR = '1') then
```

```
59                  L_COUNTER       <= X"00000000";
60              elsif (L_COUNTER >= LIMIT) then
61                  L_COUNTER       <= L_COUNTER - LIMIT;
62              else
63                  L_COUNTER       <= L_COUNTER + BAUD_16;
64              end if;
65          end if;
66      end process;
67
68      baud1: process(I_CLK)
69      begin
70          if (rising_edge(I_CLK)) then
71              if (I_CLR = '1') then
72                  L_CNT_16       <= "0000";
73              elsif (L_CE_16 = '1') then
74                  L_CNT_16       <= L_CNT_16 + "0001";
75              end if;
76          end if;
77      end process;
78
79      L_CE_16       <= '1' when (L_COUNTER >= LIMIT) else '0';
80      Q_CE_16       <= L_CE_16;
81      Q_CE_1       <= L_CE_16 when L_CNT_16 = "1111" else '0';
82
83  end behavioral;
84
```

src/baudgen.vhd

13 LISTING OF common.vhd

```
1     -------------------------------------------------------------------------------
2     --
3     -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4     --
5     --  This code is free software: you can redistribute it and/or modify
6     --  it under the terms of the GNU General Public License as published by
7     --  the Free Software Foundation, either version 3 of the License, or
8     --  (at your option) any later version.
9     --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    common
23    -- Create Date:    13:51:24 11/07/2009
24    -- Description:    constants shared by different modules.
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.STD_LOGIC_1164.all;
30
31    package common is
32
33        -----------------------------------------------------------------------
34
35        -- ALU operations
36        --
37        constant ALU_ADC    : std_logic_vector(4 downto 0) := "00000";
38        constant ALU_ADD    : std_logic_vector(4 downto 0) := "00001";
39        constant ALU_ADIW   : std_logic_vector(4 downto 0) := "00010";
40        constant ALU_AND    : std_logic_vector(4 downto 0) := "00011";
41        constant ALU_ASR    : std_logic_vector(4 downto 0) := "00100";
42        constant ALU_BLD    : std_logic_vector(4 downto 0) := "00101";
43        constant ALU_BIT_CS : std_logic_vector(4 downto 0) := "00110";
44        constant ALU_COM    : std_logic_vector(4 downto 0) := "00111";
45        constant ALU_DEC    : std_logic_vector(4 downto 0) := "01000";
46        constant ALU_EOR    : std_logic_vector(4 downto 0) := "01001";
47        constant ALU_MV_16  : std_logic_vector(4 downto 0) := "01010";
48        constant ALU_INC    : std_logic_vector(4 downto 0) := "01011";
49        constant ALU_INTR   : std_logic_vector(4 downto 0) := "01100";
50        constant ALU_LSR    : std_logic_vector(4 downto 0) := "01101";
51        constant ALU_D_MV_Q : std_logic_vector(4 downto 0) := "01110";
52        constant ALU_R_MV_Q : std_logic_vector(4 downto 0) := "01111";
53        constant ALU_MULT   : std_logic_vector(4 downto 0) := "10000";
54        constant ALU_NEG    : std_logic_vector(4 downto 0) := "10001";
55        constant ALU_OR     : std_logic_vector(4 downto 0) := "10010";
56        constant ALU_PC_1   : std_logic_vector(4 downto 0) := "10011";
57        constant ALU_PC_2   : std_logic_vector(4 downto 0) := "10100";
58        constant ALU_ROR    : std_logic_vector(4 downto 0) := "10101";
```

```
59          constant ALU_SBC    : std_logic_vector(4 downto 0) := "10110";
60          constant ALU_SBIW   : std_logic_vector(4 downto 0) := "10111";
61          constant ALU_SREG   : std_logic_vector(4 downto 0) := "11000";
62          constant ALU_SUB    : std_logic_vector(4 downto 0) := "11001";
63          constant ALU_SWAP   : std_logic_vector(4 downto 0) := "11010";
64
65          -----------------------------------------------------------------------
66          --
67          -- PC manipulations
68          --
69          constant PC_NEXT    : std_logic_vector(2 downto 0) := "000";    -- PC += 1
70          constant PC_BCC     : std_logic_vector(2 downto 0) := "001";    -- PC ?= IMM
71          constant PC_LD_I    : std_logic_vector(2 downto 0) := "010";    -- PC = IMM
72          constant PC_LD_Z    : std_logic_vector(2 downto 0) := "011";    -- PC = Z
73          constant PC_LD_S    : std_logic_vector(2 downto 0) := "100";    -- PC = (SP)
74          constant PC_SKIP_Z  : std_logic_vector(2 downto 0) := "101";    -- SKIP if Z
75          constant PC_SKIP_T  : std_logic_vector(2 downto 0) := "110";    -- SKIP if T
76
77          -----------------------------------------------------------------------
78          --
79          -- Addressing modes. An address mode consists of two sub-fields,
80          -- which are the source of the address and an offset from the source.
81          -- Bit 3 indicates if the address will be modified.
82
83          -- address source
84          constant AS_SP  : std_logic_vector(2 downto 0) := "000";     -- SP
85          constant AS_Z   : std_logic_vector(2 downto 0) := "001";     -- Z
86          constant AS_Y   : std_logic_vector(2 downto 0) := "010";     -- Y
87          constant AS_X   : std_logic_vector(2 downto 0) := "011";     -- X
88          constant AS_IMM : std_logic_vector(2 downto 0) := "100";     -- IMM
89
90          -- address offset
91          constant AO_0   : std_logic_vector(5 downto 3) := "000";     -- as is
92          constant AO_Q   : std_logic_vector(5 downto 3) := "010";     -- +q
93          constant AO_i   : std_logic_vector(5 downto 3) := "001";     -- +1
94          constant AO_ii  : std_logic_vector(5 downto 3) := "011";     -- +2
95          constant AO_d   : std_logic_vector(5 downto 3) := "101";     -- -1
96          constant AO_dd  : std_logic_vector(5 downto 3) := "111";     -- -2
97          --                                                  |
98          --                                              +--+
99          -- address updated ?                            |
100         --                                              v
101         constant AM_WX : std_logic_vector(3 downto 0) := '1' & AS_X;  -- X ++ or --
102         constant AM_WY : std_logic_vector(3 downto 0) := '1' & AS_Y;  -- Y ++ or --
103         constant AM_WZ : std_logic_vector(3 downto 0) := '1' & AS_Z;  -- Z ++ or --
104         constant AM_WS : std_logic_vector(3 downto 0) := '1' & AS_SP; -- SP ++/--
105
106         -- address modes used
107         --
108         constant AMOD_ABS : std_logic_vector(5 downto 0) := AO_0  & AS_IMM; -- IMM
109         constant AMOD_X   : std_logic_vector(5 downto 0) := AO_0  & AS_X;   -- (X)
110         constant AMOD_Xq  : std_logic_vector(5 downto 0) := AO_Q  & AS_X;   -- (X+q)
111         constant AMOD_Xi  : std_logic_vector(5 downto 0) := AO_i  & AS_X;   -- (X++)
112         constant AMOD_dX  : std_logic_vector(5 downto 0) := AO_d  & AS_X;   -- (--X)
113         constant AMOD_Y   : std_logic_vector(5 downto 0) := AO_0  & AS_Y;   -- (Y)
114         constant AMOD_Yq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Y;   -- (Y+q)
115         constant AMOD_Yi  : std_logic_vector(5 downto 0) := AO_i  & AS_Y;   -- (Y++)
116         constant AMOD_dY  : std_logic_vector(5 downto 0) := AO_d  & AS_Y;   -- (--Y)
117         constant AMOD_Z   : std_logic_vector(5 downto 0) := AO_0  & AS_Z;   -- (Z)
118         constant AMOD_Zq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Z;   -- (Z+q)
119         constant AMOD_Zi  : std_logic_vector(5 downto 0) := AO_i  & AS_Z;   -- (Z++)
120         constant AMOD_dZ  : std_logic_vector(5 downto 0) := AO_d  & AS_Z;   -- (--Z)
```

```
121         constant AMOD_SPi : std_logic_vector(5 downto 0) := AO_i  & AS_SP;  -- (SP++)
122         constant AMOD_SPii: std_logic_vector(5 downto 0) := AO_ii & AS_SP;  -- (SP++)
123         constant AMOD_dSP : std_logic_vector(5 downto 0) := AO_d  & AS_SP;  -- (--SP)
124         constant AMOD_ddSP: std_logic_vector(5 downto 0) := AO_dd & AS_SP;  -- (--SP)
125
126         -----------------------------------------------------------------------
127
128         -- Stack pointer manipulations.
129         --
130         constant SP_NOP : std_logic_vector(2 downto 0) := "000";
131         constant SP_ADD1: std_logic_vector(2 downto 0) := "001";
132         constant SP_ADD2: std_logic_vector(2 downto 0) := "010";
133         constant SP_SUB1: std_logic_vector(2 downto 0) := "011";
134         constant SP_SUB2: std_logic_vector(2 downto 0) := "100";
135
136         -----------------------------------------------------------------------
137         --
138         -- ALU multiplexers.
139         --
140         constant RS_REG : std_logic_vector(1 downto 0) := "00";
141         constant RS_IMM : std_logic_vector(1 downto 0) := "01";
142         constant RS_DIN : std_logic_vector(1 downto 0) := "10";
143
144         -----------------------------------------------------------------------
145         --
146         -- Multiplier variants. F means FMULT (as opposed to MULT).
147         -- S and U means signed vs. unsigned operands.
148         --
149         constant MULT_UU  : std_logic_vector(2 downto 0) := "000";
150         constant MULT_SU  : std_logic_vector(2 downto 0) := "010";
151         constant MULT_SS  : std_logic_vector(2 downto 0) := "011";
152         constant MULT_FUU : std_logic_vector(2 downto 0) := "100";
153         constant MULT_FSU : std_logic_vector(2 downto 0) := "110";
154         constant MULT_FSS : std_logic_vector(2 downto 0) := "111";
155
156         -----------------------------------------------------------------------
157
158     end common;
159
```

src/common.vhd

14 LISTING OF cpu_core.vhd

```
1     -------------------------------------------------------------------------------
2     --
3     -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4     --
5     --  This code is free software: you can redistribute it and/or modify
6     --  it under the terms of the GNU General Public License as published by
7     --  the Free Software Foundation, either version 3 of the License, or
8     --  (at your option) any later version.
9     --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    cpu_core - Behavioral
23    -- Create Date:    13:51:24 11/07/2009
24    -- Description:    the instruction set implementation of a CPU.
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.STD_LOGIC_1164.ALL;
30    use IEEE.STD_LOGIC_ARITH.ALL;
31    use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33    entity cpu_core is
34        port (  I_CLK       : in  std_logic;
35                I_CLR       : in  std_logic;
36                I_INTVEC    : in  std_logic_vector( 5 downto 0);
37                I_DIN       : in  std_logic_vector( 7 downto 0);
38
39                Q_OPC       : out std_logic_vector(15 downto 0);
40                Q_PC        : out std_logic_vector(15 downto 0);
41                Q_DOUT      : out std_logic_vector( 7 downto 0);
42                Q_ADR_IO    : out std_logic_vector( 7 downto 0);
43                Q_RD_IO     : out std_logic;
44                Q_WE_IO     : out std_logic);
45    end cpu_core;
46
47    architecture Behavioral of cpu_core is
48
49    component opc_fetch
50        port(   I_CLK       : in  std_logic;
51
52                I_CLR       : in  std_logic;
53                I_INTVEC    : in  std_logic_vector( 5 downto 0);
54                I_NEW_PC    : in  std_logic_vector(15 downto 0);
55                I_LOAD_PC   : in  std_logic;
56                I_PM_ADR    : in  std_logic_vector(11 downto 0);
57                I_SKIP      : in  std_logic;
58
```

```
 59                Q_OPC        : out std_logic_vector(31 downto 0);
 60                Q_PC         : out std_logic_vector(15 downto 0);
 61                Q_PM_DOUT    : out std_logic_vector( 7 downto 0);
 62                Q_T0         : out std_logic);
 63     end component;
 64
 65     signal F_PC             : std_logic_vector(15 downto 0);
 66     signal F_OPC            : std_logic_vector(31 downto 0);
 67     signal F_PM_DOUT        : std_logic_vector( 7 downto 0);
 68     signal F_T0             : std_logic;
 69
 70     component opc_deco is
 71         port (  I_CLK        : in  std_logic;
 72
 73                 I_OPC        : in  std_logic_vector(31 downto 0);
 74                 I_PC         : in  std_logic_vector(15 downto 0);
 75                 I_T0         : in  std_logic;
 76
 77                 Q_ALU_OP     : out std_logic_vector( 4 downto 0);
 78                 Q_AMOD       : out std_logic_vector( 5 downto 0);
 79                 Q_BIT        : out std_logic_vector( 3 downto 0);
 80                 Q_DDDDD      : out std_logic_vector( 4 downto 0);
 81                 Q_IMM        : out std_logic_vector(15 downto 0);
 82                 Q_JADR       : out std_logic_vector(15 downto 0);
 83                 Q_OPC        : out std_logic_vector(15 downto 0);
 84                 Q_PC         : out std_logic_vector(15 downto 0);
 85                 Q_PC_OP      : out std_logic_vector( 2 downto 0);
 86                 Q_PMS        : out std_logic;  -- program memory select
 87                 Q_RD_M       : out std_logic;
 88                 Q_RRRRR      : out std_logic_vector( 4 downto 0);
 89                 Q_RSEL       : out std_logic_vector( 1 downto 0);
 90                 Q_WE_01      : out std_logic;
 91                 Q_WE_D       : out std_logic_vector( 1 downto 0);
 92                 Q_WE_F       : out std_logic;
 93                 Q_WE_M       : out std_logic_vector( 1 downto 0);
 94                 Q_WE_XYZS    : out std_logic);
 95     end component;
 96
 97     signal D_ALU_OP         : std_logic_vector( 4 downto 0);
 98     signal D_AMOD           : std_logic_vector( 5 downto 0);
 99     signal D_BIT            : std_logic_vector( 3 downto 0);
100     signal D_DDDDD          : std_logic_vector( 4 downto 0);
101     signal D_IMM            : std_logic_vector(15 downto 0);
102     signal D_JADR           : std_logic_vector(15 downto 0);
103     signal D_OPC            : std_logic_vector(15 downto 0);
104     signal D_PC             : std_logic_vector(15 downto 0);
105     signal D_PC_OP          : std_logic_vector(2 downto 0);
106     signal D_PMS            : std_logic;
107     signal D_RD_M           : std_logic;
108     signal D_RRRRR          : std_logic_vector( 4 downto 0);
109     signal D_RSEL           : std_logic_vector( 1 downto 0);
110     signal D_WE_01          : std_logic;
111     signal D_WE_D           : std_logic_vector( 1 downto 0);
112     signal D_WE_F           : std_logic;
113     signal D_WE_M           : std_logic_vector( 1 downto 0);
114     signal D_WE_XYZS        : std_logic;
115
116     component data_path
117         port(   I_CLK        : in    std_logic;
118
119                 I_ALU_OP     : in  std_logic_vector( 4 downto 0);
120                 I_AMOD       : in  std_logic_vector( 5 downto 0);
```

```
121                 I_BIT       : in  std_logic_vector( 3 downto 0);
122                 I_DDDDD     : in  std_logic_vector( 4 downto 0);
123                 I_DIN       : in  std_logic_vector( 7 downto 0);
124                 I_IMM       : in  std_logic_vector(15 downto 0);
125                 I_JADR      : in  std_logic_vector(15 downto 0);
126                 I_PC_OP     : in  std_logic_vector( 2 downto 0);
127                 I_OPC       : in  std_logic_vector(15 downto 0);
128                 I_PC        : in  std_logic_vector(15 downto 0);
129                 I_PMS       : in  std_logic;  -- program memory select
130                 I_RD_M      : in  std_logic;
131                 I_RRRRR     : in  std_logic_vector( 4 downto 0);
132                 I_RSEL      : in  std_logic_vector( 1 downto 0);
133                 I_WE_01     : in  std_logic;
134                 I_WE_D      : in  std_logic_vector( 1 downto 0);
135                 I_WE_F      : in  std_logic;
136                 I_WE_M      : in  std_logic_vector( 1 downto 0);
137                 I_WE_XYZS   : in  std_logic;
138
139                 Q_ADR       : out std_logic_vector(15 downto 0);
140                 Q_DOUT      : out std_logic_vector( 7 downto 0);
141                 Q_INT_ENA   : out std_logic;
142                 Q_LOAD_PC   : out std_logic;
143                 Q_NEW_PC    : out std_logic_vector(15 downto 0);
144                 Q_OPC       : out std_logic_vector(15 downto 0);
145                 Q_PC        : out std_logic_vector(15 downto 0);
146                 Q_RD_IO     : out std_logic;
147                 Q_SKIP      : out std_logic;
148                 Q_WE_IO     : out std_logic);
149     end component;
150
151     signal R_INT_ENA        : std_logic;
152     signal R_NEW_PC         : std_logic_vector(15 downto 0);
153     signal R_LOAD_PC        : std_logic;
154     signal R_SKIP           : std_logic;
155     signal R_ADR            : std_logic_vector(15 downto 0);
156
157     -- local signals
158     --
159     signal L_DIN            : std_logic_vector( 7 downto 0);
160     signal L_INTVEC_5       : std_logic;
161
162     begin
163
164         opcf : opc_fetch
165         port map(   I_CLK       => I_CLK,
166
167                     I_CLR       => I_CLR,
168                     I_INTVEC(5) => L_INTVEC_5,
169                     I_INTVEC(4 downto 0) => I_INTVEC(4 downto 0),
170                     I_LOAD_PC   => R_LOAD_PC,
171                     I_NEW_PC    => R_NEW_PC,
172                     I_PM_ADR    => R_ADR(11 downto 0),
173                     I_SKIP      => R_SKIP,
174
175                     Q_PC        => F_PC,
176                     Q_OPC       => F_OPC,
177                     Q_T0        => F_T0,
178                     Q_PM_DOUT   => F_PM_DOUT);
179
180         odec : opc_deco
181         port map(   I_CLK       => I_CLK,
182
```

```
183                 I_OPC       => F_OPC,
184                 I_PC        => F_PC,
185                 I_T0        => F_T0,
186
187                 Q_ALU_OP    => D_ALU_OP,
188                 Q_AMOD      => D_AMOD,
189                 Q_BIT       => D_BIT,
190                 Q_DDDDD     => D_DDDDD,
191                 Q_IMM       => D_IMM,
192                 Q_JADR      => D_JADR,
193                 Q_OPC       => D_OPC,
194                 Q_PC        => D_PC,
195                 Q_PC_OP     => D_PC_OP,
196                 Q_PMS       => D_PMS,
197                 Q_RD_M      => D_RD_M,
198                 Q_RRRRR     => D_RRRRR,
199                 Q_RSEL      => D_RSEL,
200                 Q_WE_01     => D_WE_01,
201                 Q_WE_D      => D_WE_D,
202                 Q_WE_F      => D_WE_F,
203                 Q_WE_M      => D_WE_M,
204                 Q_WE_XYZS   => D_WE_XYZS);
205
206     dpath : data_path
207     port map(   I_CLK       => I_CLK,
208
209                 I_ALU_OP    => D_ALU_OP,
210                 I_AMOD      => D_AMOD,
211                 I_BIT       => D_BIT,
212                 I_DDDDD     => D_DDDDD,
213                 I_DIN       => L_DIN,
214                 I_IMM       => D_IMM,
215                 I_JADR      => D_JADR,
216                 I_OPC       => D_OPC,
217                 I_PC        => D_PC,
218                 I_PC_OP     => D_PC_OP,
219                 I_PMS       => D_PMS,
220                 I_RD_M      => D_RD_M,
221                 I_RRRRR     => D_RRRRR,
222                 I_RSEL      => D_RSEL,
223                 I_WE_01     => D_WE_01,
224                 I_WE_D      => D_WE_D,
225                 I_WE_F      => D_WE_F,
226                 I_WE_M      => D_WE_M,
227                 I_WE_XYZS   => D_WE_XYZS,
228
229                 Q_ADR       => R_ADR,
230                 Q_DOUT      => Q_DOUT,
231                 Q_INT_ENA   => R_INT_ENA,
232                 Q_NEW_PC    => R_NEW_PC,
233                 Q_OPC       => Q_OPC,
234                 Q_PC        => Q_PC,
235                 Q_LOAD_PC   => R_LOAD_PC,
236                 Q_RD_IO     => Q_RD_IO,
237                 Q_SKIP      => R_SKIP,
238                 Q_WE_IO     => Q_WE_IO);
239
240     L_DIN       <= F_PM_DOUT when (D_PMS = '1') else I_DIN(7 downto 0);
241     L_INTVEC_5       <= I_INTVEC(5) and R_INT_ENA;
242     Q_ADR_IO       <= R_ADR(7 downto 0);
243
244 end Behavioral;
```

```
245
```

src/cpu_core.vhd

15 LISTING OF data_mem.vhd

```
 1    -------------------------------------------------------------------------------
 2    --
 3    -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4    --
 5    --  This code is free software: you can redistribute it and/or modify
 6    --  it under the terms of the GNU General Public License as published by
 7    --  the Free Software Foundation, either version 3 of the License, or
 8    --  (at your option) any later version.
 9    --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    data_mem - Behavioral
23    -- Create Date:    14:09:04 10/30/2009
24    -- Description:    the data mempry of a CPU.
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.STD_LOGIC_1164.ALL;
30    use IEEE.STD_LOGIC_ARITH.ALL;
31    use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33    entity data_mem is
34        port (  I_CLK       : in  std_logic;
35
36                I_ADR       : in  std_logic_vector(10 downto 0);
37                I_DIN       : in  std_logic_vector(15 downto 0);
38                I_WE        : in  std_logic_vector( 1 downto 0);
39
40                Q_DOUT      : out std_logic_vector(15 downto 0));
41    end data_mem;
42
43    architecture Behavioral of data_mem is
44
45    constant zero_256 : bit_vector := X"00000000000000000000000000000000"
46                                    & X"00000000000000000000000000000000";
47    constant nine_256 : bit_vector := X"99999999999999999999999999999999"
48                                    & X"99999999999999999999999999999999";
49
50    component RAMB4_S4_S4
51        generic(INIT_00 : bit_vector := zero_256;
52                INIT_01 : bit_vector := zero_256;
53                INIT_02 : bit_vector := zero_256;
54                INIT_03 : bit_vector := zero_256;
55                INIT_04 : bit_vector := zero_256;
56                INIT_05 : bit_vector := zero_256;
57                INIT_06 : bit_vector := zero_256;
58                INIT_07 : bit_vector := zero_256;
```

```
 59                INIT_08 : bit_vector := zero_256;
 60                INIT_09 : bit_vector := zero_256;
 61                INIT_0A : bit_vector := zero_256;
 62                INIT_0B : bit_vector := zero_256;
 63                INIT_0C : bit_vector := zero_256;
 64                INIT_0D : bit_vector := zero_256;
 65                INIT_0E : bit_vector := zero_256;
 66                INIT_0F : bit_vector := zero_256);
 67
 68        port(   DOA     : out std_logic_vector(3 downto 0);
 69                DOB     : out std_logic_vector(3 downto 0);
 70                ADDRA   : in  std_logic_vector(9 downto 0);
 71                ADDRB   : in  std_logic_vector(9 downto 0);
 72                CLKA    : in  std_ulogic;
 73                CLKB    : in  std_ulogic;
 74                DIA     : in  std_logic_vector(3 downto 0);
 75                DIB     : in  std_logic_vector(3 downto 0);
 76                ENA     : in  std_ulogic;
 77                ENB     : in  std_ulogic;
 78                RSTA    : in  std_ulogic;
 79                RSTB    : in  std_ulogic;
 80                WEA     : in  std_ulogic;
 81                WEB     : in  std_ulogic);
 82    end component;
 83
 84    signal L_ADR_0      : std_logic;
 85    signal L_ADR_E      : std_logic_vector(10 downto 1);
 86    signal L_ADR_O      : std_logic_vector(10 downto 1);
 87    signal L_DIN_E      : std_logic_vector( 7 downto 0);
 88    signal L_DIN_O      : std_logic_vector( 7 downto 0);
 89    signal L_DOUT_E     : std_logic_vector( 7 downto 0);
 90    signal L_DOUT_O     : std_logic_vector( 7 downto 0);
 91    signal L_WE_E       : std_logic;
 92    signal L_WE_O       : std_logic;
 93
 94    begin
 95
 96        sr_0 : RAMB4_S4_S4 ---------------------------------------------------------
 97        generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
 98                    INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
 99                    INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
100                    INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
101                    INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
102                    INIT_0F => nine_256)
103
104        port map(   ADDRA => L_ADR_E,               ADDRB => "0000000000",
105                    CLKA  => I_CLK,                 CLKB  => I_CLK,
106                    DIA   => L_DIN_E(3 downto 0),   DIB   => "0000",
107                    ENA   => '1',                   ENB   => '0',
108                    RSTA  => '0',                   RSTB  => '0',
109                    WEA   => L_WE_E,                WEB   => '0',
110                    DOA   => L_DOUT_E(3 downto 0),  DOB   => open);
111
112        sr_1 : RAMB4_S4_S4 ---------------------------------------------------------
113        generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
114                    INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
115                    INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
116                    INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
117                    INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
118                    INIT_0F => nine_256)
119
120        port map(   ADDRA => L_ADR_E,               ADDRB => "0000000000",
```

```
121                 CLKA  => I_CLK,                  CLKB  => I_CLK,
122                 DIA   => L_DIN_E(7 downto 4),   DIB   => "0000",
123                 ENA   => '1',                    ENB   => '0',
124                 RSTA  => '0',                    RSTB  => '0',
125                 WEA   => L_WE_E,                 WEB   => '0',
126                 DOA   => L_DOUT_E(7 downto 4),  DOB   => open);
127
128         sr_2 : RAMB4_S4_S4 ---------------------------------------------------------
129         generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
130                     INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
131                     INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
132                     INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
133                     INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
134                     INIT_0F => nine_256)
135
136         port map(   ADDRA => L_ADR_O,                ADDRB => "0000000000",
137                     CLKA  => I_CLK,                  CLKB  => I_CLK,
138                     DIA   => L_DIN_O(3 downto 0),   DIB   => "0000",
139                     ENA   => '1',                    ENB   => '0',
140                     RSTA  => '0',                    RSTB  => '0',
141                     WEA   => L_WE_O,                 WEB   => '0',
142                     DOA   => L_DOUT_O(3 downto 0),  DOB   => open);
143
144         sr_3 : RAMB4_S4_S4 ---------------------------------------------------------
145         generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
146                     INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
147                     INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
148                     INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
149                     INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
150                     INIT_0F => nine_256)
151
152         port map(   ADDRA => L_ADR_O,                ADDRB => "0000000000",
153                     CLKA  => I_CLK,                  CLKB  => I_CLK,
154                     DIA   => L_DIN_O(7 downto  4),  DIB   => "0000",
155                     ENA   => '1',                    ENB   => '0',
156                     RSTA  => '0',                    RSTB  => '0',
157                     WEA   => L_WE_O,                 WEB   => '0',
158                     DOA   => L_DOUT_O(7 downto  4), DOB   => open);
159
160
161         -- remember ADR(0)
162         --
163         adr0: process(I_CLK)
164         begin
165             if (rising_edge(I_CLK)) then
166                 L_ADR_0      <= I_ADR(0);
167             end if;
168         end process;
169
170         -- we use two memory blocks _E and _O (even and odd).
171         -- This gives us a memory with ADR and ADR + 1 at th same time.
172         -- The second port is currently unused, but may be used later,
173         -- e.g. for DMA.
174         --
175
176         L_ADR_O      <= I_ADR(10 downto 1);
177         L_ADR_E      <= I_ADR(10 downto 1) + ("000000000" & I_ADR(0));
178
179         L_DIN_E      <= I_DIN( 7 downto 0) when (I_ADR(0) = '0') else I_DIN(15 downto 8);
180         L_DIN_O      <= I_DIN( 7 downto 0) when (I_ADR(0) = '1') else I_DIN(15 downto 8);
181
182         L_WE_E      <= I_WE(1) or (I_WE(0) and not I_ADR(0));
```

```
183         L_WE_O       <= I_WE(1) or (I_WE(0) and     I_ADR(0));
184
185         Q_DOUT( 7 downto 0)       <= L_DOUT_E when (L_ADR_0 = '0') else L_DOUT_O;
186         Q_DOUT(15 downto 8)       <= L_DOUT_E when (L_ADR_0 = '1') else L_DOUT_O;
187
188     end Behavioral;
189
```

src/data_mem.vhd

16 LISTING OF data_path.vhd

```
1   -------------------------------------------------------------------------------
2   --
3   -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4   --
5   --  This code is free software: you can redistribute it and/or modify
6   --  it under the terms of the GNU General Public License as published by
7   --  the Free Software Foundation, either version 3 of the License, or
8   --  (at your option) any later version.
9   --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  -------------------------------------------------------------------------------
20  -------------------------------------------------------------------------------
21  --
22  -- Module Name:    data_path - Behavioral
23  -- Create Date:    13:24:10 10/29/2009
24  -- Description:    the data path of a CPU.
25  --
26  -------------------------------------------------------------------------------
27  --
28  library IEEE;
29  use IEEE.std_logic_1164.ALL;
30  use IEEE.std_logic_ARITH.ALL;
31  use IEEE.std_logic_UNSIGNED.ALL;
32
33  use work.common.ALL;
34
35  entity data_path is
36      port(   I_CLK          : in  std_logic;
37
38              I_ALU_OP    : in  std_logic_vector( 4 downto 0);
39              I_AMOD      : in  std_logic_vector( 5 downto 0);
40              I_BIT       : in  std_logic_vector( 3 downto 0);
41              I_DDDDD     : in  std_logic_vector( 4 downto 0);
42              I_DIN       : in  std_logic_vector( 7 downto 0);
43              I_IMM       : in  std_logic_vector(15 downto 0);
44              I_JADR      : in  std_logic_vector(15 downto 0);
45              I_OPC       : in  std_logic_vector(15 downto 0);
46              I_PC        : in  std_logic_vector(15 downto 0);
47              I_PC_OP     : in  std_logic_vector( 2 downto 0);
48              I_PMS       : in  std_logic;  -- program memory select
49              I_RD_M      : in  std_logic;
50              I_RRRRR     : in  std_logic_vector( 4 downto 0);
51              I_RSEL      : in  std_logic_vector( 1 downto 0);
52              I_WE_01     : in  std_logic;
53              I_WE_D      : in  std_logic_vector( 1 downto 0);
54              I_WE_F      : in  std_logic;
55              I_WE_M      : in  std_logic_vector( 1 downto 0);
56              I_WE_XYZS   : in  std_logic;
57
58              Q_ADR       : out std_logic_vector(15 downto 0);
```

```
 59                 Q_DOUT       : out std_logic_vector( 7 downto 0);
 60                 Q_INT_ENA    : out std_logic;
 61                 Q_LOAD_PC    : out std_logic;
 62                 Q_NEW_PC     : out std_logic_vector(15 downto 0);
 63                 Q_OPC        : out std_logic_vector(15 downto 0);
 64                 Q_PC         : out std_logic_vector(15 downto 0);
 65                 Q_RD_IO      : out std_logic;
 66                 Q_SKIP       : out std_logic;
 67                 Q_WE_IO      : out std_logic);
 68     end data_path;
 69
 70     architecture Behavioral of data_path is
 71
 72     component alu
 73         port (  I_ALU_OP     : in  std_logic_vector( 4 downto 0);
 74                 I_BIT        : in  std_logic_vector( 3 downto 0);
 75                 I_D          : in  std_logic_vector(15 downto 0);
 76                 I_D0         : in  std_logic;
 77                 I_DIN        : in  std_logic_vector( 7 downto 0);
 78                 I_FLAGS      : in  std_logic_vector( 7 downto 0);
 79                 I_IMM        : in  std_logic_vector( 7 downto 0);
 80                 I_PC         : in  std_logic_vector(15 downto 0);
 81                 I_R          : in  std_logic_vector(15 downto 0);
 82                 I_R0         : in  std_logic;
 83                 I_RSEL       : in  std_logic_vector( 1 downto 0);
 84
 85                 Q_FLAGS      : out std_logic_vector( 9 downto 0);
 86                 Q_DOUT       : out std_logic_vector(15 downto 0));
 87     end component;
 88
 89     signal A_DOUT            : std_logic_vector(15 downto 0);
 90     signal A_FLAGS           : std_logic_vector( 9 downto 0);
 91
 92     component register_file
 93         port (  I_CLK        : in  std_logic;
 94
 95                 I_AMOD       : in  std_logic_vector( 5 downto 0);
 96                 I_COND       : in  std_logic_vector( 3 downto 0);
 97                 I_DDDDD      : in  std_logic_vector( 4 downto 0);
 98                 I_DIN        : in  std_logic_vector(15 downto 0);
 99                 I_FLAGS      : in  std_logic_vector( 7 downto 0);
100                 I_IMM        : in  std_logic_vector(15 downto 0);
101                 I_RRRR       : in  std_logic_vector( 4 downto 1);
102                 I_WE_01      : in  std_logic;
103                 I_WE_D       : in  std_logic_vector( 1 downto 0);
104                 I_WE_F       : in  std_logic;
105                 I_WE_M       : in  std_logic;
106                 I_WE_XYZS    : in  std_logic;
107
108                 Q_ADR        : out std_logic_vector(15 downto 0);
109                 Q_CC         : out std_logic;
110                 Q_D          : out std_logic_vector(15 downto 0);
111                 Q_FLAGS      : out std_logic_vector( 7 downto 0);
112                 Q_R          : out std_logic_vector(15 downto 0);
113                 Q_S          : out std_logic_vector( 7 downto 0);
114                 Q_Z          : out std_logic_vector(15 downto 0));
115     end component;
116
117     signal F_ADR             : std_logic_vector(15 downto 0);
118     signal F_CC              : std_logic;
119     signal F_D               : std_logic_vector(15 downto 0);
120     signal F_FLAGS           : std_logic_vector( 7 downto 0);
```

```
121     signal F_R              : std_logic_vector(15 downto 0);
122     signal F_S              : std_logic_vector( 7 downto 0);
123     signal F_Z              : std_logic_vector(15 downto 0);
124
125     component data_mem
126         port (  I_CLK       : in  std_logic;
127
128                 I_ADR       : in  std_logic_vector(10 downto 0);
129                 I_DIN       : in  std_logic_vector(15 downto 0);
130                 I_WE        : in  std_logic_vector( 1 downto 0);
131
132                 Q_DOUT      : out std_logic_vector(15 downto 0));
133     end component;
134
135     signal M_DOUT           : std_logic_vector(15 downto 0);
136
137     signal L_DIN            : std_logic_vector( 7 downto 0);
138     signal L_WE_SRAM        : std_logic_vector( 1 downto 0);
139     signal L_FLAGS_98       : std_logic_vector( 9 downto 8);
140
141     begin
142
143         alui : alu
144         port map(   I_ALU_OP    => I_ALU_OP,
145                     I_BIT       => I_BIT,
146                     I_D         => F_D,
147                     I_D0        => I_DDDDD(0),
148                     I_DIN       => L_DIN,
149                     I_FLAGS     => F_FLAGS,
150                     I_IMM       => I_IMM(7 downto 0),
151                     I_PC        => I_PC,
152                     I_R         => F_R,
153                     I_R0        => I_RRRRR(0),
154                     I_RSEL      => I_RSEL,
155
156                     Q_FLAGS     => A_FLAGS,
157                     Q_DOUT      => A_DOUT);
158
159         regs : register_file
160         port map(   I_CLK       => I_CLK,
161
162                     I_AMOD      => I_AMOD,
163                     I_COND(3)   => I_OPC(10),
164                     I_COND(2 downto 0)=> I_OPC(2 downto 0),
165                     I_DDDDD     => I_DDDDD,
166                     I_DIN       => A_DOUT,
167                     I_FLAGS     => A_FLAGS(7 downto 0),
168                     I_IMM       => I_IMM,
169                     I_RRRR      => I_RRRRR(4 downto 1),
170                     I_WE_01     => I_WE_01,
171                     I_WE_D      => I_WE_D,
172                     I_WE_F      => I_WE_F,
173                     I_WE_M      => I_WE_M(0),
174                     I_WE_XYZS   => I_WE_XYZS,
175
176                     Q_ADR       => F_ADR,
177                     Q_CC        => F_CC,
178                     Q_D         => F_D,
179                     Q_FLAGS     => F_FLAGS,
180                     Q_R         => F_R,
181                     Q_S         => F_S,   -- Q_Rxx(F_ADR)
182                     Q_Z         => F_Z);
```

```
183
184         sram : data_mem
185         port map(   I_CLK   => I_CLK,
186
187                     I_ADR   => F_ADR(10 downto 0),
188                     I_DIN   => A_DOUT,
189                     I_WE    => L_WE_SRAM,
190
191                     Q_DOUT  => M_DOUT);
192
193         -- remember A_FLAGS(9 downto 8) (within the current instruction).
194         --
195         flg98: process(I_CLK)
196         begin
197             if (rising_edge(I_CLK)) then
198                 L_FLAGS_98      <= A_FLAGS(9 downto 8);
199             end if;
200         end process;
201
202         -- whether PC shall be loaded with NEW_PC or not.
203         -- I.e. if a branch shall be taken or not.
204         --
205         process(I_PC_OP, F_CC)
206         begin
207             case I_PC_OP is
208                 when PC_BCC  => Q_LOAD_PC      <= F_CC;      -- maybe (PC on I_JADR)
209                 when PC_LD_I => Q_LOAD_PC      <= '1';       -- yes: new PC on I_JADR
210                 when PC_LD_Z => Q_LOAD_PC      <= '1';       -- yes: new PC in Z
211                 when PC_LD_S => Q_LOAD_PC      <= '1';       -- yes: new PC on stack
212                 when others  => Q_LOAD_PC      <= '0';       -- no.
213             end case;
214         end process;
215
216         -- whether the next instruction shall be skipped or not.
217         --
218         process(I_PC_OP, L_FLAGS_98, F_CC)
219         begin
220             case I_PC_OP is
221                 when PC_BCC    => Q_SKIP      <= F_CC;            -- if cond met
222                 when PC_LD_I   => Q_SKIP      <= '1';             -- yes
223                 when PC_LD_Z   => Q_SKIP      <= '1';             -- yes
224                 when PC_LD_S   => Q_SKIP      <= '1';             -- yes
225                 when PC_SKIP_Z => Q_SKIP      <= L_FLAGS_98(8);  -- if Z set
226                 when PC_SKIP_T => Q_SKIP      <= L_FLAGS_98(9);  -- if T set
227                 when others    => Q_SKIP      <= '0';             -- no.
228             end case;
229         end process;
230
231         Q_ADR           <= F_ADR;
232         Q_DOUT          <= A_DOUT(7 downto 0);
233         Q_INT_ENA       <= A_FLAGS(7);
234         Q_OPC           <= I_OPC;
235         Q_PC            <= I_PC;
236
237         Q_RD_IO         <= '0'                        when (F_ADR <X"20")
238                 else (I_RD_M and not I_PMS) when (F_ADR       <X"5D")
239                 else '0';
240         Q_WE_IO          <= '0'                        when (F_ADR <X"20")
241                 else I_WE_M(0)              when (F_ADR       <X"5D")
242                 else '0';
243         L_WE_SRAM        <= "00"   when  (F_ADR <X"0060") else I_WE_M;
244         L_DIN            <= I_DIN  when (I_PMS = '1')
```

```
245                 else F_S    when  (F_ADR      <X"0020")
246                 else I_DIN  when  (F_ADR       <X"005D")
247                 else F_S    when  (F_ADR       <X"0060")
248                 else M_DOUT(7 downto 0);
249
250         -- compute potential new PC value from Z, (SP), or IMM.
251         --
252         Q_NEW_PC        <= F_Z    when I_PC_OP = PC_LD_Z        -- IJMP, ICALL
253                else M_DOUT when I_PC_OP = PC_LD_S       -- RET, RETI
254                else I_JADR;                             -- JMP adr
255
256     end Behavioral;
257
```

src/data_path.vhd

17 LISTING OF io.vhd

```
 1   -------------------------------------------------------------------------------
 2   --
 3   -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4   --
 5   --  This code is free software: you can redistribute it and/or modify
 6   --  it under the terms of the GNU General Public License as published by
 7   --  the Free Software Foundation, either version 3 of the License, or
 8   --  (at your option) any later version.
 9   --
10   --  This code is distributed in the hope that it will be useful,
11   --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12   --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13   --  GNU General Public License for more details.
14   --
15   --  You should have received a copy of the GNU General Public License
16   --  along with this code (see the file named COPYING).
17   --  If not, see http://www.gnu.org/licenses/.
18   --
19   -------------------------------------------------------------------------------
20   -------------------------------------------------------------------------------
21   --
22   -- Module Name:    io - Behavioral
23   -- Create Date:    13:59:36 11/07/2009
24   -- Description:    the I/O of a CPU (uart and general purpose I/O lines).
25   --
26   -------------------------------------------------------------------------------
27   --
28   library IEEE;
29   use IEEE.STD_LOGIC_1164.ALL;
30   use IEEE.STD_LOGIC_ARITH.ALL;
31   use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33   entity io is
34       port (  I_CLK       : in  std_logic;
35
36               I_CLR       : in  std_logic;
37               I_ADR_IO    : in  std_logic_vector( 7 downto 0);
38               I_DIN       : in  std_logic_vector( 7 downto 0);
39               I_SWITCH    : in  std_logic_vector( 7 downto 0);
40               I_RD_IO     : in  std_logic;
41               I_RX        : in  std_logic;
42               I_WE_IO     : in  std_logic;
43
44               Q_7_SEGMENT : out std_logic_vector( 6 downto 0);
45               Q_DOUT      : out std_logic_vector( 7 downto 0);
46               Q_INTVEC    : out std_logic_vector( 5 downto 0);
47               Q_LEDS      : out std_logic_vector( 1 downto 0);
48               Q_TX        : out std_logic);
49   end io;
50
51   architecture Behavioral of io is
52
53   component uart
54       generic(CLOCK_FREQ  : std_logic_vector(31 downto 0);
55               BAUD_RATE   : std_logic_vector(27 downto 0));
56       port(   I_CLK       : in  std_logic;
57               I_CLR       : in  std_logic;
58               I_RD        : in  std_logic;
```

```
 59                I_WE        : in  std_logic;
 60                I_RX        : in  std_logic;
 61                I_TX_DATA   : in  std_logic_vector(7 downto 0);
 62
 63                Q_RX_DATA   : out std_logic_vector(7 downto 0);
 64                Q_RX_READY  : out std_logic;
 65                Q_TX        : out std_logic;
 66                Q_TX_BUSY   : out std_logic);
 67    end component;
 68
 69    signal U_RX_READY       : std_logic;
 70    signal U_TX_BUSY        : std_logic;
 71    signal U_RX_DATA        : std_logic_vector( 7 downto 0);
 72
 73    signal L_INTVEC         : std_logic_vector( 5 downto 0);
 74    signal L_LEDS           : std_logic;
 75    signal L_RD_UART        : std_logic;
 76    signal L_RX_INT_ENABLED : std_logic;
 77    signal L_TX_INT_ENABLED : std_logic;
 78    signal L_WE_UART        : std_logic;
 79
 80    begin
 81        urt: uart
 82        generic map(CLOCK_FREQ  => std_logic_vector(conv_unsigned(25000000, 32)),
 83                    BAUD_RATE   => std_logic_vector(conv_unsigned(   38400, 28)))
 84        port map(   I_CLK      => I_CLK,
 85                    I_CLR      => I_CLR,
 86                    I_RD       => L_RD_UART,
 87                    I_WE       => L_WE_UART,
 88                    I_TX_DATA  => I_DIN(7 downto 0),
 89                    I_RX       => I_RX,
 90
 91                    Q_TX       => Q_TX,
 92                    Q_RX_DATA  => U_RX_DATA,
 93                    Q_RX_READY => U_RX_READY,
 94                    Q_TX_BUSY  => U_TX_BUSY);
 95
 96        -- IO read process
 97        --
 98        iord: process(I_ADR_IO, I_SWITCH,
 99                      U_RX_DATA, U_RX_READY, L_RX_INT_ENABLED,
100                      U_TX_BUSY, L_TX_INT_ENABLED)
101        begin
102            -- addresses for mega8 device (use iom8.h or #define __AVR_ATmega8__).
103            --
104            case I_ADR_IO is
105                when X"2A"  => Q_DOUT      <=             -- UCSRB:
106                                   L_RX_INT_ENABLED  -- Rx complete int enabled.
107                                 & L_TX_INT_ENABLED  -- Tx complete int enabled.
108                                 & L_TX_INT_ENABLED  -- Tx empty int enabled.
109                                 & '1'               -- Rx enabled
110                                 & '1'               -- Tx enabled
111                                 & '0'               -- 8 bits/char
112                                 & '0'               -- Rx bit 8
113                                 & '0';              -- Tx bit 8
114                when X"2B"  => Q_DOUT      <=             -- UCSRA:
115                                   U_RX_READY       -- Rx complete
116                                 & not U_TX_BUSY    -- Tx complete
117                                 & not U_TX_BUSY    -- Tx ready
118                                 & '0'              -- frame error
119                                 & '0'              -- data overrun
120                                 & '0'              -- parity error
```

```
121                              & '0'              -- double dpeed
122                              & '0';             -- multiproc mode
123                 when X"2C"  => Q_DOUT       <= U_RX_DATA; -- UDR
124                 when X"40"  => Q_DOUT       <=             -- UCSRC
125                                '1'              -- URSEL
126                              & '0'              -- asynchronous
127                              & "00"             -- no parity
128                              & '1'              -- two stop bits
129                              & "11"             -- 8 bits/char
130                              & '0';             -- rising clock edge
131
132                 when X"36"  => Q_DOUT       <= I_SWITCH;  -- PINB
133                 when others => Q_DOUT       <= X"AA";
134             end case;
135         end process;
136
137         -- IO write process
138         --
139         iowr: process(I_CLK)
140         begin
141             if (rising_edge(I_CLK)) then
142                 if (I_CLR = '1') then
143                     L_RX_INT_ENABLED       <= '0';
144                     L_TX_INT_ENABLED       <= '0';
145                 elsif (I_WE_IO = '1') then
146                     case I_ADR_IO is
147                         when X"38"  => Q_7_SEGMENT      <= I_DIN(6 downto 0);    -- PORTB
148                                        L_LEDS      <= not L_LEDS;
149                         when X"40"  =>  -- handled by uart
150                         when X"41"  =>  -- handled by uart
151                         when X"43"  => L_RX_INT_ENABLED      <= I_DIN(0);
152                                        L_TX_INT_ENABLED      <= I_DIN(1);
153                         when others =>
154                     end case;
155                 end if;
156             end if;
157         end process;
158
159         -- interrupt process
160         --
161         ioint: process(I_CLK)
162         begin
163             if (rising_edge(I_CLK)) then
164                 if (I_CLR = '1') then
165                     L_INTVEC      <= "000000";
166                 else
167                     if (L_RX_INT_ENABLED and U_RX_READY) = '1' then
168                         if (L_INTVEC(5) = '0') then     -- no interrupt pending
169                             L_INTVEC      <= "101011";       -- _VECTOR(11)
170                         end if;
171                     elsif (L_TX_INT_ENABLED and not U_TX_BUSY) = '1' then
172                         if (L_INTVEC(5) = '0') then     -- no interrupt pending
173                             L_INTVEC      <= "101100";       -- _VECTOR(12)
174                         end if;
175                     else                                -- no interrupt
176                         L_INTVEC      <= "000000";
177                     end if;
178                 end if;
179             end if;
180         end process;
181
182         L_WE_UART      <= I_WE_IO when (I_ADR_IO = X"2C") else '0'; -- write UART UDR
```

```
183         L_RD_UART       <= I_RD_IO when (I_ADR_IO = X"2C") else '0'; -- read  UART UDR
184
185         Q_LEDS(1)       <= L_LEDS;
186         Q_LEDS(0)       <= not L_LEDS;
187         Q_INTVEC        <= L_INTVEC;
188
189     end Behavioral;
190
```

src/io.vhd

18 Listing of opc_deco.vhd

```
 1  -------------------------------------------------------------------------------
 2  --
 3  -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4  --
 5  --  This code is free software: you can redistribute it and/or modify
 6  --  it under the terms of the GNU General Public License as published by
 7  --  the Free Software Foundation, either version 3 of the License, or
 8  --  (at your option) any later version.
 9  --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  -------------------------------------------------------------------------------
20  -------------------------------------------------------------------------------
21  --
22  -- Module Name:    opc_deco - Behavioral
23  -- Create Date:    16:05:16 10/29/2009
24  -- Description:    the opcode decoder of a CPU.
25  --
26  -------------------------------------------------------------------------------
27  --
28  library IEEE;
29  use IEEE.STD_LOGIC_1164.ALL;
30  use IEEE.STD_LOGIC_ARITH.ALL;
31  use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33  use work.common.ALL;
34
35  entity opc_deco is
36      port (  I_CLK       : in  std_logic;
37
38              I_OPC       : in  std_logic_vector(31 downto 0);
39              I_PC        : in  std_logic_vector(15 downto 0);
40              I_T0        : in  std_logic;
41
42              Q_ALU_OP    : out std_logic_vector( 4 downto 0);
43              Q_AMOD      : out std_logic_vector( 5 downto 0);
44              Q_BIT       : out std_logic_vector( 3 downto 0);
45              Q_DDDDD     : out std_logic_vector( 4 downto 0);
46              Q_IMM       : out std_logic_vector(15 downto 0);
47              Q_JADR      : out std_logic_vector(15 downto 0);
48              Q_OPC       : out std_logic_vector(15 downto 0);
49              Q_PC        : out std_logic_vector(15 downto 0);
50              Q_PC_OP     : out std_logic_vector( 2 downto 0);
51              Q_PMS       : out std_logic;  -- program memory select
52              Q_RD_M      : out std_logic;
53              Q_RRRRR     : out std_logic_vector( 4 downto 0);
54              Q_RSEL      : out std_logic_vector( 1 downto 0);
55              Q_WE_01     : out std_logic;
56              Q_WE_D      : out std_logic_vector( 1 downto 0);
57              Q_WE_F      : out std_logic;
58              Q_WE_M      : out std_logic_vector( 1 downto 0);
```

```
 59                Q_WE_XYZS    : out std_logic);
 60    end opc_deco;
 61
 62    architecture Behavioral of opc_deco is
 63
 64    begin
 65
 66        process(I_CLK)
 67        begin
 68        if (rising_edge(I_CLK)) then
 69            --
 70            -- set the most common settings as default.
 71            --
 72            Q_ALU_OP        <= ALU_D_MV_Q;
 73            Q_AMOD          <= AMOD_ABS;
 74            Q_BIT           <= I_OPC(10) & I_OPC(2 downto 0);
 75            Q_DDDDD         <= I_OPC(8 downto 4);
 76            Q_IMM           <= X"0000";
 77            Q_JADR          <= I_OPC(31 downto 16);
 78            Q_OPC           <= I_OPC(15 downto  0);
 79            Q_PC            <= I_PC;
 80            Q_PC_OP         <= PC_NEXT;
 81            Q_PMS           <= '0';
 82            Q_RD_M          <= '0';
 83            Q_RRRRR         <= I_OPC(9) & I_OPC(3 downto 0);
 84            Q_RSEL          <= RS_REG;
 85            Q_WE_D          <= "00";
 86            Q_WE_01         <= '0';
 87            Q_WE_F          <= '0';
 88            Q_WE_M          <= "00";
 89            Q_WE_XYZS       <= '0';
 90
 91            case I_OPC(15 downto 10) is
 92                when "000000" =>
 93                    case I_OPC(9 downto 8) is
 94                        when "00" =>
 95                            --
 96                            -- 0000 0000 0000 0000 - NOP
 97                            -- 0000 0000 001v vvvv - INTERRUPT
 98                            --
 99                            if (I_OPC(5)) = '1' then   -- interrupt
100                                Q_ALU_OP       <= ALU_INTR;
101                                Q_AMOD       <= AMOD_ddSP;
102                                Q_JADR       <= "0000000000" & I_OPC(4 downto 0) & "0";
103                                Q_PC_OP       <= PC_LD_I;
104                                Q_WE_F       <= '1';
105                                Q_WE_M       <= "11";
106                            end if;
107
108                        when "01" =>
109                            --
110                            -- 0000 0001 dddd rrrr - MOVW
111                            --
112                            Q_DDDDD       <= I_OPC(7 downto 4) & "0";
113                            Q_RRRRR       <= I_OPC(3 downto 0) & "0";
114                            Q_ALU_OP       <= ALU_MV_16;
115                            Q_WE_D       <= "11";
116
117                        when "10" =>
118                            --
119                            -- 0000 0010 dddd rrrr - MULS
120                            --
```

```
121                             Q_DDDDD       <= "1" & I_OPC(7 downto 4);
122                             Q_RRRRR       <= "1" & I_OPC(3 downto 0);
123                             Q_ALU_OP       <= ALU_MULT;
124                             Q_IMM(7 downto 5)       <= MULT_SS;
125                             Q_WE_01       <= '1';
126                             Q_WE_F       <= '1';
127
128                         when others =>
129                             --
130                             -- 0000 0011 0ddd 0rrr - _MULSU  SU "010"
131                             -- 0000 0011 0ddd 1rrr - FMUL    UU "100"
132                             -- 0000 0011 1ddd 0rrr - FMULS   SS "111"
133                             -- 0000 0011 1ddd 1rrr - FMULSU  SU "110"
134                             --
135                             Q_DDDDD(4 downto 3)       <= "10";    -- regs 16 to 23
136                             Q_RRRRR(4 downto 3)       <= "10";    -- regs 16 to 23
137                             Q_ALU_OP       <= ALU_MULT;
138                             if I_OPC(7) = '0' then
139                                 if I_OPC(3) = '0' then
140                                     Q_IMM(7 downto 5)       <= MULT_SU;
141                                 else
142                                     Q_IMM(7 downto 5)       <= MULT_FUU;
143                                 end if;
144                             else
145                                 if I_OPC(3) = '0' then
146                                     Q_IMM(7 downto 5)       <= MULT_FSS;
147                                 else
148                                     Q_IMM(7 downto 5)       <= MULT_FSU;
149                                 end if;
150                             end if;
151                             Q_WE_01       <= '1';
152                             Q_WE_F       <= '1';
153                     end case;
154
155                 when "000001" | "000010" =>
156                     --
157                     -- 0000 01rd dddd rrrr - CPC = SBC without Q_WE_D
158                     -- 0000 10rd dddd rrrr - SBC
159                     --
160                     Q_ALU_OP       <= ALU_SBC;
161                     Q_WE_D       <= '0' & I_OPC(11);  -- write Rd if SBC.
162                     Q_WE_F       <= '1';
163
164                 when "000011" =>
165                     --
166                     -- 0000 11rd dddd rrrr - ADD
167                     --
168                     Q_ALU_OP       <= ALU_ADD;
169                     Q_WE_D       <= "01";
170                     Q_WE_F       <= '1';
171
172                 when "000100" => -- CPSE
173                     Q_ALU_OP       <= ALU_SUB;
174                     Q_RD_M       <= I_T0;
175                     if (I_T0 = '0') then        -- second cycle.
176                         Q_PC_OP       <= PC_SKIP_Z;
177                     end if;
178
179                 when "000101" | "000110" =>
180                     --
181                     -- 0001 01rd dddd rrrr - CP = SUB without Q_WE_D
182                     -- 0000 10rd dddd rrrr - SUB
```

```
183                 --
184                 Q_ALU_OP        <= ALU_SUB;
185                 Q_WE_D        <= '0' & I_OPC(11);   -- write Rd if SUB.
186                 Q_WE_F        <= '1';
187
188             when "000111" =>
189                 --
190                 -- 0001 11rd dddd rrrr - ADC
191                 --
192                 Q_ALU_OP        <= ALU_ADC;
193                 Q_WE_D        <= "01";
194                 Q_WE_F        <= '1';
195
196             when "001000" =>
197                 --
198                 -- 0010 00rd dddd rrrr - AND
199                 --
200                 Q_ALU_OP        <= ALU_AND;
201                 Q_WE_D        <= "01";
202                 Q_WE_F        <= '1';
203
204             when "001001" =>
205                 --
206                 -- 0010 01rd dddd rrrr - EOR
207                 --
208                 Q_ALU_OP        <= ALU_EOR;
209                 Q_WE_D        <= "01";
210                 Q_WE_F        <= '1';
211
212             when "001010" => -- OR
213                 --
214                 -- 0010 10rd dddd rrrr - OR
215                 --
216                 Q_ALU_OP        <= ALU_OR;
217                 Q_WE_D        <= "01";
218                 Q_WE_F        <= '1';
219
220             when "001011" =>
221                 --
222                 -- 0010 11rd dddd rrrr - MOV
223                 --
224                 Q_ALU_OP        <= ALU_R_MV_Q;
225                 Q_WE_D        <= "01";
226
227             when "001100" | "001101" | "001110" | "001111"
228                | "010100" | "010101" | "010110" | "010111" =>
229                 --
230                 -- 0011 KKKK dddd KKKK - CPI
231                 -- 0101 KKKK dddd KKKK - SUBI
232                 --
233                 Q_ALU_OP        <= ALU_SUB;
234                 Q_IMM(7 downto 0)        <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
235                 Q_RSEL        <= RS_IMM;
236                 Q_DDDDD(4)        <= '1';     -- Rd = 16...31
237                 Q_WE_D        <= '0' & I_OPC(14);
238                 Q_WE_F        <= '1';
239
240             when "010000" | "010001" | "010010" | "010011" =>
241                 --
242                 -- 0100 KKKK dddd KKKK - SBCI
243                 --
244                 Q_ALU_OP        <= ALU_SBC;
```

```
245                 Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
246                 Q_RSEL      <= RS_IMM;
247                 Q_DDDDD(4)      <= '1';    -- Rd = 16...31
248                 Q_WE_D      <= "01";
249                 Q_WE_F      <= '1';
250
251             when "011000" | "011001" | "011010" | "011011" =>
252                 --
253                 -- 0110 KKKK dddd KKKK - ORI
254                 --
255                 Q_ALU_OP      <= ALU_OR;
256                 Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
257                 Q_RSEL      <= RS_IMM;
258                 Q_DDDDD(4)      <= '1';    -- Rd = 16...31
259                 Q_WE_D      <= "01";
260                 Q_WE_F      <= '1';
261
262             when "011100" | "011101" | "011110" | "011111" =>
263                 --
264                 -- 0111 KKKK dddd KKKK - ANDI
265                 --
266                 Q_ALU_OP      <= ALU_AND;
267                 Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
268                 Q_RSEL      <= RS_IMM;
269                 Q_DDDDD(4)      <= '1';    -- Rd = 16...31
270                 Q_WE_D      <= "01";
271                 Q_WE_F      <= '1';
272
273             when "100000" | "100001" | "100010" | "100011"
274                | "101000" | "101001" | "101010" | "101011" =>
275                 --
276                 -- LDD (Y + q) == LD (y) if q == 0
277                 --
278                 -- 10q0 qq0d dddd 1qqq  LDD (Y + q)
279                 -- 10q0 qq0d dddd 0qqq  LDD (Z + q)
280                 -- 10q0 qq1d dddd 1qqq  SDD (Y + q)
281                 -- 10q0 qq1d dddd 0qqq  SDD (Z + q)
282                 --        L/      Z/
283                 --        S       Y
284                 --
285                 Q_IMM(5)      <= I_OPC(13);
286                 Q_IMM(4 downto 3)      <= I_OPC(11 downto 10);
287                 Q_IMM(2 downto 0)      <= I_OPC( 2 downto  0);
288
289                 if (I_OPC(3) = '0') then    Q_AMOD      <= AMOD_Zq;
290                 else                        Q_AMOD      <= AMOD_Yq;
291                 end if;
292
293                 Q_RD_M      <= not I_OPC(9);             -- '1'  if LDD
294                 Q_WE_M      <= '0' & I_OPC(9);           -- "01" if STD
295
296             when "100100" =>
297                 Q_IMM      <= I_OPC(31 downto 16);   -- absolute address for LDS/STS
298                 if (I_OPC(9) = '0') then        -- LDD / POP
299                     --
300                     -- 1001 00-0d dddd 0000 - LDS
301                     -- 1001 00-0d dddd 0001 - LD Rd, Z+
302                     -- 1001 00-0d dddd 0010 - LD Rd, -Z
303                     -- 1001 00-0d dddd 0100 - (ii)  LPM Rd, (Z)
304                     -- 1001 00-0d dddd 0101 - (iii) LPM Rd, (Z+)
305                     -- 1001 00-0d dddd 0110 - ELPM Z        --- not mega8
306                     -- 1001 00-0d dddd 0111 - ELPM Z+       --- not mega8
```

```
307                     -- 1001 00-0d dddd 1001 - LD Rd, Y+
308                     -- 1001 00-0d dddd 1010 - LD Rd, -Y
309                     -- 1001 00-0d dddd 1100 - LD Rd, X
310                     -- 1001 00-0d dddd 1101 - LD Rd, X+
311                     -- 1001 00-0d dddd 1110 - LD Rd, -X
312                     -- 1001 00-0d dddd 1111 - POP Rd
313                     --
314                     Q_RSEL      <= RS_DIN;
315                     Q_RD_M      <= I_T0;
316                     Q_WE_D      <= '0' & not I_T0;
317                     Q_WE_XYZS      <= not I_T0;
318                     Q_PMS      <= (not I_OPC(3)) and I_OPC(2) and (not I_OPC(1));
319                     case I_OPC(3 downto 0) is
320                         when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
321                         when "0001" => Q_AMOD      <= AMOD_Zi;
322                         when "0100" => Q_AMOD      <= AMOD_Z;    Q_WE_XYZS <= '0';
323                         when "0101" => Q_AMOD      <= AMOD_Zi;
324                         when "1001" => Q_AMOD      <= AMOD_Yi;
325                         when "1010" => Q_AMOD      <= AMOD_dY;
326                         when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
327                         when "1101" => Q_AMOD      <= AMOD_Xi;
328                         when "1110" => Q_AMOD      <= AMOD_dX;
329                         when "1111" => Q_AMOD      <= AMOD_SPi;
330                         when others =>                        Q_WE_XYZS      <= '0';
331                     end case;
332                 else                        -- STD / PUSH
333                     --
334                     -- 1001 00-1r rrrr 0000 - STS
335                     -- 1001 00-1r rrrr 0001 - ST Z+. Rr
336                     -- 1001 00-1r rrrr 0010 - ST -Z. Rr
337                     -- 1001 00-1r rrrr 1000 - ST Y. Rr
338                     -- 1001 00-1r rrrr 1001 - ST Y+. Rr
339                     -- 1001 00-1r rrrr 1010 - ST -Y. Rr
340                     -- 1001 00-1r rrrr 1100 - ST X. Rr
341                     -- 1001 00-1r rrrr 1101 - ST X+. Rr
342                     -- 1001 00-1r rrrr 1110 - ST -X. Rr
343                     -- 1001 00-1r rrrr 1111 - PUSH Rr
344                     --
345                     Q_ALU_OP      <= ALU_D_MV_Q;
346                     Q_WE_M      <= "01";
347                     Q_WE_XYZS      <= '1';
348                     case I_OPC(3 downto 0) is
349                         when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
350                         when "0001" => Q_AMOD      <= AMOD_Zi;
351                         when "0010" => Q_AMOD      <= AMOD_dZ;
352                         when "1001" => Q_AMOD      <= AMOD_Yi;
353                         when "1010" => Q_AMOD      <= AMOD_dY;
354                         when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
355                         when "1101" => Q_AMOD      <= AMOD_Xi;
356                         when "1110" => Q_AMOD      <= AMOD_dX;
357                         when "1111" => Q_AMOD      <= AMOD_dSP;
358                         when others =>
359                     end case;
360                 end if;
361
362             when "100101" =>
363                 if (I_OPC(9) = '0') then
364                     if (I_OPC(3) = '0') then
365                         --
366                         --  1001 010d dddd 0000 - COM
367                         --  1001 010d dddd 0001 - NEG
368                         --  1001 010d dddd 0010 - SWAP
```

```
369                             --  1001 010d dddd 0011 - INC
370                             --  1001 010d dddd 0101 - ASR
371                             --  1001 010d dddd 0110 - LSR
372                             --  1001 010d dddd 0111 - ROR
373                             --
374                             case I_OPC(2 downto 0) is
375                                 when "000"  => Q_ALU_OP      <= ALU_COM;
376                                 when "001"  => Q_ALU_OP      <= ALU_NEG;
377                                 when "010"  => Q_ALU_OP      <= ALU_SWAP;
378                                 when "011"  => Q_ALU_OP      <= ALU_INC;
379                                 when "101"  => Q_ALU_OP      <= ALU_ASR;
380                                 when "110"  => Q_ALU_OP      <= ALU_LSR;
381                                 when "111"  => Q_ALU_OP      <= ALU_ROR;
382                                 when others =>
383                             end case;
384                             Q_WE_D      <= "01";
385                             Q_WE_F      <= '1';
386                         else
387                             case I_OPC(2 downto 0) is
388                                 when "000"  =>
389                                     if (I_OPC(8)) = '0' then
390                                         --
391                                         --  1001 0100 0sss 1000 - BSET
392                                         --  1001 0100 1sss 1000 - BCLR
393                                         --
394                                         Q_BIT(3 downto 0)      <= I_OPC(7 downto 4);
395                                         Q_ALU_OP      <= ALU_SREG;
396                                         Q_WE_F      <= '1';
397                                     else
398                                         --
399                                         --  1001 0101 0000 1000 - RET
400                                         --  1001 0101 0001 1000 - RETI
401                                         --  1001 0101 1000 1000 - SLEEP
402                                         --  1001 0101 1001 1000 - BREAK
403                                         --  1001 0101 1100 1000 - LPM     [ R0,(Z) ]
404                                         --  1001 0101 1101 1000 - ELPM   not mega8
405                                         --  1001 0101 1110 1000 - SPM
406                                         --  1001 0101 1111 1000 - SPM #2
407                                         --  1001 0101 1010 1000 - WDR
408                                         --
409                                         case I_OPC(7 downto 4) is
410                                             when "0000" =>  -- RET
411                                                 Q_AMOD      <= AMOD_SPii;
412                                                 if (I_T0 = '1') then
413                                                     Q_RD_M      <= '1';
414                                                 else
415                                                     Q_PC_OP      <= PC_LD_S;
416                                                     Q_WE_XYZS      <= not I_T0;
417                                                 end if;
418
419                                             when "0001" =>  -- RETI
420                                                 Q_ALU_OP      <= ALU_INTR;
421                                                 Q_IMM(6)      <= '1';
422                                                 Q_AMOD      <= AMOD_SPii;
423                                                 if (I_T0 = '1') then
424                                                     Q_RD_M      <= '1';
425                                                 else
426                                                     Q_PC_OP      <= PC_LD_S;
427                                                     Q_WE_XYZS      <= not I_T0;
428                                                 end if;
429
430                                             when "1000" =>  -- (i) LPM R0, (Z)
```

```
431                                     Q_DDDDD      <= "00000";
432                                     Q_AMOD       <= AMOD_Z;
433                                     Q_PMS        <= '1';
434                                     Q_WE_D       <= '0' & not I_T0;
435
436                                 when "1110" =>  -- SPM
437                                     Q_DDDDD      <= "00000";
438                                     Q_AMOD       <= AMOD_Z;
439                                     Q_PMS        <= '1';
440                                     Q_WE_M       <= "01";
441
442                                 when "1111" =>  -- SPM #2
443                                     -- page write: not su[pported
444
445                                 when others =>
446                             end case;
447                         end if;
448
449                     when "001"  =>
450                         --
451                         --  1001 0100 0000 1001 IJMP
452                         --  1001 0100 0001 1001 EIJMP   -- not mega8
453                         --  1001 0101 0000 1001 ICALL
454                         --  1001 0101 0001 1001 EICALL   -- not mega8
455                         --
456                         Q_PC_OP      <= PC_LD_Z;
457                         if (I_OPC(8) = '1') then        -- ICALL
458                             Q_ALU_OP      <= ALU_PC_1;
459                             Q_AMOD       <= AMOD_ddSP;
460                             Q_WE_M       <= "11";
461                             Q_WE_XYZS      <= '1';
462                         end if;
463
464                     when "010"  =>
465                         --
466                         --  1001 010d dddd 1010 - DEC
467                         --
468                         Q_ALU_OP      <= ALU_DEC;
469                         Q_WE_D       <= "01";
470                         Q_WE_F       <= '1';
471
472                     when "011"  =>
473                         --
474                         --  1001 0100 KKKK 1011 - DES   -- not mega8
475                         --
476
477                     when "100" | "101"  =>
478                         --
479                         --  1001 010k kkkk 110k - JMP (k = 0 for 16 bit)
480                         --  kkkk kkkk kkkk kkkk
481                         --
482                         Q_PC_OP      <= PC_LD_I;
483
484                     when "110" | "111"  =>
485                         --
486                         --  1001 010k kkkk 111k - CALL (k = 0)
487                         --  kkkk kkkk kkkk kkkk
488                         --
489                         Q_ALU_OP      <= ALU_PC_2;
490                         Q_AMOD       <= AMOD_ddSP;
491                         Q_PC_OP      <= PC_LD_I;
492                         Q_WE_M       <= "11";     -- both PC bytes
```

```
493                             Q_WE_XYZS       <= '1';
494
495                         when others =>
496                     end case;
497                 end if;
498             else
499                 --
500                 --  1001 0110 KKdd KKKK - ADIW
501                 --  1001 0111 KKdd KKKK - SBIW
502                 --
503                 if (I_OPC(8) = '0') then    Q_ALU_OP       <= ALU_ADIW;
504                 else                        Q_ALU_OP       <= ALU_SBIW;
505                 end if;
506                 Q_IMM(5 downto 4)       <= I_OPC(7 downto 6);
507                 Q_IMM(3 downto 0)       <= I_OPC(3 downto 0);
508                 Q_RSEL       <= RS_IMM;
509                 Q_DDDDD       <= "11" & I_OPC(5 downto 4) & "0";
510
511                 Q_WE_D       <= "11";
512                 Q_WE_F       <= '1';
513             end if; -- I_OPC(9) = 0/1
514
515         when "100110" =>
516             --
517             --  1001 1000 AAAA Abbb - CBI
518             --  1001 1001 AAAA Abbb - SBIC
519             --  1001 1010 AAAA Abbb - SBI
520             --  1001 1011 AAAA Abbb - SBIS
521             --
522             Q_ALU_OP       <= ALU_BIT_CS;
523             Q_AMOD       <= AMOD_ABS;
524             Q_BIT(3)       <= I_OPC(9);   -- set/clear
525
526             -- IMM = AAAAAA + 0x20
527             --
528             Q_IMM(4 downto 0)       <= I_OPC(7 downto 3);
529             Q_IMM(6 downto 5)       <= "01";
530
531             Q_RD_M       <= I_T0;
532             if ((I_OPC(8) = '0') ) then      -- CBI or SBI
533                 Q_WE_M(0)       <= '1';
534             else                            -- SBIC or SBIS
535                 if (I_T0 = '0') then        -- second cycle.
536                     Q_PC_OP       <= PC_SKIP_T;
537                 end if;
538             end if;
539
540         when "100111" => -- MUL
541             --
542             --  1001 11rd dddd rrrr - MUL
543             --
544              Q_ALU_OP       <= ALU_MULT;
545              Q_IMM(7 downto 5)       <= "000"; --  -MUL UU;
546              Q_WE_01       <= '1';
547              Q_WE_F       <= '1';
548
549         when "101100" | "101101" =>
550             --
551             -- 1011 0AAd dddd AAAA - IN
552             --
553             Q_RSEL       <= RS_DIN;
554             Q_AMOD       <= AMOD_ABS;
```

```
555
556                         -- IMM = AAAAAA
557                         --     + 010000 (0x20)
558                         Q_IMM(3 downto 0)      <= I_OPC(3 downto 0);
559                         Q_IMM(4)      <= I_OPC(9);
560                         Q_IMM(6 downto 5)      <= "01" + ('0' & I_OPC(10 downto 10));
561
562                         Q_WE_D      <= "01";
563
564                     when "101110" | "101111" =>
565                         --
566                         -- 1011 1AAr rrrr AAAA - OUT
567                         --
568                         Q_ALU_OP      <= ALU_D_MV_Q;
569                         Q_AMOD      <= AMOD_ABS;
570
571                         -- IMM = AAAAAA
572                         --     + 010000 (0x20)
573                         --
574                         Q_IMM(3 downto 0)      <= I_OPC(3 downto 0);
575                         Q_IMM(4)      <= I_OPC(9);
576                         Q_IMM(6 downto 5)      <= "01" + ('0' & I_OPC(10 downto 10));
577                         Q_WE_M      <= "01";
578
579                     when "110000" | "110001" | "110010" | "110011" =>
580                         --
581                         -- 1100 kkkk kkkk kkkk - RJMP
582                         --
583                         Q_JADR      <= I_PC + (I_OPC(11) & I_OPC(11) & I_OPC(11) & I_OPC(11)
584                                         & I_OPC(11 downto 0)) + X"0001";
585                         Q_PC_OP      <= PC_LD_I;
586
587                     when "110100" | "110101" | "110110" | "110111" =>
588                         --
589                         -- 1101 kkkk kkkk kkkk - RCALL
590                         --
591                         Q_JADR      <= I_PC + (I_OPC(11) & I_OPC(11) & I_OPC(11) & I_OPC(11)
592                                         & I_OPC(11 downto 0)) + X"0001";
593                         Q_ALU_OP      <= ALU_PC_1;
594                         Q_AMOD      <= AMOD_ddSP;
595                         Q_PC_OP      <= PC_LD_I;
596                         Q_WE_M      <= "11";     -- both PC bytes
597                         Q_WE_XYZS      <= '1';
598
599                     when "111000" | "111001" | "111010" | "111011" => -- LDI
600                         --
601                         -- 1110 KKKK dddd KKKK - LDI Rd, K
602                         --
603                         Q_ALU_OP      <= ALU_R_MV_Q;
604                         Q_RSEL      <= RS_IMM;
605                         Q_DDDDD      <= '1' & I_OPC(7 downto 4);     -- 16..31
606                         Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
607                         Q_WE_D      <= "01";
608
609                     when "111100" | "111101" =>
610                         --
611                         -- 1111 00kk kkkk kbbb - BRBS
612                         -- 1111 01kk kkkk kbbb - BRBC
613                         --       v
614                         -- bbb: status register bit
615                         -- v: value (set/cleared) of status register bit
616                         --
```

```
617                     Q_JADR      <= I_PC + (I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
618                                        & I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
619                                        & I_OPC(9) & I_OPC(9 downto 3)) + X"0001";
620                     Q_PC_OP      <= PC_BCC;
621
622                 when "111110" =>
623                     --
624                     -- 1111 100d dddd 0bbb - BLD
625                     -- 1111 101d dddd 0bbb - BST
626                     --
627                     if I_OPC(9) = '0' then  -- BLD: T flag to register
628                         Q_ALU_OP      <= ALU_BLD;
629                         Q_WE_D      <= "01";
630                     else                    -- BST: register to T flag
631                         Q_AMOD      <= AMOD_ABS;
632                         Q_BIT(3)      <= I_OPC(10);
633                         Q_IMM(4 downto 0)      <= I_OPC(8 downto 4);
634                         Q_ALU_OP      <= ALU_BIT_CS;
635                         Q_WE_F      <= '1';
636                     end if;
637
638                 when "111111" =>
639                     --
640                     -- 1111 110r rrrr 0bbb - SBRC
641                     -- 1111 111r rrrr 0bbb - SBRS
642                     --
643                     -- like SBIC, but and general purpose regs instead of I/O regs.
644                     --
645                     Q_ALU_OP      <= ALU_BIT_CS;
646                     Q_AMOD      <= AMOD_ABS;
647                     Q_BIT(3)      <= I_OPC(9);   -- set/clear bit
648                     Q_IMM(4 downto 0)      <= I_OPC(8 downto 4);
649                     if (I_T0 = '0') then
650                         Q_PC_OP      <= PC_SKIP_T;
651                     end if;
652
653                 when others =>
654             end case;
655         end if;
656         end process;
657
658     end Behavioral;
659
```

src/opc_deco.vhd

19 LISTING OF opc_fetch.vhd

```
 1    -------------------------------------------------------------------------------
 2    --
 3    -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4    --
 5    --  This code is free software: you can redistribute it and/or modify
 6    --  it under the terms of the GNU General Public License as published by
 7    --  the Free Software Foundation, either version 3 of the License, or
 8    --  (at your option) any later version.
 9    --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    opc_fetch - Behavioral
23    -- Create Date:    13:00:44 10/30/2009
24    -- Description:    the opcode fetch stage of a CPU.
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.std_logic_1164.ALL;
30    use IEEE.std_logic_ARITH.ALL;
31    use IEEE.std_logic_UNSIGNED.ALL;
32
33    entity opc_fetch is
34        port (  I_CLK       : in  std_logic;
35
36                I_CLR       : in  std_logic;
37                I_INTVEC    : in  std_logic_vector( 5 downto 0);
38                I_LOAD_PC   : in  std_logic;
39                I_NEW_PC    : in  std_logic_vector(15 downto 0);
40                I_PM_ADR    : in  std_logic_vector(11 downto 0);
41                I_SKIP      : in  std_logic;
42
43                Q_OPC       : out std_logic_vector(31 downto 0);
44                Q_PC        : out std_logic_vector(15 downto 0);
45                Q_PM_DOUT   : out std_logic_vector( 7 downto 0);
46                Q_T0        : out std_logic);
47    end opc_fetch;
48
49    architecture Behavioral of opc_fetch is
50
51    component prog_mem
52        port (  I_CLK       : in  std_logic;
53
54                I_WAIT      : in  std_logic;
55                I_PC        : in  std_logic_vector (15 downto 0);
56                I_PM_ADR    : in  std_logic_vector (11 downto 0);
57
58                Q_OPC       : out std_logic_vector (31 downto 0);
```

```
 59                 Q_PC         : out std_logic_vector (15 downto 0);
 60                 Q_PM_DOUT    : out std_logic_vector ( 7 downto 0));
 61     end component;
 62
 63     signal P_OPC             : std_logic_vector(31 downto 0);
 64     signal P_PC              : std_logic_vector(15 downto 0);
 65
 66     signal L_INVALIDATE      : std_logic;
 67     signal L_LONG_OP         : std_logic;
 68     signal L_NEXT_PC         : std_logic_vector(15 downto 0);
 69     signal L_PC              : std_logic_vector(15 downto 0);
 70     signal L_T0              : std_logic;
 71     signal L_WAIT            : std_logic;
 72
 73     begin
 74
 75         pmem : prog_mem
 76         port map(   I_CLK       => I_CLK,
 77
 78                     I_WAIT      => L_WAIT,
 79                     I_PC        => L_NEXT_PC,
 80                     I_PM_ADR    => I_PM_ADR,
 81
 82                     Q_OPC       => P_OPC,
 83                     Q_PC        => P_PC,
 84                     Q_PM_DOUT   => Q_PM_DOUT);
 85
 86        lpc: process(I_CLK)
 87         begin
 88             if (rising_edge(I_CLK)) then
 89                 L_PC      <= L_NEXT_PC;
 90                 L_T0      <= not L_WAIT;
 91             end if;
 92         end process;
 93
 94         L_NEXT_PC      <= X"0000"        when (I_CLR     = '1')
 95                 else L_PC           when (L_WAIT    = '1')
 96                 else I_NEW_PC       when (I_LOAD_PC = '1')
 97                 else L_PC + X"0002" when (L_LONG_OP = '1')
 98                 else L_PC + X"0001";
 99
100         -- Two word opcodes:
101         --
102         --        9       3210
103         -- 1001 000d dddd 0000 kkkk kkkk kkkk kkkk - LDS
104         -- 1001 001d dddd 0000 kkkk kkkk kkkk kkkk - SDS
105         -- 1001 010k kkkk 110k kkkk kkkk kkkk kkkk - JMP
106         -- 1001 010k kkkk 111k kkkk kkkk kkkk kkkk - CALL
107         --
108         L_LONG_OP      <= '1' when (((P_OPC(15 downto  9) = "1001010") and
109                                      (P_OPC( 3 downto  2) = "11"))       -- JMP, CALL
110                                 or  ((P_OPC(15 downto 10) = "100100") and
111                                      (P_OPC( 3 downto  0) = "0000")))    -- LDS, STS
112                 else '0';
113
114         -- Two cycle opcodes:
115         --
116         -- 1001 000d dddd .... - LDS etc.
117         -- 1001 0101 0000 1000 - RET
118         -- 1001 0101 0001 1000 - RETI
119         -- 1001 1001 AAAA Abbb - SBIC
120         -- 1001 1011 AAAA Abbb - SBIS
```

```
121         -- 1111 110r rrrr 0bbb - SBRC
122         -- 1111 111r rrrr 0bbb - SBRS
123         --
124         L_WAIT      <= '0'  when (L_INVALIDATE = '1')
125              else '0'  when (I_INTVEC(5)  = '1')
126              else L_T0 when ((P_OPC(15 downto   9) = "1001000" )    -- LDS etc.
127                          or  (P_OPC(15 downto   8) = "10010101")    -- RET etc.
128                          or  ((P_OPC(15 downto 10) = "100110")      -- SBIC, SBIS
129                            and P_OPC(8) = '1')
130                          or  (P_OPC(15 downto  10) = "111111"))     -- SBRC, SBRS
131             else  '0';
132
133         L_INVALIDATE      <= I_CLR or I_SKIP;
134
135         Q_OPC      <= X"00000000" when (L_INVALIDATE = '1')
136             else P_OPC       when (I_INTVEC(5) = '0')
137             else (X"000000" & "00" & I_INTVEC);     -- "interrupt opcode"
138
139         Q_PC      <= P_PC;
140         Q_T0      <= L_T0;
141
142     end Behavioral;
143
```

src/opc_fetch.vhd

20 LISTING OF prog_mem_content.vhd

```
 1
 2    library IEEE;
 3    use IEEE.STD_LOGIC_1164.all;
 4
 5    package prog_mem_content is
 6
 7    -- content of pe_0 -------------------------------------------------------------------------
 8    constant pe_0_00 : BIT_VECTOR := X"F180C8135798FC118181E0CA1010905100EF1D2C5F8CCCCCCCCCCC
 9    constant pe_0_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFF3E5864BEFFF1194EECFF8514CE38
10    constant pe_0_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
11    constant pe_0_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
12    constant pe_0_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
13    constant pe_0_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
14    constant pe_0_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
15    constant pe_0_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
16    constant pe_0_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
17    constant pe_0_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
18    constant pe_0_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
19    constant pe_0_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
20    constant pe_0_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
21    constant pe_0_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
22    constant pe_0_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
23    constant pe_0_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
24
25    -- content of pe_1 -------------------------------------------------------------------------
26    constant pe_1_00 : BIT_VECTOR := X"3F930800000088842808F0CBEA0ADA0FB1DC10726440000000000
27    constant pe_1_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFAC9CBC0DC0C0DC8F0ED109C2FF00
28    constant pe_1_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
29    constant pe_1_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
30    constant pe_1_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
31    constant pe_1_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
32    constant pe_1_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
33    constant pe_1_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
34    constant pe_1_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
35    constant pe_1_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
36    constant pe_1_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
37    constant pe_1_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
38    constant pe_1_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
39    constant pe_1_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
40    constant pe_1_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
41    constant pe_1_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
42
43    -- content of pe_2 -------------------------------------------------------------------------
44    constant pe_2_00 : BIT_VECTOR := X"F7100B00000044404050F0007606760000F54AC0C0544444444444
45    constant pe_2_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFF0101B55F5117B71141335B711F05
46    constant pe_2_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
47    constant pe_2_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
48    constant pe_2_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
49    constant pe_2_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
50    constant pe_2_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
51    constant pe_2_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
52    constant pe_2_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
53    constant pe_2_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
54    constant pe_2_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
55    constant pe_2_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
56    constant pe_2_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
57    constant pe_2_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
58    constant pe_2_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
 59    constant pe_2_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 60
 61    -- content of pe_3 ----------------------------------------------------------------------
 62    constant pe_3_00 : BIT_VECTOR := X"4FEECBCCCCCC33FFFF9EC000F3CEF39EEEBE20626249999999999 9
 63    constant pe_3_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFC00009EEBE9990F90909990F90CC9
 64    constant pe_3_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 65    constant pe_3_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 66    constant pe_3_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 67    constant pe_3_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 68    constant pe_3_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 69    constant pe_3_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 70    constant pe_3_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 71    constant pe_3_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 72    constant pe_3_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 73    constant pe_3_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 74    constant pe_3_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 75    constant pe_3_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 76    constant pe_3_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 77    constant pe_3_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 78
 79    -- content of po_0 ----------------------------------------------------------------------
 80    constant po_0_00 : BIT_VECTOR := X"3F1A0560796F99750980CDCCE1D001D2A0D4F047C0C33333333333
 81    constant po_0_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFF6EAE400D48FFE081B2CFF1421CDC
 82    constant po_0_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 83    constant po_0_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 84    constant po_0_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 85    constant po_0_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 86    constant po_0_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 87    constant po_0_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 88    constant po_0_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 89    constant po_0_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 90    constant po_0_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 91    constant po_0_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 92    constant po_0_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 93    constant po_0_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 94    constant po_0_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 95    constant po_0_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 96
 97    -- content of po_1 ----------------------------------------------------------------------
 98    constant po_1_00 : BIT_VECTOR := X"82082B8888885408888985000B1B1B00EACD10656045555555555 5
 99    constant po_1_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFF0F00D1CD01DCC82508C0082025A
100    constant po_1_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
101    constant po_1_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
102    constant po_1_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
103    constant po_1_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
104    constant po_1_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
105    constant po_1_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
106    constant po_1_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
107    constant po_1_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
108    constant po_1_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
109    constant po_1_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
110    constant po_1_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
111    constant po_1_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
112    constant po_1_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
113    constant po_1_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
114
115    -- content of po_2 ----------------------------------------------------------------------
116    constant po_2_00 : BIT_VECTOR := X"0F7609BCC88F400424409B444720072096F0E01FF8C00000000000
117    constant po_2_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFF4F4000F05111B36001337B369B1
118    constant po_2_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
119    constant po_2_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
120    constant po_2_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
121     constant po_2_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
122     constant po_2_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
123     constant po_2_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
124     constant po_2_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
125     constant po_2_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
126     constant po_2_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
127     constant po_2_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
128     constant po_2_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
129     constant po_2_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
130     constant po_2_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
131     constant po_2_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
132
133     -- content of po_3 ---------------------------------------------------------------------
134     constant po_3_00 : BIT_VECTOR := X"E59EE9EEEEEEFFC3333EB999909EE09CEEBEB02664400000000000
135     constant po_3_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFC9C9CEEBE99901290C0999129B90
136     constant po_3_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
137     constant po_3_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
138     constant po_3_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
139     constant po_3_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
140     constant po_3_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
141     constant po_3_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
142     constant po_3_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
143     constant po_3_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
144     constant po_3_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
145     constant po_3_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
146     constant po_3_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
147     constant po_3_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
148     constant po_3_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
149     constant po_3_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
150
151     end prog_mem_content;
152
```

src/prog_mem_content.vhd

21 LISTING OF prog_mem.vhd

```
1    -------------------------------------------------------------------------------
2    --
3    -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4    --
5    --  This code is free software: you can redistribute it and/or modify
6    --  it under the terms of the GNU General Public License as published by
7    --  the Free Software Foundation, either version 3 of the License, or
8    --  (at your option) any later version.
9    --
10   --  This code is distributed in the hope that it will be useful,
11   --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12   --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13   --  GNU General Public License for more details.
14   --
15   --  You should have received a copy of the GNU General Public License
16   --  along with this code (see the file named COPYING).
17   --  If not, see http://www.gnu.org/licenses/.
18   --
19   -------------------------------------------------------------------------------
20   -------------------------------------------------------------------------------
21   --
22   -- Module Name:    prog_mem - Behavioral
23   -- Create Date:    14:09:04 10/30/2009
24   -- Description:    the program memory of a CPU.
25   --
26   ----------------------------------------------------------------------------------
27   library IEEE;
28   use IEEE.STD_LOGIC_1164.ALL;
29   use IEEE.STD_LOGIC_ARITH.ALL;
30   use IEEE.STD_LOGIC_UNSIGNED.ALL;
31
32   -- the content of the program memory.
33   --
34   use work.prog_mem_content.all;
35
36   entity prog_mem is
37       port (  I_CLK       : in  std_logic;
38
39               I_WAIT      : in  std_logic;
40               I_PC        : in  std_logic_vector(15 downto 0); -- word address
41               I_PM_ADR    : in  std_logic_vector(11 downto 0); -- byte address
42
43               Q_OPC       : out std_logic_vector(31 downto 0);
44               Q_PC        : out std_logic_vector(15 downto 0);
45               Q_PM_DOUT   : out std_logic_vector( 7 downto 0));
46   end prog_mem;
47
48   architecture Behavioral of prog_mem is
49
50   constant zero_256 : bit_vector := X"00000000000000000000000000000000"
51                                   & X"00000000000000000000000000000000";
52
53   component RAMB4_S4_S4
54       generic(INIT_00 : bit_vector := zero_256;
55               INIT_01 : bit_vector := zero_256;
56               INIT_02 : bit_vector := zero_256;
57               INIT_03 : bit_vector := zero_256;
58               INIT_04 : bit_vector := zero_256;
```

```
 59                 INIT_05 : bit_vector := zero_256;
 60                 INIT_06 : bit_vector := zero_256;
 61                 INIT_07 : bit_vector := zero_256;
 62                 INIT_08 : bit_vector := zero_256;
 63                 INIT_09 : bit_vector := zero_256;
 64                 INIT_0A : bit_vector := zero_256;
 65                 INIT_0B : bit_vector := zero_256;
 66                 INIT_0C : bit_vector := zero_256;
 67                 INIT_0D : bit_vector := zero_256;
 68                 INIT_0E : bit_vector := zero_256;
 69                 INIT_0F : bit_vector := zero_256);
 70
 71         port(   ADDRA   : in  std_logic_vector(9 downto 0);
 72                 ADDRB   : in  std_logic_vector(9 downto 0);
 73                 CLKA    : in  std_ulogic;
 74                 CLKB    : in  std_ulogic;
 75                 DIA     : in  std_logic_vector(3 downto 0);
 76                 DIB     : in  std_logic_vector(3 downto 0);
 77                 ENA     : in  std_ulogic;
 78                 ENB     : in  std_ulogic;
 79                 RSTA    : in  std_ulogic;
 80                 RSTB    : in  std_ulogic;
 81                 WEA     : in  std_ulogic;
 82                 WEB     : in  std_ulogic;
 83
 84                 DOA     : out std_logic_vector(3 downto 0);
 85                 DOB     : out std_logic_vector(3 downto 0));
 86     end component;
 87
 88     signal M_OPC_E      : std_logic_vector(15 downto 0);
 89     signal M_OPC_O      : std_logic_vector(15 downto 0);
 90     signal M_PMD_E      : std_logic_vector(15 downto 0);
 91     signal M_PMD_O      : std_logic_vector(15 downto 0);
 92
 93     signal L_WAIT_N     : std_logic;
 94     signal L_PC_0       : std_logic;
 95     signal L_PC_E       : std_logic_vector(10 downto 1);
 96     signal L_PC_O       : std_logic_vector(10 downto 1);
 97     signal L_PMD        : std_logic_vector(15 downto 0);
 98     signal L_PM_ADR_1_0 : std_logic_vector( 1 downto 0);
 99
100     begin
101
102         pe_0 : RAMB4_S4_S4 ---------------------------------------------------------
103         generic map(INIT_00 => pe_0_00, INIT_01 => pe_0_01, INIT_02 => pe_0_02,
104                     INIT_03 => pe_0_03, INIT_04 => pe_0_04, INIT_05 => pe_0_05,
105                     INIT_06 => pe_0_06, INIT_07 => pe_0_07, INIT_08 => pe_0_08,
106                     INIT_09 => pe_0_09, INIT_0A => pe_0_0A, INIT_0B => pe_0_0B,
107                     INIT_0C => pe_0_0C, INIT_0D => pe_0_0D, INIT_0E => pe_0_0E,
108                     INIT_0F => pe_0_0F)
109         port map(ADDRA => L_PC_E,                 ADDRB => I_PM_ADR(11 downto 2),
110                  CLKA  => I_CLK,                  CLKB  => I_CLK,
111                  DIA   => "0000",                 DIB   => "0000",
112                  ENA   => L_WAIT_N,               ENB   => '1',
113                  RSTA  => '0',                    RSTB  => '0',
114                  WEA   => '0',                    WEB   => '0',
115                  DOA   => M_OPC_E(3 downto 0),    DOB   => M_PMD_E(3 downto 0));
116
117         pe_1 : RAMB4_S4_S4 ---------------------------------------------------------
118         generic map(INIT_00 => pe_1_00, INIT_01 => pe_1_01, INIT_02 => pe_1_02,
119                     INIT_03 => pe_1_03, INIT_04 => pe_1_04, INIT_05 => pe_1_05,
120                     INIT_06 => pe_1_06, INIT_07 => pe_1_07, INIT_08 => pe_1_08,
```

```
121                  INIT_09 => pe_1_09, INIT_0A => pe_1_0A, INIT_0B => pe_1_0B,
122                  INIT_0C => pe_1_0C, INIT_0D => pe_1_0D, INIT_0E => pe_1_0E,
123                  INIT_0F => pe_1_0F)
124         port map(ADDRA => L_PC_E,                   ADDRB => I_PM_ADR(11 downto 2),
125                  CLKA  => I_CLK,                    CLKB  => I_CLK,
126                  DIA   => "0000",                   DIB   => "0000",
127                  ENA   => L_WAIT_N,                 ENB   => '1',
128                  RSTA  => '0',                      RSTB  => '0',
129                  WEA   => '0',                      WEB   => '0',
130                  DOA   => M_OPC_E(7 downto 4),      DOB   => M_PMD_E(7 downto 4));
131
132         pe_2 : RAMB4_S4_S4 ---------------------------------------------------------
133         generic map(INIT_00 => pe_2_00, INIT_01 => pe_2_01, INIT_02 => pe_2_02,
134                  INIT_03 => pe_2_03, INIT_04 => pe_2_04, INIT_05 => pe_2_05,
135                  INIT_06 => pe_2_06, INIT_07 => pe_2_07, INIT_08 => pe_2_08,
136                  INIT_09 => pe_2_09, INIT_0A => pe_2_0A, INIT_0B => pe_2_0B,
137                  INIT_0C => pe_2_0C, INIT_0D => pe_2_0D, INIT_0E => pe_2_0E,
138                  INIT_0F => pe_2_0F)
139         port map(ADDRA => L_PC_E,                   ADDRB => I_PM_ADR(11 downto 2),
140                  CLKA  => I_CLK,                    CLKB  => I_CLK,
141                  DIA   => "0000",                   DIB   => "0000",
142                  ENA   => L_WAIT_N,                 ENB   => '1',
143                  RSTA  => '0',                      RSTB  => '0',
144                  WEA   => '0',                      WEB   => '0',
145                  DOA   => M_OPC_E(11 downto 8),     DOB   => M_PMD_E(11 downto 8));
146
147         pe_3 : RAMB4_S4_S4 ---------------------------------------------------------
148         generic map(INIT_00 => pe_3_00, INIT_01 => pe_3_01, INIT_02 => pe_3_02,
149                  INIT_03 => pe_3_03, INIT_04 => pe_3_04, INIT_05 => pe_3_05,
150                  INIT_06 => pe_3_06, INIT_07 => pe_3_07, INIT_08 => pe_3_08,
151                  INIT_09 => pe_3_09, INIT_0A => pe_3_0A, INIT_0B => pe_3_0B,
152                  INIT_0C => pe_3_0C, INIT_0D => pe_3_0D, INIT_0E => pe_3_0E,
153                  INIT_0F => pe_3_0F)
154         port map(ADDRA => L_PC_E,                   ADDRB => I_PM_ADR(11 downto 2),
155                  CLKA  => I_CLK,                    CLKB  => I_CLK,
156                  DIA   => "0000",                   DIB   => "0000",
157                  ENA   => L_WAIT_N,                 ENB   => '1',
158                  RSTA  => '0',                      RSTB  => '0',
159                  WEA   => '0',                      WEB   => '0',
160                  DOA   => M_OPC_E(15 downto 12),    DOB   => M_PMD_E(15 downto 12));
161
162         po_0 : RAMB4_S4_S4 ---------------------------------------------------------
163         generic map(INIT_00 => po_0_00, INIT_01 => po_0_01, INIT_02 => po_0_02,
164                  INIT_03 => po_0_03, INIT_04 => po_0_04, INIT_05 => po_0_05,
165                  INIT_06 => po_0_06, INIT_07 => po_0_07, INIT_08 => po_0_08,
166                  INIT_09 => po_0_09, INIT_0A => po_0_0A, INIT_0B => po_0_0B,
167                  INIT_0C => po_0_0C, INIT_0D => po_0_0D, INIT_0E => po_0_0E,
168                  INIT_0F => po_0_0F)
169         port map(ADDRA => L_PC_O,                   ADDRB => I_PM_ADR(11 downto 2),
170                  CLKA  => I_CLK,                    CLKB  => I_CLK,
171                  DIA   => "0000",                   DIB   => "0000",
172                  ENA   => L_WAIT_N,                 ENB   => '1',
173                  RSTA  => '0',                      RSTB  => '0',
174                  WEA   => '0',                      WEB   => '0',
175                  DOA   => M_OPC_O(3 downto 0),      DOB   => M_PMD_O(3 downto 0));
176
177         po_1 : RAMB4_S4_S4 ---------------------------------------------------------
178         generic map(INIT_00 => po_1_00, INIT_01 => po_1_01, INIT_02 => po_1_02,
179                  INIT_03 => po_1_03, INIT_04 => po_1_04, INIT_05 => po_1_05,
180                  INIT_06 => po_1_06, INIT_07 => po_1_07, INIT_08 => po_1_08,
181                  INIT_09 => po_1_09, INIT_0A => po_1_0A, INIT_0B => po_1_0B,
182                  INIT_0C => po_1_0C, INIT_0D => po_1_0D, INIT_0E => po_1_0E,
```

```
183                 INIT_0F => po_1_0F)
184         port map(ADDRA => L_PC_O,                 ADDRB => I_PM_ADR(11 downto 2),
185                  CLKA  => I_CLK,                  CLKB  => I_CLK,
186                  DIA   => "0000",                 DIB   => "0000",
187                  ENA   => L_WAIT_N,               ENB   => '1',
188                  RSTA  => '0',                    RSTB  => '0',
189                  WEA   => '0',                    WEB   => '0',
190                  DOA   => M_OPC_O(7 downto 4),    DOB   => M_PMD_O(7 downto 4));
191
192         po_2 : RAMB4_S4_S4 ---------------------------------------------------------
193         generic map(INIT_00 => po_2_00, INIT_01 => po_2_01, INIT_02 => po_2_02,
194                     INIT_03 => po_2_03, INIT_04 => po_2_04, INIT_05 => po_2_05,
195                     INIT_06 => po_2_06, INIT_07 => po_2_07, INIT_08 => po_2_08,
196                     INIT_09 => po_2_09, INIT_0A => po_2_0A, INIT_0B => po_2_0B,
197                     INIT_0C => po_2_0C, INIT_0D => po_2_0D, INIT_0E => po_2_0E,
198                     INIT_0F => po_2_0F)
199         port map(ADDRA => L_PC_O,                 ADDRB => I_PM_ADR(11 downto 2),
200                  CLKA  => I_CLK,                  CLKB  => I_CLK,
201                  DIA   => "0000",                 DIB   => "0000",
202                  ENA   => L_WAIT_N,               ENB   => '1',
203                  RSTA  => '0',                    RSTB  => '0',
204                  WEA   => '0',                    WEB   => '0',
205                  DOA   => M_OPC_O(11 downto 8),   DOB   => M_PMD_O(11 downto 8));
206
207         po_3 : RAMB4_S4_S4 ---------------------------------------------------------
208         generic map(INIT_00 => po_3_00, INIT_01 => po_3_01, INIT_02 => po_3_02,
209                     INIT_03 => po_3_03, INIT_04 => po_3_04, INIT_05 => po_3_05,
210                     INIT_06 => po_3_06, INIT_07 => po_3_07, INIT_08 => po_3_08,
211                     INIT_09 => po_3_09, INIT_0A => po_3_0A, INIT_0B => po_3_0B,
212                     INIT_0C => po_3_0C, INIT_0D => po_3_0D, INIT_0E => po_3_0E,
213                     INIT_0F => po_3_0F)
214         port map(ADDRA => L_PC_O,                 ADDRB => I_PM_ADR(11 downto 2),
215                  CLKA  => I_CLK,                  CLKB  => I_CLK,
216                  DIA   => "0000",                 DIB   => "0000",
217                  ENA   => L_WAIT_N,               ENB   => '1',
218                  RSTA  => '0',                    RSTB  => '0',
219                  WEA   => '0',                    WEB   => '0',
220                  DOA   => M_OPC_O(15 downto 12),  DOB   => M_PMD_O(15 downto 12));
221
222         -- remember I_PC0 and I_PM_ADR for the output mux.
223         --
224         pc0: process(I_CLK)
225         begin
226             if (rising_edge(I_CLK)) then
227                 Q_PC      <= I_PC;
228                 L_PM_ADR_1_0      <= I_PM_ADR(1 downto 0);
229                 if ((I_WAIT = '0')) then
230                     L_PC_0      <= I_PC(0);
231                 end if;
232             end if;
233         end process;
234
235         L_WAIT_N       <= not I_WAIT;
236
237         -- we use two memory blocks _E and _O (even and odd).
238         -- This gives us a quad-port memory so that we can access
239         -- I_PC, I_PC + 1, and PM simultaneously.
240         --
241         -- I_PC and I_PC + 1 are handled by port A of the memory while PM
242         -- is handled by port B.
243         --
244         -- Q_OPC(15 ... 0) shall contain the word addressed by I_PC, while
```

```
245         -- Q_OPC(31 ... 16) shall contain the word addressed by I_PC + 1.
246         --
247         -- There are two cases:
248         --
249         -- case A: I_PC     is even, thus I_PC + 1 is odd
250         -- case B: I_PC + 1 is odd , thus I_PC is even
251         --
252         L_PC_O      <= I_PC(10 downto 1);
253         L_PC_E      <= I_PC(10 downto 1) + ("000000000" & I_PC(0));
254         Q_OPC(15 downto  0)      <= M_OPC_E when L_PC_0 = '0' else M_OPC_O;
255         Q_OPC(31 downto 16)      <= M_OPC_E when L_PC_0 = '1' else M_OPC_O;
256
257         L_PMD      <= M_PMD_E               when (L_PM_ADR_1_0(1) = '0') else M_PMD_O;
258         Q_PM_DOUT      <= L_PMD(7 downto 0) when (L_PM_ADR_1_0(0) = '0')
259                 else L_PMD(15 downto 8);
260
261     end Behavioral;
262
```

src/prog_mem.vhd

22 LISTING OF reg_16.vhd

```
 1  --------------------------------------------------------------------------------
 2  --
 3  -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4  --
 5  --  This code is free software: you can redistribute it and/or modify
 6  --  it under the terms of the GNU General Public License as published by
 7  --  the Free Software Foundation, either version 3 of the License, or
 8  --  (at your option) any later version.
 9  --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  --------------------------------------------------------------------------------
20  --------------------------------------------------------------------------------
21  --
22  -- Module Name:    Register - Behavioral
23  -- Create Date:    12:37:55 10/28/2009
24  -- Description:    a register pair of a CPU.
25  --
26  -----------------------------------------------------------------------------------
27  library IEEE;
28  use IEEE.STD_LOGIC_1164.ALL;
29  use IEEE.STD_LOGIC_ARITH.ALL;
30  use IEEE.STD_LOGIC_UNSIGNED.ALL;
31
32  entity reg_16 is
33      port (  I_CLK       : in  std_logic;
34
35              I_D         : in  std_logic_vector (15 downto 0);
36              I_WE        : in  std_logic_vector ( 1 downto 0);
37
38              Q           : out std_logic_vector (15 downto 0));
39  end reg_16;
40
41  architecture Behavioral of reg_16 is
42
43  signal L                : std_logic_vector (15 downto 0) := X"7777";
44  begin
45
46      process(I_CLK)
47      begin
48          if (rising_edge(I_CLK)) then
49              if (I_WE(1) = '1') then
50                  L(15 downto 8)      <= I_D(15 downto 8);
51              end if;
52              if (I_WE(0) = '1') then
53                  L( 7 downto 0)      <= I_D( 7 downto 0);
54              end if;
55          end if;
56      end process;
57
58      Q      <= L;
```

```
59
60      end Behavioral;
61
```

src/reg_16.vhd

23 LISTING OF register_file.vhd

```
1     -------------------------------------------------------------------------------
2     --
3     -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4     --
5     --  This code is free software: you can redistribute it and/or modify
6     --  it under the terms of the GNU General Public License as published by
7     --  the Free Software Foundation, either version 3 of the License, or
8     --  (at your option) any later version.
9     --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    RegisterFile - Behavioral
23    -- Create Date:    12:43:34 10/28/2009
24    -- Description:    a register file (16 register pairs) of a CPU.
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.STD_LOGIC_1164.ALL;
30    use IEEE.STD_LOGIC_ARITH.ALL;
31    use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33    use work.common.ALL;
34
35    entity register_file is
36        port (  I_CLK       : in  std_logic;
37
38                I_AMOD      : in  std_logic_vector( 5 downto 0);
39                I_COND      : in  std_logic_vector( 3 downto 0);
40                I_DDDDD     : in  std_logic_vector( 4 downto 0);
41                I_DIN       : in  std_logic_vector(15 downto 0);
42                I_FLAGS     : in  std_logic_vector( 7 downto 0);
43                I_IMM       : in  std_logic_vector(15 downto 0);
44                I_RRRR      : in  std_logic_vector( 4 downto 1);
45                I_WE_01     : in  std_logic;
46                I_WE_D      : in  std_logic_vector( 1 downto 0);
47                I_WE_F      : in  std_logic;
48                I_WE_M      : in  std_logic;
49                I_WE_XYZS   : in  std_logic;
50
51                Q_ADR       : out std_logic_vector(15 downto 0);
52                Q_CC        : out std_logic;
53                Q_D         : out std_logic_vector(15 downto 0);
54                Q_FLAGS     : out std_logic_vector( 7 downto 0);
55                Q_R         : out std_logic_vector(15 downto 0);
56                Q_S         : out std_logic_vector( 7 downto 0);
57                Q_Z         : out std_logic_vector(15 downto 0));
58    end register_file;
```

```
 59
 60     architecture Behavioral of register_file is
 61
 62     component reg_16
 63         port (  I_CLK       : in    std_logic;
 64
 65                 I_D         : in    std_logic_vector(15 downto 0);
 66                 I_WE        : in    std_logic_vector( 1 downto 0);
 67
 68                 Q           : out   std_logic_vector(15 downto 0));
 69     end component;
 70
 71     signal R_R00            : std_logic_vector(15 downto 0);
 72     signal R_R02            : std_logic_vector(15 downto 0);
 73     signal R_R04            : std_logic_vector(15 downto 0);
 74     signal R_R06            : std_logic_vector(15 downto 0);
 75     signal R_R08            : std_logic_vector(15 downto 0);
 76     signal R_R10            : std_logic_vector(15 downto 0);
 77     signal R_R12            : std_logic_vector(15 downto 0);
 78     signal R_R14            : std_logic_vector(15 downto 0);
 79     signal R_R16            : std_logic_vector(15 downto 0);
 80     signal R_R18            : std_logic_vector(15 downto 0);
 81     signal R_R20            : std_logic_vector(15 downto 0);
 82     signal R_R22            : std_logic_vector(15 downto 0);
 83     signal R_R24            : std_logic_vector(15 downto 0);
 84     signal R_R26            : std_logic_vector(15 downto 0);
 85     signal R_R28            : std_logic_vector(15 downto 0);
 86     signal R_R30            : std_logic_vector(15 downto 0);
 87     signal R_SP             : std_logic_vector(15 downto 0);    -- stack pointer
 88
 89     component status_reg is
 90         port (  I_CLK       : in  std_logic;
 91
 92                 I_COND      : in  std_logic_vector ( 3 downto 0);
 93                 I_DIN       : in  std_logic_vector ( 7 downto 0);
 94                 I_FLAGS     : in  std_logic_vector ( 7 downto 0);
 95                 I_WE_F      : in  std_logic;
 96                 I_WE_SR     : in  std_logic;
 97
 98                 Q           : out std_logic_vector ( 7 downto 0);
 99                 Q_CC        : out std_logic);
100     end component;
101
102     signal S_FLAGS          : std_logic_vector( 7 downto 0);
103
104     signal L_ADR            : std_logic_vector(15 downto 0);
105     signal L_BASE           : std_logic_vector(15 downto 0);
106     signal L_DDDD           : std_logic_vector( 4 downto 1);
107     signal L_DSP            : std_logic_vector(15 downto 0);
108     signal L_DX             : std_logic_vector(15 downto 0);
109     signal L_DY             : std_logic_vector(15 downto 0);
110     signal L_DZ             : std_logic_vector(15 downto 0);
111     signal L_PRE            : std_logic_vector(15 downto 0);
112     signal L_POST           : std_logic_vector(15 downto 0);
113     signal L_S              : std_logic_vector(15 downto 0);
114     signal L_WE_SP_AMOD     : std_logic;
115     signal L_WE             : std_logic_vector(31 downto 0);
116     signal L_WE_A           : std_logic;
117     signal L_WE_D           : std_logic_vector(31 downto 0);
118     signal L_WE_D2          : std_logic_vector( 1 downto 0);
119     signal L_WE_DD          : std_logic_vector(31 downto 0);
120     signal L_WE_IO          : std_logic_vector(31 downto 0);
```

```
121     signal L_WE_MISC        : std_logic_vector(31 downto 0);
122     signal L_WE_X           : std_logic;
123     signal L_WE_Y           : std_logic;
124     signal L_WE_Z           : std_logic;
125     signal L_WE_SP          : std_logic_vector( 1 downto 0);
126     signal L_WE_SR          : std_logic;
127     signal L_XYZS           : std_logic_vector(15 downto 0);
128
129     begin
130
131         r00: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 1 downto  0), I_D => I_DIN, Q => R
132         r02: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 3 downto  2), I_D => I_DIN, Q => R
133         r04: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 5 downto  4), I_D => I_DIN, Q => R
134         r06: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 7 downto  6), I_D => I_DIN, Q => R
135         r08: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 9 downto  8), I_D => I_DIN, Q => R
136         r10: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(11 downto 10), I_D => I_DIN, Q => R
137         r12: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(13 downto 12), I_D => I_DIN, Q => R
138         r14: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(15 downto 14), I_D => I_DIN, Q => R
139         r16: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(17 downto 16), I_D => I_DIN, Q => R
140         r18: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(19 downto 18), I_D => I_DIN, Q => R
141         r20: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(21 downto 20), I_D => I_DIN, Q => R
142         r22: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(23 downto 22), I_D => I_DIN, Q => R
143         r24: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(25 downto 24), I_D => I_DIN, Q => R
144         r26: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(27 downto 26), I_D => L_DX,  Q => R
145         r28: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(29 downto 28), I_D => L_DY,  Q => R
146         r30: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(31 downto 30), I_D => L_DZ,  Q => R
147         sp:  reg_16 port map(I_CLK => I_CLK, I_WE => L_WE_SP,            I_D => L_DSP, Q => R
148
149         sr: status_reg
150         port map(   I_CLK       => I_CLK,
151                     I_COND      => I_COND,
152                     I_DIN       => I_DIN(7 downto 0),
153                     I_FLAGS     => I_FLAGS,
154                     I_WE_F      => I_WE_F,
155                     I_WE_SR     => L_WE_SR,
156                     Q           => S_FLAGS,
157                     Q_CC        => Q_CC);
158
159         -- The output of the register selected by L_ADR.
160         --
161         process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
162                 R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
163                 R_SP, S_FLAGS, L_ADR(6 downto 1))
164         begin
165             case L_ADR(6 downto 1) is
166                 when "000000" => L_S      <= R_R00;
167                 when "000001" => L_S      <= R_R02;
168                 when "000010" => L_S      <= R_R04;
169                 when "000011" => L_S      <= R_R06;
170                 when "000100" => L_S      <= R_R08;
171                 when "000101" => L_S      <= R_R10;
172                 when "000110" => L_S      <= R_R12;
173                 when "000111" => L_S      <= R_R14;
174                 when "001000" => L_S      <= R_R16;
175                 when "001001" => L_S      <= R_R18;
176                 when "001010" => L_S      <= R_R20;
177                 when "001011" => L_S      <= R_R22;
178                 when "001100" => L_S      <= R_R24;
179                 when "001101" => L_S      <= R_R26;
180                 when "001110" => L_S      <= R_R28;
181                 when "001111" => L_S      <= R_R30;
182                 when "101111" => L_S      <= R_SP ( 7 downto 0) & X"00";     -- SPL
```

```
183                 when others   => L_S      <= S_FLAGS & R_SP (15 downto 8);   -- SR/SPH
184             end case;
185         end process;
186 
187         -- The output of the register pair selected by I_DDDDD.
188         --
189         process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
190                 R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
191                 I_DDDDD(4 downto 1))
192         begin
193             case I_DDDDD(4 downto 1) is
194                 when "0000" => Q_D      <= R_R00;
195                 when "0001" => Q_D      <= R_R02;
196                 when "0010" => Q_D      <= R_R04;
197                 when "0011" => Q_D      <= R_R06;
198                 when "0100" => Q_D      <= R_R08;
199                 when "0101" => Q_D      <= R_R10;
200                 when "0110" => Q_D      <= R_R12;
201                 when "0111" => Q_D      <= R_R14;
202                 when "1000" => Q_D      <= R_R16;
203                 when "1001" => Q_D      <= R_R18;
204                 when "1010" => Q_D      <= R_R20;
205                 when "1011" => Q_D      <= R_R22;
206                 when "1100" => Q_D      <= R_R24;
207                 when "1101" => Q_D      <= R_R26;
208                 when "1110" => Q_D      <= R_R28;
209                 when others => Q_D      <= R_R30;
210             end case;
211         end process;
212 
213         -- The output of the register pair selected by I_RRRR.
214         --
215         process(R_R00, R_R02, R_R04,  R_R06, R_R08, R_R10, R_R12, R_R14,
216                 R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30, I_RRRR)
217         begin
218             case I_RRRR is
219                 when "0000" => Q_R      <= R_R00;
220                 when "0001" => Q_R      <= R_R02;
221                 when "0010" => Q_R      <= R_R04;
222                 when "0011" => Q_R      <= R_R06;
223                 when "0100" => Q_R      <= R_R08;
224                 when "0101" => Q_R      <= R_R10;
225                 when "0110" => Q_R      <= R_R12;
226                 when "0111" => Q_R      <= R_R14;
227                 when "1000" => Q_R      <= R_R16;
228                 when "1001" => Q_R      <= R_R18;
229                 when "1010" => Q_R      <= R_R20;
230                 when "1011" => Q_R      <= R_R22;
231                 when "1100" => Q_R      <= R_R24;
232                 when "1101" => Q_R      <= R_R26;
233                 when "1110" => Q_R      <= R_R28;
234                 when others => Q_R      <= R_R30;
235             end case;
236         end process;
237 
238         -- the base value of the X/Y/Z/SP register as per I_AMOD.
239         --
240         process(I_AMOD(2 downto 0), I_IMM, R_SP, R_R26, R_R28, R_R30)
241         begin
242             case I_AMOD(2 downto 0) is
243                 when AS_SP  => L_BASE      <= R_SP;
244                 when AS_Z   => L_BASE      <= R_R30;
```

```
245                 when AS_Y   => L_BASE      <= R_R28;
246                 when AS_X   => L_BASE      <= R_R26;
247                 when AS_IMM => L_BASE      <= I_IMM;
248                 when others => L_BASE      <= X"0000";
249             end case;
250         end process;
251 
252         -- the value of the X/Y/Z/SP register after a potential PRE-decrement
253         -- (by 1 or 2) and POST-increment (by 1 or 2).
254         --
255         process(I_AMOD(5 downto 3), I_IMM)
256         begin
257             case I_AMOD(5 downto 3) is
258                 when AO_0   => L_PRE      <= X"0000";    L_POST <= X"0000";
259                 when AO_i   => L_PRE      <= X"0000";    L_POST <= X"0001";
260                 when AO_ii  => L_PRE      <= X"0000";    L_POST <= X"0002";
261                 when AO_q   => L_PRE      <= I_IMM;      L_POST <= X"0000";
262                 when AO_d   => L_PRE      <= X"FFFF";    L_POST <= X"FFFF";
263                 when AO_dd  => L_PRE      <= X"FFFE";    L_POST <= X"FFFE";
264                 when others => L_PRE      <= X"0000";    L_POST <= X"0000";
265             end case;
266         end process;
267 
268         L_XYZS      <= L_BASE + L_POST;
269         L_ADR       <= L_BASE + L_PRE;
270 
271         L_WE_A      <= I_WE_M when (L_ADR(15 downto 5) = "00000000000") else '0';
272         L_WE_SR         <= I_WE_M when (L_ADR = X"005F") else '0';
273         L_WE_SP_AMOD      <= I_WE_XYZS when (I_AMOD(2 downto 0) = AS_SP) else '0';
274         L_WE_SP(1)      <= I_WE_M when (L_ADR = X"005E") else L_WE_SP_AMOD;
275         L_WE_SP(0)      <= I_WE_M when (L_ADR = X"005D") else L_WE_SP_AMOD;
276 
277         L_DX       <= L_XYZS when (L_WE_MISC(26) = '1')        else I_DIN;
278         L_DY       <= L_XYZS when (L_WE_MISC(28) = '1')        else I_DIN;
279         L_DZ       <= L_XYZS when (L_WE_MISC(30) = '1')        else I_DIN;
280         L_DSP      <= L_XYZS when (I_AMOD(3 downto 0) = AM_WS) else I_DIN;
281 
282         -- the WE signals for the differen registers.
283         --
284         -- case 1: write to an 8-bit register addressed by DDDDD.
285         --
286         -- I_WE_D(0) = '1' and I_DDDDD matches,
287         --
288         L_WE_D( 0)      <= I_WE_D(0) when (I_DDDDD = "00000") else '0';
289         L_WE_D( 1)      <= I_WE_D(0) when (I_DDDDD = "00001") else '0';
290         L_WE_D( 2)      <= I_WE_D(0) when (I_DDDDD = "00010") else '0';
291         L_WE_D( 3)      <= I_WE_D(0) when (I_DDDDD = "00011") else '0';
292         L_WE_D( 4)      <= I_WE_D(0) when (I_DDDDD = "00100") else '0';
293         L_WE_D( 5)      <= I_WE_D(0) when (I_DDDDD = "00101") else '0';
294         L_WE_D( 6)      <= I_WE_D(0) when (I_DDDDD = "00110") else '0';
295         L_WE_D( 7)      <= I_WE_D(0) when (I_DDDDD = "00111") else '0';
296         L_WE_D( 8)      <= I_WE_D(0) when (I_DDDDD = "01000") else '0';
297         L_WE_D( 9)      <= I_WE_D(0) when (I_DDDDD = "01001") else '0';
298         L_WE_D(10)      <= I_WE_D(0) when (I_DDDDD = "01010") else '0';
299         L_WE_D(11)      <= I_WE_D(0) when (I_DDDDD = "01011") else '0';
300         L_WE_D(12)      <= I_WE_D(0) when (I_DDDDD = "01100") else '0';
301         L_WE_D(13)      <= I_WE_D(0) when (I_DDDDD = "01101") else '0';
302         L_WE_D(14)      <= I_WE_D(0) when (I_DDDDD = "01110") else '0';
303         L_WE_D(15)      <= I_WE_D(0) when (I_DDDDD = "01111") else '0';
304         L_WE_D(16)      <= I_WE_D(0) when (I_DDDDD = "10000") else '0';
305         L_WE_D(17)      <= I_WE_D(0) when (I_DDDDD = "10001") else '0';
306         L_WE_D(18)      <= I_WE_D(0) when (I_DDDDD = "10010") else '0';
```

```
307         L_WE_D(19)       <= I_WE_D(0) when (I_DDDDD = "10011") else '0';
308         L_WE_D(20)       <= I_WE_D(0) when (I_DDDDD = "10100") else '0';
309         L_WE_D(21)       <= I_WE_D(0) when (I_DDDDD = "10101") else '0';
310         L_WE_D(22)       <= I_WE_D(0) when (I_DDDDD = "10110") else '0';
311         L_WE_D(23)       <= I_WE_D(0) when (I_DDDDD = "10111") else '0';
312         L_WE_D(24)       <= I_WE_D(0) when (I_DDDDD = "11000") else '0';
313         L_WE_D(25)       <= I_WE_D(0) when (I_DDDDD = "11001") else '0';
314         L_WE_D(26)       <= I_WE_D(0) when (I_DDDDD = "11010") else '0';
315         L_WE_D(27)       <= I_WE_D(0) when (I_DDDDD = "11011") else '0';
316         L_WE_D(28)       <= I_WE_D(0) when (I_DDDDD = "11100") else '0';
317         L_WE_D(29)       <= I_WE_D(0) when (I_DDDDD = "11101") else '0';
318         L_WE_D(30)       <= I_WE_D(0) when (I_DDDDD = "11110") else '0';
319         L_WE_D(31)       <= I_WE_D(0) when (I_DDDDD = "11111") else '0';
320
321         --
322         -- case 2: write to a 16-bit register pair addressed by DDDD.
323         --
324         -- I_WE_DD(1) = '1' and L_DDDD matches,
325         --
326         L_DDDD       <= I_DDDDD(4 downto 1);
327         L_WE_D2       <= I_WE_D(1) & I_WE_D(1);
328         L_WE_DD( 1 downto  0)       <= L_WE_D2 when (L_DDDD = "0000") else "00";
329         L_WE_DD( 3 downto  2)       <= L_WE_D2 when (L_DDDD = "0001") else "00";
330         L_WE_DD( 5 downto  4)       <= L_WE_D2 when (L_DDDD = "0010") else "00";
331         L_WE_DD( 7 downto  6)       <= L_WE_D2 when (L_DDDD = "0011") else "00";
332         L_WE_DD( 9 downto  8)       <= L_WE_D2 when (L_DDDD = "0100") else "00";
333         L_WE_DD(11 downto 10)       <= L_WE_D2 when (L_DDDD = "0101") else "00";
334         L_WE_DD(13 downto 12)       <= L_WE_D2 when (L_DDDD = "0110") else "00";
335         L_WE_DD(15 downto 14)       <= L_WE_D2 when (L_DDDD = "0111") else "00";
336         L_WE_DD(17 downto 16)       <= L_WE_D2 when (L_DDDD = "1000") else "00";
337         L_WE_DD(19 downto 18)       <= L_WE_D2 when (L_DDDD = "1001") else "00";
338         L_WE_DD(21 downto 20)       <= L_WE_D2 when (L_DDDD = "1010") else "00";
339         L_WE_DD(23 downto 22)       <= L_WE_D2 when (L_DDDD = "1011") else "00";
340         L_WE_DD(25 downto 24)       <= L_WE_D2 when (L_DDDD = "1100") else "00";
341         L_WE_DD(27 downto 26)       <= L_WE_D2 when (L_DDDD = "1101") else "00";
342         L_WE_DD(29 downto 28)       <= L_WE_D2 when (L_DDDD = "1110") else "00";
343         L_WE_DD(31 downto 30)       <= L_WE_D2 when (L_DDDD = "1111") else "00";
344
345         --
346         -- case 3: write to an 8-bit register pair addressed by an I/O address.
347         --
348         -- L_WE_A = '1' and L_ADR(4 downto 0) matches
349         --
350         L_WE_IO( 0)       <= L_WE_A when (L_ADR(4 downto 0) = "00000") else '0';
351         L_WE_IO( 1)       <= L_WE_A when (L_ADR(4 downto 0) = "00001") else '0';
352         L_WE_IO( 2)       <= L_WE_A when (L_ADR(4 downto 0) = "00010") else '0';
353         L_WE_IO( 3)       <= L_WE_A when (L_ADR(4 downto 0) = "00011") else '0';
354         L_WE_IO( 4)       <= L_WE_A when (L_ADR(4 downto 0) = "00100") else '0';
355         L_WE_IO( 5)       <= L_WE_A when (L_ADR(4 downto 0) = "00101") else '0';
356         L_WE_IO( 6)       <= L_WE_A when (L_ADR(4 downto 0) = "00110") else '0';
357         L_WE_IO( 7)       <= L_WE_A when (L_ADR(4 downto 0) = "00111") else '0';
358         L_WE_IO( 8)       <= L_WE_A when (L_ADR(4 downto 0) = "01000") else '0';
359         L_WE_IO( 9)       <= L_WE_A when (L_ADR(4 downto 0) = "01001") else '0';
360         L_WE_IO(10)       <= L_WE_A when (L_ADR(4 downto 0) = "01010") else '0';
361         L_WE_IO(11)       <= L_WE_A when (L_ADR(4 downto 0) = "01011") else '0';
362         L_WE_IO(12)       <= L_WE_A when (L_ADR(4 downto 0) = "01100") else '0';
363         L_WE_IO(13)       <= L_WE_A when (L_ADR(4 downto 0) = "01101") else '0';
364         L_WE_IO(14)       <= L_WE_A when (L_ADR(4 downto 0) = "01110") else '0';
365         L_WE_IO(15)       <= L_WE_A when (L_ADR(4 downto 0) = "01111") else '0';
366         L_WE_IO(16)       <= L_WE_A when (L_ADR(4 downto 0) = "10000") else '0';
367         L_WE_IO(17)       <= L_WE_A when (L_ADR(4 downto 0) = "10001") else '0';
368         L_WE_IO(18)       <= L_WE_A when (L_ADR(4 downto 0) = "10010") else '0';
```

```
369         L_WE_IO(19)       <= L_WE_A when (L_ADR(4 downto 0) = "10011") else '0';
370         L_WE_IO(20)       <= L_WE_A when (L_ADR(4 downto 0) = "10100") else '0';
371         L_WE_IO(21)       <= L_WE_A when (L_ADR(4 downto 0) = "10101") else '0';
372         L_WE_IO(22)       <= L_WE_A when (L_ADR(4 downto 0) = "10110") else '0';
373         L_WE_IO(23)       <= L_WE_A when (L_ADR(4 downto 0) = "10111") else '0';
374         L_WE_IO(24)       <= L_WE_A when (L_ADR(4 downto 0) = "11000") else '0';
375         L_WE_IO(25)       <= L_WE_A when (L_ADR(4 downto 0) = "11001") else '0';
376         L_WE_IO(26)       <= L_WE_A when (L_ADR(4 downto 0) = "11010") else '0';
377         L_WE_IO(27)       <= L_WE_A when (L_ADR(4 downto 0) = "11011") else '0';
378         L_WE_IO(28)       <= L_WE_A when (L_ADR(4 downto 0) = "11100") else '0';
379         L_WE_IO(29)       <= L_WE_A when (L_ADR(4 downto 0) = "11101") else '0';
380         L_WE_IO(30)       <= L_WE_A when (L_ADR(4 downto 0) = "11110") else '0';
381         L_WE_IO(31)       <= L_WE_A when (L_ADR(4 downto 0) = "11111") else '0';
382
383         -- case 4 special cases.
384         -- 4a. WE_01 for register pair 0/1 (multiplication opcode).
385         -- 4b. I_WE_XYZS for X (register pairs 26/27) and I_AMOD matches
386         -- 4c. I_WE_XYZS for Y (register pairs 28/29) and I_AMOD matches
387         -- 4d. I_WE_XYZS for Z (register pairs 30/31) and I_AMOD matches
388         --
389         L_WE_X       <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WX) else '0';
390         L_WE_Y       <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WY) else '0';
391         L_WE_Z       <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WZ) else '0';
392         L_WE_MISC       <= L_WE_Z & L_WE_Z &      -- -Z and Z+ address modes  r30
393                      L_WE_Y & L_WE_Y &      -- -Y and Y+ address modes  r28
394                      L_WE_X & L_WE_X &      -- -X and X+ address modes  r26
395                      X"000000" &            -- never                    r24 - r02
396                      I_WE_01 & I_WE_01;     -- multiplication result    r00
397
398         L_WE       <= L_WE_D or L_WE_DD or L_WE_IO or L_WE_MISC;
399
400         Q_S       <= L_S( 7 downto 0) when (L_ADR(0) = '0') else L_S(15 downto 8);
401         Q_FLAGS       <= S_FLAGS;
402         Q_Z       <= R_R30;
403         Q_ADR       <= L_ADR;
404
405     end Behavioral;
406
```

src/register_file.vhd

24 LISTING OF segment7.vhd

```
1     -------------------------------------------------------------------------------
2     --
3     -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4     --
5     --  This code is free software: you can redistribute it and/or modify
6     --  it under the terms of the GNU General Public License as published by
7     --  the Free Software Foundation, either version 3 of the License, or
8     --  (at your option) any later version.
9     --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    -------------------------------------------------------------------------------
20    -------------------------------------------------------------------------------
21    --
22    -- Module Name:    segment7 - Behavioral
23    -- Create Date:    12:52:16 11/11/2009
24    -- Description:    a 7 segment LED display interface.
25    --
26    -------------------------------------------------------------------------------
27    --
28    library IEEE;
29    use IEEE.STD_LOGIC_1164.ALL;
30    use IEEE.STD_LOGIC_ARITH.ALL;
31    use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33    entity segment7 is
34        port ( I_CLK        : in  std_logic;
35
36               I_CLR        : in  std_logic;
37               I_OPC        : in  std_logic_vector(15 downto 0);
38               I_PC         : in  std_logic_vector(15 downto 0);
39
40               Q_7_SEGMENT : out std_logic_vector( 6 downto 0));
41    end segment7;
42
43    --      Signal      Loc Alt
44    ---------------------------
45    --      SEG_LED(0)  V3  A
46    --      SEG_LED(1)  V4  B
47    --      SEG_LED(2)  W3  C
48    --      SEG_LED(3)  T4  D
49    --      SEG_LED(4)  T3  E
50    --      SEG_LED(5)  U3  F
51    --      SEG_LED(6)  U4  G
52    --
53    architecture Behavioral of segment7 is
54
55    function lmap(VAL: std_logic_vector( 3 downto 0))
56             return std_logic_vector is
57    begin
58        case VAL is         --      6543210
```

```
 59             when "0000" =>  return "0111111";   -- 0
 60             when "0001" =>  return "0000110";   -- 1
 61             when "0010" =>  return "1011011";   -- 2
 62             when "0011" =>  return "1001111";   -- 3
 63             when "0100" =>  return "1100110";   -- 4    ----A----       ----0----
 64             when "0101" =>  return "1101101";   -- 5    |       |       |       |
 65             when "0110" =>  return "1111101";   -- 6    F       B       5       1
 66             when "0111" =>  return "0000111";   -- 7    |       |       |       |
 67             when "1000" =>  return "1111111";   -- 8    +---G---+       +---6---+
 68             when "1001" =>  return "1101111";   -- 9    |       |       |       |
 69             when "1010" =>  return "1110111";   -- A    E       C       4       2
 70             when "1011" =>  return "1111100";   -- b    |       |       |       |
 71             when "1100" =>  return "0111001";   -- C    ----D----       ----3----
 72             when "1101" =>  return "1011110";   -- d
 73             when "1110" =>  return "1111001";   -- E
 74             when others =>  return "1110001";   -- F
 75         end case;
 76     end;
 77
 78     signal L_CNT            : std_logic_vector(27 downto 0);
 79     signal L_OPC            : std_logic_vector(15 downto 0);
 80     signal L_PC             : std_logic_vector(15 downto 0);
 81     signal L_POS            : std_logic_vector( 3 downto 0);
 82
 83     begin
 84
 85         process(I_CLK)    -- 20 MHz
 86         begin
 87             if (rising_edge(I_CLK)) then
 88                 if (I_CLR = '1') then
 89                     L_POS       <= "0000";
 90                     L_CNT       <= X"0000000";
 91                     Q_7_SEGMENT       <= "1111111";
 92                 else
 93                     L_CNT       <= L_CNT + X"0000001";
 94                     if (L_CNT =  X"0C00000") then
 95                         Q_7_SEGMENT       <= "1111111";       -- blank
 96                     elsif (L_CNT =  X"1000000") then
 97                         L_CNT       <= X"0000000";
 98                         L_POS       <= L_POS + "0001";
 99                         case L_POS is
100                             when "0000" =>  -- blank
101                                 Q_7_SEGMENT       <= "1111111";
102                             when "0001" =>
103                                 L_PC       <= I_PC;        -- sample PC
104                                 L_OPC       <= I_OPC;      -- sample OPC
105                                 Q_7_SEGMENT       <= not lmap(L_PC(15 downto 12));
106                             when "0010" =>
107                                 Q_7_SEGMENT       <= not lmap(L_PC(11 downto  8));
108                             when "0011" =>
109                                 Q_7_SEGMENT       <= not lmap(L_PC( 7 downto  4));
110                             when "0100" =>
111                                 Q_7_SEGMENT       <= not lmap(L_PC( 3 downto  0));
112                             when "0101" =>  -- minus
113                                 Q_7_SEGMENT       <= "0111111";
114                             when "0110" =>
115                                 Q_7_SEGMENT       <= not lmap(L_OPC(15 downto 12));
116                             when "0111" =>
117                                 Q_7_SEGMENT       <= not lmap(L_OPC(11 downto  8));
118                             when "1000" =>
119                                 Q_7_SEGMENT       <= not lmap(L_OPC( 7 downto  4));
120                             when "1001" =>
```

```
121                                 Q_7_SEGMENT      <= not lmap(L_OPC( 3 downto  0));
122                                 L_POS      <= "0000";
123                             when others =>
124                                 L_POS      <= "0000";
125                         end case;
126                     end if;
127                 end if;
128             end if;
129         end process;
130
131     end Behavioral;
132
```

src/segment7.vhd

25 LISTING OF status_reg.vhd

```
 1  --------------------------------------------------------------------------------
 2  --
 3  -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4  --
 5  --  This code is free software: you can redistribute it and/or modify
 6  --  it under the terms of the GNU General Public License as published by
 7  --  the Free Software Foundation, either version 3 of the License, or
 8  --  (at your option) any later version.
 9  --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  --------------------------------------------------------------------------------
20  --------------------------------------------------------------------------------
21  --
22  -- Module Name:    Register - Behavioral
23  -- Create Date:    16:15:54 12/26/2009
24  -- Description:    the status register of a CPU.
25  --
26  ----------------------------------------------------------------------------------
27  library IEEE;
28  use IEEE.STD_LOGIC_1164.ALL;
29  use IEEE.STD_LOGIC_ARITH.ALL;
30  use IEEE.STD_LOGIC_UNSIGNED.ALL;
31
32  entity status_reg is
33      port (  I_CLK       : in  std_logic;
34
35              I_COND      : in  std_logic_vector ( 3 downto 0);
36              I_DIN       : in  std_logic_vector ( 7 downto 0);
37              I_FLAGS     : in  std_logic_vector ( 7 downto 0);
38              I_WE_F      : in  std_logic;
39              I_WE_SR     : in  std_logic;
40
41              Q           : out std_logic_vector ( 7 downto 0);
42              Q_CC        : out std_logic);
43  end status_reg;
44
45  architecture Behavioral of status_reg is
46
47  signal L                : std_logic_vector ( 7 downto 0);
48  begin
49
50      process(I_CLK)
51      begin
52          if (rising_edge(I_CLK)) then
53              if (I_WE_F = '1') then          -- write flags (from ALU)
54                  L       <= I_FLAGS;
55              elsif (I_WE_SR = '1') then      -- write I/O
56                  L       <= I_DIN;
57              end if;
58          end if;
```

```
 59         end process;
 60
 61         cond: process(I_COND, L)
 62         begin
 63             case I_COND(2 downto 0) is
 64                 when "000"  => Q_CC       <= L(0) xor I_COND(3);
 65                 when "001"  => Q_CC       <= L(1) xor I_COND(3);
 66                 when "010"  => Q_CC       <= L(2) xor I_COND(3);
 67                 when "011"  => Q_CC       <= L(3) xor I_COND(3);
 68                 when "100"  => Q_CC       <= L(4) xor I_COND(3);
 69                 when "101"  => Q_CC       <= L(5) xor I_COND(3);
 70                 when "110"  => Q_CC       <= L(6) xor I_COND(3);
 71                 when others => Q_CC       <= L(7) xor I_COND(3);
 72             end case;
 73         end process;
 74
 75         Q       <= L;
 76
 77     end Behavioral;
 78
```

src/status_reg.vhd

26 LISTING OF uart_rx.vhd

```
1   -------------------------------------------------------------------------------
2   --
3   -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4   --
5   --  This code is free software: you can redistribute it and/or modify
6   --  it under the terms of the GNU General Public License as published by
7   --  the Free Software Foundation, either version 3 of the License, or
8   --  (at your option) any later version.
9   --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  -------------------------------------------------------------------------------
20  -------------------------------------------------------------------------------
21  --
22  -- Create Date:    14:22:28 11/07/2009
23  -- Design Name:
24  -- Module Name:    uart_rx - Behavioral
25  -- Description:    a UART receiver.
26  --
27  -------------------------------------------------------------------------------
28  --
29  library IEEE;
30  use IEEE.STD_LOGIC_1164.ALL;
31  use IEEE.STD_LOGIC_ARITH.ALL;
32  use IEEE.STD_LOGIC_UNSIGNED.ALL;
33
34  entity uart_rx is
35      PORT(   I_CLK       : in  std_logic;
36              I_CLR       : in  std_logic;
37              I_CE_16     : in  std_logic;            -- 16 times baud rate
38              I_RX        : in  std_logic;            -- Serial input line
39
40              Q_DATA      : out std_logic_vector(7 downto 0);
41              Q_FLAG      : out std_logic);       -- toggle on every byte received
42  end uart_rx;
43
44  architecture Behavioral of uart_rx is
45
46  signal L_POSITION       : std_logic_vector(7 downto 0);     --  sample position
47  signal L_BUF            : std_logic_vector(9 downto 0);
48  signal L_FLAG           : std_logic;
49  signal L_SERIN          : std_logic;                -- double clock the input
50  signal L_SER_HOT        : std_logic;                -- double clock the input
51
52  begin
53
54      -- double clock the input data...
55      --
56      process(I_CLK)
57      begin
58          if (rising_edge(I_CLK)) then
```

```
 59                 if (I_CLR = '1') then
 60                     L_SERIN      <= '1';
 61                     L_SER_HOT      <= '1';
 62                 else
 63                     L_SERIN        <= I_RX;
 64                     L_SER_HOT      <= L_SERIN;
 65                 end if;
 66             end if;
 67         end process;
 68
 69         process(I_CLK, L_POSITION)
 70             variable START_BIT : boolean;
 71             variable STOP_BIT  : boolean;
 72             variable STOP_POS  : boolean;
 73
 74         begin
 75         START_BIT := L_POSITION(7 downto 4) = X"0";
 76         STOP_BIT  := L_POSITION(7 downto 4) = X"9";
 77         STOP_POS  := STOP_BIT and L_POSITION(3 downto 2) = "11"; -- 3/4 of stop bit
 78
 79             if (rising_edge(I_CLK)) then
 80                 if (I_CLR = '1') then
 81                     L_FLAG      <= '0';
 82                     L_POSITION      <= X"00";    -- idle
 83                     L_BUF           <= "1111111111";
 84                     Q_DATA          <= "00000000";
 85                 elsif (I_CE_16 = '1') then
 86                     if (L_POSITION = X"00") then            -- uart idle
 87                         L_BUF       <= "1111111111";
 88                         if (L_SER_HOT = '0')  then          -- start bit received
 89                             L_POSITION      <= X"01";
 90                         end if;
 91                     else
 92                         L_POSITION      <= L_POSITION + X"01";
 93                         if (L_POSITION(3 downto 0) = "0111") then       -- 1/2 bit
 94                             L_BUF      <= L_SER_HOT & L_BUF(9 downto 1);     -- sample data
 95                             --
 96                             -- validate start bit
 97                             --
 98                             if (START_BIT and L_SER_HOT = '1') then     -- 1/2 start bit
 99                                 L_POSITION      <= X"00";
100                             end if;
101
102                             if (STOP_BIT) then                          -- 1/2 stop bit
103                                 Q_DATA      <= L_BUF(9 downto 2);
104                             end if;
105                         elsif (STOP_POS) then                       -- 3/4 stop bit
106                             L_FLAG      <= L_FLAG xor (L_BUF(9) and not L_BUF(0));
107                             L_POSITION      <= X"00";
108                         end if;
109                     end if;
110                 end if;
111             end if;
112         end process;
113
114         Q_FLAG      <= L_FLAG;
115
116     end Behavioral;
117
```

src/uart_rx.vhd

27 LISTING OF uart_tx.vhd

```
 1  -------------------------------------------------------------------------------
 2  --
 3  -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4  --
 5  --  This code is free software: you can redistribute it and/or modify
 6  --  it under the terms of the GNU General Public License as published by
 7  --  the Free Software Foundation, either version 3 of the License, or
 8  --  (at your option) any later version.
 9  --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  -------------------------------------------------------------------------------
20  -------------------------------------------------------------------------------
21  --
22  -- Module Name:    uart_tx - Behavioral
23  -- Create Date:    14:21:59 11/07/2009
24  -- Description:    a UART receiver.
25  --
26  -------------------------------------------------------------------------------
27  --
28  library IEEE;
29  use IEEE.STD_LOGIC_1164.ALL;
30  use IEEE.STD_LOGIC_ARITH.ALL;
31  use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33  entity uart_tx is
34      port(   I_CLK       : in  std_logic;
35              I_CLR       : in  std_logic;              -- RESET
36              I_CE_1      : in  std_logic;              -- BAUD rate clock enable
37              I_DATA      : in  std_logic_vector(7 downto 0);   -- DATA to be sent
38              I_FLAG      : in  std_logic;              -- toggle to send data
39              Q_TX        : out std_logic;              -- Serial output line
40              Q_FLAG      : out std_logic);             -- Transmitting Flag
41  end uart_tx;
42
43  architecture Behavioral of uart_tx is
44
45  signal L_BUF            : std_logic_vector(7 downto 0);
46  signal L_TODO           : std_logic_vector(3 downto 0);     -- bits to send
47  signal L_FLAG           : std_logic;
48
49  begin
50
51      process(I_CLK)
52      begin
53          if (rising_edge(I_CLK)) then
54              if (I_CLR = '1') then
55                  Q_TX        <= '1';
56                  L_BUF       <= "11111111";
57                  L_TODO      <= "0000";
58                  L_FLAG      <= I_FLAG;                    -- idle
```

```
 59                 elsif (I_CE_1 = '1') then
 60                     if (L_TODO /= "0000") then          -- transmitting
 61                         Q_TX      <= L_BUF(0);                -- next bit
 62                         L_BUF          <= '1' & L_BUF(7 downto 1);
 63                         if (L_TODO = "0001") then
 64                             L_FLAG      <= I_FLAG;
 65                         end if;
 66                         L_TODO      <= L_TODO - "0001";
 67                     elsif (L_FLAG /= I_FLAG) then       -- new byte
 68                         Q_TX      <= '0';                     -- start bit
 69                         L_BUF      <= I_DATA;                 -- data bits
 70                         L_TODO      <= "1001";
 71                     end if;
 72                 end if;
 73             end if;
 74         end process;
 75
 76         Q_FLAG      <= L_FLAG;
 77
 78     end Behavioral;
 79
```

src/uart_tx.vhd

28 LISTING OF RAMB4_S4_S4.vhd

```
 1    --------------------------------------------------------------------------------
 2    --
 3    -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
 4    --
 5    --  This code is free software: you can redistribute it and/or modify
 6    --  it under the terms of the GNU General Public License as published by
 7    --  the Free Software Foundation, either version 3 of the License, or
 8    --  (at your option) any later version.
 9    --
10    --  This code is distributed in the hope that it will be useful,
11    --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12    --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13    --  GNU General Public License for more details.
14    --
15    --  You should have received a copy of the GNU General Public License
16    --  along with this code (see the file named COPYING).
17    --  If not, see http://www.gnu.org/licenses/.
18    --
19    --------------------------------------------------------------------------------
20    --------------------------------------------------------------------------------
21    --
22    -- Module Name:    prog_mem - Behavioral
23    -- Create Date:    14:09:04 10/30/2009
24    -- Description:    a block memory module
25    --
26    --------------------------------------------------------------------------------
27
28    library IEEE;
29    use IEEE.STD_LOGIC_1164.ALL;
30    use IEEE.STD_LOGIC_ARITH.ALL;
31    use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33    entity RAMB4_S4_S4 is
34        generic(INIT_00 : bit_vector := X"00000000000000000000000000000000"
35                                      &  "00000000000000000000000000000000";
36                INIT_01 : bit_vector := X"00000000000000000000000000000000"
37                                      & X"00000000000000000000000000000000";
38                INIT_02 : bit_vector := X"00000000000000000000000000000000"
39                                      & X"00000000000000000000000000000000";
40                INIT_03 : bit_vector := X"00000000000000000000000000000000"
41                                      & X"00000000000000000000000000000000";
42                INIT_04 : bit_vector := X"00000000000000000000000000000000"
43                                      & X"00000000000000000000000000000000";
44                INIT_05 : bit_vector := X"00000000000000000000000000000000"
45                                      & X"00000000000000000000000000000000";
46                INIT_06 : bit_vector := X"00000000000000000000000000000000"
47                                      & X"00000000000000000000000000000000";
48                INIT_07 : bit_vector := X"00000000000000000000000000000000"
49                                      & X"00000000000000000000000000000000";
50                INIT_08 : bit_vector := X"00000000000000000000000000000000"
51                                      & X"00000000000000000000000000000000";
52                INIT_09 : bit_vector := X"00000000000000000000000000000000"
53                                      & X"00000000000000000000000000000000";
54                INIT_0A : bit_vector := X"00000000000000000000000000000000"
55                                      & X"00000000000000000000000000000000";
56                INIT_0B : bit_vector := X"00000000000000000000000000000000"
57                                      & X"00000000000000000000000000000000";
58                INIT_0C : bit_vector := X"00000000000000000000000000000000"
```

```
59                                     & X"00000000000000000000000000000000";
60                 INIT_0D : bit_vector := X"00000000000000000000000000000000"
61                                     & X"00000000000000000000000000000000";
62                 INIT_0E : bit_vector := X"00000000000000000000000000000000"
63                                     & X"00000000000000000000000000000000";
64                 INIT_0F : bit_vector := X"00000000000000000000000000000000"
65                                     & X"00000000000000000000000000000000");
66
67         port(   ADDRA    : in  std_logic_vector(9 downto 0);
68                 ADDRB    : in  std_logic_vector(9 downto 0);
69                 CLKA     : in  std_ulogic;
70                 CLKB     : in  std_ulogic;
71                 DIA      : in  std_logic_vector(3 downto 0);
72                 DIB      : in  std_logic_vector(3 downto 0);
73                 ENA      : in  std_ulogic;
74                 ENB      : in  std_ulogic;
75                 RSTA     : in  std_ulogic;
76                 RSTB     : in  std_ulogic;
77                 WEA      : in  std_ulogic;
78                 WEB      : in  std_ulogic;
79
80                 DOA      : out std_logic_vector(3 downto 0);
81                 DOB      : out std_logic_vector(3 downto 0));
82     end RAMB4_S4_S4;
83
84     architecture Behavioral of RAMB4_S4_S4 is
85
86     function cv(A : bit) return std_logic is
87     begin
88        if (A = '1') then return '1';
89        else              return '0';
90        end if;
91     end;
92
93     function cv1(A : std_logic) return bit is
94     begin
95        if (A = '1') then return '1';
96        else              return '0';
97        end if;
98     end;
99
100    signal DATA : bit_vector(4095 downto 0) :=
101        INIT_0F & INIT_0E & INIT_0D & INIT_0C & INIT_0B & INIT_0A & INIT_09 & INIT_08 &
102        INIT_07 & INIT_06 & INIT_05 & INIT_04 & INIT_03 & INIT_02 & INIT_01 & INIT_00;
103
104    begin
105
106        process(CLKA, CLKB)
107        begin
108            if (rising_edge(CLKA)) then
109                if (ENA = '1') then
110                    DOA(3)      <= cv(DATA(conv_integer(ADDRA & "11")));
111                    DOA(2)      <= cv(DATA(conv_integer(ADDRA & "10")));
112                    DOA(1)      <= cv(DATA(conv_integer(ADDRA & "01")));
113                    DOA(0)      <= cv(DATA(conv_integer(ADDRA & "00")));
114                    if (WEA = '1') then
115                        DATA(conv_integer(ADDRA & "11"))      <= cv1(DIA(3));
116                        DATA(conv_integer(ADDRA & "10"))      <= cv1(DIA(2));
117                        DATA(conv_integer(ADDRA & "01"))      <= cv1(DIA(1));
118                        DATA(conv_integer(ADDRA & "00"))      <= cv1(DIA(0));
119                    end if;
120               end if;
```

```
121             end if;
122
123             if (rising_edge(CLKB)) then
124                 if (ENB = '1') then
125                     DOB(3)      <= cv(DATA(conv_integer(ADDRB & "11")));
126                     DOB(2)      <= cv(DATA(conv_integer(ADDRB & "10")));
127                     DOB(1)      <= cv(DATA(conv_integer(ADDRB & "01")));
128                     DOB(0)      <= cv(DATA(conv_integer(ADDRB & "00")));
129                     if (WEB = '1') then
130                         DATA(conv_integer(ADDRB & "11"))      <= cv1(DIB(3));
131                         DATA(conv_integer(ADDRB & "10"))      <= cv1(DIB(2));
132                         DATA(conv_integer(ADDRB & "01"))      <= cv1(DIB(1));
133                         DATA(conv_integer(ADDRB & "00"))      <= cv1(DIB(0));
134                     end if;
135                 end if;
136             end if;
137         end process;
138
139     end Behavioral;
140
```

test/RAMB4_S4_S4.vhd

29 LISTING OF test_tb.vhd

```
1   -------------------------------------------------------------------------------
2   --
3   -- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
4   --
5   --  This code is free software: you can redistribute it and/or modify
6   --  it under the terms of the GNU General Public License as published by
7   --  the Free Software Foundation, either version 3 of the License, or
8   --  (at your option) any later version.
9   --
10  --  This code is distributed in the hope that it will be useful,
11  --  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  --  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  --  GNU General Public License for more details.
14  --
15  --  You should have received a copy of the GNU General Public License
16  --  along with this code (see the file named COPYING).
17  --  If not, see http://www.gnu.org/licenses/.
18  --
19  -------------------------------------------------------------------------------
20  -------------------------------------------------------------------------------
21  --
22  -- Module Name:    alu - Behavioral
23  -- Create Date:    16:47:24 12/29/2009
24  -- Description:    arithmetic logic unit of a CPU
25  --
26  -------------------------------------------------------------------------------
27  --
28  library IEEE;
29  use IEEE.STD_LOGIC_1164.ALL;
30  use IEEE.STD_LOGIC_ARITH.ALL;
31  use IEEE.STD_LOGIC_UNSIGNED.ALL;
32
33  entity testbench is
34  end testbench;
35
36  architecture Behavioral of testbench is
37
38  component avr_fpga
39      port (  I_CLK_100   : in  std_logic;
40              I_SWITCH    : in  std_logic_vector(9 downto 0);
41              I_RX        : in  std_logic;
42
43              Q_7_SEGMENT : out std_logic_vector(6 downto 0);
44              Q_LEDS      : out std_logic_vector(3 downto 0);
45              Q_TX        : out std_logic);
46  end component;
47
48  signal L_CLK_100            : std_logic;
49  signal L_LEDS               : std_logic_vector(3 downto 0);
50  signal L_7_SEGMENT          : std_logic_vector(6 downto 0);
51  signal L_RX                 : std_logic;
52  signal L_SWITCH             : std_logic_vector(9 downto 0);
53  signal L_TX                 : std_logic;
54
55  signal  L_CLK_COUNT         : integer := 0;
56
57  begin
58
```

```
 59         fpga: avr_fpga
 60         port map(   I_CLK_100   => L_CLK_100,
 61                     I_SWITCH    => L_SWITCH,
 62                     I_RX        => L_RX,
 63
 64                     Q_LEDS      => L_LEDS,
 65                     Q_7_SEGMENT => L_7_SEGMENT,
 66                     Q_TX        => L_TX);
 67
 68         process -- clock process for CLK_100,
 69         begin
 70             clock_loop : loop
 71                 L_CLK_100      <= transport '0';
 72                 wait for 5 ns;
 73
 74                 L_CLK_100      <= transport '1';
 75                 wait for 5 ns;
 76             end loop clock_loop;
 77         end process;
 78
 79         process(L_CLK_100)
 80         begin
 81             if (rising_edge(L_CLK_100)) then
 82                 case L_CLK_COUNT is
 83                     when 0 => L_SWITCH      <= "0011100000";   L_RX <= '0';
 84                     when 2 => L_SWITCH(9 downto 8)      <= "11";
 85                     when others =>
 86                 end case;
 87                 L_CLK_COUNT      <= L_CLK_COUNT + 1;
 88             end if;
 89         end process;
 90     end Behavioral;
 91
```

test/test_tb.vhd

30 LISTING OF avr_fpga.ucf

```
 1    NET    I_CLK_100      PERIOD = 10 ns;
 2    NET    L_CLK          PERIOD = 35 ns;
 3
 4    NET    I_CLK_100      TNM_NET = I_CLK_100;
 5    NET    L_CLK          TNM_NET = L_CLK;
 6
 7    NET    I_CLK_100      LOC = AA12;
 8    NET    I_RX           LOC = M3;
 9    NET    Q_TX           LOC = M4;
10
11    # 7 segment LED display
12    #
13    NET    Q_7_SEGMENT       LOC = V3;
14    NET    Q_7_SEGMENT       LOC = V4;
15    NET    Q_7_SEGMENT       LOC = W3;
16    NET    Q_7_SEGMENT       LOC = T4;
17    NET    Q_7_SEGMENT       LOC = T3;
18    NET    Q_7_SEGMENT       LOC = U3;
19    NET    Q_7_SEGMENT       LOC = U4;
20
21    # single LEDs
22    #
23    NET    Q_LEDS            LOC = N1;
24    NET    Q_LEDS            LOC = N2;
25    NET    Q_LEDS            LOC = P1;
26    NET    Q_LEDS            LOC = P2;
27
28    # DIP switch(0 ... 7) and two pushbuttons (8, 9)
29    #
30    NET    I_SWITCH          LOC = H2;
31    NET    I_SWITCH          LOC = H1;
32    NET    I_SWITCH          LOC = J2;
33    NET    I_SWITCH          LOC = J1;
34    NET    I_SWITCH          LOC = K2;
35    NET    I_SWITCH          LOC = K1;
36    NET    I_SWITCH          LOC = L2;
37    NET    I_SWITCH          LOC = L1;
38    NET    I_SWITCH          LOC = R1;
39    NET    I_SWITCH          LOC = R2;
40
41    NET    I_SWITCH          PULLUP;
42
```

src/avr_fpga.ucf

31 LISTING OF Makefile

```
 1    PROJECT=avr_core
 2
 3    # the vhdl source files (except testbench)
 4    #
 5    FILES           += src/*.vhd
 6
 7    # the testbench sources and binary.
 8    #
 9    SIMFILES        = test/test_tb.vhd test/RAMB4_S4_S4.vhd
10    SIMTOP          = testbench
11
12    # When to stop the simulation
13    #
14    # GHDL_SIM_OPT  = --assert-level=error
15    GHDL_SIM_OPT    = --stop-time=40us
16
17    SIMDIR          = simu
18
19    FLAGS           = --ieee=synopsys --warn-no-vital-generic -fexplicit --std=93c
20
21    all:
22            make compile
23            make run 2>& 1 | grep -v std_logic_arith
24            make view
25
26    compile:
27            @mkdir -p simu
28            @echo -----------------------------------------------------------------
29            ghdl -i $(FLAGS) --workdir=simu --work=work $(SIMFILES) $(FILES)
30            @echo
31            @echo -----------------------------------------------------------------
32            ghdl -m $(FLAGS) --workdir=simu --work=work $(SIMTOP)
33            @echo
34            @mv $(SIMTOP) simu/$(SIMTOP)
35
36    run:
37            @$(SIMDIR)/$(SIMTOP) $(GHDL_SIM_OPT) --vcdgz=$(SIMDIR)/$(SIMTOP).vcdgz
38
39    view:
40            gunzip --stdout $(SIMDIR)/$(SIMTOP).vcdgz | gtkwave --vcd gtkwave.save
41
42    clean:
43            ghdl --clean --workdir=simu
44
```

Makefile

32 LISTING OF hello.c

```
 1     #include "stdint.h"
 2     #include "avr/io.h"
 3     #include "avr/pgmspace.h"
 4
 5     #undef F_CPU
 6     #define F_CPU 25000000UL
 7     #include "util/delay.h"
 8
 9
10          //----------------------------------------------------------------------//
11         //                                                                      //
12        //   print char cc on UART.                                             //
13       //    return number of chars printed (i.e. 1).                          //
14      //                                                                      //
15     //----------------------------------------------------------------------//
16     uint8_t
17     uart_putc(uint8_t cc)
18     {
19         while ((UCSRA & (1      <<UDRE)) == 0)      ;
20         UDR = cc;
21         return 1;
22     }
23
24          //----------------------------------------------------------------------//
25         //                                                                      //
26        //   print char cc on 7 segment display.                                //
27       //    return number of chars printed (i.e. 1).                          //
28      //                                                                      //
29     //----------------------------------------------------------------------//
30     // The segments of the display are encoded like this:
31     //
32     //
33     //      segment     PORT B
34     //      name        Bit number
35     //      ----A----   ----0----
36     //      |       |   |       |
37     //      F       B   5       1
38     //      |       |   |       |
39     //      ----G----   ----6----
40     //      |       |   |       |
41     //      E       C   4       2
42     //      |       |   |       |
43     //      ----D----   ----3----
44     //
45     //-----------------------------------------------------------------------------
46
47     #define SEG7(G, F, E, D, C, B, A)   (~(G      <
app/hello.c
```

33 LISTING OF make_mem.cc

```
 1     #include "assert.h"
 2     #include "stdio.h"
 3     #include "stdint.h"
 4     #include "string.h"
 5
 6     const char * hex_file = 0;
 7     const char * vhdl_file = 0;
 8
 9     uint8_t buffer[0x10000];
10
11     //---------------------------------------------------------------------------------
12     uint32_t
13     get_byte(const char *  cp)
14     {
15     uint32_t value;
16     const char cc[3] = { cp[0], cp[1], 0 };
17     const int cnt = sscanf(cc, "%X", &value);
18        assert(cnt == 1);
19        return value;
20     }
21     //---------------------------------------------------------------------------------
22     void
23     read_file(FILE * in)
24     {
25        memset(buffer, 0xFF, sizeof(buffer));
26     char line[200];
27        for (;;)
28            {
29              const char * s = fgets(line, sizeof(line) - 2, in);
30              if (s == 0)   return;
31              assert(*s++ == ':');
32              const uint32_t len     = get_byte(s);
33              const uint32_t ah      = get_byte(s + 2);
34              const uint32_t al      = get_byte(s + 4);
35              const uint32_t rectype = get_byte(s + 6);
36              const char * d = s + 8;
37              const uint32_t addr = ah      <<8 | al;
38
39              uint32_t csum = len + ah + al + rectype;
40              assert((addr + len)       <= 0x10000);
41              for (uint32_t l = 0; l      <len; ++l)
42                  {
43                    const uint32_t byte = get_byte(d);
44                    d += 2;
45                    buffer[addr + l] = byte;
46                    csum += byte;
47                  }
48
49              csum = 0xFF & -csum;
50              const uint32_t sum = get_byte(d);
51              assert(sum == csum);
52            }
53     }
54     //---------------------------------------------------------------------------------
55     void
56     write_vector(FILE * out, bool odd, uint32_t mem, uint32_t v)
57     {
58     const uint8_t * base = buffer;
```

```
59
60        // total memory is 2 even bytes, 2 odd bytes, 2 even bytes, ...
61        //
62        if (odd)   base += 2;
63
64        // total memory is 4 kByte organized into 8 memories.
65        // thus each of the 16 vectors covers 256 bytes.
66        //
67        base += v*256;
68
69        // memories 0 and 1 are the low byte of the opcode while
70        // memories 2 and 3 are the high byte.
71        //
72        if (mem >= 2)   ++base;
73
74     const char * px = odd ? "po" : "pe";
75        fprintf(out, "constant %s_%u_%2.2X : BIT_VECTOR := X\"", px, mem, v);
76        for (int32_t d = 63; d >= 0; --d)
77            {
78              uint32_t q = base[4*d];
79              if (mem & 1)   q >>= 4;     // high nibble
80              else           q &= 0x0F;   // low nibble
81              fprintf(out, "%X", q);
82            }
83
84        fprintf(out, "\";\r\n");
85     }
86     //-----------------------------------------------------------------------------
87     void
88     write_mem(FILE * out, bool odd, uint32_t mem)
89     {
90     const char * px = odd ? "po" : "pe";
91
92        fprintf(out, "-- content of %s_%u --------------------------------------"
93                     "---------------------------------------------\r\n", px, mem);
94
95        for (uint32_t v = 0; v       <16; ++v)
96            write_vector(out, odd, mem, v);
97
98        fprintf(out, "\r\n");
99     }
100    //-----------------------------------------------------------------------------
101    void
102    write_file(FILE * out)
103    {
104       fprintf(out,
105    "\r\n"
106    "library IEEE;\r\n"
107    "use IEEE.STD_LOGIC_1164.all;\r\n"
108    "\r\n"
109    "package prog_mem_content is\r\n"
110    "\r\n");
111
112       for (uint32_t m = 0; m       <4; ++m)
113           write_mem(out, false, m);
114
115       for (uint32_t m = 0; m       <4; ++m)
116           write_mem(out, true,  m);
117
118       fprintf(out,
119    "end prog_mem_content;\r\n"
120    "\r\n");
```

```
121     }
122     //-----------------------------------------------------------------------------
123     int
124     main(int argc, char * argv[])
125     {
126        if (argc > 1)   hex_file = argv[1];
127        if (argc > 2)   vhdl_file = argv[2];
128
129     FILE * in = stdin;
130        if (hex_file)   in = fopen(hex_file, "r");
131        assert(in);
132        read_file(in);
133        fclose(in);
134
135     FILE * out = stdout;
136        if (vhdl_file)   out = fopen(vhdl_file, "w");
137        write_file(out);
138        assert(out);
139     }
140     //-----------------------------------------------------------------------------
```

tools/make_mem.cc

34 LISTING OF end_conv.cc

```
 1      #include "assert.h"
 2      #include "ctype.h"
 3      #include "stdio.h"
 4      #include "string.h"
 5
 6      //-----------------------------------------------------------------------------
 7      int
 8      main(int argc, const char * argv)
 9      {
10      char buffer[2000];
11      int pc, val, val2;
12
13         for (;;)
14             {
15               char * s = fgets(buffer, sizeof(buffer) - 2, stdin);
16               if (s == 0)   return 0;
17
18               // map lines '  xx:' and 'xxxxxxxx; to 2* the hex value.
19               //
20               if (
21                   (isxdigit(s[0]) || s[0] == ' ') &&
22                   (isxdigit(s[1]) || s[1] == ' ') &&
23                   (isxdigit(s[2]) || s[2] == ' ') &&
24                    isxdigit(s[3]) && s[4] == ':')   // '  xx:'
25                  {
26                    assert(1 == sscanf(s, " %x:", &pc));
27                    if (pc & 1)       printf("%4X+:", pc/2);
28                    else              printf("%4X:", pc/2);
29                    s += 5;
30                  }
31               else if (isxdigit(s[0]) && isxdigit(s[1]) && isxdigit(s[2]) &&
32                        isxdigit(s[3]) && isxdigit(s[4]) && isxdigit(s[5]) &&
33                        isxdigit(s[6]) && isxdigit(s[7]))             // 'xxxxxxxx'
34                  {
35                    assert(1 == sscanf(s, "%x", &pc));
36                    if (pc & 1)   printf("%8.8X+:", pc/2);
37                    else          printf("%8.8X:", pc/2);
38                    s += 8;
39                  }
40               else                              // other: copy verbatim
41                  {
42                    printf("%s", s);
43                    continue;
44                  }
45
46                while (isblank(*s))   printf("%c", *s++);
47
48                // endian swap.
49                //
50                while (isxdigit(s[0]) &&
51                       isxdigit(s[1]) &&
52                                s[2] == ' ' &&
53                       isxdigit(s[3]) &&
54                       isxdigit(s[4]) &&
55                                s[5] == ' ')
56                   {
57                    assert(2 == sscanf(s, "%x %x ", &val, &val2));
58                    printf("%2.2X%2.2X  ", val2, val);
```

```
59                      s += 6;
60                     }
61
62                char * s1 = strstr(s, ".+");
63                char * s2 = strstr(s, ".-");
64                if (s1)
65                   {
66                     assert(1 == sscanf(s1 + 2, "%d", &val));
67                     assert((val & 1) == 0);
68                     sprintf(s1, " 0x%X", (pc + val)/2 + 1);
69                     printf(s);
70                     s = s1 + strlen(s1) + 1;
71                   }
72                else if (s2)
73                   {
74                     assert(1 == sscanf(s2 + 2, "%d", &val));
75                     assert((val & 1) == 0);
76                     sprintf(s2, " 0x%X", (pc - val)/2 + 1);
77                     printf(s);
78                     s = s2 + strlen(s2) + 1;
79                   }
80
81                printf("%s", s);
82              }
83      }
84      //-----------------------------------------------------------------------------
```

tools/end_conv.cc