The NEO430 Processor

by Dipl.-Ing. Stephan Nolting

A very small, powerful and highly customizable
open-source 16-bit soft-core microcontroller,
compatible to TI's MSP430© ISA

Processor Hardware Version: 0x0320

Proprietary / Legal Notice

License / Disclaimer

A big shout out to all the ones, who contributed to this project! =)

The **NEO430 Processor** Project
© 11/19 Dipl.-Ing. Stephan Nolting, Hannover, Germany
For any kind of feedback, feel free to drop me a line: stnolting@gmail.com

The most recent version of the NEO430Processor project and the according documentary can be found at
https://github.com/stnolting/neo430

Table of Content

1. Introduction

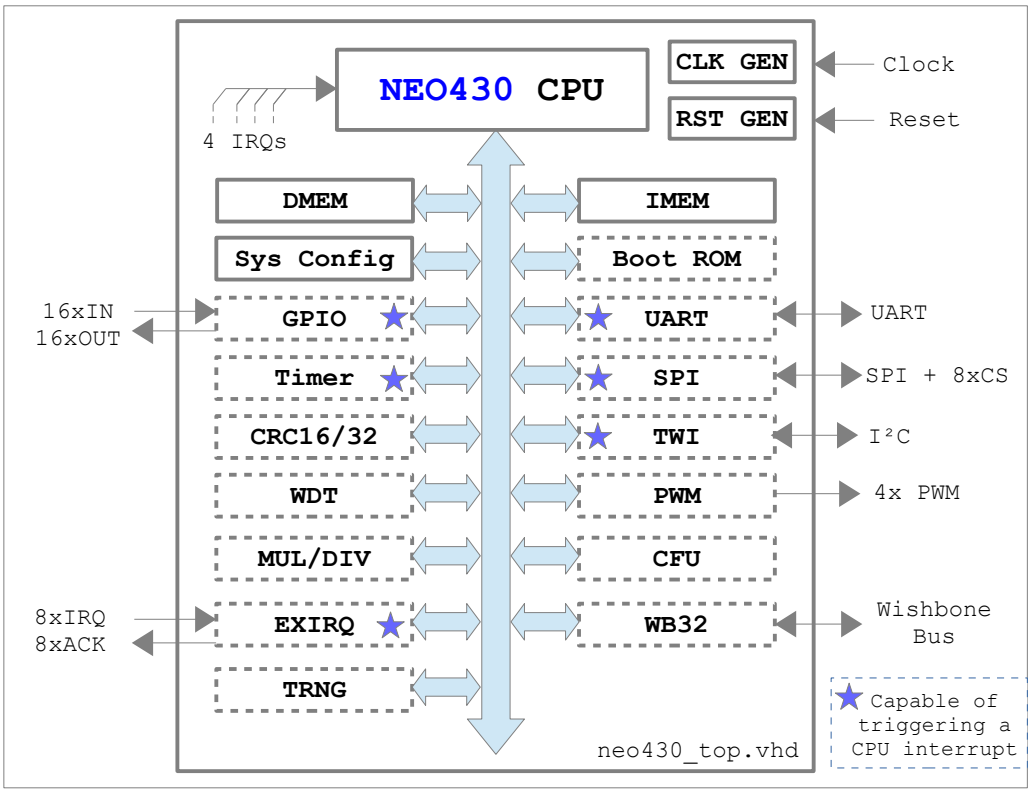

Figure 1: NEO430 processor block diagram, optional modules are marked using dashed lines

Welcome to **The NEO430 Processor** project!

You need a small but still powerful, customizable and microcontroller-like processor system for your next FPGA design? Then the NEO430 is the right choice for you!

This processor is based on the Texas Instruments MSP430 ISA and provides 100% compatibility with the original instruction set. The processor features a very small outline, already implementing standard features like a timer, a watchdog, UART, TWI and SPI serial interfaces, general purpose IO ports, an internal bootloader and of course internal memory for program code and data. All of the peripheral modules are optional – so if you do not need them, you can just exclude them from implementation to reduce the footprint of the system. Any additional modules, which make a more customized system, can be connected via a Wishbone-compatible bus interface or you can add them as custom functions unit to the processor core itself. By this you can build a system that perfectly fits your needs.

The high-level software development is based on the free TI msp430-gcc compiler tool chain. You can either use Windows Powershell, the Windows Bash Subsystem or directly Linux as build environment for your applications. The example software folder of this project features several demo programs from which you can start creating your own NEO430 applications.

This project is intended to work "out of the box". Just synthesize the test setup from this project, upload it to your FPGA board and start exploring the capabilities of the NEO430 processor. Application program generation (and even ROM-installation) works by executing a single "make" command. Jump to **4. Let's Get It Started!**, where a lot of guides and tutorials are provided to make your first NEO430 setup run.

1.1. Processor Features

- ✔ 16-bit open source soft-core microcontroller-like processor system
- ✔ Code-efficient RISC-like ISA with powerful CISC-like addressing capabilities
- ✔ Full support of the original MSP430 instruction set architecture
- ✔ Tool chain based on free TI msp430-gcc compiler
- ✔ Application compilation scripts for Windows Powershell, Windows Subsystem for Linux & Linux
- ✔ Completely described in behavioral, platform-independent VHDL (no macros, primitives, attributes, etc. used), tested on Intel, Xilinx and Lattice FPGAs
- ✔ Fully synchronous design[1], no latches, no gated clocks
- ✔ Very low hardware resource requirements and high operating frequency
- ✔ Internal data (**DMEM**) and instruction memories (**IMEM**) with user-configurable sizes
- ✔ One external interrupt line with acknowledge signal
- ✔ Customizable processor hardware configuration
- ✔ Optional multiplier and divider unit (**MULDIV**)
- ✔ Optional high-precision timer (**TIMER**)
- ✔ Optional universal asynchronous receiver and transmitter unit (**UART**)
- ✔ Optional serial peripheral interface master (**SPI**)
- ✔ Optional two wire serial interface master (**TWI**), compatible to the I²C standard
- ✔ Optional general purpose parallel IO port (**GPIO**), 16 inputs & 16 outputs with pin-change interrupt
- ✔ Optional 32-bit Wishbone bus interface adapter (**WB32**) – including bridges to Avalon and AXI-Lite
- ✔ Optional watchdog timer (**WDT**)
- ✔ Optional custom functions unit (**CFU**) to implement user-defined extensions
- ✔ Optional 16 or 32 bit cyclic redundancy checksum computation unit (**CRC16/32**)
- ✔ Optional 4 channel PWM controller with 4 or 8 bit resolution (**PWM**)
- ✔ Optional GARO-based true random number generator (**TRNG**)
- ✔ Optional 8 channel external interrupts controller (**EXIRQ**)
- ✔ Optional internal bootloader (**BOOTLD**) with UART or SPI EEPROM boot option

1 Except for the true random number generator.

1.2. Main Differences to TI's Original MSP430 Architecture

Since the NEO430 is not intended as a MSP430 processor clone, there are severe differences to TI's original MSP430 product lines. The *main* differences are:

- ✗ Completely different processor peripheral modules with different functionality
- ✗ No *"compiler support"* of the hardware multiplier/divider (the multiplier/divider unit can still be utilized by using specific C functions provided by the NEO430 C-library)
- ✗ Maximum of 48kB instruction memory and 12kB of data memory
- ✗ NEO430 tool chain (makefiles, boot-code and linker script) required
- ✗ Custom binary executable format
- ✗ No hardware debugging interface
- ✗ No analog components
- ✗ No support of TI's Code Composer Studio
- ✗ No support of the CPU's DADD instruction (BCD addition) by default. However, the instruction can be enabled in the processor's package
- ✗ Just 4 CPU interrupt channels
- ✗ Single clock domain for complete processor
- ✗ Different numbers of instruction execution cycles
- ✗ Only one power-down (sleep) mode
- ✗ Internal bootloader with user interface (via UART serial port)
- ✗ Extended ALU functions (if enabled, disabled by default)

1.3. Project Folder Structure

```
|
├─doc              This folder contains a copy of the implemented Wishbone
| |                specifications as well as the processor documentary (the document you
| |                are currently reading).
| |
| └─figures        Some figures for the web page.
|
├─rtl              All the rtl files of the project can be found here.
| |
| ├─core           This folder contains all the rtl (VHDL) core files of the NEO430
| |                processor. Make sure to add ALL of them to your FPGA EDA project.
| |
| └─top_templates  Here you can find several different exemplary top entities of the
|                  NEO430 (like a simple system top entity that you can use for your
|                  first FPGA test implementation of the NEO430 processor or a top
|                  entity using only resolved std_logic signal types).
|
├─sim              The sim folder contains a simple VHDL testbench and additional
| |                simulation files.
| |
| ├─ISIM           Here you can find a default Xilinx ISIM/Vivado simulator waveform
| |                configuration file.
| |
| └─modelsim       Waveform configuration file and simulation compilation script for
|                  Mentor Graphic's ModelSim.
|
└─sw               The software folder contains example programs, software libraries,
  |                compilation scripts and of course several example code projects to
  |                start from.
  |
  ├─bootloader     Sources and compilation scripts of the NEO430-internal bootloader.
  |
  ├─common         Application linker script and CPU startup code.
  |
  ├─example        Here you can find several example programs. Each project folder
  | |              includes the program's C sources and a makefile for compilation. Add
  | └─...          your own projects to this folder.
  |
  |
  ├─lib            This folder contains different C software libraries.
  | |
  | └─neo430       Here you can find the NEO430 library's C source files as well as the
  |   |            according include files.
  |   └─src/inc
  |
  └─tools          This folder contains helper programs.
    |
    └─image_gen    This program either generates an executable binary (for uploading via
                   the bootloader) or an executable VHDL ROM initialization image for
                   the bootloader ROM or the actual application ROM/RAM.
```

Do not change the project's folder structure unless you really know what you are doing. Changing the structure might corrupt the file dependencies.

1.4. Processor VHDL File Hierarchy

All necessary VHDL hardware description files are located in the project's **`rtl/core`** folder. The top entity of the entire processor including all the required configuration generics is **`neo430_top.vhd`**. Make sure to add all files to your project and assign them to a **library** called "**`neo430`**".

```
neo430_top.vhd                  Processor core top entity
|
├─neo430_boot_rom.vhd           Bootloader ROM
| |
| └─neo430_bootloader_image.vhd Boot ROM initialization image for the bootloader[2]
|
├─neo430_cfu.vhd                Custom functions units for user-defined extension
|
├─neo430_crc.vhd                Checksum computation unit (CRC16/32)
|
├─neo430_dmem.vhd               DMEM: Internal RAM for storing data
|
├─neo430_exirq.vhd              External interrupts controller
|
├─neo430_gpio.vhd               General purpose parallel IO port
|
├─neo430_imem.vhd               IMEM: Internal RAM/ROM for the application code
| |
| └─neo430_application_image.vhd IMEM application initialization image[3]
|
├─neo430_muldiv.vhd             Multiplier and divider unit
|
├─neo430_package.vhd            Processor VHDL package file
|
├─neo430_pwm.vhd                Pulse-width modulation controller
|
├─neo430_spi.vhd                Serial peripheral interface
|
├─neo430_sysconfig.vhd          IRQ vector configuration and system information
|
├─neo430_timer.vhd              High-precision timer
|
├─neo430_trng.vhd               True random number generator
|
├─neo430_twi.vhd                Two wire serial interface
|
├─neo430_uart.vhd               Universal asynchronous receiver/transmitter
|
├─neo430_wb_interface.vhd       Wishbone bus adapter
|
├─neo430_wdt.vhd                Watchdog timer
|
└─neo430_cpu.vhd                CPU's top entity
  |
  ├─neo430_addr_gen.vhd         Address generator unit
  |
  ├─neo430_alu.vhd              Arithmetic/logic unit
  |
  ├─neo430_control.vhd          CPU control finite state machine
  |
  └─neo430_reg_file.vhd         Register file
```

2 Auto generated by the bootloader compile scripts.
3 Auto generated by the application compile scripts.

1.5. Processor Top Entity – Signals

The following table shows all interface ports of the processor top entity (**neo430_top.vhd**). The type of all signals is **std_ulogic** or **std_ulogic_vector**, respectively (except for the TWI signals). If you need a top entity with resolved signals, take a look at the alternative top entities in the **rtl/top_templates** folder. Make sure to tie all unused inputs to zero and leave all unused outputs open.

Signal name	Width	Direction	Function	Module
Global Control				
clk_i	1	Input	Global clock line, all registers triggering on rising edge	global
rst_i	1	Input	Global reset, low-active	global
General Purpose Inputs & Outputs				
gpio_o	16	Output	General purpose parallel output[4]	GPIO
gpio_i	16	Input	General purpose parallel input	GPIO
PWM Channels				
pwm_o	4	Output	Pulse Width Modulation channels	PWM
Serial Communication				
uart_txd_o	1	Output	UART serial transmitter	UART
uart_rxd_i	1	Input	UART serial receiver	UART
spi_sclk_o	1	Output	SPI master clock line	SPI
spi_mosi_o	1	Output	SPI serial data output	SPI
spi_miso_i	1	Input	SPI serial data input	SPI
spi_cs_o	8	Output	SPI dedicated chip select lines 0..7[5] (low-active)	SPI
twi_sda_io	1	InOut	TWI serial data line	TWI
twi_scl_io	1	InOut	TWI serial clock line	TWI
Wishbone Bus (on-chip)				
wb_adr_o	32	Output	Slave address	WISHBONE
wb_dat_i	32	Input	Write data	WISHBONE
wb_dat_o	32	Output	Read data	WISHBONE
wb_we_o	1	Output	Write enable ('0' = read transfer)	WISHBONE
wb_sel_o	4	Output	Byte enable	WISHBONE
wb_stb_o	1	Output	Strobe	WISHBONE
wb_cyc_o	1	Output	Valid cycle	WISHBONE
wb_ack_i	1	Input	Transfer acknowledge	WISHBONE
External Interrupt Request				
ext_irq_i	8	Input	Interrupt request signals, high-level or rising-edge trigger	EXIRQ
ext_ack_o	8	Output	Interrupt request acknowledges, single-shot	EXIRQ

Table 1: neo430_top.vhd – processor's top entity interface ports

4 Bit #0 is used by the bootloader to drive a high-active status LED.
5 Chip select #0 is used by the bootloader to access the external boot SPI EEPROM.

1.6. Processor Top Entity – Configuration Generics

The following table shows the configuration generics of the processor top entity (**neo430_top.vhd**).

Generic name	Type	Default	Function
General Configuration			
CLOCK_SPEED	natural	100000000	Clock speed of signal clk_i in Hz (Hertz)
IMEM_SIZE	natural	4*1024	Size of internal instruction memory (max 48kB) in bytes
DMEM_SIZE	natural	2*1024	Size of internal data memory (max 12kB) in bytes
Additional Configuration			
USER_CODE	std_ulogic_vector	x"0000"	16-bit custom user code, can be read by application software
Module Configuration			
MULDIV_USE	boolean	**true**	Implement multiplier/divider unit
WB32_USE	boolean	**true**	Implement Wishbone interface adapter
WDT_USE	boolean	**true**	Implement watchdog timer
GPIO_USE	boolean	**true**	Implement parallel GPIO port
TIMER_USE	boolean	**true**	Implement high-precision timer
UART_USE	boolean	**true**	Implement UART serial communication unit
CRC_USE	boolean	**true**	Implement checksum computation unit
CFU_USE	boolean	**false**	Implement custom functions unit
PWM_USE	boolean	**true**	Implement pulse width controller
TWI_USE	boolean	**true**	Implement two wire serial interface unit
SPI_USE	boolean	**true**	Implement serial peripheral interface unit
TRNG_USE	boolean	**false**	Implement true random number generator
EXIRQ_USE	boolean	**true**	Implement external interrupts contoller
Boot Configuration			
BOOTLD_USE	boolean	**true**	Implement and auto start internal bootloader
IMEM_AS_ROM	boolean	**false**	Implement internal instruction memory as read-only memory

Table 2: neo430_top.vhd – processor's top entity configuration generics

1.7. Alternative Top Entities

Besides the actual top entity of the processor (**rtl/core/neo430_top.vhd**), there are several other entities that can be used instead. These alternative top entities are located in: **rtl/top_templates**

Top entity file	Description
neo430_test.vhd	This setup is meant as “hello world” test project for your first contact with the neo430 processor. This setup is used in the Let’s Get It Started tutorial.
neo430_top_avm.vhd	This top entity features an Avalon master interface, which is generated by converting the processor’s Wishbone bus. All signals are **std_logic**.
neo430_top_std_logic.vhd	Same as the original top entity, but using only **std_logic** signal types.
neo430_top_axi_lite.vhd	This top entity features an AXI-Lite-compatible master itnerface, which is generated by converting the processor’s Wishbone bus. All signals are **std_logic**.

Table 3: Alternative top entities

1.8. FPGA Implementation Results

This chapter shows some exemplary implementation results of the NEO430 processor for different FPGA platforms, EDA tool chains and configurations.

1.8.1. Full Implementation (Default Configuration)

Configuration

Hardware Version: **0x0320**

- IMEM_SIZE: **4*1024**
- DMEM_SIZE: **2*1024**
- MULDIV_USE: **true**
- WB32_USE: **true**
- WDT_USE: **true**
- GPIO_USE: **true**
- TIMER_USE: **true**
- UART_USE: **true**
- CRC_USE: **true**
- CFU_USE: **false**
- PWM_USE: **true**
- TWI_USE: **true**
- SPI_USE: **true**
- TRNG_USE: **false**
- EXIRQ_USE: **true**
- BOOTLD_USE: **true**
- IMEM_AS_ROM: **false**

FPGA Tools

- Intel Quartus Prime Lite 17.0 (“balanced implementation”)
- Xilinx Vivado 2017.3, default strategies
- Lattice Radiant 1.0 (Synplify)

Implementation Results

Resource	Altera Cyclone IV *EP4CE22F17C6N*	Xilinx Artix-7 *XC7a35TICS324-1L*	Lattice iCE40 UltraPlus *iCE40UP5K-SG48I*
LUTs/LEs:	1648 / 22320 (7%)	983 / 20800 (4.7%)	2600 / 5280 (49%)
FFs/ Registers:	990 / 22320 (4%)	1014 / 41600 (2.35%)	1152 / 5280 (21%)
Total memory bits / Block RAMs / EBRs:	65800 / 608256 (11%)	2.5 / 50 (5%)	16 / 30 (53%)
DSP-Blocks:	0	0	0
Maximum Frequency:	122 MHz (slow 1200mV 0°C model)	100 MHz (constrained)	20 MHz (constrained)

Table 4: Hardware utilization – full / default configuration

1.8.2. Minimal Configuration

This is the minimal configuration of the NEO430 processor that is still able to do “useful” stuff.

Configuration

Hardware Version: **0x0320**

• IMEM_SIZE:	**4*1024**	• CFU_USE:	**false**
• DMEM_SIZE:	**2*1024**	• PWM_USE:	**false**
• MULDIV_USE:	**false**	• TWI_USE:	**false**
• WB32_USE:	**false**	• SPI_USE:	**false**
• WDT_USE:	**false**	• TRNG_USE:	**false**
• GPIO_USE:	**true**	• EXIRQ_USE:	**false**
• TIMER_USE:	**false**	• BOOTLD_USE:	**false**
• UART_USE:	**false**	• IMEM_AS_ROM:	**true**
• CRC_USE:	**false**		

FPGA Tools

- Intel Quartus Prime Lite 17.0 (“balanced implementation”)
- Xilinx Vivado 2017.3, default strategies
- Lattice Radiant 1.0 (Synplify)

Implementation Results

Resource	**Altera Cyclone IV** ***EP4CE22F17C6N***	**Xilinx Artix-7** ***XC7a35TICS324-1L***	**Lattice iCE40 UltraPlus** ***iCE40UP5K-SG48I***
LUTs/LEs:	596 / 22320 (3%)	685 / 20800 (3.3%)	1365 / 5280 (25%)
FFs/ Registers:	233 / 22320 (1%)	290 / 41600 (0.7%)	493 / 5280 (9%)
Total memory bits / Block RAMs / EBRs:	49408 / 608256 (8%)	1 / 50 (2%)	13 / 30 (40%)
DSP-Blocks:	0	0	0
Maximum Frequency:	126 MHz (slow 1200mV 0°C model)	100 MHz (constrained)	20 MHz (constrained)

Table 5: Hardware utilization – minimal configuration

1.8.3. Resource Utilization by Entity

This table shows the required resources for each entity of the processor system. Logic functions of different modules might be merged between entity boundaries, so the total number might vary a bit.

Configuration

Hardware Version: **0x0320**

- IMEM_SIZE: **4*1024**
- DMEM_SIZE: **2*1024**
- MULDIV_USE: **true**
- WB32_USE: **true**
- WDT_USE: **true**
- GPIO_USE: **true**
- TIMER_USE: **true**
- UART_USE: **true**
- CRC_USE: **true**
- CFU_USE: **false**
- PWM_USE: **true**
- TWI_USE: **true**
- SPI_USE: **true**
- TRNG_USE: **true**
- EXIRQ_USE: **true**
- BOOTLD_USE: **true**
- IMEM_AS_ROM: **false**

Implementation Results

Altera Cyclone IV EP4CE22F17C6N, Intel Quartus Prime Lite 17.0 (“balanced implementation”)					
Entity/Module	**Function**	**LEs**	**FFs**	**MEM bits**	**DSPs**
CPU	Central processing unit	506	171	256	0
IMEM (4kB)	Instruction memory (RAM)	4	1	32768	0
DMEM (2kB)	Data memory (RAM)	6	1	16384	0
Boot ROM (2kB)	Bootloader ROM	2	1	16384	0
SYSCONFIG	System information	15	13	0	0
GPIO	GPIO parallel in/out ports	49	45	0	0
MULDIV	Multiplier/divider unit	184	131	0	0
WDT	Watchdog timer	49	36	0	0
TIMER	High-precision timer	70	55	0	0
UART	Universal asynchronous RX & TX	129	89	0	0
WB32	32-bit Wishbone bus interface	128	117	0	0
CRC	Cyclic redundancy checksum unit	110	94	0	0
CFU	Custom functions unit	-	-	-	-
PWM	Pulse width modulation unit	80	66	0	0
TWI	Two wire serial interface	80	41	0	0
SPI	Serial peripheral interface	57	43	0	0
TRNG	True random number generator	44	36	0	0
EXIRQ	External Interrupts Controller	73	60	0	0

Table 6: Hardware utilization by entity

2. Hardware Architecture

The NEO430 processor system is constructed from the CPU itself and several different memory and peripheral modules. This chapter takes a closer look at these modules and their specific functionality.

Address Space

Although the NEO430 is fully compatible to the original TI MSP430 instruction set architecture, the implemented modules are completely different from the original design. Hence, the provided modules and the resulting address space layout are completely new. The figure below shows the general layout of the 16-bit address space of the CPU.

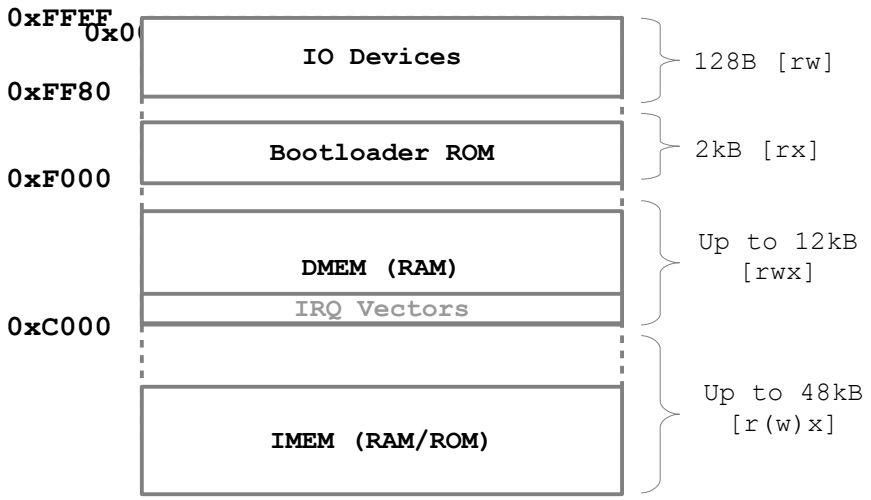

Figure 2: General NEO430 address space layout

In general, the address space in separated into four groups: At the beginning of the address space the instruction memory (**IMEM**) is located. This memory (can be implemented as RAM or ROM) stores the instructions of the actual application. The data memory (**DMEM**) starts in the middle of the address space. This memory stores global variables, the stack and the heap. Additionally, the interrupt vectors are located at the beginning of the DMEM. The custom NEO430 linker script ensures that these locations are not used by the application program. Close to the end of the address space the bootloader ROM can be found. This optional memory contains the image of the interactive bootloader. Finally, all the IO devices (timer, UART, GPIO, …) are located at the very end of the address space.

Peripheral/IO Devices

In contrast to the original MSP430, the NEO430 does not have any special function registers at the beginning of the memory space. Instead, all 'special functions' – like peripheral/IO devices, control registers and interrupt enable configurations – are located inside the according hardware units. These IO units (devices) are located at the end of the memory space in the so-called *IO region*. This region is 128 bytes large. A special linker script as well as the dedicated NEO430 software library abstract the specific memory layout for the user. All IO devices are reset by software by writing zero to the according unit's control register.

Separated Instruction (IMEM) and Data (DMEM) Memories

Just like the original MSP430, the NEO430 uses separated memories for storing data and instructions, but both memories are accessed via the same bus. The DMEM is implemented as normal RAM, the IMEM is also implemented as RAM, but one can only write to it if a special bit in the CPU's status register is set. Normally, this bit is only set by the bootloader for transferring an image from an external flash/EEPROM or via the UART into the IMEM during the boot process. Alternatively, the IMEM can be implemented as true ROM (via the `IMEM_AS_ROM` generic). In this case, the actual executable application image is included during the synthesis process and persists as non-volatile image in the IMEM. Thus, a bootloader ROM is no longer required (only for development purpose, maybe).

Word and Byte Accesses

All internal memories (IMEM, DMEM, bootloader ROM) can be accesses in byte and word mode. All of the peripheral devices in the IO region can only be accessed using full-word mode.

All IO modules (peripheral modules) can only be accessed in full-word mode (full 16-bit accesses).

Internal Reset Generator

All processor-internal modules – except for the CPU and the watchdog timer – do not require a dedicated reset signal. However, all devices can be reset by software by clearing the corresponding unit's control register. The automatically included application start-up code will perform such a software-reset of all modules to ensure a clean system reset state.

The hardware reset signal of the processor can either be triggered via the external reset pin (**LOW-active**) or by the internal watchdog timer (if implemented). Before the external reset signal is applied to the system, it is filtered (so no spike can generate a reset, a minimum active reset period of one clock cycle is required) and extended to have a minimal duration of four clock cycles.

Internal Clock Generator

An internal clock divider generates 8 clock signals derived from the main clock input. These derived clock signals are not actual *clock signals*. Instead, they are derived from a simple counter and are used as "clock enable" signal by the different processor modules. Thus, the whole design operates using only the main clock signal (single clock domain). Some of the processor modules (like the timer or the UART) can select one of the derived clock enabled signals for their internal operation. If none of the connected modules require a clock signal from the generator, the clock divider is automatically deactivated to reduce dynamic power.

The available clock frequencies from the clock generator can be used by many modules (like the TWI, SPI, UART, Timer and WDT) by setting a certain 3-bit prescaler in the module's control register. The mapping of the selection bits to the actually obtained clock are shown in the table below. "f" represents the main clock.

Prescaler bits:	000	001	010	011	100	101	110	111
Resulting clock	f/2	f/4	f/8	f/64	f/128	f/1024	f/2048	f/4096

2.1. NEO430 CPU

The CPU is the heart of the NEO430 processor. It implements all the instructions, emulated instructions and addressing modes of the original TI MSP430 instruction set architecture (ISA). There are small differences to the original architecture when it comes to instruction execution cycles, status register bits, power down modes and interrupt behavior.

Data Path

Instruction execution is conducted by performing several tiny steps – so-called *micro operations*. Thus, the NEO430 implements a multi-cycle architecture: The CPU requires several consecutive cycles to complete a single instruction. An accurate listing of the required processing cycles for each instruction is given in the following chapter. The execution of the micro operations is controlled by the central control arbiter, which implements a complex finite state machine (FSM). This FSM generates the control signals for the data path, that processes the data. This data path is constructed from the register file, the primary data ALU and the address generator unit. The image below shows the simplified architecture of this data path.

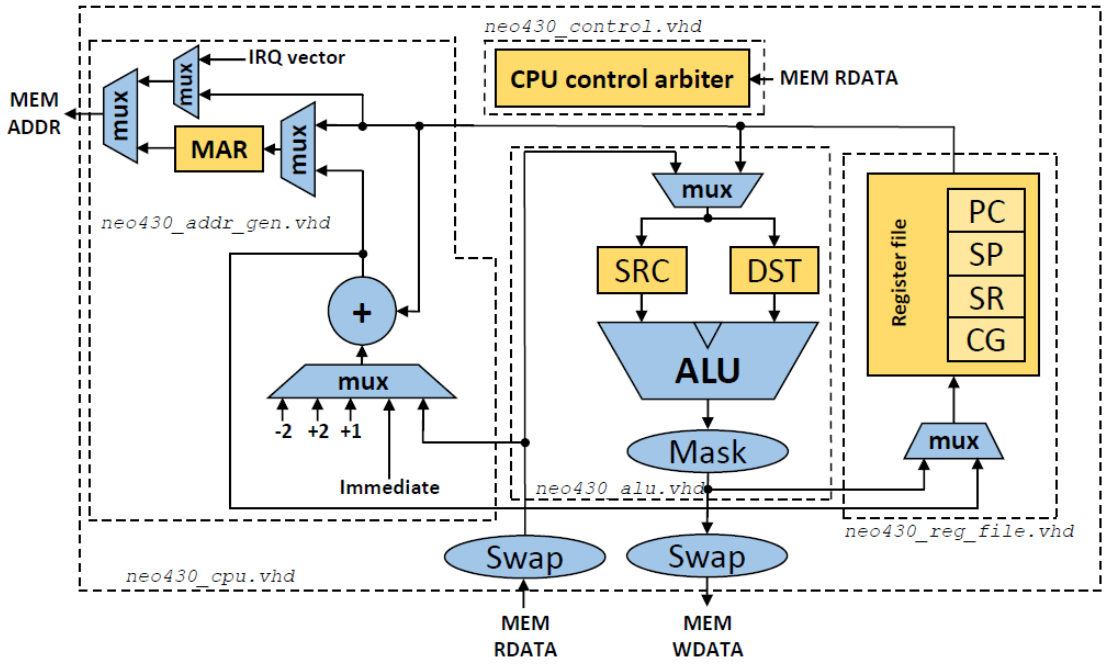

Figure 3: Data path of the NEO430 CPU

Control Path

The NEO430 CPU was implemented as multi-cycle architecture, requiring several consecutive clock cycles to process a single instruction. Obviously, this kind of implementation scheme reduces the overall performance when compared to a pipelined one. There are two main points for this design decision: First, a multi cycle architecture generally requires less logic since you don not need any pipeline conflict handling (e.g., forwarding). Second, the MSP430 ISA uses CISC-like features, which makes it nearly impossible to implement in using a classic 5-stage pipeline (e.g., single instructions requiring 3 memory accesses).

2.1.1. Register File

The NEO430 provides a register file with 16 registers, each having a data width of 16 bit. Each register can be accessed in 16-bit or 8-bit mode, but always a full 16-bit word is written back. The first four registers have a special purpose:

- R0: Program counter
- R1: Stack pointer, managed by hardware, can also be used as general purpose data register
- R2: Status register (see later)
- R4: Constant generator; not an actual register, writing data to this register has no effect

The remaining 12 registers can be used as general purpose data registers.

2.1.2. Arithmetic / Logic Unit

The ALU is the data processing core of the processor. It implements arithmetical, logical, transfer, exchange, test and compare operations. All these operations are derived from the original TI MSP430. Most of the ALU operations generate some kind of status information that is stored to the status register.

Extended ALU functions

The NEO430 ALU provides some features/function which are not part of the original MSP430 processor ISA. At the moment, the only extended ALU function is the computation of a parity flag conducted for each ALU operation (including MOVs). For this, all result bits are XNORed and the result is stored to a normally reserved bit in the status register (see next chapter). The neo430_cpu.h library files provides a function to easily utilize this feature.

By default, these option are excluded from synthesis but can be enabled via the processor's VHDL package file (rtl\core\neo430_package.vhd):

```
constant use_xalu_c : boolean := false; -- implement extended ALU function
```

Binary-Coded Decimal Addition (DADD instruction)

By default, the binary coded decimal addition operation (DADD) is disabled, since it requires a lot of hardware for implementation, increases the critical path is rarely used. And by saying "rarely" I mean never. The compiler won't generate this command if it is not explicitly used by inline assembly – at least that is what I have noticed.

However, the required DADD logic is already present in the core's ALU, but is disabled by default. If you want to enable the implementation of the DADD instruction, you can activate it in the processor's package file (rtl\core\neo430_package.vhd):

```
constant use_dadd_cmd_c : boolean := false; -- implement CPU's DADD instruction
```

Compiler Warning Regarding DADD

The application compilation makefiles will output a warning when the DADD instruction might be used by a program.

```
NEO430: WARNING! 'DADD' instruction might be used! Make sure it is synthesized!
```

The makefile only scans the generated assembly listing file for the "DADD" keyword. If the makefile outputs this warning, there might be a DADD instruction in the actual program code. However, it is more likely that a part from the read-only section of the program (like a string or other constants) was interpreted by the disassembler as DADD instruction. In case of doubt check the assembly listing file *.s by yourself.

2.1.3. Status Register

The status register (SR = R2) represents the ALU execution status flags and CPU control flags. The carry **C**, zero **Z**, negative **N** and the overflow **V** flags correspond to the result of the last ALU operation.

Via the **I** flag interrupts can be globally activated or deactivated. If this flag is cleared, all further interrupt requests to the CPU are queued and finally executed when the **I** flag is set again. When setting the **Q** flag, all pending interrupts in the CPU's interrupt request queue are deleted. This flag is write-only (read as zero) and automatically clears after being set.

The **S** flag is used to bring the CPU into power-down (sleep) mode. When this flag is set, the CPU is completely deactivated while all processor modules – like the timer – keep operating. An interrupt request from any IRQ channel will reactivate the CPU, clears the **S** flag and the processor resumes operations with the next consecutive instruction.

The **R** flag is used to control write access to the internal instruction memory (IMEM). When set, the IMEM behaves as a RAM, otherwise the IMEM behaves like a true read-only memory. If the IMEM is implemented as true ROM (`IMEM_AS_ROM` generic), this flag is always zero and no write access to the IMEM is possible at all. All other bits of the status register do not have a specific function yet. Hence, they are reserved for future use and should not be used and are always read as zero.

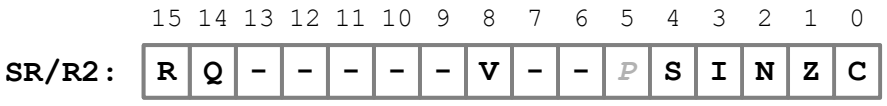

Figure 4: Processor status register

Bit#	Name	R/W	Function
0	**C_FLAG**	R/W	**C**arry flag
1	**Z_FLAG**	R/W	**Z**ero flag
2	**N_FLAG**	R/W	**N**egative flag
3	**I_FLAG**	R/W	Global **I**nterrupt enable
4	**S_FLAG**	R/W	**S**leep mode (CPU off)
5	**P_FLAG**	R/W	**P**arity flag[6] (see previous. chapter)
6..7	-	R/-	*Reserved, read as 0*
8	**V_FLAG**	R/W	O**V**erflow flag
9..13	-	R/-	*Reserved, read as 0*
14	**Q_FLAG**	-/W	Clears pending interrupt buffer when set[7]
15	**R_FLAG**	R/W	Allow write access to IMEM "**R**OM"

Table 7: Bits of the status register

6 This flag is only available if the extended ALU operations switch is activated. See chapter Arithmetic / Logic Unit for more information.
7 This flag is automatically cleared in the next clock cycle and is always read as zero.

2.1.4. Interrupts

The NEO430 CPU features 4 independent interrupts via 4 interrupt request signals. When triggered, each of these requests start a unique interrupt handler. The interrupt-causing sources can be the TIMER overflow interrupt, the UART or the TWI or the SPI transmission done interrupt, the GPIO pin-change interrupt or the external interrupts controller. The base addresses of the according interrupt handlers have to be stored in advance to the interrupt vector configuration table, which is located at the beginning of the DMEM. The application linker script ensures these locations are not used by the actual program.

Address	IRQ Vector Name	Priority	Source
0xC000	IRQVEC_TIMER	1 (highest)	The timer generates a threshold match
0xC002	IRQVEC_SERIAL	2	UART Rx available **OR** UART TX done **OR** SPI transmission done **OR** TWI transmission done
0xC004	IRQVEC_GPIO	3	GPIO input pin change
0xC006	IRQVEC_EXT	4 (lowest)	External interrupts controller IRQ

Table 8: Interrupt sources, priorities and handler base addresses with configuration register names

Operation

All interrupts can be globally disabled by clearing the `I` flag in the processor's status register. An interrupt handler can only execute, when the `I` flag is set and the corresponding enable flag of the interrupt source is activated inside the according source's control register (e.g., the timer unit).

If an interrupt is triggered, the according handler is executed and the interrupt request is deleted from the queue as soon as the handler starts executing. If the same interrupt request triggers again during the execution of the handler, the request is stored and is executed after the handler has finished. Any other pending interrupt requests with lower priority will be further queued. Whenever an interrupt is triggered and the corresponding handler is entered, the `I` flag of the status register is cleared to avoid an interruption of the executed handler. If more than one interrupt channel is triggered at the same time, the one with the highest priority is executed while the other requests are queued. When the handler of the interrupt with the highest priority exits, the handler of the interrupt with the next lower priority is started afterwards. Of course, you can reactivate the global interrupt enable flag inside an interrupt handler to implement a nested interrupt behavior. When setting the **Q** flag, all pending interrupt request in the buffer are deleted.

Interrupt Vector Configuration

The interrupt vectors can be initialized by a program by using the provided register aliases:

```
// interrupt vector table setup
IRQVEC_TIMER  = (uint16_t)(&timer_irq_handler);  // Timer IRQ handler address
IRQVEC_SERIAL = (uint16_t)(&serial_irq_handler); // UART/SPI/TWI IRQ handler address
IRQVEC_GPIO   = (uint16_t)(&gpio_irq_handler);   // GPIO IRQ handler address
IRQVEC_EXT    = (uint16_t)(&ext_irq_handler);    // External IRQ handler address
```

Using Interrupts in C Code

The interrupt handlers have to use specific attributes to ensure they perform stack spilling and also to use the RETI instruction when finishing. Each handler must not have any passed arguments nor a return value. The following example shows an exemplary setup:

```
  int main(void) {

    IRQVEC_TIMER  = (uint16_t)(&timer_irq_handler); // Timer IRQ handler address
    …
  }

/* ------------------------------------------------------------
 * INFO Timer interrupt handler
 * ------------------------------------------------------------ */
  void __attribute__((__interrupt__)) timer_irq_handler(void) {

    time_ms++;
  }
```

CPU Behavior

When a valid interrupt is received by the CPU, the hardware first stores the return address and afterwards the current state of the status register (including the set interrupt enable flag) to the stack. After that, the global interrupt flag and the sleep flag (if set) are cleared. Thus, the status register keeps its value, which was set in the interrupted program, except for the cleared sleep and interrupt enable flags. Finally, the according interrupt handler address is moved to the program counter to start execution of the handler. When the handler finishes execution (by the RETI instruction), the CPU reloads the old state of the status register (with the set interrupt enable flag and maybe the set sleep flag) and the return address from the stack to continue normal program execution.

When an interrupt handler finishes execution, at least one instruction from the interrupted program is executed before the same or another interrupt handler can start execution.

Maximum Interrupt Latency

The maximum interrupt latency – starting from a valid interrupt request to the CPU until the actual interrupt service routine is started – is defined by the latency of the interrupt management system of the CPU, the latency of the currently executed instruction and the latency required for fetching and starting the according interrupt vector.

The interrupt management system requires 2 cycles to process a valid interrupt request. The worst-case latency for an instruction (see next sub chapter) is presented by a certain CALL instruction, which requires 11 cycles for completion. If an interrupt is triggered, 8 cycles are required to fetch the corresponding interrupt vector and to perform a jump to that address. Thus, the **maximum interrupt latency is 21 cycles**. In contrast, the **minimum interrupt latency is 10 cycles**.

External Interrupts

In its default configuration, the NEO430 processor features eight external interrupt request signals (top entity port `ext_irq_i`) with eight according interrupt request acknowledge signals (top entity port `ext_ack_o`). The IRQ signals are evaluated by the external interrupts controller (EXIRQ) and are passed to the CPU's IRQVEC_EXT interrupt request line.

When the external interrupts controller (EXIRQ) is excluded from synthesis, the CPU IRQ line triggering IRQVEC_EXT is directly connected to the port `ext_irq_i(0)` while all other interrupt input lines are unused. Also, only `ext_ack_o(0)` is connected.

For more information regarding the use of the external interrupts via the EXIRQ take a look at chapter 2.18. External Interrupts Controller (EXIRQ).

2.1.5. Instruction Set

The instruction set of the NEO430 CPU is fully compatible to the original TI MSP430 instruction set architecture. The only exception is presented by the CPU's BCD addition instruction "DADD". A cheat sheet featuring all instructions can be found in the "instruction_set.pdf" in the doc folder.

2.1.6. Instruction Timing

A fully registered data path, which is subdivided into several micro operation cycles, is implemented by the NEO430 processor. This allows the system to operate at very high clock rates, but of course this also requires a splitting of the instruction execution into several sub cycles. The tables below show the required execution cycles for the different operand classes and addressing modes.

		SRC			
		Register direct R	**Indexed [R+n]**	**Indirect [R]**	**Indirect auto inc [R++]**
DST	**Register direct R**	6	9	7	7
	Indexed [R+n]	9	10	10	10

*Table 9: Double-operand (**format I**) instruction execution cycles*

		SRC = DST			
		Register direct R	**Indexed [R+n]**	**Indirect [R]**	**Indirect auto inc [R++]**
Operation	**CALL**	8	11	9	9
	PUSH	7	10	8	8
	Others	6	9	7	7

*Table 10: Single-operand (**format II**) instruction execution cycles (except `RETI`)*

Instruction / Operation	**Branches**	3
	RETI	8
	Interrupt	6

Table 11: Special instructions / operations execution cycles

Average Instruction Execution Time

If all instruction types and formats are executed in an equally distributed manner, the average CPI (cycles per instruction) evaluates to **8.14 cycles per instruction (worst case)**.

The double-operand "decimal addition" instruction (**DADD**) requires an **additional execution cycle** (for all addressing modes) to complete – if the instruction is enabled for synthesis at all.

2.1.7. System Bus

All components of the NEO430 processor are connected to the CPU via the main system bus. Since the connected devices are accessed using a memory-mapped scheme, simple load and store operations are used to transfer data to or from the devices.

Name	Width	Dir	Function
WREN	2	out	Write enable for each of the two transferred bytes
RDEN	1	out	Read enable (always full-word)
ADDR	16	out	Address signal
DO	16	out	Write data
DI	16	in	Read data (one cycle latency)

Table 12: System bus signals (direction seen from CPU)

In the figure below you can see the signal timings when performing a write or read transaction. When conducting a write operation to a specific module, the actual 16-bit address and the data, that shall be written, are applied together with the write enable signals. For single byte transmission, only the corresponding bit of the WREN signal is set. A complete write transaction only requires a single cycle to complete. Read operations require two clock cycles to complete. Here, the read enable signals is applied together with the source address. In the next cycle, the accessed data word is read. Even when performing an explicit read operation of a single byte, the full 16-bit word is transferred.

The data output signals of all devices are OR-ed together before the resulting signal is fed to the CPU. Hence, only the actually accessed device must generate an output different than 0x0000. Therefore, read transactions are subdivided into two consecutive cycles: In the first cycle, the address and the read enabled signal are applied. Now, each device can check whether it is accessed or not. If there is an address match, the according device fetches data from the accessed location and applies it to its data output port in the *next cycle*. In any other situation, the data output of that module must be set to 0x0000.

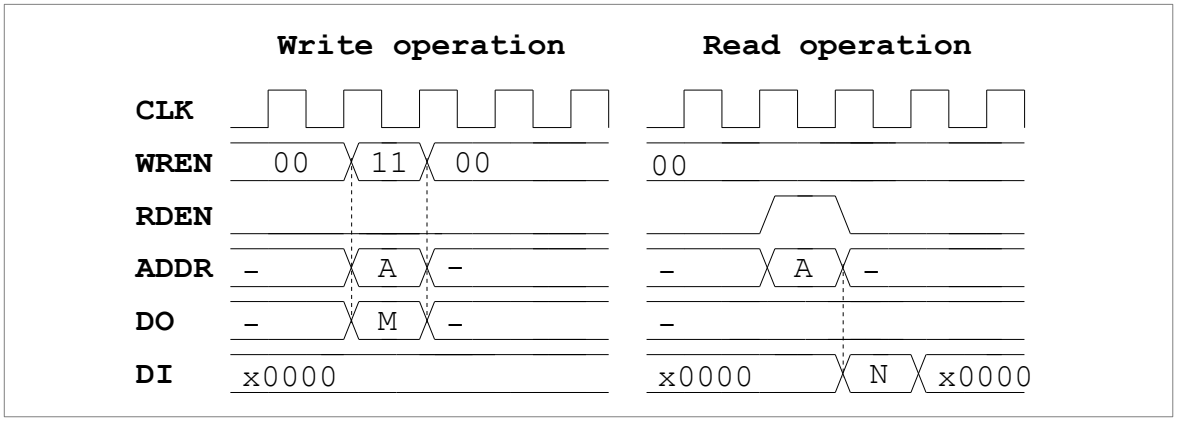

Figure 5: Write and read bus cycles (full word transfers); Write: M → [A]; Read: [A] = N

You can add (or replace) custom modules to the processor-internal bus, but that requires a good understanding of the address space layout and the general NEO430 architecture. Instead, I encourage you to use the custom functions unit (CFU) or the Wishbone bus interface to implement or attach custom logic.

2.2. Internal Instruction Memory (IMEM)

The internal instruction memory (VHDL component **neo430_imem.vhd**) stores the code of the actual user program. It is located at base address 0x0000 of the address space. The actual IMEM size can be configured via the IMEM_SIZE generic (see below) of the processor top entity (see cut-out below). Make sure the IMEM size **does not exceed 48 kB**. During run time the size can be obtained by a program by reading a specific CPUID register from the SYSCONFIG module (will be discussed later).

```
IMEM_SIZE => 4*1024, -- internal IMEM size in bytes, max 48kB (default=4kB)
```

By default, the IMEM is implemented as RAM, so the content can be modified during run time. This is required when using a bootloader that can update the content of the IMEM at any time. With the default implementation as RAM the **r_flag** in the CPU's status register has to be set in order to allow write accesses to the instruction memory. If you do not need the bootloader, because your application development is done and you want the program to permanently reside in the IMEM, the IMEM can also be implemented as true read-only memory. In this case set the IMEM_AS_ROM generic of the processor's top entity to "true".

```
IMEM_AS_ROM : boolean := false -- implement IMEM as read-only memory? (default=false)
```

When the IMEM is implemented as ROM, it will be initialized during synthesis with the actual application program. The toolchain will generate a VHDL initialization file (neo430_application_image.vhd) from your application, which is automatically inserted into the IMEM. If the IMEM is implemented as RAM, the memory will not be initialized at all.

2.3. Internal Data Memory (DMEM)

The internal data memory (VHDL component **neo430_dmem.vhd**) serves as general data memory / RAM for the currently executed program. It is located at base address 0xC000 of the address space. This address is fixed and must not be altered. The actual RAM size can be configured via the DMEM_SIZE generic of the processor top entity (see cut-out below). Make sure the RAM size **does not exceed 12 kB**. During run time, the size can be obtained by a program by reading a specific CPUID register from the SYSCONFIG module (will be discussed later). Remember, that the first 8 byte of the DMEM are used by the hardware for storing the interrupt vectors and are not available for the actual program.

```
DMEM_SIZE => 2*1024, -- internal DMEM size in bytes, max 12kB (default=2kB)
```

2.4. Boot ROM

As the name already suggests, the boot ROM (VHDL component **neo430_boot_rom.vhd**) contains the read-only bootloader image, which is executed right after a system reset. It is located at address 0xF000 of the address space. This address is fixed and must not be modified, since it represents the hardware-defined boot address. The ROM size can be configured in the neo430_package.vhd file (see cut-out below) if you want to write your own custom bootloader, but the size **must not exceed 2kB**. During synthesis, the VHDL boot ROM is initialized using the neo430_bootloader_image.vhd file (which is generated by the image generator auxiliary program during compilation process).

```
constant boot_size_c : natural := 2*1024; -- bytes, max 2kB (default=2kB)
```

If you are using the IMEM as true ROM – initialized with your application code during synthesis – the bootloader is (in most cases) no longer necessary. In this case you can disable the implementation of the bootloader. Use the BOOTLD_USE generic of the processor top entity (see cut-out below) to exclude it. If the bootloader implementation is deactivated, the CPU starts booting your application from address 0x0000 instead from the base address of the boot ROM at 0xF000.

```
BOOTLD_USE => true, -- implement and use bootloader? (default=true)
```

Boot Configuration

The default configuration of the NEO430 processor includes all optional modules (except for the custom functions unit) and also provides a build-in serial bootloader. This bootloader is very suitable for the evaluation process, since the application program can be re-uploaded at every time using a standard UART connection. Furthermore, the bootloader provides an automatic boot configuration, which automatically boots from an external SPI EEPROM after a specific UART console timeout. This feature allows to implement a non-volatile program storage, which can still can be altered after implementation.

For a mature design the bootloader feature might not be required anymore. For this scenario the bootloader can be excluded from the design via the according generic configuration switch (BOOTLD_USE). If the bootloader is disabled, your application code will be directly executed after reset. Therefore, the application program image remains permanently in the internal instruction memory (IMEM). Note that modifications of the IMEM are still possible when the IMEM_AS_ROM switch is not disabled.

More information regarding the boot configuration can be found in chapter 3.2. Internal Bootloader.

2.5. Wishbone Bus Interface (WB32)

The default NEO430 processor setup includes a Wishbone bus interface adapter (VHDL component **neo430_wb_interface.vhd**). Several IP blocks (e.g., from opencores.org) provide a Wishbone interface. Hence, a custom system-on-chip can be build using this bus standard. The Wishbone adapter features 32-bit wide address and data buses. If required, only a subsection of the address and/or data buses can be connected to create a Wishbone bus with smaller data and/or address buses. The neo430_wishbone.h library file in the sw/lib/neo430 folder already implements the most common Wishbone transfer operations. These "driver functions" also take care of setting the according byte enable signals manipulating the final address when performing 16-bit or even byte-aligned accesses.

Wishbone Bus Protocol

A detailed description of the implemented Wishbone bus protocol and the according interface signals can be found in the opencores.org documentation data sheet "*Wishbone B4 – WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*". A copy of this document can be found in the doc folder of this project.

Implementation Control

Use the WB32_USE generic of the processor top entity (see cut-out below) to control implementation. When disabled, the dedicated Wishbone interface signals (see table) are not functional. In this case, set all input signals to low level and leave all output signals unconnected.

```
WB32_USE => true, -- implement WB32 unit? (default=true)
```

Wishbone Transactions

To perform a Wishbone transaction, several tiny steps are required. At first, the according byte enable signals (WB32_CT_WBSELx) and the transfer cycle type (see later) must be configured in the control register. The byte enable signals directly drive the wb_sel_o bus. Setting any of these four bits will also activate the Wishbone adapter. In contrast, the adapter can be deactivated (for example if an addressed slave is not responding) by clearing the control register.

In case of a write transfer, the data, which shall be written, must be loaded into the WB32_LD (low 16-bit part) and WB32_HD (high 16-bit part) registers. Together, these registers directly drive the wb_dat_o data output bus. To start the actual transfer, the address is written to the WB32_LWA and WB32_HWA register. The actual store access to the high address word register initiates the actual transfer. For a read transfer, the low part of the address is stored to the WB32_LRA and WB32_HRA register. Just like before, a write access to the high word register triggers the actual read transaction.

As soon as the transaction is started, the WB32_CT_PENDING bit in the unit's control register is set to indicate a pending transfer. The transfer was successfully completed when this bit returns to zero. A transfer is completed if the accesses slaves acknowledges the cycle by setting the ACK signal. If you wish to abort a pending transfer, you must disable the device by clearing the WB32_CT control register.

The Wishbone bus interface of the NEO430 processor **cannot** be used for instruction fetch or direct data access via the standard memory-access instructions. Instead, the bus interface is a communication device, that can be used by a program, which is executed from the internal memory, to access processor-external peripheral devices like mass-storage memories, hardware accelerators or additional communication interfaces.

The Wishbone interface adapter supports the standard or "classic" Wishbone transfer mode when connecting slaves, which apply their acknowledge signal in an asynchronous way. When the accessed slave cannot apply the acknowledge immediately – for example by registering the output signals in order to shorten the critical path – the implemented Wishbone protocol differs from the standard. The NEO430 Wishbone adapter **applies its STB signal only for one single cycle** of an active transfer. The cycle is terminated when the selected slave sets the acknowledge signal. This allows to implement slaves with registered outputs (higher frequency!) while also allowing correct accesses to IP cores, which trigger their operation on the STB signal (for instance a FIFO).

Long story short: When the slave applies its acknowledge asynchronously signal in the **same cycle**, the Wishbone transfer fulfills the **classic/standard mode protocol specification**. When the slave applies its acknowledge signal **one ore more cycles later**, the Wishbone transfer fulfills the specification of the **pipelined mode** protocol.

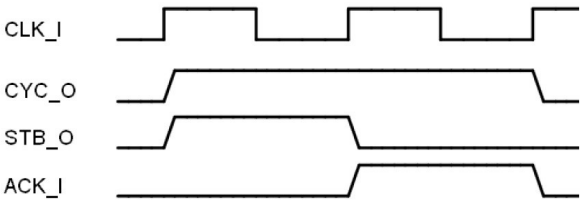

Figure 6: Pipelined Wishbone cycle

Wishbone Bus Interface Signals

Signal name	Width (#bits)	Direction	Function
wb_adr_o	32	Output	Access address
wb_dat_i	32	Input	Read data input
wb_dat_o	32	Output	Write data output
wb_we_o	1	Output	Read/write access
wb_sel_o	4	Output	Byte enable
wb_stb_o	1	Output	Strobe signal
wb_cyc_o	1	Output	Valid cycle indicator
wb_ack_i	1	Input	Cycle acknowledge

Table 13: Wishbone bus interface adapter signals seen from the processor

The Wishbone bus uses the processor's main clock for data bus operations.

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFF90	WB32_CT	0	WB32_CT_WBSEL0	-/W	Byte 0 transfer enable → wb_sel_o(0)
		1	WB32_CT_WBSEL1	-/W	Byte 1 transfer enable → wb_sel_o(1)
		2	WB32_CT_WBSEL2	-/W	Byte 2 transfer enable → wb_sel_o(2)
		3	WB32_CT_WBSEL3	-/W	Byte 3 transfer enable → wb_sel_o(3)
		4..14		-/-	*reserved*
		15	WB32_CT_PENDING	R/-	Pending Wishbone transfer flag
0xFF92	WB32_LRA	0..15		-/W	Low address word for read transfer
0xFF94	WB32_HRA	0..15		-/W	High address word for read transfer (+trigger)
0xFF96	WB32_LWA	0..15		-/W	Low address word for write transfer
0xFF98	WB32_HWA	0..15		-/W	High address word for write transfer (+trigger)
0xFF9A	WB32_LD	0..15		R/W	Low word of read/write data, can be accessed in byte mode
0xFF9C	WB32_HD	0..15		R/R	High word of read/write data, can be accessed in byte mode
0xFF9E	-	0..15		-/-	*Reserved*

Table 14: Wishbone32 interface adapter address map

This unit needs to be reset by software before you can use it. The reset is performed by writing zero to the unit's control register.

Do not read from registers, which do not provide a read access feature (e.g., when R/W is -/W), since such accesses return undefined data.

Regarding Wishbone Bus Sizes

Although the wishbone adapter implements 32-bit wide data and address buses, you do not need to actually use this sizes. If you have only some accessible addresses in you Wishbone network, then 16-bit address width might by sufficient. Also, if you only use slaves with e.g., 16-bit data width, you do not actually need the full32-bit provided by the adapter. By constraining the bus sizes to your actual needs the hardware can be simplified by the synthesis tool to save hardware resources. Furthermore, if you use the according Wishbone access functions for smaller data width (e.g., 16-bit transfers only), the Wishbone access can be conducted in less CPU cycles increasing the performance.

2.6. General Purpose Input and Output Ports (GPIO)

The general purpose parallel IO controller (VHDL component **neo430_gpio.vhd**) provides a simple 16-bit parallel input port and a 16-bit parallel output port. These ports can be used chip-externally (e.g. to drive LEDs, connect buttons, etc.) or system-internally to provide control signals for other IP modules.

Implementation Control

I know, hardware resources are precious. The GPIO module does not require a lot of logic and you won't have this fancy blinking bootloader status signal, when you disable the module. However, if you do not need the included GPIO ports, you can exclude the module from synthesis. Use the `GPIO_USE` generic of the processor top entity (see cut-out below) to control implementation. If the unit is excluded from synthesis, all parallel output signals are set to low level and the pin change interrupt is permanently disabled.

```
GPIO_USE => true, -- implement GPIO unit? (default=true)
```

Pin-Change Interrupt

The parallel input port features a single pin-change interrupt. To select which input pins can cause an interrupt, the `GPIO_IRQMAS` register allows to select only the desired input pins. The pin change interrupt is deactivated if all bits of the mask register are cleared. When enabled, the interrupt will trigger if there is any transition (rising edge or falling edge on one of the masked input pins. Therefore, it is not possible to directly determine which input pin caused the interrupt. This mus be done by reading the input data and examining the new state. Use the `neo430_gpio.h` library file in the `sw/lib/neo430` folder to get access to some of the most common IO operations like bit set, clear or toggle.

Address	IRQ Vector Name	Priority	Source
0xC004	IRQVEC_GPIO	3	GPIO input pin change.

Table 15: GPIO pin-change interrupt vector

Register Map

Address	Name	Bit(s) (Name)	R/W	Function
0xFFA8	-	0..15	-/-	*Reserved*
0xFFAA	GPIO_IRQMASK	0..15	-/W	Enable according input pin(s) as interrupt trigger
0xFFAC	GPIO_IN	0..15	R/-	Parallel input port
0xFFAE	GPIO_OUT	0..15	R/W	Parallel output port

Table 16: GPIO module register map

PWM Modulation of the GPIO Output Port

The whole GPIO output port can be modulated by using the processor's PWM controller. See chapter 2.16. Pulse Width Modulation Controller (PWM) for more information.

2.7. Universal Asynchronous Receiver and Transmitter (UART)

In most cases, the UART interface is used to establish a communication channel between the computer/user and an application running in the processor. Of course, you can also use the UART for interfacing chip-external peripheral devices. A standard configuration is used for the UART protocol layout: 8 data bits, 1 stop bit and no parity bit. These values are fixed and cannot be altered. The actual Baudrate is configurable by software. This configuration is explained later.

After configuring the Baud rate, the UART must be activated by setting the `UART_CT_EN` in the UART control register `UART_CT`. Now you can transmit a character by writing it to the `UART_RTX` register. The transfer is in progress if the `UART_CT_TX_BUSY` bit in the control register is set. A received char is available when bit #15 of the `UART_RTX` register is set. When reading this register, the available flag is automatically cleared and you have your received character – all done using a single access! That's cool, huh? A “char available” or “transmission completed” interrupt can be activated by setting the according bits in the control register. Note, that both interrupt sources trigger the same interrupt handler (`IRQVEC_SERIAL`)! To make the usage of the UART a little bit easier, the `neo430_uart.h` library in the `sw/lib/neo430` folder features elementary function for sending and receiving data. The only thing you have to do by hand is to enable the UART and call the Baud rate configuration. Well, actually you don't have to do this, since it is already done by the bootloader. The bootloader computes the according Baud value based on the clock speed from the *SYSCONFIG* for a final Baud rate of 19200.

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFFA0	UART_CT	0	UART_CT_BAUD0	R/W	Baud value config bit 0
		1	UART_CT_BAUD1	R/W	Baud value config bit 1
		2	UART_CT_BAUD2	R/W	Baud value config bit 2
		3	UART_CT_BAUD3	R/W	Baud value config bit 3
		4	UART_CT_BAUD4	R/W	Baud value config bit 4
		5	UART_CT_BAUD5	R/W	Baud value config bit 5
		6	UART_CT_BAUD6	R/W	Baud value config bit 6
		7	UART_CT_BAUD7	R/W	Baud value config bit 7
		8	UART_CT_PRSC0	R/W	Baud prescaler select bit 0
		9	UART_CT_PRSC1	R/W	Baud prescaler select bit 1
		10	UART_CT_PRSC2	R/W	Baud prescaler select bit 2
		12	UART_CT_EN	R/W	UART enable
		13	UART_CT_RX_IRQ	R/W	Enable RX interrupt
		14	UART_CT_TX_IRQ	R/W	Enable TX interrupt
		15	UART_CT_TX_BUSY	R/-	Trasmitter busy
0xFFA2	UART_RTX	0..7	UART RX/TX data	R/W	SPI Rx/Tx data
		15	UART_RTX_AVAIL	R/-	RX data available

Table 17: UART regitser map

Implementation Control

By default, the UARt is always synthesized. You can use the `UART_USE` generic of the processor top entity (see cut-out below) to control implementation.

```
UART_USE => true, -- implement UART? (default=true)
```

UART Baudrate

The actual transfer speed – the Baud rate – can be arbitrarily configured via the `UART_CT_BAUDx` bits in the control register. These bits define a prescaler value (**PRSC**) toegther with the prescaler selection bits `UART_CT_PRSCx`. The actual Baud rate of the UART interface is computed using the following formula:

$$Baudrate = \frac{f_{main}[Hz]}{PRSC * BAUD}$$

The BAUD parameter can be obtained by finding the largest number for a given clock frequency and a selected prescaler, that fits into 8 bits. The following table shows different **BAUD** values for 8 common Baudrates using one of the 8 prescaler configurations. This setup assumes a clock frequency of **50MHz**. The red highlighted values are invalid, since they cannot fit into the 8-bit wide BAUD register. In contrast, the green values are valid for the according prescaler selection, but you should always use the highest possible BAUD value to minimize the Baudrate error.

Baudrate / Prescaler / BAUD Value Look-up-Table

Prescaler bits configuration:	000	001	010	011	100	101	110	111
Resulting clk prescaler *PRSC*:	2	4	8	64	128	1024	2048	4096
Baudrate = **1200**	20833	10417	5208	651	326	41	20	10
Baudrate = **2400**	10417	5208	2604	326	163	20	10	5
Baudrate = **4800**	5208	2604	1302	163	81	10	5	3
Baudrate = **9600**	2604	1302	651	81	41	5	3	1
Baudrate = **19200**	1302	651	326	41	20	3	1	1
Baudrate = **28800**	868	434	217	27	14	2	1	0
Baudrate = **57600**	434	217	109	14	7	1	0	0
Baudrate = **115200**	217	109	54	7	3	0	0	0

In this table the green highlighted numbers represent valid values for the 8-bit wide Baud value configuration (BAUD).

Example Baudrate Computation

Clock frequency: **50MHz**
Desired Baudrate: **19200**
Prescaler PRSC: **64 → 0b011**
BAUD value: **41**

Actually, you do not have to worry about the configuration of the UART Baud rate at all. A function from the `neo430_uart.h` library does all the work for you – just call it once at program start.

UART Interrupt

The UART features a single interrupt output, which can be used to indicate a *UART RX data available* status and/or *UART TX done* status. When enabling all sources at the same time, you have to check the UART control register to determine the actual causing event. Note, that this interrupt channel can also be used by the SPI or the TWI module.

Address	IRQ Vector Name	Priority	Source
`0xC002`	`IRQVEC_SERIAL`	2	UART Rx available **OR** UART Tx done **OR** SPI transmission done **OR** TWI transmission done

Table 18: Serial register vector

2.8. Serial Peripheral Interface (SPI)

Just like the UART, the SPI is a standard interface for accessing a wide variety of external devices. The data transfer quantity is fixed to 8-bit for a single transfer. However, larger data 'packets' can be implemented, since the actual transferred data size is determined by the actual number of send/received bytes during an active chip select (CS) of the connected device.

Operation

A transmission is started when writing a data byte to the `SPI_RTX` register. The `SPI_CT_BUSY` bit of the control register indicates a transfer being in progress. The received data can be obtained by reading the `SPI_RTX` register as soon as the transmission is done and the busy flag is cleared. A “transmission done” interrupt can be activated by setting the `SPI_CT_IRQ` bit. Note, that this interrupt also triggers the same interrupt handler as the interrupt sources from the UART module and the TWI module. The SPI module implements already six dedicated chip select lines, which are directly accessible via the unit's control register, but additional CS lines can be created using the GPIO controller or a specific Wishbone device.

Implementation Control

By default, the SPI is always synthesized. You can use the `SPI_USE` generic of the processor top entity (see cut-out below) to control implementation.

```
SPI_USE => true, -- implement SPI? (default=true)
```

Register Map

Address	Name	Bit(s)	(Name)	R/W	Function
0xFFA4	SPI_CT	0	SPI_CT_EN	R/W	SPI enable
		1	SPI_CT_CPHA	R/W	Clock phase (“idle clock polarity”)
		2	SPI_CT_IRQ	R/W	Transmission done interrupt enable
		3	SPI_CT_PRSC0	R/W	Clock prescaler select bit 0
		4	SPI_CT_PRSC1	R/W	Clock prescaler select bit 1
		5	SPI_CT_PRSC2	R/W	Clock prescaler select bit 2
		6	SPI_CT_CS_SEL0	R/W	Chip select select bit 0
		7	SPI_CT_CS_SEL1	R/W	Chip select select bit 1
		8	SPI_CT_CS_SEL2	R/W	Chip select select bit 2
		9	SPI_CT_CS_SET	R/W	Activate selected CS (set low)
		10	SPI_CT_DIR	R/W	Shift direction (0: MSB, 1: LSB fisrt)
0xFFA6	SPI_RTX	0..7	RX/TX data	R/W	Send/receive data

Table 19: SPI register map

Transmission Configuration

The SPI transmission provides several configuration options. The actual clock phase can be determined via the `SPI_CT_CPHA` bit. The clock polarity is fixed to a low idle level. If you wish to use a high idle level, invert the SPI clock signal in your top design. If the `SPI_CT_DIR` flag is zero, the MSB of the data word is send/sampled first. When the flag is set, the LSB is send/sampled first. Mirror your data byte if you wish to send the LSB first. The actual transmission speed is set via the three prescaler selection bits `SPI_CT_PRSCx` of the control register. The resulting main clock prescaler value *PRSC* is shown in the table below:

Prescaler bits configuration:	000	001	010	011	100	101	110	111
Resulting prescaler *PRSC*:	2	4	8	64	128	1024	2048	4096

Based on the **PRSC** prescaler value, the actual SPI clock frequency is determined by:

$$f_{SPI} = \frac{f_{main}[Hz]}{2 * PRSC}$$

In the `neo430_spi.h` library file you can find elementary functions for performing an SPI transmission and for controlling the dedicated chip-select signals of the SPI module.

Dedicated SPI Chip Select (CS) Lines

The SPI controller features eight dedicated chip select lines (signal `spi_cs_o` from the processor's top entity) so you can directly connect up to six SPI slaves to the controller without using e.g., GPIO pins as chip select signals. The six lines are accessible via the `SPI_CT_CS_SELx` bits in the SPI control register. The tree bits are used to index exactly one of the eight CS lines. The selected CS line will be activated (set low) when the `SPI_CT_CS_SET` bit in the control register is set.

SPI Interrupt

The SPI features a single interrupt output, which can be used to indicate a *SPI transmission done* status. . Note, that this interrupt channel can also be used by the UART or the TWI module.

Address	IRQ Vector Name	Priority	Source
0xC002	IRQVEC_SERIAL	2	UART Rx available **OR** UART Tx done **OR** SPI transmission done **OR** TWI transmission done

Table 20: Serial register vector

2.9. Two Wire Serial Interface Master (TWI)

The two wire interface – actually called I²C – is a quite famous interface for connecting several on-board components. Since this interface only needs two signals (the serial data line SDA and the serial clock line SCL) – despite of the number of connected devices – it allows easy interconnections of several slave nodes.

This unit implements a **TWI master,** which can be used to access several slave devices. The TWI module features "**clock stretching**", so a slow slave can halt the transmission by pulling the SCL line low. **Currently no multi-master support it available.**

Implementation Control

By default, the TWI is always synthesized. You can use the `TWI_USE` generic of the processor top entity (see cut-out below) to control implementation.

```
TWI_USE => true, -- implement TWI? (default=true)
```

Electrical Requirements

Since the serial clock (SCL) and the serial data (SDA) lines can only be actively driven low by the master. Hence, an external pull-up resistor is required for each signal. The resistance of the pull-ups can be estimated by the following formula:

$$R_{pullup} = \frac{V_{cc} - 0.4\,V}{0.003\,A}$$

Operation

After the device has been enabled, the program can start / terminate a transmission by issuing a START or STOP condition. These conditions are generated by setting the according bit in the devices control register. Data is send by writing a byte to the `TWI_RTX` register. Received data can also be obtained from this register. The TWI master is busy (transmitting or performing a START or STOP condition) as long as the `TWI_CT_BUSY` bit in the control register is set. An accessed slave has to acknowledge each transferred byte. When the `TWI_DT_ACK` bit in the `TWI_RTX` register is set after a completed transmission, the accessed slave has send an acknowledge. If it is cleared after a transmission, the slave send a no-acknowledge (NACK).

In summary, the following independent TWI operations can be triggered by the application program:

- send START condition (also as REPEATED START condition)
- send STOP condition
- send (at least) one byte while also sampling one byte from the bus

In the `neo430_twi.h` library file you can find elementary functions for performing TWI transmissions.

Transmission Configuration

The TWI transmission speed (or the clock frequency of the SCL line) is configured via the three prescaler selection bits `TWI_CT_PRSCx` of the control register. The resulting main clock prescaler value *PRSC* is shown in the table below:

Prescaler bits configuration:	000	001	010	011	100	101	110	111
Resulting prescaler *PRSC*:	2	4	8	64	128	1024	2048	4096

Based on the **PRSC** prescaler value, the actual TWI clock frequency is determined by:

$$f_{SCL} = \frac{f_{main}[Hz]}{4 * PRSC}$$

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFFE8	TWI_CT	0	TWI_CT_EN	R/W	Enable TWI
		1	TWI_CT_START	-/W	Generate START condition
		2	TW_CT_STOP	-/W	Generate STOP condition
		3	TWI_CT_BUSY	R/-	TWI is currently busy
		4	TWI_CT_PRSC0	R/W	Clock prescaler select bit 0
		5	TWI_CT_PRSC1	R/W	Clock prescaler select bit 1
		6	TWI_CT_PRSC2	R/W	Clock prescaler select bit 2
		7	TWI_IRQ_EN	R/W	Transmission done interrupt enable
0xFFEA	TWI_RTX	0..7	RX/TX data	R/W	Send/receive data
		15	TWI_DT_ACK	R/-	Set when an ACK has been received

Table 21: TWI register map

TWI Interrupt

The TWI features a single interrupt output, which can be used to indicate a *TWI transmission done* status. Note, that this interrupt channel can also be used by the UART or the SPI module.

Address	IRQ Vector Name	Priority	Source
0xC002	IRQVEC_SERIAL	2	UART Rx available **OR** UART Tx done **OR** SPI transmission done **OR** TWI transmission done

Table 22: Serial interrupt vector

2.10. High-Precision Timer (TIMER)

A high-precision timer (VHDL component **`neo430_timer.vhd`**) is required by many real-time applications. The included device features a simple but powerful module to generate an interrupt in specific time intervals. Besides selecting the main clock-based prescaler, a timer threshold can be configured to accomplish highly-accurate timing.

Implementation Control

By default, the timer is always synthesized. You can use the `TIMER_USE` generic of the processor top entity (see cut-out below) to control implementation.

```
TIMER_USE => true, -- implement timer? (default=true)
```

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFFB0	TMR_CT	0	TMR_CT_EN	R/W	Timer enable
		1	TMR_CT_ARST	R/W	Automatic reset on timer match
		2	TMR_CT_IRQ	R/W	IRQ enable
		3	TMR_CT_PRSC0	R/W	Timer counter increment clock prescaler PRSC
		4	TMR_CT_PRSC1	R/W	
		5	TMR_CT_PRSC2	R/W	
		7..15		R/-	*Reserved, read as 0*
0xFFB2	TMR_CNT	0..15		R/-	Counter register, read-only!
0xFFB4	TMR_THRES	0..15		R/W	Threshold register, IRQ on match
0xFFB6	-	0..15		R/-	reserved

Table 23: High-precision timer register map

This unit needs to be reset by software before you can use it. The reset is performed by writing zero to the unit's control register.

Timer Operation

An exact timer period is configured using the clock select prescaler bits `TMR_CT_PRSCx` in the control register `TMR_CT` and setting a timer threshold `TMR_THRES`. Corresponding to the three prescaler selection bits, one of 8 different prescaler values can be selected:

Prescaler bits configuration:	000	001	010	011	100	101	110	111
Resulting prescaler *PRSC*:	2	4	8	64	128	1024	2048	4096

Based on the ***PRSC*** prescaler value and the ***THRES*** threshold value, the resulting interrupt "tick frequency" (reciprocal time between two interrupts) is given by:

$$f_{tick} = \frac{f_{main}}{PRSC \cdot (THRES + 1)}$$

Example: The desired tick frequency may be 2Hz at a main clock of 100MHz. Using the max. prescaler value (0b111 → PRSC = 4096), the threshold value is computed by:

$$THRES = \frac{f_{main}}{f_{tick} \cdot PRSC} - 1 = \frac{100000000\ Hz}{2\ Hz \cdot 4096} - 1 = 12207$$

After the timer is enabled via the `TMR_CT_EN` bit, the timer increments the counter register with the specified (→ prescaler bits) frequency. Whenever it reaches the value stored in the threshold register, an interrupt request is asserted (when enabled via the `TMR_CT_IRQ` bit). If the auto-reset bit `TMR_CT_ARST` is set, the counter register is cleared when the threshold value is reached and counting starts again. If not, the user has to clear the counter register manually within the interrupt service routine.

Timer Interrupt

When the `TMR_IRQ_EN` bit in the timer's control register is set, an interrupt request is generated whenever a counter match occurs (`TMR_THRES == TMR_CNT`). If the timer unit has been excluded from synthesis (disabled), the timer's interrupt request signal is always connected to 0.

Address	IRQ Vector Name	Priority	Source
0xC000	IRQVEC_TIMER	1 (highest)	The timer generates a threshold match.

Table 24: Timer interrupt vector

2.11. Watchdog Timer (WDT)

The WDT (VHDL component **neo430_wdt.vhd**) implements a watchdog timer. When enabled, an internal 16-bit counter is started. A program can reset this counter at any time. If the counter is not reset, a system wide hardware reset is executed when the timer overflows. The watchdog is enabled by setting the WDT_EN bit. Each write access to the watchdog must contain the watchdog access password (=0x47) in the upper 8 bits of the written data word. If the password is wrong and the watchdog is disabled, the access is simply ignored. If the password is wrong and the watchdog is enabled, a hardware reset is generated. A user can determine the cause of the last processor reset by reading the WDT_RCAUSE bti. If the bit is set, the last reset was generated by the watchdog. If the bit is cleared, the reset was generated via the external reset signal. When the watchdog caused the reset, the WDT_RPW_FAIL flag indicates if this reset was caused by a normal timeout ('0') or a failed access ('1') due to a wrong password. You can find elementary functions for performing watchdog operations in the neo430_wdt.h library.

To control the timeout period, one can select 1 of 8 different timeout periods via the WDT_PRSCx bits:

CLKSELx bits configuration:	000	001	010	011	100	101	110	111
Main clock prescaler:	2	4	8	64	128	1024	2048	4096
Timeout period in main clock cycles:	131 072	262 144	524 288	4 194 304	8 388 608	67 108 864	134 217 728	268 435 456

Implementation Control

Use the WDT_USE generic of the processor top entity (see cut-out below) to control implementation.

```
WDT_USE => true, -- implement WDT? (default=true)
```

Register Map

Address	Name	Bit(s)	Name	R/W	Function
0xFFD8	WDT_CT	0..2	WDT_CT_PRSCx	R/W	Timeout interval selection
		3	WDT_CT_EN	R/W	Watchdog enable bit
		4	WDT_CT_RCAUSE	R/-	Cause of last processor reset (0=external reset, 1=watchdog timeout)
		5	WDT_CT_RPWFAIL	R/-	Reset caused by wrong-password WDT access when 1
		6..7	-	R/-	*Reserved, read as 0*
		8..15	WDT_CT_PASSWORD	-/W	Access password, has to be 0b01000111 (0x47)

Table 25: Watchdog timer register map

2.12. System Configuration Module (SYSCONFIG)

The system information module (VHDL component **`neo430_sysconfig.vhd`**) gives access to various system information, which are mainly defined by the generics of the processor's top entity.

The module, which is implemented as simple ROM with eight 16-bit locations, provides information regarding the processor hardware configuration. By accessing this component, a program can determine the available RAM space, check if specific instructions or hardware modules are implemented and compute timings (like the UART Baud rate) based on the actual clock speed during run time. Also, a custom user code ca be checked. Most of these parameters are set using the configuration generics of the NEO430 top entity during instantiation.

Address	Name	Bit(s) (Name)		R/W	Function
0xFFF0	CPUID0	0..15: HW_VERSION		R/-	Hardware version
0xFFF2	CPUID1	0	SYS_MULDIV_EN	R/-	Set if **MULDIV** is implemented
		1	SYS_WB32_EN	R/-	Set if **WB32** is implemented
		2	SYS_WDT_EN	R/-	Set if **WDT** is implemented
		3	SYS_GPIO_EN	R/-	Set if **GPIO** is implemented
		4	SYS_TIMER_EN	R/-	Set if **TIMER** is implemented
		5	SYS_UART_EN	R/-	Set if **UART** is implemented
		6	SYS_DADD_EN	R/-	Set if **DADD instruction** (CPU) is implemented
		7	SYS_BTLD_EN	R/-	Set if **bootloader** is implemented and used
		8	SYS_IROM_EN	R/-	Set if **IMEM** is implemented **as true ROM**
		9	SYS_CRC_EN	R/-	Set if **CRC unit** is implemented
		10	SYS_CFU_EN	R/-	Set if **CFU** is implemented
		11	SYS_PWM_EN	R/-	Set if **PWM controller** is implemented
		12	SYS_TWI_EN	R/-	Set if **TWI** is implemented
		13	SYS_SPI_EN	R/-	Set if **SPI** is implemented
		14	SYS_TRNG_EN	R/-	Set if **TRNG** is implemented
		15	SYS_EXIRQ_EN	R/-	Set if **EXIRQ** is implemented
0xFFF4	CPUID2	0..15: USER_CODE		R/-	Custom user code, defined via top's generic
0xFFF6	CPUID3	0..15: IMEM_SIZE		R/-	Size of IMEM in bytes
0xFFF8	CPUID4	0..15: *reserved*		R/-	*reserved*
0xFFFA	CPUID5	0..15: DMEM_SIZE		R/-	Size of DMEM in bytes
0xFFFC	CPUID6	0..15: CLOCKSPEED_LO		R/-	Low word of clock speed (in Hz)
0xFFFE	CPUID7	0..15: CLOCKSPEED_HI		R/-	High word of clock speed (in Hz)

Table 26: System information memory register map

The information provided by the system information ROM is used by the bootloader to perform a system initialization (configure Baud rate, setup the timer interval, check connectivity, …). Furthermore, the application start-up code (`crt0.asm`) uses the system information ROM for the minimal-required hardware setup.

2.13. Multiplier and Divider Unit (MULDIV)

By default the NEO430 processor includes a *serial* unsigned multiplier and divider unit (VHDL component **neo430_muldiv.vhd**). This unit allows to compute 16-bit division (with remainder) and 16x16-bit = 32-bit multiplication much faster than computing the same operations in software only. This accelerator is **NOT compatible to the MSP430-GCC compiler**, hence it cannot be used by describing general multiplications and divisions in software. Instead, the accelerator has to be used by explicitly using specific functions provided by the `neo430_muldiv.h` library. The multiplier/divider unit only performs unsigned operations. If signed operations are required, the provided library functions take care of the necessary conversions.

Implementation Control

In case you do not need the multiply and divide unit, you can use the `MULDIV_USE` generic of the processor top entity (see cut-out below) to control implementation.

```
MULDIV_USE => true, -- implement multiplier/divider unit? (default=true)
```

By default, the multiplier core is implemented using general FPGA logic. If you want to use dedicated DSP blocks instead, activate this option in the processor package file (`neo430_package.vhd`):

```
constant use_dsp_mul_c : boolean := false; -- use DSP blocks for MULDIV's
```

Operation

The first operand (first factor for multiplication or the dividend for division) is always written to the `MULDIV_OPA` register. When writing the second operand (here, the divisor) to the `MULDIV_OPB_DIV` register, a division is triggered. When writing the second operand (here, the second factor) to the `MULDIV_OPB_MUL` register, a multiplication is triggered. The according result (32-bit product or the remainder and the quotient) can be read from the `MULDIV_RESX` and `MULDIV_RESY` registers. The hardware operation itself needs 18 CPU cycles to generate the requested results.

Register Map

Address	Name	Bit(s)	R/W	Function
0xFF80	MULDIV_OPA	0..15	-/W	First operand (dividend for divisions, first factor for multiplication)
0xFF82	MULDIV_OPB_DIV	0..15	-/W	Dividend, write access triggers the actual division
0xFF84	MULDIV_OPB_MUL	0..15	-/W	Second factor, write access triggers the actual multiplication
0xFF8C	MULDIV_RESX	0..15	R/-	Low word of the multiplication product or the division's quotient
0xFF8E	MULDIV_RESY	0..15	R/-	High word of the multiplication product or the division's remainder

Table 27: Multiplier/divider unit register map

2.14. Cyclic Redundancy Checksum Unit (CRC)

Ever been in need to verify a data stream? Then the CRC unit (VHDL component **`neo430_crc.vhd`**) is just right for you! This unit implements a 32-bit shift register, 32-bit XOR mask and an 8-bit data input shift register allowing to compute any CRC16 or CRC32 checksum. The unit operates on chunks of 8-bit input data and can compute the programmed checksum very quickly. Furthermore, the start value of the internal computation shift register can be set in order to specify custom init values.

Implementation Control

In case you do not need the checksum computation unit, you can use the `CRC_USE` generic of the processor top entity (see cut-out below) to control implementation.

```
CRC_USE => true, -- implement CRC unit? (default=true)
```

Operation

At first, the actual polynomial must be written as according XOR mask to the `CRC_POLY_LO` and `CRC_POLY_HI` registers. If you are using a CRC16 checksum, you only need to configure the lower polynomial register. After that, you can specify an initial seed for the internal 32-bit CRC shift register (`CRC_RESX` and `CRC_RESY`). In most cases the stat value is set to zero. After the initial configuration new input data in chunks of 8-bit (so only thw lowest 8 bits are used) can be written to the `CRC_CRC16IN` register for 16-bit CRC computations or to the `CRC_CRC32IN` register for 32-bit CRC computations. The final results can be obtained from the `CRC_RESX` register for 32-bit CRC computations and also from the `CRC_RESY` register for 32-bit CRC computations. The provided `neo430_crc.h` library features some of the mostly required CRC computations functions that also allow an easy and hardware abstract handling of the CRC unit.

Register Map

Address	Name	Bit(s)	R/W	Function
0xFFC0	CRC_POLY_LO	0..15	-/W	Low 16-bit of the polynomial XOR mask
0xFFC2	CRC_POLY_HI	0..15	-/W	High 16-bit of the polynomial XOR mask
0xFFC4	CRC_CRC16IN	0..7	-/W	8-bit input data for 16-bit CRC computation + operation trigger
0xFFC6	CRC_CRC32IN	0..7	-/W	8-bit input data for 32-bit CRC computation + operation trigger
0xFFC8	-	0..15	-/-	*reserved*
0xFFCA	-	0..15	-/-	*reserved*
0xFFCC	CRC_RESX	0..15	R/W	CRC shift register (result / init value) low part
0xFFCE	CRC_RESY	0..15	R/W	CRC shift register (result / init value) high part

Table 28: CRC unit register map

2.15. Custom Functions Unit (CFU)

The custom functions unit (VHDL component **neo430_cfu.vhd**) is dedicated for user-defined processor extensions. In contrast to specific hardware accelerators connected to the Wishbone bus interface, the CFU allows the implementation of low-latency and tightly-coupled hardware extensions.

Implementation Control

In case you want to use the custom functions unit to implement user-defined hardware extensions use the CFU_USE generic of the processor top entity (see cut-out below) to control implementation. By default, the CFU will **NOT** be synthesized since the provided The actual CFU template does not implement any "useful" operations- it is up to you to implement them ;)

```
CFU_USE => false, -- implement custom functions unit? (default=false)
```

Operation

From a software point of view, the CFU implements 8 16-bit wide registers, that can be used for writing and reading data. These register can only be accessed using full 16-bit-word read/write transfers. The default CFU from the project's rtl folder implements these 8 with no additional computation logic. Therefore, this CFU behaves like a simple register file.

If you want to implement a custom functions, for instance some kind of crypto computations, the actual computations have to made based on data, which is available via one of the 8 CFU register addresses. Take a look at the other processor modules like the GPIO controller or the Timer to get an idea on how to use the register interface. Also make sure to get familiar with the CPU bus protocol, introduced at the beginning of this chapter.

Register Map

Address	Name	Bit(s)	R/W	Function
0xFFD0	CFU_REG0	0..15	R/W	CFU user register 0
0xFFD2	CFU_REG1	0..15	R/W	CFU user register 1
0xFFD4	CFU_REG2	0..15	R/W	CFU user register 2
0xFFD6	CFU_REG3	0..15	R/W	CFU user register 3
0xFFD8	CFU_REG4	0..15	R/W	CFU user register 4
0xFFDA	CFU_REG5	0..15	R/W	CFU user register 5
0xFFDC	CFU_REG6	0..15	R/W	CFU user register 6
0xFFDE	CFU_REG7	0..15	R/W	CFU user register 7

Table 29: Custom functions unit register map

2.16. Pulse Width Modulation Controller (PWM)

The PWM controller (VHDL component **neo430_pwm.vhd**) implements a pulse width modulation controller with four independent channels and up to 8-bit resolution per channel. It is based on an 8-bit counter with four programmable threshold comparators that define the actual duty cycle of each channel. The output signals are available in the processor's top entity via the `pwm_o` port. The controller can be used to drive a fancy RGB-LED with 24-bit true color, to dim LCD backlight or even for motor control. An external integrator (RC low-pass filter) can be used to smooth the generated "analog" signals.

The width of the internal counter can be either set to 4 bit or 8 bit y writing a flag in the unit's control register. By this, the resolution of the PWM module's channels is reduced but also the sampling frequency is increased. The `neo430_pwm.h` library provides basic functions for using the PWM controller.

Implementation Control

If you do not need the processor-internal PWM controller (maybe you have attached a far more complex one to the Wishbone bus), you can exclude it from synthesis using the `PWM_USE` generic of the processor top entity (see cut-out below).

```
PWM_USE => true, -- implement PWM controller? (default=true)
```

Operation

The PWM controller is activated by setting the `PWM_CT_EN` bit in the module's control register `PWM_CT`. When this flag is cleared, the unit is reset and all output channels are set to zero.

The 8-bit duty cycle for each channel, which represents the channel's "intensity", can be specified via the according `PWM_CHxx` register. Note, that one control register is used to define the duty cycle for two channels at once, so the `PWM_CH10` register defines the duty cycle for PWM outputs 0 and 1 and the `PWM_CH23` register defines the duty cycle for PWM outputs 3 and 4.

The effective bit width of the internal PWM counter can be defined to be 4 to 8 bit wide. The width is configured via the `PWM_CT_SIZE_SEL` flag in the unit's control register. When this flag is set to zero the effective PWM counter bit width is 4 bit wide. When the flag is set the effective PWM counter bit width is 8 bit wide. This configuration is used for all four PWM channels. Note, that a small effective bit width will increase the sampling rate but will also decrease the resolution.

Based on the duty cycles `PWM_Chxx` the according analog output voltage (relative to the IO supply voltage) of each channel can be computed by the formulaabelow.

$$Intensity_{xx} = \frac{\text{PWM\_CHxx}}{2^4}\% \quad for\ 4-bit\ counter\ width$$

$$Intensity_{xx} = \frac{\text{PWM\_CHxx}}{2^8}\% \quad for\ 8-bit\ counter\ width$$

The frequency of the generated PWM signals is defined by the effective counter bit width and the PWM operating frequency. This operating frequency is derived from the main system clock and divided by a prescaler via the three `PWM_CT_PRSCx` bits in the unit's control register. The following prescalers are available:

Prescaler bits configuration:	000	001	010	011	100	101	110	111
Resulting prescaler *PRSC*:	2	4	8	64	128	1024	2048	4096

The resulting PWM frequency is defined by:

$$f_{PWM} = \frac{f_{main}}{2^4 * PRSC} \quad for\ 4-bit\ counter\ width$$

$$f_{PWM} = \frac{f_{main}}{2^8 * PRSC} \quad for\ 8-bit\ counter\ width$$

Example

The system operates at 100MHz, the clock is divided by a prescaler of 2 (`PWM_CT_SIZEx` = 0b000) and the effective bit width is set to 4 bits (`PWM_CT_SIZE_SEL` = 0):

$$f_{PWM} = \frac{100\,MHz}{2^4 * 2} = 3.125\,MHz$$

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFFE0	PWM_CT	0	PWM_CT_EN	-/W	Enable (activate) PWM controller
		1	PWM_CT_PRSC0	-/W	Clock prescaler select bit 0
		2	PWM_CT_PRSC1	-/W	Clock prescaler select bit 1
		3	PWM_CT_PRSC2	-/W	Clock prescaler select bit 2
		4	PWM_CT_GPIO_PWM	-/W	Use channel 3 to modulate GPIO unit's output
		5	PWM_CT_SIZE_SEL	-/W	Counter size (1: 8bit, 0: 4bit)
		6..15: *reserved*		-/-	*Reserved*
0xFFE2	PWM_CH10	0..7		R/W	Duty cycle for channel 0
		8..15		R/W	Duty cycle for channel 1
0xFFE4	PWM_CH32	0..7		R/W	Duty cycle for channel 2
		8..15		R/w	Duty cycle for channel 3
0xFFE6	-	0..15		-/-	*reserved*

Table 30: PWM controller register map

PWM Modulation of the GPIO Unit's Output Port

Channel 3 of the PWM controller can alternatively be used to modulate the output port of NEO430 general purpose input/output controller (GPIO). By setting the `PWM_CT_GPIO_PWM` bit in the PWM controller's control register, output channel 3 (top entity port `pwm_o(3)`) is permanently set to zero and the according PWM signal is routed to the GPIO unit to modulate the whole output port.

For example, this feature can be used to uniquely control several LEDs via the GPIO controller while also controlling their intensity via the PWM controller.

2.17. True Random Number Generator (TRNG)

The NEO430 true random number generator (VHDL component **`neo430_trng.vhd`**) provides true random numbers for your application. Instead of using a pseudo RNG like a LFSR, the TRNG of the processor uses a simple, straight-forward ring oscillator as physical entropy source. Hence, voltage and thermal fluctuations are used to provide true physical random data. It features a platform independent architecture based on two papers which are cited at the bottom of the following pages.

Implementation Control

By default, the TRNG will **not** be synthesized. You can enable synthesis by using the `TRNG_USE` generic of the processor top entity (see cut-out below).

```
TRNG_USE => false, -- implement TRNG? (default=false)
```

Operation

The TRNG features a single control register (`TRNG_CT`) to control operation and to read the generated random data. When the `TRNG_CT_EN` bit is set, the TRNG starts operation. After activation, the user should wait some clock cycles to ensure all elements of the inverter chain are enabled and nicely oscillating.

The generated random bytes can be read from the lowest 8 bits of `TRNG_CT_EN` register. Note, that the TRNG needs 8 clock cycles to generate a new random byte. During this sampling time the current output random data is kept in the output register until the sampling of the new byte has completed.

The `neo430_trng.h` library provides basic functions for using the TRNG.

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFFEC	TRNG_CT	0..7		R/-	Random data byte output
		8..14		-/-	*Reserved*
		15	TRNG_CT_EN	R/W	Enable (activate) TRNG

Table 31: TRNG register map

Architecture

The NEO430 TRNG is based on the *GARO **Ga**lois **R**ing **O**scillator TRNG*[8]. Basically, this architecture is an asynchronous LFSR constructed from a chain of inverters. Before the output signal of one oscillator is passed to the input of the next one, the signal can be XORed with the final output signal of the inverter chain (see image below) using a switching mask (f).

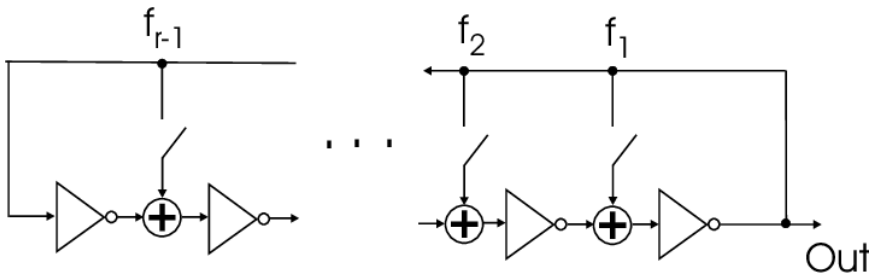

Figure 7: GARO architecture, image from: High-Speed True Random Number Generation with Logic Gates Only, by Markus Dichtl, Siemens. Published in: Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings (pp.45-62)

The default setup of the NEO430 TRNG uses a total of 5 inverters and f = 0101 as switching mask. If you wish, you can modify this setup using the unit's VHDL "user configuration" parameters in the `rtl/core/neo430_trng.vhd` file:

```
constant num_oscs_c  : natural := 5; -- number of oscillators (default=5)
constant garo_taps_c : std_ulogic_vector(num_oscs_c-2 downto 0) := "0101";
```

To prevent the synthesis tool from doing logic optimization and thus, removing all but one inverter, the TRNG uses simple latches to decouple an inverter and its actual output. By this, the latches introduce an additional logic delay, but also allow to implement the circuit without the EDA tool complaining. The latches are reset when the TRNG is disabled and are enabled one by one by a simple shift register when the TRNG is activated. By this, the TRNG provides a platform independent architecture[9] since no specific VHDL attributes are required.

The single-bit output signal of the GARO array is fed through two sampling flip flops to eliminate any metastability beyond this point. This sampled signal is used as input for a simple 8-bit Fibonacci LFSR post-processing to provide a 'better' uniform distribution: The random bit from the GARO array is XORed with bits 7, 5, 4 and 3 of the LFSR and then used as new input bit for the LSB of the shift register. As soon as 8 bits have bin sampled, the resulting data byte is moved to the output register and is available for fetching by the CPU bus. The LFSR is also configurable by using constants in the TRNG's VHDL source file:

```
constant use_lfsr_c  : boolean := true; -- use LFSR for post-processing (default=true)
constant lfsr_taps_c : std_ulogic_vector(7 downto 0) := "10111000"; -- LFSR feedback
```

8 "Enhancing the Randomness of a Combined True Random Number Generator Based on the Ring Oscillator Sampling Method" by Mieczyslaw Jessa and Lukasz Matuszewski

9 "Extended Abstract: The Butterfly PUF Protecting IP on every FPGA" by Sandeep S. Kumar, Jorge Guajardo, Roel Maesyz, Geert-Jan Schrijen and Pim Tuyls, Philips Research Europe, 2008

Quality of the Random Numbers

I'm not a math guy, so statistical analysis is not my favorite waste of time... However, I'll try to show some elementary statistical properties of the TRNG's random numbers. The data used for evaluation was generated by a NEO430 setup on an Intel Cyclone IV FPGA running at 100 MHz.

The histogram in the figure below shows the relative (percentage) occurrence of all possible random data values (8-bit → 256 different values → 0 to 255) for *2 000 000 000* samples (blue boxes, "real"). For a perfect uniform distribution, each possible value would account for $\bar{x}_{perfect} = 100\,\% \div 256 = 0.390625\,\%$ with a standard deviation of $\sigma_{perfect} = 0\,\%$ of the total sample amount (orange line, "perfect"). The arithmetic mean of the real occurrences also sum up to $\bar{x}_{real} = 0.390625\,\%$ (due to the post-processing LFSR) but with a standard deviation of $\sigma_{real} = 0.00077993\,\%$ caused by the imperfect entropy source.

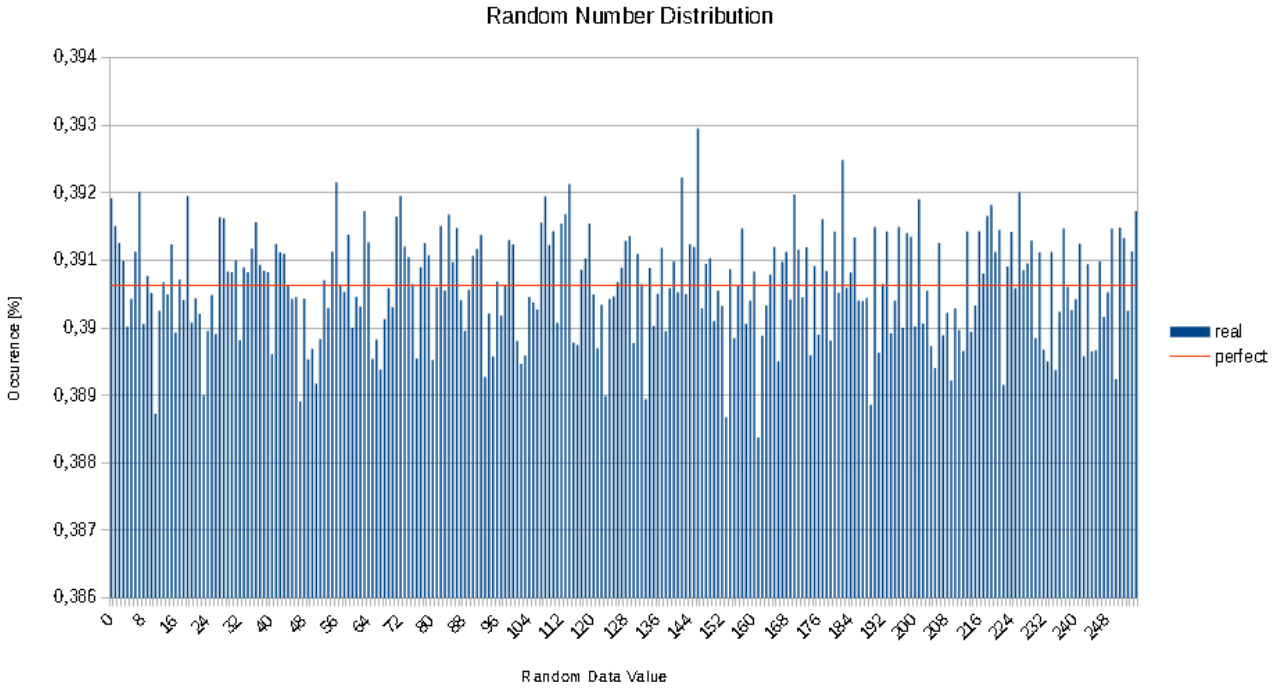

Figure 8: NEO430 TRNG relative random number distribution histogram (2 000 000 000 samples)

I cannot guarantee the NEO430 TRNG is a cryptographically secure random number generator since I have not conducted any kind of a more sophisticated analysis. You can try to enhance the theoretical entropy by increasing the number of inverters in the GARO array. A better analysis regarding the TRNG cryptographical quality could be done by using dedicated test suites.

2.18. External Interrupts Controller (EXIRQ)

To expand the IRQ capabilities of the CPU, the processor features a controller for external interrupts (VHDL component **neo430_exirq.vhd**). This controller features eight external interrupt request lines with according acknowledge signals. By configuring a global trigger, the IRQ lines can be either triggered by a high-level or a rising edge. Each interrupt channel can be individually activated or deactivated.

If your application requires more than eight interrupt lines, you should disabled this unit for synthesis and connected a more sophisticated interrupt controller via the Wishbone bus.

Implementation Control

By default, the EXIRQ will always be synthesized. You can enable synthesis by using the `EXIRQ_USE` generic of the processor top entity (see cut-out below).

```
EXIRQ_USE => true, -- implement EXIRQ? (default=true)
```

External Interrupt

Whenever an enabled interrupt channel of the controller is triggered, the according interrupt request is brought to the CPU via the `IRQVEC_EXT` vector. So, regardless which channel is triggered, always the same interrupt handler will be started. However, the EXIRQ driver software library (`neo430_exirq.h`) allows to define unique handler functions for each external interrupt channel. More details are presented in the further course of this chapter.

Address	IRQ Vector Name	Priority	Source
0xC006	IRQVEC_EXT	4 (lowest)	External interrupt request

Table 32: EXIRQ interrupt vector

When the external interrupts controller (EXIRQ) is excluded from synthesis, the CPU IRQ line triggering IRQVEC_EXT is directly connected to the port `ext_irq_i(0)` while all other interrupt input lines are unused. Also, only `ext_ack_o(0)` is connected.

Operation

The EXIRQ controller features a single control register (`EXIRQ_CT`). The unit is enabled when setting the `EXIRQ_CT_EN` bit. If this bit is cleared during operation, all buffered interrupt requests are deleted. Each interrupt request channel features a unique enable signal (`EXIRQ_CT_IRQx_EN`), which activated the according channel when set. The actual trigger can only be set globally for all channels. Writing a 1 to the `EXIRQ_CT_TRIG` bit configures the channels to trigger on a rising edge. By writing a zero to this bit the channels trigger on a high-level.

Channel 0 (`ext_irq_i(0)`) has the highest priority, while channel 7 has the lowest. If several interrupt request arise at the same time, the one with highest priority will be processed while the remaining ones are

buffered. When an interrupt is signaled to the CPU, the application program can determine which channel caused the request by checking the `EXIRQ_CT_SRCx` bits. A “000” indicates channel 0, a “001” channel 1 and so on. During the read-process of the control register the current interrupt request is also acknowledged and the according bit of the `ext_ack_i` output port is set for one clock cycle.

The `neo430_exirq.h` library provides all required functions for configuring and using the external interrupts controller.

Using Low-Level or Falling-Edge as Trigger

The EXIRQ only provides a global trigger type, which can be either a high-level or a rising edge. If your application requires the counterparts, you need to manually add inverters to the according interrupt input channels.

Register Map

Address	Name	Bit(s)	(Name)	R/W	Function
0xFFEE	EXIRQ_CT	0	EXIRQ_CT_SRC0	R/-	IRQ source bit 0
		1	EXIRQ_CT_SRC1	R/-	IRQ source bit 1
		2	EXIRQ_CT_SRC2	R/-	IRQ source bit 2
		3	EXIRQ_CT_TRIG	R/W	Global trigger type (0: high level, 1: rising edge)
		4	EXIRQ_CT_EN	R/W	Global unit enable bit
		5..7		-/-	*Reserved*
		8	EXIRQ_CT_IRQ0_EN	R/W	Enable IRQ channel 0
		9	EXIRQ_CT_IRQ1_EN	R/W	Enable IRQ channel 1
		10	EXIRQ_CT_IRQ2_EN	R/W	Enable IRQ channel 2
		11	EXIRQ_CT_IRQ3_EN	R/W	Enable IRQ channel 3
		12	EXIRQ_CT_IRQ4_EN	R/W	Enable IRQ channel 4
		13	EXIRQ_CT_IRQ5_EN	R/W	Enable IRQ channel 5
		14	EXIRQ_CT_IRQ6_EN	R/W	Enable IRQ channel 6
		15	EXIRQ_CT_IRQ7_EN	R/W	Enable IRQ channel 7

Table 33: EXIRQ register map

Using the EXIRQ Controller in Software

The complexity of the EXIRQ controller is hidden by the `neo430_exirq.h` library. For configuring the EXIRQ controller you only need to use the `neo430_exirq_config()` function from this library. A specific struct (`struct neo430_exirq_config`) is passed to this function that includes the actual configuration:

- **`uint16_t address[8]`** : A 16-bit wide and 8 entries deep array that stores the base address of the handler functions for the external interrupt channels 0 to 7.
- **`uint8_t enable`** : An 8-bit wide variable to enable/disable each of the interrupt channels, i.e. bit #0 enables channel 0 and its according address(0) and so on.
- **`uint8_t trigger`** : This variable can be set to 1 for all channels triggering on a rising edge or to 0 for all channels triggering on a high level.

When the EXIRQ configuration function is called, the actual configuration stored in the struct is stored to a hidden global variable. Also, an actual interrupt handler that handles the interrupt request from the EXIRQ controller is included and its base address is automatically stored to the according interrupt vector location (`IRQVEC_EXT`). Whenever the EXIRQ triggers this interrupt, the interrupt handler is started. This handler also reads the according source bits from the EXIRQ and **calls** the according function (defined by the configuration struct). Hence, the called handler for the external interrupt channels must be normal functions, as the calling interrupt handler already does all the required stack spilling.

A simple example for the configuration is shown below (a complete example project for using the EXIRQ can be found in `sw\example\exirq_test`):

```
// use this struct for configuring the EXIRQ controller
struct neo430_exirq_config_t exirq_config;

// initialize handler addresses
exirq_config.address[0] = (uint16_t)(&ext_irq_ch0_handler);
exirq_config.address[1] = (uint16_t)(&ext_irq_ch1_handler);
exirq_config.address[2] = (uint16_t)(&ext_irq_ch2_handler);
exirq_config.address[3] = (uint16_t)(&ext_irq_ch3_handler);
exirq_config.address[4] = 0; // set unused vectors to zero
exirq_config.address[5] = 0;
exirq_config.address[6] = 0;
exirq_config.address[7] = (uint16_t)(&ext_irq_ch7_handler);

// enable only actually used IRQ channels
exirq_config.enable = 0b10001111;

// use rising edge as trigger for all channels
exirq_config.trigger = 1;

// send configuration
neo430_exirq_config(exirq_config);

// activate EXIRQ controller
neo430_exirq_enable();

// enable global interrupts and go to sleep
neo430_eint();
while(1) {
  neo430_sleep();
}
```

The actual handler functions for the external interrupts have to be “normal functions” - so no specific attributes like “interrupt” must be used. Also, these handler functions must not have any parameters or return values. One handler function (for channel 0) is shown below:

```
void ext_irq_ch0_handler(void) {

  neo430_gpio_pin_toggle(0);
}
```

Do not use “interrupt” attributes for the external interrupt channel handler functions that are called by the EXIRQ controller as this will corrupt the stack.

3. Software Architecture

Software development for the NEO430 is based on the freely-available **TI msp430-gcc compiler toolchain**, which can be downloaded from (use the "compiler only" package):

http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPGCC/latest/index_FDS.html

With the compiler tool chain, you can turn your C/C++ programs into an NEO430 executable. Generating an executable is done in several consecutive steps (all done by the provided compilation scripts):

1. The application start-up code (*crt0.asm*) is assembled into an object file. This start-up codes is responsible for the minimal required hardware initialization.

2. The actual application program is compiled together with all included files and libraries. The code is optimized for size (**-Os**) by default. If required, all library functions are also compiled.

3. In the next step, all generated object files are linked together using the special NEO430 linker script (`neo430_linker_script.x`). This specific linker script generates a final object file, that already represents the actual memory layout of the NEO430. Also, an ASM listing file is generated (`main.s`) for debugging.

4. The final program object file is generated.

5. In the last step, the program image is converted into a NEO430 executable (`main.bin`) binary. This file can be uploaded and executed by the NEO430 bootloader. Additionally, an executable VHDL memory initialization image for the IMEM is generated and directly installed into the `neo430_application_image.vhd` file – no manual copy required. This is only relevant if the instruction memory is configured as true ROM.

The last step is done by a small C program, which is located in the `sw/tools/image_gen` folder. It is automatically compiled when you are using the NEO430 toolchain for the first time.

The size of the final executable, which is printed in your console by the make script, only represents the size of the executable image. Additional RAM is required for allocating dynamic memory for the stack and the head (actual size depends on the application program).

3.1. Executable Program Image

As the last step of the program compilation flow, the NEO430 executables are generated. The binary version can be uploaded to the processor to be executed directly and/or programmed into an externa flash SPI or EEPROM. The executable VHDL IMEM memory initialization data is directly inserted into the processor's IMEM image VHDL file. The compilation script uses a specific linker script to generate the final image:

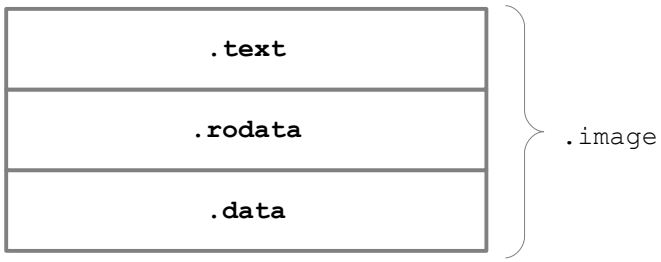

Figure 9: Construction of the final program image

3.1.1. Image Sections

The final executable image consists of the following three sections:

`.text`	Executable instructions, including start-up, application and termination code
`.rodata`	Read-only data (constants like strings)
`.data`	Pre-initialized variables (will be copied into RAM during start-up)

3.1.2. Dynamic Memory

The remaining memory – the memory after the `.data` section until the end of the RAM – is used for the dynamic data during run time. This data includes the stack and the heap. The stack grows from the end of the memory down to the end of the `.data` section. The heap grows from the end of the `.data` section up to the end of the memory. Make sure there is no collision between the heap and the stack when using dynamic memory allocation!

3.1.3. Application Start-Up Code

During the linking process, the application start-up code `crt0.asm` is placed right before the actual application. The resulting code represents the applications `.text` segment and thus, the final executable. The start-up code implements a basic system initialization:

- Setup the stack-pointer according to the memory size configuration from the `CPUID` registers
- Set all IO device registers to 0x0000
- Clear complete DMEM, including `.bss` segment and the interrupt vectors, copy the `.data` section from IMEM to DMEM
- Initialize all CPU data registers
- Call the application's main function
- If the main function returns, the watchdog timer is deactivated, interrupts are disabled and the CPU is set to eternal sleep mode

3.1.4. Executable Image Formats

The auxiliary image generator program (`sw\tools\image_gen`) is used to either create an executable binary or an executable VHDL memory initialization image. The actual conversion target is given by the first argument when executing the image generator. Valid target options are listed below. The second argument determines the input file and the third argument specifies the output file.

`-app_bin`	Generates an executable binary "main.bin" (for UART uploading via the bootloader) in the project's folder (including a file header!!!)
`-app_img`	Generates an executable VHDL memory initialization image for the IMEM. This function is meant to generate the "neo430_application_image.vhd" file.
`-bld_img`	Generates an executable VHDL memory initialization image for the DMEM. This function is meant to generate the "neo430_bootloader_image.vhd" file.

There is a special thing about the binary executable format: This executable version has a very small header consisting of three 16-bit words located right at the beginning of the file. The first word (**red**) is the signature word and is always **`0xCAFE`**. Based on this word, the bootloader can identify a valid image file. The next word (**green**) represents the size in bytes of the program image (so this value is always 6 bytes less than the actual file size). A simple XOR checksum of the program data is given by the third word (**blue**). This checksum is computed by XOR-ing all program data words (no header data!) of the program image. Below you can see an exemplary binary executable.

```
CA FE 01 7A 19 59 43 03 42 18 FF E8 42 19 FF EA 58 09 43 02 49 01 83 21 43 82 FF
9E 43 82 FF A6 43 82 FF B4 43 82 FF B2 43 82 FF C4 40 B2 47 00 FF D0 98 09 24 04
43 88 00 00 53 28 3F FA 40 35 01 7A 40 36 01 7A 40 37 80 00 95 06 24 04 45 B7 00
00 53 27 3F FA 43 04 43 05 43 06 43 07 43 08 43 09 43 0A 43 0B 43 0C 43 0D 43 0E
43 0F 12 B0 00 72 43 02 D0 32 00 10 42 1F FF EE 42 1B FF EC 43 0C 43 0D 4F 0D 4B
0E 43 0F DE 0C DF 0D 3C 04 50 3C 6A 00 63 3D 53 1F 93 1D 2F FA 90 3C 96 00 2F F7
43 4E 3C 0E 93 6E 24 02 92 6E 20 07 C3 12 10 0F C3 12 10 0F C3 12 10 0F 3C 02 C3
12 10 0F 53 5E 90 3F 01 00 2F EF 4E 4E 10 8E DF 0E 4E 82 FF A4 40 B2 FE 81 FF A6
40 3F 01 3E 12 B0 01 16 B2 B2 FF E2 20 07 40 3F 01 5A 12 B0 01 16 43 1F 40 30 01
14 43 82 FF B2 43 0F 4F 0E F0 3E 00 FF 53 1F 4E 82 FF B2 40 3E 00 0B 3C 04 43 3D
43 03 53 3D 23 FD 53 3E 23 FA 3F F0 41 30 3C 0F 90 7E 00 0A 20 06 B2 B2 FF A6 23
FD 40 B2 00 0D FF A2 B2 B2 FF A6 23 FD 11 8E 4E 82 FF A2 4F 7E 93 4E 23 EE 41 30
42 0A 69 6C 6B 6E 6E 69 20 67 45 4C 20 44 65 64 6F 6D 70 20 6F 72 72 67 6D 61 00
0A 72 45 6F 72 21 72 4E 20 20 6F 49 50 20 4F 6E 75 74 69 73 20 6E 79 68 74 73 65
7A 69 64 65 00 21
```

Hex-view of an executable binary image including colorized header

3.2. Internal Bootloader

The bootloader requires at least the TIMER and the UART units to be included into the design! The GPIO unit is optional since is used just for status indication.

The included bootloader of the NEO430 processor allows you to upload new program images at every time. If you have an external SPI EEPROM connected to the processor, you can store the program image to it and the system can directly boot it after reset without any user interaction. But we will talk about that later…

To interact with the bootloader, attach the UART signals of the processor via a COM port (-adapter) to a computer, configure your terminal program using the following settings and perform a reset of the processor.

Terminal console settings (19200-8-N-1):
- **19200** Baud
- **8** data bits
- **N**o parity bit
- **1** stop bit
- Newline on "`\r\n`" (carriage return, newline)
- No transfer protocol for sending data, just the raw byte stuff

The bootloader uses bit #0 of the GPIO output port as high-active status LED (all other outputs are set to low level by the bootloader). After reset, this LED will start blinking at ~2Hz and the following intro screen should show up in your terminal:

```
<< NEO430 Bootloader >>

BLV: Nov 19 2019
HWV: 0x0320
USR: 0x4788
CLK: 0x05F5E100
ROM: 0x1000
RAM: 0x0800
SYS: 0xFABF

Autoboot in 8s. Press key to abort.
```

NEO430 bootloader start-up screen

This start-up screen gives some brief information about the bootloader version and several system parameters (all in hexadecimal representation):

- **BLV**: Bootloader version (compile date)
- **HWV**: Hardware version
- **USR**: User code defined by the `USER_CODE` generic of the top entity
- **CLK**: Clock speed in Hz
- **ROM**: Size of internal IMEM in bytes
- **RAM**: Size of internal DMEM in bytes
- **SYS**: System features (synthesized modules)

Now you have 8 seconds to press any key. Otherwise, the bootloader starts the auto boot sequence (see next chapter).

When you press any key within the 8 seconds, the actual bootloader user console starts:

```
<< NEO430 Bootloader >>

BLV: Nov 19 2019
HWV: 0x0320
USR: 0x4788
CLK: 0x05F5E100
ROM: 0x1000
RAM: 0x0800
SYS: 0xFABF

Autoboot in 8s. Press key to abort.
Aborted.

Commands:
 d: Dump MEM
 e: Load EEPROM
 h: Help
 p: Store EEPROM
 r: Restart
 s: Start app
 u: Upload
CMD:>
```

NEO430 bootloader console

The auto-boot countdown is stopped and now you can enter a command from the list to perform the corresponding operation:

- **d**: Core dump of full address space (can be aborted at any time by pressing any key)
- **e**: Load application image from SPI EEPROM (at SPI.CS0) into IMEM
- **h**: Show the help text (again)
- **p**: Store the complete IMEM content as boot image to the SPI EEPROM (at SPI.CS0)
- **r**: Restart the bootloader
- **s**: Start the application, which is currently stored in IMEM
- **u**: Upload new program executable image (**raw** *.bin file) via UART into the IMEM

A new program is uploaded to the NEO430 by using the upload function. The compile scripts of this project generate a compatible binary executable (*.bin format), which must be transmitted by your terminal program without using any kind of protocol – just raw data. When the image is completely uploaded, it resides in the IMEM and you can start executing it using the “Start app” option. If you want to take at look at the whole address space, perform a “Core dump”. This is very useful for simple debugging or if you just want to see what's going on.

The complete content of the IMEM can be stored in an external SPI EEPROM (program it via “Store EEPROM”). The bootloader can copy the image from the EEPROM at start up and automatically launches it. Of course, you can also load it manually using the “Load EEPROM” option.

When the bootloader is implemented (enabled via the `BOOTLD_USE` generic) the IMEM is not initialized by the bitstream at all. This allows a mapping of the IMEM to memory primitives, that do not support initialization during bitstream upload.

3.2.1. Auto Boot Sequence

When you reset the NEO430 processor, the bootloader waits 8 seconds for a console user input before it starts the automatic boot sequence. This sequence tries to fetch a valid boot image from the external SPI EEPROM, connected to SPI chip select bit #0. If a valid boot image is found and can be successfully transferred into the internal IMEM, it is automatically started. If no SPI EEPROM was detected or if there was no valid boot image found, the bootloader stalls and the status LED is permanently activated.

3.2.2. Error Codes

If something goes wrong during the bootloader operation, an error code is shown. In this case, the processor stalls, a bell command and one of the following error codes is send to the terminal, the status LED is permanently activated and the system must be manually reset.

- **ERR_00**: This error occurs if the attached EEPROM cannot be accessed during write transfers. Make sure you have the right type of EEPROM and that it is connected properly to the NEO430's SPI port at chip select #0 (CS0).
- **ERR_01**: If you have implemented the IMEM as true ROM (so it cannot be written) this error pops up when trying to install a new application image (e.g. via the UART). Set the `IMEM_AS_ROM` configuration generic of the processor top entity to 'false' to implement the IMEM as writable RAM.
- **ERR_02**: If you try to transfer an invalid executable (via UART or from EEPROM), this error message shows up. Also, if no EEPROM was found during a boot attempt, this message will be displayed.
- **ERR_04**: Your program is way too big for the internal IMEM. Increase the IMEM size of your NEO430 project or reduce your application code.
- **ERR_08**: This indicates a checksum error. Something went wrong during the transfer of the program image (upload via UART or loading it from EEPROM). If the error was caused by a UART upload, just try it again. When the error was generated during an EEPROM access, the stored image might be corrupted.

3.3. Software Libraries

The NEO430 project provides a set of C libraries that allow an easy usage of all of the core's peripheral and CPU features. All you need to do is to include the main NEO430 library file in your application's main C file. The main library file as well as all sub-libraries are located in `sw/lib/neo430` and are automatically added by the makefile to the include path.

```
#include <neo430.h>
```

This main include file will automatically include all driver libraries (e.g. the drivers for the UART). The following list shows all the included driver libraries, which are located in the `sw/lib/neo430/include` folder. The according source files are located in `sw/lib/neo430/source`. Take a look at the according library file when you want to use the according hardware unit – you will find a rich set of handy functions :)

Library file	HW module
neo430.h	All (mandatory for all other libraries)
neo430_cpu.h/.c	CPU
neo430_crc_.h/.c	Cyclic redundancy check unit (CRC)
neo430_exirq.h/.c	External interrupts controller (EXIRQ)
neo430_gpio.h/.c	General purpose input/output unit (GPIO)
neo430_muldiv.h/.c	Multiplier/divider unit (MULDIV)
neo430_pwm.h/.c	Pulse-width modulation unit (PWM)
neo430_spi.h/.c	Serial peripheral interface (SPI)
neo430_timer.h/.c	High-precision timer (TIMER)
neo430_trng.h/.c	True random number generator (TRNG)
neo430_twi.h/.c	Two wire serial interface (TWI)
neo430_uart.h/.c	Universal asynchronous receiver and transmitter (UART)
neo430_wdt.h/.c	Watchdog timer unit (WDT)
neo430_wishbone.h/.c	Wishbone bus interface unit (WB32)

Table 34:NEO430 software library files

The NEO430 software library provides driver functions for the CPU and all peripheral devices. Check the `sw\lib\neo430\source` folder – each driver library is highly commented to explain how to use the different functions.

4. Let's Get It Started!

To make your NEO430 project run, follow the guides from the upcoming sections. There are several guides for the application compilation and all details of the project.

4.1. General Hardware Setup

Follow these steps to build the FPGA hardware of your NEO430 project. In this tutorial, we will use a test implementation of the processor – using most of the processor's optional modules but just propagating the minimal signals to the outer world. Hence, this guide is intended as evaluation project to check out the NEO430. A little note: The order of the following steps might be a little different for your specific EDA tool.

1. Create a new project with your FPGA EDA tool of choice (Xilinx Vivado, Intel Quartus, Lattice Diamond/Radiant, …).

2. Add all VHDL files from the project's **`rtl/core`** folder to your project. Make sure to *reference* the files only – do not copy them.

3. Make sure to add all the rtl files to a new **`library`** called "**`neo430`**".

4. The **`neo430_top.vhd`** file is the top entity of the NEO430 processor. If you already have a design, instantiate this unit into your design and proceed. If you do not have a design yet and just want to check out the NEO430 – no problem! Use the **`neo430_test.vhd`** file from the **`rtl/top_templates`** folder as top entity. Of course, you also need to add this file to your project. This tutorial assumes to use this test entity as top entity, but the basic steps are the same when using the core itself as part of your project.

5. The configuration of the NEO430 processor is done using the generics of the instantiated processor top entity (here, done in the `neo430_test.vhd` file). Let's keep things simple at first and use the default configuration (see below). But there is one generic, that has to be set according to your FPGA / board: The clock frequency of the top's clock input signal (`clk_i`). Use the **`CLOCK_SPEED`** generic to specify your clock source's frequency in Hertz (Hz). The default value, that you need to adapt, is marked in **red**:

```
neo430_top_test_inst: neo430_top
generic map (
  -- general configuration --
  CLOCK_SPEED => 100000000,     -- main clock in Hz
  IMEM_SIZE   => 4*1024,        -- internal IMEM size in bytes, max 48kB (default=4kB)
  DMEM_SIZE   => 2*1024,        -- internal DMEM size in bytes, max 12kB (default=2kB)
  -- additional configuration --
  USER_CODE   => x"4788",       -- custom user code
  -- module configuration --
  MULDIV_USE  => true,  -- implement multiplier/divider unit? (default=true)
  WB32_USE    => true,  -- implement WB32 unit? (default=true)
  WDT_USE     => true,  -- implement WDT? (default=true)
  GPIO_USE    => true,  -- implement GPIO unit? (default=true)
  TIMER_USE   => true,  -- implement timer? (default=true)
  UART_USE    => true,  -- implement UART? (default=true)
  CRC_USE     => true,  -- implement CRC unit? (default=true)
  CFU_USE     => false, -- implement custom functions unit? (default=false)
  PWM_USE     => true,  -- implement PWM controller? (default=true)
  TWI_USE     => true,  -- implement two wire serial interface? (default=true)
  SPI_USE     => true,  -- implement serial peripheral interface? (default=true)
  TRNG_USE    => false, -- implement true random number generator? (default=false)
  EXIRQ_USE   => true,  -- implement external interrupts controller? (default=true)
  -- boot configuration --
  BOOTLD_USE  => true,          -- implement and use bootloader? (default=true)
  IMEM_AS_ROM => false          -- implement IMEM as read-only memory? (default=false)
)
```

6. If you feel like it – or if your FPGA does not provide enough resources – you can modify the memory sizes (IMEM and DMEM) or exclude certain modules from implementation. But as mentioned above, let's keep things simple and use the standard configuration for now. We will come back to the customization of all those configuration generics in later chapters.

7. Depending on your FPGA tool of choice, it is time now (or later?) to assign the signals of the test setup top entity to the according pins of your FPGA board. All the signals can be found in the entity:

```
entity neo430_test is
  port (
    -- global control --
    clk_i      : in  std_ulogic; -- global clock, rising edge
    rst_i      : in  std_ulogic; -- global reset, async, LOW-active
    -- gpio --
    gpio_o     : out std_ulogic_vector(07 downto 0); -- parallel output
    -- serial com --
    uart_txd_o : out std_ulogic; -- UART send data
    uart_rxd_i : in  std_ulogic  -- UART receive data
  );
end neo430_test;
```

8. Attach the clock input to your clock source and connect the reset line to a button of your FPGA board. Check whether it is low-active or high-active – the reset signal of the processor must be **low-active**, so maybe you need to invert the input signal. If possible, connected at least bit #0 of the GPIO output port to a high-active LED (invert the signal when your LEDs are low-active). Finally, connect the UART signals to your serial host interface (dedicated pins, USB-to-serial converter, etc.). The final test setup is illustrated in the figure below.

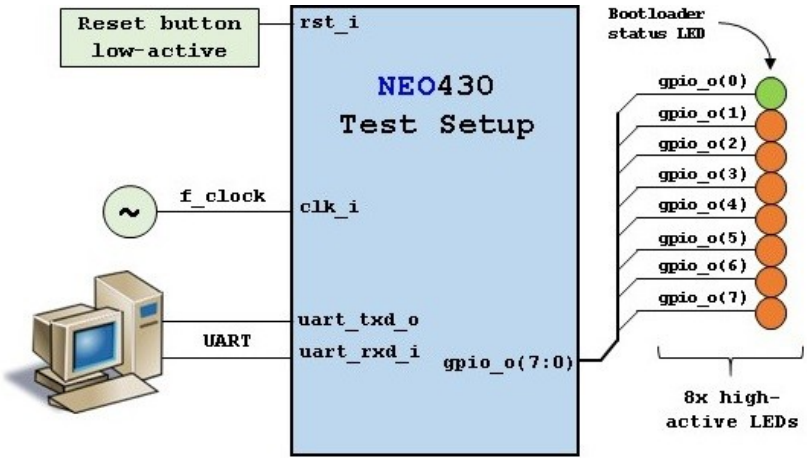

Figure 10: External hardware configuration of the NEO430 test implementation (neo430_test.vhd)

9. Perform the project HDL compilation (synthesis, mapping, bitstream generation).

10. Download the generated bitstream into your FPGA (“program” it) and press the reset button (just to make sure everything is sync).

11. Done! If you have assigned the bootloader status LED (bit #0 of the GPIO output port), it should be flashing now and you should receive the bootloader start prompt via the UART.

4.2. General Software Setup

So, the hardware thing is done. Now it is time to prepare the general part of the software flow.

1. At first, download the latest version of the **TI msp430-gcc compiler tool chain**. Make sure to download the Windows version when using the Windows Powershell. Download the Linux version when you are using the Windows Bash Subsystem or a native Linux system. You can downloaded the compiler without registration (select the "compiler only" package) from **http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPGCC/latest/index_FDS.html**.

2. Extract/install all files into a folder somewhere in your file system. Remember where you have installed the compiler, since this will be important for the setup of compilation scripts in the next chapter(s).

3. **Changing Memory Sizes:** You need to tell the linker the size of the internal RAM (the data memory, "DMEM", `DMEM_SIZE` generic) and the internal ROM (instruction memory, "IMEM", `IMEM_SIZE` generic) of the NEO430 (you defined that during the previous tutorial). Open the **`neo430_linker_script.x`** in the **`sw/common`** folder with a text editor and set the parameter `LENGTH` of the ROM memory section according to the previously configured `IMEM_SIZE` generic and the RAM memory section according to the previously configured `DMEM_SIZE` generic (hexadecimal representation!). The cut-out below shows the default configuration – if you have not changed the memory sizes before you can keep everything in its current state and proceed.

```
MEMORY
{
  rom  (rx) : ORIGIN = 0x0000, LENGTH = 0x1000
  ram (rwx) : ORIGIN = 0xC008, LENGTH = 0x0800 - 8
}
```

(Only edit the values marked in **red**!)

Make sure you **do not** delete the "`-8`" right after the length of the RAM! This subtraction is required due to the interrupt vectors, which are located at the beginning of the DMEM. Additionally, the origin of the DMEM is set to 0xC008 *for the compiler* so it does not use the first 8 bytes at all. Of course, the "real" base address of the data memory module is still 0xC000.

Every time you change the hardware configuration of the IMEM and/or DMEM you need to do the same modifications in the linker script file.

4. Make sure to have a native CCC compiler installed. The native GCC is required to compile the image generator helper tool. Furthermore, GNU Make is required. When using the Windows Powershell you can use MinGW to get GCC and GNU Make. Make sure all these tools are up-to-date.

5. Well, this is all you need to do for the general software setup. In the next chapter(s), we will take a closer look on the application compile script.

4.3. Application Program Compilation using Windows Powershell

Use this guide if you want to compile programs using the **Windows Powershell**.

1. Double-check you have downloaded and installed the Windows version of the TI MSP430GCC compiler.

2. Now open a Windows Powershell console. Just hit the Windows key, type “Powershell” and press enter.

3. Next we need to add the MSP430-gcc compiler binaries to the system’s **PATH environment variable** so the makefiles can actually use all the compiler tools. In the following example the binary compiler sources are located in the folder `C:\msp430-gcc-8.3.0.16_win64\bin`. Make sure to use the **absolute** path here.

```
PS c:\> $env:PATH="$env:PATH;C:\msp430-gcc-8.3.0.16_win64\bin"
```

4. This environment variable is only modified for the current console session. After closing and re-opening the console you need to assign the variable again. However, you can also add it permanently to you console. **(MORE TO COME HERE)**

5. That’s all for now! Now you can start compiling programs. At first, we will begin with a simple example program. Navigate with your console to the `blink_led` folder in the project's software examples folder: `sw\example`

6. Execute the actual compilation `make` script in the current example folder to see all provided targets:

```
...\sw\example\blink_led> make
NEO430 Application Compilation Script
Make sure to add the absolute path of the msp430-gcc bin folder to your PATH variable.
Targets:
 help      - show this text"
 compile   - compile and generate *.bin executable for upload via bootloader"
 install   - compile, generate and install VHDL boot image"
 all       - compile and generate *.bin executable for upload via bootloader and generate
and install VHDL boot image"
 clean     - clean up project"
 clean_all - clean up project, core libraries and helper tools"
```

7. Execute a “**make clean_all all**”. If an error regarding the msp430-elf-objdump appears, simply execute “**make all**” again (I’m working on this…). See the following tutorial for more information regarding the available targets.

8. That’s all, you have just compiled your first application.

4.4. Application Program Compilation using the Windows Bash Subsystem or a Native Linux System

Use this guide if you want to compile programs using the **Windows Subsystem for Linux** or native **Linux**.

1. Double-check you have downloaded and installed the Linux version of the TI MSP430GCC compiler.

2. Open a terminal console. When using the Windows Subsystem for Linux you can do this by executing “bash” in a normal CMD console.

3. Next we need to add the MSP430-gcc compiler binaries to the system’s **PATH environment variable** so the makefiles can actually use all the compiler tools. In the following exampl example (Windows Bash) the binary compiler sources are located in the folder `/mnt/c/msp430-gcc-8.3.0.16_linux64/bin`. Make sure to use the **absolute** path here.

```
$ export PATH=$PATH:/mnt/c/msp430-gcc-8.3.0.16_linux64/bin
```

4. This environment variable is only set for the current console session. After closing and re-opening the console you need to assign the variable again. Alternatively, you can add this command to your *bashrc*. By this, the environment variable is automatically configured when opening a new console.

5. That’s all for now! Now you can start compiling programs. At first, we will begin with a simple example program. Open a terminal and navigate to the `blink_led` folder in the project's software examples folder: `sw/example`

6. Execute the actual compilation `make` script in the current example folder to see all provided targets:

```
.../sw/example/blink_led$ make
NEO430 Application Compilation Script
Make sure to add the absolute path of the msp430-gcc bin folder to your PATH variable.
Targets:
 help      - show this text"
 compile   - compile and generate *.bin executable for upload via bootloader"
 install   - compile, generate and install VHDL boot image"
 all       - compile and generate *.bin executable for upload via bootloader and generate
and install VHDL boot image"
 clean     - clean up project"
 clean_all - clean up project, core libraries and helper tools"
```

7. The target “help” will display the same text again. “compile” compiles the current project and generates a binary file, that can be uploaded via the NEO430 bootloader. The target “install” also compiles the project and will also create a VHDL memory initialization file, which can be directly used for booting the application when not using the bootloader. “all” will execute both of these steps. Via “clean” you can clean the current project. A “clean_all” will also clean the NEO430 libraries and the auxiliary tools. For now, execute a “**make clean_all all**”.

```
.../sw/example/blink_led$ make clean_all all
Memory utilization:
   text    data     bss     dec     hex filename
    680       0       0     680     2a8 main.elf
Installing application image to rtl/core/neo430_application_image.vhd
```

8. At first, the memory utilization/distribution is shown (in bytes). After that, a status message is shown, that confirms the “installation” process of the generated program image into the instruction memory VHDL component using the `neo430_application_image.vhd` file in the rtl folder.

9. That’s it!

4.5. Uploading and Starting of a Binary Executable Image via UART

When compiling an application, two final files are generated in the project folder:

- **`main.bin`** – The binary executable used for uploading via the bootloader.
- **`main.s`** – The ASM listing file of the compiled application (for debugging).

The generated binary executable must be uploaded to the NEO430 to be executed. This tutorial uses **TeraTerm** as an exemplary serial terminal program for **Windows**, but the general procedure is the same for other terminal programs and/or build environments / operating systems.

1. Connect the UART interface of your FPGA (board) to a COM port of your computer or use an USB-to-serial adapter.

2. Start a terminal program. In this tutorial, I am using TeraTerm for Windows. You can download it from: **https://ttssh2.osdn.jp/index.html.en**

3. Open a connection to the corresponding COM port. Configure the terminal according to the following parameters:

   - **19200 Baud**
   - **8 data bits**
   - **1 stop bit**
   - **No parity bits**
   - **No transmission/flow control protocol** (just raw byte mode)
   - Newline on "\r\n" = carriage return & newline (if configurable at all)

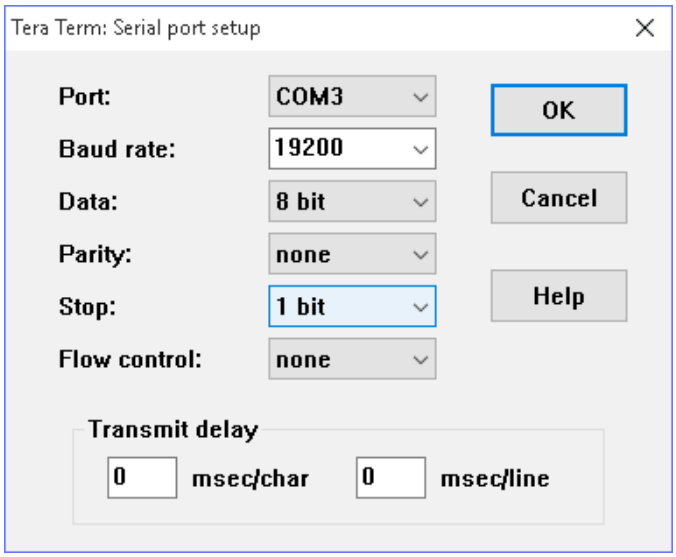

Figure 11: Serial configuration of TeraTerm

4. Also make sure, that single chars are transmitted without any consecutive "new line" or "carriage return" commands (this is highly dependent on your terminal application of choice, TeraTerm only sends the raw chars by default).

5. Press the NEO430's reset button to restart the bootloader. The status LED starts blinking and the bootloader intro screen appears in your console. Hurry up and press any key (hit space!) to abort the automatic boot sequence and to start the actual bootloader user interface console.

```
<< NEO430 Bootloader >>

BLV: Nov 19 2019
HWV: 0x0320
USR: 0x4788
CLK: 0x05F5E100
ROM: 0x1000
RAM: 0x0800
SYS: 0xFABF

Autoboot in 8s. Press key to abort.
Aborted.

Commands:
 d: Dump MEM
 e: Load EEPROM
 h: Help
 p: Store EEPROM
 r: Restart
 s: Start app
 u: Upload
CMD:>
```

6. Execute the "Upload" command by typing **`u`**. Now, the bootloader is waiting for a binary executable to be send.

```
CMD:> u
Awaiting BINEXE...
```

7. Use the "send file" option of your terminal program to transmit the previously generated binary executable (**`main.bin`**) from the `sw\example\blink_led` folder to the NEO430.

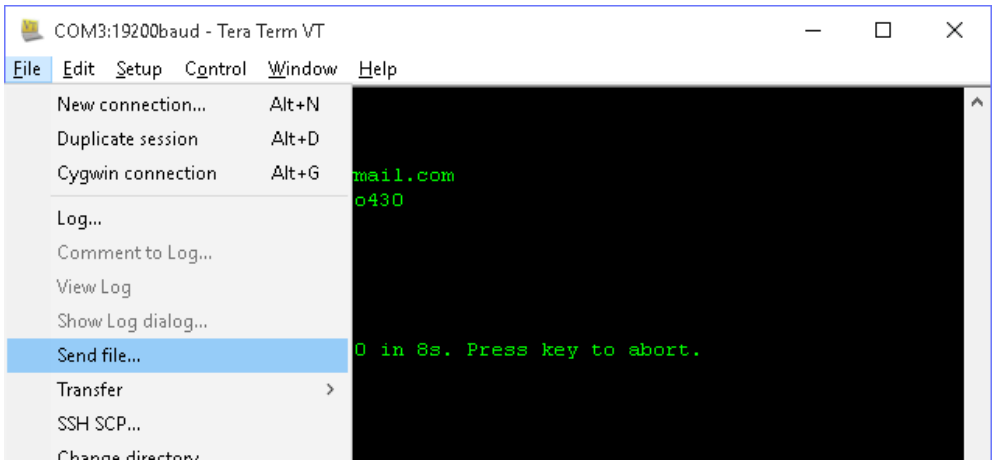

Figure 12: Sending a file using TeraTerm

8. Make sure to transmit the executable in **raw binary mode** (no transfer protocol, no additional header stuff). When using TeraTerm, select the "binary" option in the send file dialog:

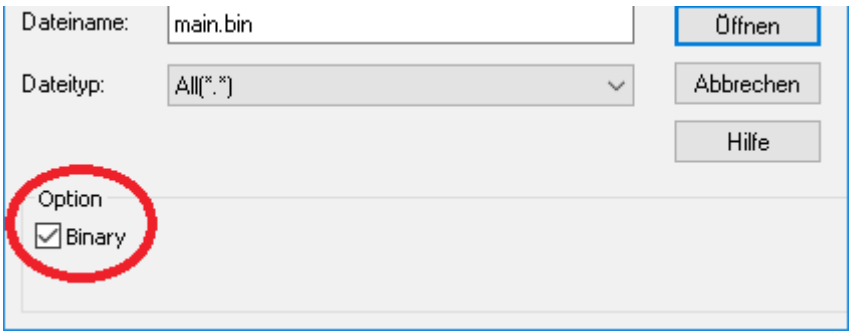

Figure 13: Transfer executable in binary mode (German version of TeraTerm)

9. If everything went fine, `OK` will appear in your terminal:

```
CMD:> u
Awaiting BINEXE... OK
```

10. The program image now resides in the internal IMEM of your NEO430. To execute the program right now, start the application by pressing **s**. The `blink_led` program starts, prints "`Blinking LED demo program`" and will begin displaying an incrementing counter on the 8 LEDs connected to the GPIO output port.

```
CMD:> s
Booting...

Blinking LED demo program
```

11. That's all. Now you are prepared to start your own project! ;)

4.6. Programming an External SPI Boot EEPROM

If you want the NEO430 bootloader to automatically fetch and execute an application image at start-up (→ auto boot configuration), you can store it to an external SPI EEPROM. The advantage of the external EEPROM is to have a non-volatile program storage, which can be re-programed at any time just by executing some bootloader commands. Thus, no FPGA bitstream recompilation is required at all.

You need an EEPROM, that is compatible to a Microchip ® SPI EEPROM like the **25LC512**, with 16-bit addresses and a 32-bit wide SPI transfer frame. The EEPROM must be at least as big as the internal IMEM.

This tutorial explains how to program the external SPI EEPROM assuming it is already connected properly to the NEO430 core top entity SPI port. Make sure to use the SPI chip select #0 signal (**`spi_cs_o(0)`**) as the chip select for the EEPROM.

1. At first, reset the NEO430 processor and wait until the bootloader start screen appears in your terminal program.

2. Abort the auto boot sequence and start the user console by pressing any key.

3. Press **`u`** to upload the program image, that you want to store to the external EEPROM. Send the binary in raw binary via your terminal program.

```
CMD:> u
Awaiting image...
```

4. When the uploaded is completed and **`OK`** appears, press **`p`** to begin programming of the EEPROM. You need to do this now – do not execute your program to prevent changes in the image!

```
CMD:> u
Awaiting image... OK
CMD:> p
Proceed (y/n)?
```

5. Now you have to confirm the writing sequence, because all previous data in the EEPROM will be lost. Press **`y`** if you want to proceed or **`n`** if you wish to abort the process. If you affirm, the actual writing process starts. This might take some time…

```
CMD:> u
Awaiting image... OK
CMD:> p
Proceed (y/n)?
Writing... OK
```

6. If **`OK`** appears in the terminal line, the writing process was successful. Now you can use the auto boot sequence to automatically boot your application from the EEPROM at system start-up without any user interaction.

4.7. Setup of a New Application Program Project

Done with all the introduction tutorials and those example programs? Then it is time to start your own application project!

1. The easiest way of creating a new project is to copy an existing one (like the **`blink_led`** project) and use that copy as starting point. Make sure to copy the folder inside the **`sw/example`** folder to keep all the file decencies in a correct manner (e.g. into the **`sw/example/my_project`** folder).

2. Now you can start modifying the **`main.c`** file according to your new project.

3. If your project contains additional program files beside the **`main.c`** file, you have to include the header files in your your **`main.c`** file using the C pre-processor macro.

```
#include <neo430.h>
#include "awesome_library.h" // one of your project's included libraries
```

4. You need to add the source folder of the included library to the sources path of the toolchain. Open the makefile in your project folder and add your sources to the APP_SRC variable:

```
# User's application sources (add additional files here)
APP_SRC += -I awesome_library/source/awesome_library.c
```

5. Additionally, you need to add the include folder of the included library to the include path of the toolchain. Open the makefile in your project folder and add your sources to the APP_INC variable:

```
# User's application include folders (don't forget the '-I before each entry)
APP_INC = -I . -I awesome_library/include
```

4.8. Simulating the Processor

If you do not have a FPGA board, if you want to check things or if you want to see what's going on, you can do a simulation of the processor. For this purpose, a simple testbench was implemented (**`neo430_tb.vhd`**, located in the project's **`sim`** folder). This testbench instantiates the top entity of the processor system (`neo430_top.vhd`) and also includes a serial UART receiver unit, which outputs the transmitted UART data to the simulator console. Additionally, the output is printed to a text file (`uart_rx_dump.txt`), which is generated in the simulator project home folder.

By default, the testbench does not simulate the system setup using the bootloader. Instead, your actual application code (in IMEM) will be simulated:

```
BOOTLD_USE => false, -- implement and use bootloader? (default=true)
```

Xilinx ISIM

In case you are using *Xilinx ISIM* simulator (or the Vivado simulator), a pre-defined waveform configuration including all relevant processor signals can be found in the `sim/ISIM` folder (`neo430_tb.wcfg`). Note, that you have to create a new project before, that needs to include all required rtl VHDL files. The generated `uart_rx_dump.txt` file (processor's UART output log file) is a little bit hard to find, but should be located in: `<Xilinx_project_home_folder>\<project_name>.sim\sim_1\behav`.

ModelSim

When you are using ModelSim, you can start a new simulation project by executing a script from the `sim/modelsim` folder. Navigate to the folder using the ModelSim simulator console and execute the following command:

```
do simulate.do
```

This will also open a pre-configured waveform to analyze the most important signals of the processor. The UART's output log file (`uart_rx_dump.txt`) will also be generated in the `sim/modelsim` folder.

4.9. Changing the Compiler's Optimization Goal

When compiling an application, the code is optimized using a given effort. By default, the optimization goal is to optimize and also to reduced code size. If you want to get more performance (with an increased code size) you can change the compilation effort / optimization goal (`-O3`).

1. Open the makefile of your current project and edit the effort variable:

```
# Compiler effort (-Os = optimize for size)
EFFORT = -Os
```

2. Perform a new compilation process of the software project to apply your changes to the generated executable.

4.10. Re-Building the Internal Bootloader

Rebuilding the bootloader is not necessary, since it is designed to work independently of the actual hardware configuration and system setup.

If you want to modify or customize the internal bootloader, you need to re-build it. Follow the upcoming steps to re-compile and re-install the modified bootloader to the boot ROM:

- After you have modified the bootloader's main source file according to your wishes, open a console and navigate to the bootloader source folder: **`sw\bootloader`**
- Open a console and execute "make clean_all all". Make sure the environment variable for the compiler's binaries is set (→ PATH).
- Now perform a new synthesis / HDL compilation to update the bitstream with your new bootloader. Done!

4.11. Building a Non-Volatile Application (Program Fixed in IMEM)

The purpose of the bootloaders is to re-upload your application code at any time via UART. Additionally, you can use an external SPI EEPROM as non-volatile program storage, that still can be updated at every time via the bootloader console. This provides a lot of flexibility, especially during development. But when you have completed your software development and your application code is *fixed*, the bootloader might not be necessary any longer. Thus, you can disable it to save hardware resources and to directly boot your application at start-up from the internal IMEM.

1. At first, compile your application code by running the **`make install`** command. This will automatically install the according memory initialization image into the IMEM.

2. Now it is time to exclude the bootloader ROM from synthesis. Set the **`BOOTLD_USE`** generic in the instantiation of the processor's top entity (**`neo430_top`**) to 'false':

```
BOOTLD_USE => false, -- implement and use bootloader? (default=true)
```

3. This will exclude the boot ROM from synthesis and also changes the CPU boot address from the beginning of the boot ROM to the beginning of the IMEM. Thus, the CPU directly executed your application code after reset.

4. The IMEM could be still modified by setting the **`R`** flag in the CPU's status register allowing write accesses. Hence, the IMEM is implemented as RAM. To prevent this and to implement the IMEM as true ROM (and eventually saving some more hardware), deactivate this feature by setting the **`IMEM_AS_ROM`** generic in the instantiation of the processor's top entity to 'false':

```
IMEM_AS_ROM => true -- implement IMEM as read-only memory? (default=false)
```

5. Perform a synthesis and upload your new bitstream. Your application code resides now unchangeable in the processor's IMEM and is directly executed after reset.

4.12. Alternative Top Entities / Avalon Bus / AXI4 Lite Connectivity

The NEO430 processor features a Wishbone-compatible 32-bit bus adapter to attach custom IP blocks. Wishbone is only one protocol for on-chip bus systems. Besides Wishbone, Avalon is a quite popular interface standard, especially in terms of Intel/Altera FPGA systems.

If you want to connect the NEO430 to IP cores using a different bus protocol you can either use a custom interface bridge or you can use one of the alternate processor top entities from the **`rtl\top_templates`** folder. These alternative top entities are a replacement of the default **`neo430_top.vhd`** as they provide the same interface ports as the default top entity. The only exception here is the actual on-chip bus protocol. Internally, the alternative top entity implement a bridging logic to convert the processor's native Wishbone interface into an **`Avalon Master`** interface.

Additionally, an alternative version of the default **`neo430_top.vhd`** top entity is provided, which only uses resolved interface types (std_logic and std_ulogic).

Alternative top entity	Description
`neo430_test.vhd`	Simple test setup for fast implementation / evaluation of the NEO430.
`neo430_top_avm.vhd`	Top entity with Avalon Master connectivity.
`neo430_top_axi4lite.vhd`	Top entity with Axi4 Lite Master connectivity.
`neo430_top_std_logic.vhd`	Top entity using only std_logic / std_logic_vector as port types.
`...`	More to come … ;)

4.13. Troubleshooting

- ✔ Have you added all HDL files from the `rtl/core` folder to your project? Make sure to add all VHDL files to a new library called “neo430”.
- ✔ Have you enabled the peripheral modules you want to use in the processor’s top entity (generics)?
- ✔ Have you selected the correct top entity (e.g. `neo430_test.vhd`)?
- ✔ Have you assigned at least the signals for the clock and reset, the status LED and the UART communication lines? Have you terminated all unused input signals (logical low)?
- ✔ Does the reset button has the correct polarity (active low)?
- ✔ Is your main clock source running at all?
- ✔ Have you made a correct configuration of all the configuration generics of the processor top entity? Especially the clock speed configuration is crucial for the test setup.
- ✔ Are the configured memory sizes in the linker script `neo430_linker_script.x` the same as in the VHDL top entity generic configuration?
- ✔ Do you want to directly execute your application from the IMEM or do you want to use the bootloader?
- ✔ Have you installed your compiler correctly and have you configured the PATH environment variable to include the absolute path to the compiler’s binaries folder? Did you install the correct compiler (TI’s msp430-gcc) version (Linux/Windows)?
- ✔ If you are communicating with the bootloader via UART, have you configured your terminal with the right settings (e.g., correct Baud rate)?
- ✔ Are you uploading the binary executable in raw-byte mode?
- ✔ Was the application compilation process successful?
- ✔ Try a “make clean_all all” in your project folder to recreate everything.
- ✔ Make sure to use an up-to-date version of all required tools (make, gcc, msp430-gcc,...).

6. Change Log

Date (DD.MM.YYYY)	HW version	Modifications
08.11.2017	`0x0125`	Added theoretical average CPI (cycles per instruction); added info regarding environmental variables for configuring the path to the MSP430-GCC binaries; added info about min/max interrupt latency
02.12.2017	`0x0140`	Replaced CFU by multiplier/divider unit. Changed documentary according to new processor design
08.12.2017	`0x0141`	25% faster execution of branches; added Xilinx implement. Results
23.12.2017	`0x0142`	Relocated Io device's control registers; all VHDL source files are part of library "neo430" now; removed USART.UART TX done interrupt (it's pointless); testbench will now executed app code by default
27.12.2017	`0x0142`	Fixed glitch-issue in UART transmitter; renamed watchdog and GPIO control registers (now they also have the "_CT" suffix)
06.01.2018	`0x0150`	Changed IO address space layout; added CRC module with according software library; added CFU; updated implementation results; added minimal configuration example
10.01.2018	`0x0154`	Moved IRQ vectors to beginning of DMEM
26.01.2018	`0x0170`	Added PWM controller; modified internal clock generator; fixed errors in address space declarations; clean-up of change log
24.04.2018	`0x0180`	r-flag is now read-only (and always zero) when implementing IMEM as true ROM
30.05.2018	`0x0182`	Added experimental low power mode; optimized IRQ controller logic; optimized SREG logic
01.06.2018	`0x0183`	Fixed bugs in Wishbone module and driver library; reduced ALU size
22.06.2018	`0x0184`	**Fixed SEVERE bug** in overflow flag computation! Thx Edward for that ;) added warning regarding not to use floating point types
03.08.2018	`0x0184`	Added NEO430 SW library description
09.08.2018	`0x0185`	Changed memory partitioning: IMEM can be up to 48kB large, DMEM up to 12kB
07.11.2018	`0x0187`	Bug-fix in latency of external IRQ line; added IRQ test to testbench; fixed some typos
17.11.2018	`0x0200`	Added TWI module; separated USART into SPI and UART; SPI now has 8 dedicated CS lines
20.09.2019	`0x0300`	PWM has now 4 channels and a variable bit width for the resolution; changed VHDL code to prevent some VHDL-2008 warnings; removed pure-windows makefiles – toolchain is now completely based on Linux makefiles, which can also be executed by Windows Powershell; modified reset generator; renamed PWM and WDT register and bit names; reworked whole documentary; split NEO430 SW library into include and source files
23.09.2019	`0x0300`	Finally fixed linking issue of math.h lib in makefiles. **Floating point types/operations are now fully supported.**
01.10.2019	`0x0300`	Updated makefiles (msp430-gcc has to be part of PATH; more targets)
04.10.2019	`0x0300`	Removed "-flto" compiler flag from makefiles; added implementation results for Lattice iCE40 UltraPlus
05.10.2019	`0x0301`	TWI modules now supports clock stretching by the slave; added note in *software setup* to make sure a native C compiler is installed

Date (DD.MM.YYYY)	HW version	Modifications
14.11.2019	`0x0304`	CPU's DADD instruction is disabled by default – can be enabled via package switch; new shift direction control for SPI module; timer CNT register is now read-only; added timer SW library files, removed old aux library; cleared old change log entries; updated implementation results; reworked several parts of this document; minor size optimization of CPU
16.11.2019	`0x0305`	ALU and IRQ controller size optimizations; added compiler warning explanation regarding DADD instruction; added notes for adding include and source files to C projects
26.11.2019	`0x0310`	**Fixed bug** in CPU interrupt controller – now an interrupted program can execute at least one instruction even if there is a permanent interrupt request; added extended ALU functions (parity computation); SREG's Q flag is write-only; added GARO TRNG incl. SW library and example project; constrained PWM counter sizes to 4 or 8 bit; added PWM option to modulate GPIO unit's output port incl. example SW project; removed DMEM_BASE from SYSCONFIG info memory
27.11.2019	`0x0311`	Added watchdog status flag to determine if a WDT reset was caused by a normal timeout or an access with wrong password; reduced TRNG to have only one accessible register
29.11.2019	`0x0320`	Added external interrupts controller (EXIRQ) incl. SW library and example project

7. Citation

If you are using the NEO430 in some kind of publication, please cite it as follows:

> S. Nolting, "The NEO430 Processor", github.com/stnolting/neo430