The NEO430 Processor

by Dipl.-Ing. Stephan Nolting

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

Processor Hardware Version: 0x0123

Proprietary Notice

This documents was created using LibreOffice.

License / Disclaimer

The **NEO430 Project**
© 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 NEO430 project and the according documentary can be found at
https://github.com/stnolting/neo430

Table of Content

1. Introduction

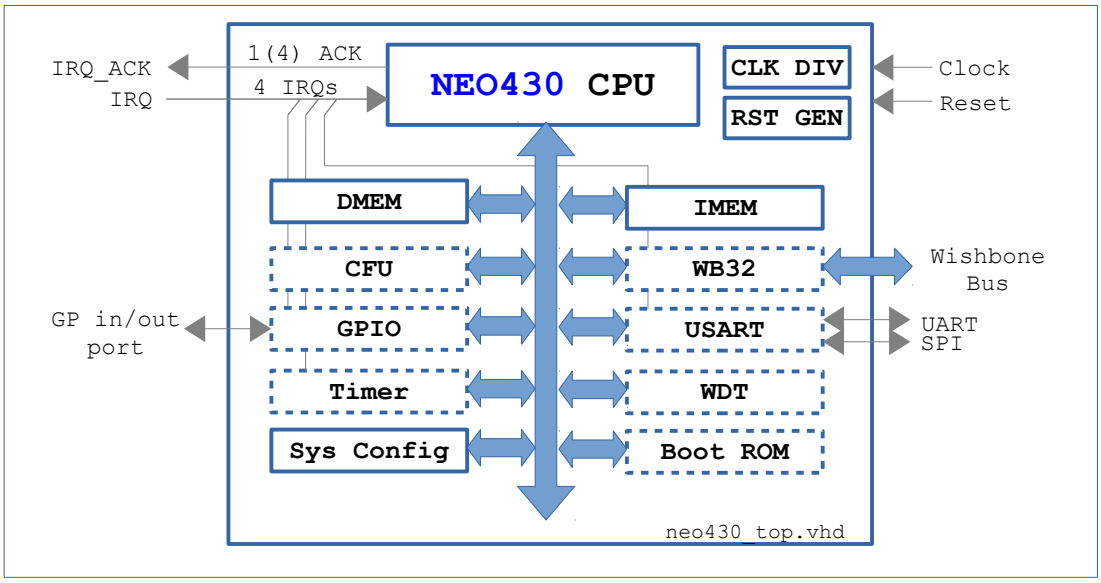

Figure 1: NEO430 processor block diagram, optional modules are marked using dashed lines

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 perfect choice for you!

This processor is based on the Texas Instruments MSP430 ISA and provides 100% compatibility with the original instruction set. The NEO430 is not an MSP430 clone – it is more a complete new implementation from the bottom up. The processor features a very small outline, already implementing standard features like a timer, a watchdog, UART 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 exclude them from implementation to reduce the size of the system. Any additional modules, which make a more customized system, can be connected via a Wishbone-compatible bus interface or you add them as custom functions unit to the processor core. By this, you can build a system, that perfectly fits your needs.

It is up to you to use the NEO430 as stand-alone, configurable and extensible microcontroller, or to include it as controller within a more complex SoC design.

The high-level software development is based on the free TI **msp430-gcc** compiler tool chain. You can either use Windows or Linux/Cygwin as build environment for your applications – the project comes with build scripts for both worlds. The example 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 of choice and start exploring the capabilities of the NEO430 processor. Application program generation (and even installation) works by executing a single "make" command. Jump to the "**Let's Get It Started**", which provides a lot of guides and tutorials to make your first NEO430 setup run.

1.1. Processor Features

- ✔ 16-bit open source soft-core microcontroller-like processor system
- ✔ Code-efficient CISC-like instruction capabilities
- ✔ Full support of the original MSP430 instruction set architecture
- ✔ Tool chain based on free TI msp430-gcc compiler
- ✔ Application compilation scripts for Windows and Linux/Cygwin
- ✔ Completely described in behavioral, platform-independent VHDL
- ✔ Fully synchronous design, no latches, no gated clocks
- ✔ Operates at high frequencies (175 MHz[1]) - no clock-domain crossing required for integration
- ✔ Very small outline and high operating frequency compared to other implementations ;)
- ✔ Internal DMEN (RAM, for data) and IMEM (RAM or ROM, for code), configurable sizes
- ✔ One external interrupt line with acknowledge signal
- ✔ Customizable processor hardware configuration
- ✔ Optional custom functions unit (CFU) to add custom memory-mapped hardware with example templates (e.g., a multiply-and-accumulate unit) including C header files
- ✔ Optional high-precision timer (TIMER)
- ✔ Optional USART interface; UART and SPI (USART)
- ✔ Optional general purpose parallel IO port (GPIO), 16 inputs, 16 outputs, with pin-change interrupt
- ✔ Optional 32-bit Wishbone bus interface adapter (WB32)
- ✔ Optional watchdog timer (WDT)
- ✔ Optional internal bootloader (2kB ROM):
  - ➔ Upload new application image via UART
  - ➔ Program external SPI EEPROM
  - ➔ Boot from external SPI EEPROM
  - ➔ Core dump
  - ➔ Automatic boot sequence

1 Xilinx Virtex-6, speed-optimized implementation

1.2. Main Differences to TI's Original MSP430 Architecture

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

- ✗ Completely different processor modules with different functionality
- ✗ No hardware multiplier support (but emulated in software) – but you could add your own ;)
- ✗ Maximum of 32kB instruction memory and 28kB of data memory
- ✗ Specific memory map – included NEO430 linker script and compilation script required
- ✗ Custom binary executable format
- ✗ Only 4 CPU interrupt channels (instead of 16)
- ✗ Single clock domain for complete processor
- ✗ Different numbers of instruction execution cycles
- ✗ Only one power-down (sleep) mode
- ✗ Internal bootloader with text interface (via UART serial port)

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 GitHub/opencores web page.
|
├─rtl              All the rtl files of the project can be found here.
| |
| ├─cfu_templates  This folder contains example templates (e.g., a multiplier) for the
| |                custom functions unit (CFU) together with the required C header file
| |                to provide the driver functions.
| |
| ├─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 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
|                  std_logic signal types=.
|
├─sim              The sim folder contains a simple VHDL testbench for simulation 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 codes to start
  |                from.
  |
  ├─bootloader     Sources and compilation scripts of the NEO430-internal bootloader.
  |
  ├─common         All required files for application program generation.
  |
  ├─example        Here you can find several example programs. Each project folder
  | |              includes the program's C sources and Windows/Linux/Cygwin scripts for
  | └─...          compilation. Add your own projects to this folder.
  |
  |
  ├─lib            This folder contains different C software libraries.
  | |
  | └─neo430       All libraries for using the different hardware modules of the NEO430
  |                as well as the neo430.h defines file, which is mandatory for every
  |                program, are located in this folder.
  |
  └─tools          This folder contains auxiliary programs for generating executables
    |              and ROM images.
    |
    └─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`**.

```
neo430_top.vhd                    Processor core top entity
|
├─neo430_boot_rom.vhd             Bootloader ROM
| |
| └─neo430_bootloader_image.vhd   Boot ROM initialization image for the bootloader.
|                                 During synthesis, the image is copied to the voot
|                                 ROM. This file is automatically generated and
|                                 copied when compiling the bootloader's sources.
|
├─neo430_cfu.vhd                 Default custom functions unit (a simple template)
|
├─neo430_dmem.vhd                DMEM: Internal RAM for storing data
|
├─neo430_gpio.vhd                General purpose parallel IO port
|
├─neo430_imem.vhd                IMEM: Internal RAM/ROM for the application code
| |
| └─neo430application_image.vhd  IMEM initialization image. During synthesis, the
|                                 initialization image is copied to the IMEM. This
|                                 file is automatically generated and copied when
|                                 compiling an application.
|
├─neo430_package.vhd             Processor VHDL package file
|
├─neo430_sysconfig.vhd           IRQ vector configuration and system information
|
├─neo430_timer.vhd               High-precision timer
|
├─neo430_usart.vhd               USART serial transceiver (SPI and UART)
|
├─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
```

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 (see notes below!).

Signal name	Width	Direction	Function	Module
Global Control				
clk_i	1	Input	Global clock line, all registers triggering on rising edge	all
rst_i	1	Input	Global reset, **asynchronous**, **low-active**	all
Parallel IO				
gpio_o	16	Output	General purpose parallel output[2]	GPIO
gpio_i	16	Input	General purpose parallel input	GPIO
Serial Communication				
uart_txd_o	1	Output	UART serial transmitter	USART.UART
uart_rxd_i	1	Input	UART serial receiver	USART.UART
spi_sclk_o	1	Output	SPI master clock line	USART.SPI
spi_mosi_o	1	Output	SPI serial data output	USART.SPI
spi_miso_i	1	Input	SPI serial data input	USART.SPI
spi_cs_o	6	Output	SPI intrinsic chip select lines #0..#5[3]	USART.SPI
Wishbone Bus				
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				
irq_i	1	Input	Interrupt request signal, high-active, single-shot	CPU
irq_ack_o	1	Output	Interrupt request acknowledge, single-shot	CPU

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

Of course, you can instantiate the processor top entity (neo430_top.vhd) in another entity – your actual system's top entity – and only connect the specific signals, that you really need, to the outer world. Set all unused input signals to logical zero and leave all unused outputs 'open'.

If you need a top entity using std_logic or std_logic_vector as signal types you do not have to make the conversion by yourself. You can use the neo430_top_std_logic.vhd from the rtl\top_templates folder as top entity instead.

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

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 32kB)
DMEM_SIZE	natural	2*1024	Size of internal data memory (max 28kB)
Additional Configuration			
USER_CODE	std_ulogic_vector	x"0000"	Custom user code, can be checked by application software
Module Configuration			
DADD_USE	boolean	true	Implement decimal addition (DADD) CPU instruction
CFU_USE	boolean	**false**	Implement custom functions 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
USART_USE	boolean	true	Implement UART/SPI serial communication unit
Boot Configuration			
BOOTLD_USE	boolean	true	Implement and use 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

Boot Configuration

The default configuration of the NEO430 processor includes all optional modules 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 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.

1.7. Implementation Results

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

1.7.1. Default Implementation

Hardware configuration:

- Hardware version: 0x0108
- Top entity file: neo430_top.vhd
- Internal IMEM: 4kB
- Internal DMEM: 2kB
- Boot ROM: 2kB (default bootloader)
- Custom function unit (CFU): Excluded
- Wishbone bus adapter (WB32): Included
- Parallel IO port (GPIO): Included
- Serial interface (USART): Included
- High-precision timer (TIMER): Included
- Watchdog timer (WDT): Included
- DADD instruction: Included

FPGA tools and settings:

- Intel/Altera Quartus Prime Lite 16.1 (“balanced implementation”)
- Xilinx ISE 14.7 (“speed-optimized”)
- Lattice Diamond 3.7.1.502

Utilization			
Resource[4]	***Altera Cyclone IV EP4CE22F17C6N***	***Xilinx Virtex-6 C6VLX240T-2FF1156***	***Lattice MachXO2 MACHXO2-1200ZE*** [5]
LUTs/LEs:	1188 / 22320 = 5%	849 / 150720 = >1%	1096 / 1280 = 86%
FFs/Registers:	557 / 22320 = 2%	652 / 301440 = >1%	519 / 1595 = 33%
Total memory bits / #Block RAMs:	79168 / 608256 = 13%	4 / 832 = >1%	6 / 7 = 86%
Maximum Frequency:	116 MHz (slow 1200mV 0°C model)	173 MHz (synthesis result)	12.09 MHz (user constrained)

Table 3: Hardware utilization for different FPGA platforms

4 Altera/Intel: LEs, registers, memory bits. Xilinx & Lattice: LUTs, FFs, block RAMs
5 Implementation with only 2kB IMEM and additional Wishbone network

1.7.2. 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 into single logic elements (LEs), so the total number of required resources might be different from the table above.

Hardware configuration:

- Hardware version: 0x0113
- Top entity file: neo430_top.vhd
- Internal IMEM: 4kB
- Internal DMEM: 2kB
- Boot ROM: 2kB (default bootloader)
- Custom function unit (CFU): Excluded
- Wishbone bus adapter (WB32): Included
- Parallel IO port (GPIO): Included
- Serial interface (USART): Included
- High-precision timer (TIMER): Included
- Watchdog timer (WDT): Included
- DADD instruction: Included

FPGA tools and settings:

- Intel/Altera Quartus Prime Lite 16.1 (“balanced implementation”)

Altera Cyclone IV EP4CE22F17C6N					
Entity/Module	**Function**	**LEs**	**FFs**	**MEM bits**	**DSPs**
CPU	Central processing unit	624	189	256	0
IMEM (4kB)	Instruction memory (RAM)	3	1	32768	0
DMEM (2kB)	Data memory (RAM)	3	1	16384	0
Boot ROM (2kB)	Bootloader ROM	1	1	20480	0
Sysconfig	System information & IRQ config	8	5	64	0
GPIO	GPIO parallel in/out ports	70	61	0	0
WDT	Watchdog timer	49	34	0	0
TIMER	High-precision timer	89	56	0	0
USART	Serial interfaces (UART + SPI)	192	124	0	0
WB32	Wishbone bus interface	99	76	0	0

Table 4: Intel Quartus Prime Lite 16.1 fitter results, “balanced implementation”, utilization by entity

2. Hardware Architecture

The NEO430 processor system is constructed from several different modules. This chapter takes a closer look 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 commercial MSP430 processors. Hence, also the provided features and the address space layout are completely new. The figure below shows the general layout of the 16-bit address space.

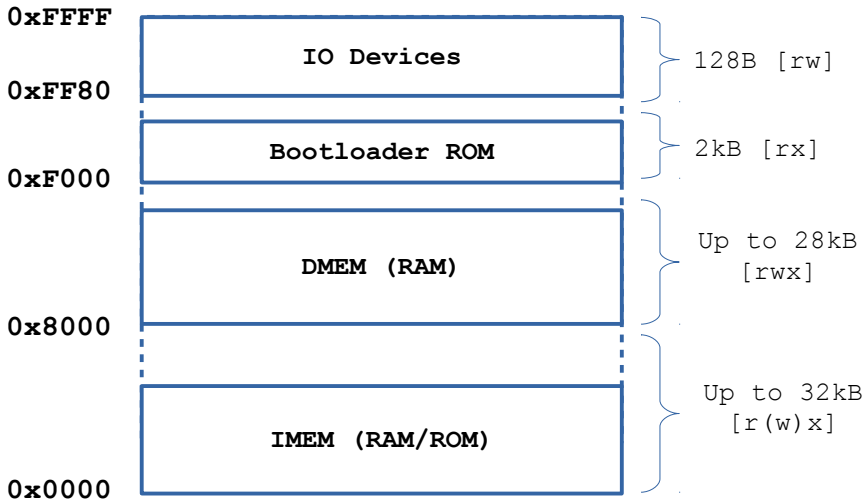

Figure 2: General NEO430 address space layout

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 and interrupt enable configurations – are located inside the according hardware units. These 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 a dedicated NEO430 include and definition files abstract the specific memory layout for the user.

Separated Instruction (IMEM) and Data (DMEM) Memories

Just like the original MSP430, the NEO430 uses separated memories for storing data and instructions. The DMEM is implemented as normal RAM, the IMEM is also implemented as RAM, but one can only write to it when 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. Alternatively, the IMEM can be implemented as true ROM (via the `IMEM_AS_ROM` generic). In this case, the actual executable application is included during the synthesis process and persists as non-volatile image in the IMEM. Thus, a bootloader ROM is no longer necessary (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 the peripheral devices in the IO region can only be accesses using full-word mode.

Internal Reset Generator

All processor-internal modules – except for the CPU – do not require a dedicated reset signal. However, to ensure correct operation, all devices are reset by software by clearing the corresponding control register before using the unit. The automatically included applications start-up code will perform this minimal-required system initialization. The hardware reset signal of the processor can either be triggered via the external reset pin (**asynchronous & LOW-active**) or by the internal watchdog timer (if implemented).

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" by the different modules. Thus, the whole design operates using only the main clock. Some of the processor modules (like the timer or the USART) can select one of the derived clock enabled signals for their internal operation. If none of the connected modules require an active clock, the clock divider is automatically deactivated to save dynamic power.

Hardware Multiplier

Some of the original TI MSP430 controllers feature a hardware multiplier, mapped to the address space. Since the NEO430 features a completely different address map, the original multiplier address mapping cannot be used. So far, I was not able to change the addresses of the multiplier used by the compiler. Therefore, a compiler-supported hardware multiplier is not available yet. However, you can implement a custom multiplier using the *Custom Functions Unit*. A later chapter will give some information regarding this unit.

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 interrupts.

Data and Control Path

Instruction execution is conducted by performing several tiny steps – so-called *micro operations*. Thus, the NEO430 is a multi-cycle architecture: The CPU needs 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.

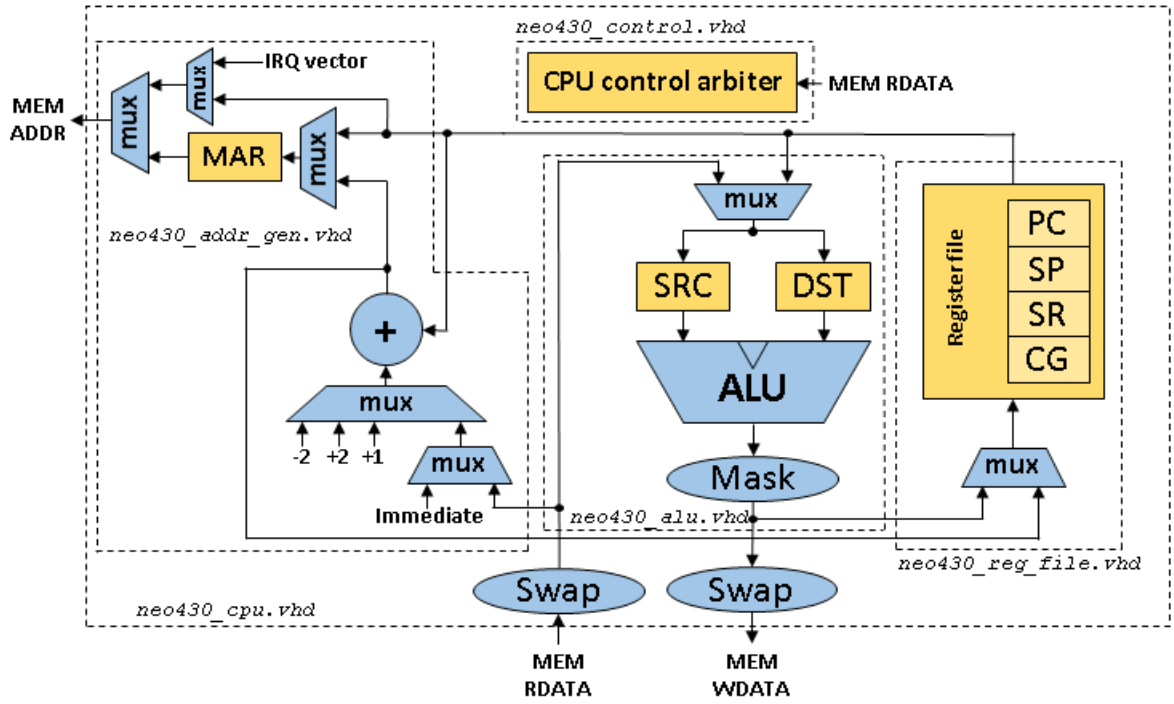

Figure 3: Data path of the NEO430 CPU

2.1.1. 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 stored and finally executed when the **I** flag is set again.

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 and clear the **S** flag again.

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), the state of this flag is irrelevant and no write accesses to the IMEM are 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. However, they are always read as zero.

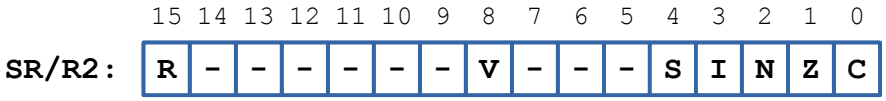

Figure 4: Processor status register

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..7	-	-/-	*Reserved, read as 0*
8	**V_FLAG**	R/W	O**V**erflow flag
9..14	-	-/-	*Reserved, read as 0*
15	**R_FLAG**	R/W	Allow write access to IMEM "**R**OM"

Table 5: Bits of the status register

The original MSP430 specs claim, that the state of the status flags after entering an interrupt is undefined. Thus, the status register should be initialized at the beginning of an interrupt service routine to provide a deterministic behavior. However, the NEO430 CPU clears all bits of the status register when entering an interrupt.

2.1.2. Interrupts

The NEO430 features 4 independent interrupts via 4 interrupt request signals. When triggered, each of these requests start a unique interrupt handler. The interrupt-causing sources are the TIMER overflow interrupt, the USART SPI transmission done and/or the UART RX/TX complete interrupt, the GPIO pin-change interrupt and the external interrupt request signal. The base addresses of these handlers have to be stored in advance to the interrupt vector configuration table, located in the SYSCONFIG module. According addresses are further explained in the SYCONFIG chapter.

All interrupts can be globally disabled by clearing the `I` flag in the processor's status register. An interrupt can only trigger, when the `I` flag is set and the corresponding enable flag of the interrupt source is activated inside the according source's control register.

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

IRQ Vector Name	Priority	Source
`IRQVEC_TIMER`	`1 (highest)`	The timer generates a threshold match.
`IRQVEC_USART`	`2`	UART Rx available **OR** UART Tx done **OR** SPI transmission done.
`IRQVEC_GPIO`	`3`	GPIO input pin change.
`IRQVEC_EXT`	`4 (lowest)`	External interrupt request via the **`irq_i`** top entity signal.

Table 6: Interrupt sources, priorities and handler base address configuration register names

External Interrupt Request Signal

The NEO430 processor features a single external interrupt request signal (`irq_i`). This signal can be used to attach an external (e.g., Wishbone-coupled) interrupt controller to add additional interrupt sources to the processors.

The external interrupt request signal of the NEO430 is level sensitive and synchronous to the processor's clock. Thus, a high-level triggers the external interrupt. The attached interrupt source (e.g., the interrupt controller) has to make sure that the IRQ signal to the processor is only active for one clock cycle. Otherwise the interrupt might be triggered several times. As soon as the CPU has received the interrupt request and the according interrupt handler is started, an acknowledge is issued via the external interrupt acknowledge signal (`irq_ack_o`). This acknowledge is also active for only one clock cycle. After that acknowledge the external interrupt request can be triggered again.

2.1.3. Instruction Set

The instruction set of the NEO430 CPU is fully compatible to the original TI MSP430 instruction set architecture. The full data sheet and the according user guide including also the pseudo-instruction can be found at www.ti.com/lit/ug/slau049f/slau049f.pdf.

2.1.4. 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 7: 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 8: Single-operand (**format II**) instruction execution cycles (except `RETI`)*

Instruction / Operation	**Branches**	4
	RETI	8
	Interrupt	6

Table 9: Special instructions / operations execution cycles

The double-operand “decimal addition” instruction (**DADD**) requires an **additional execution cycle** (for all addressing modes) to complete. **Also, the implementation of this instruction requires a lot of hardware!** If you do not use explicit elementary decimal additions in your application, you can disable this instruction. Thus, the required circuitry will be removed from the synthesis process. Of course, you have to ensure, that the `DADD` instruction is not used at all in the code (maybe check the main.s file for it). However, if it is executed, it will always generate 0. To disable the `DADD` instruction, assign “false” to the `DADD_USE` generic of the processor’s top entity.

2.1.5. 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 10: 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.

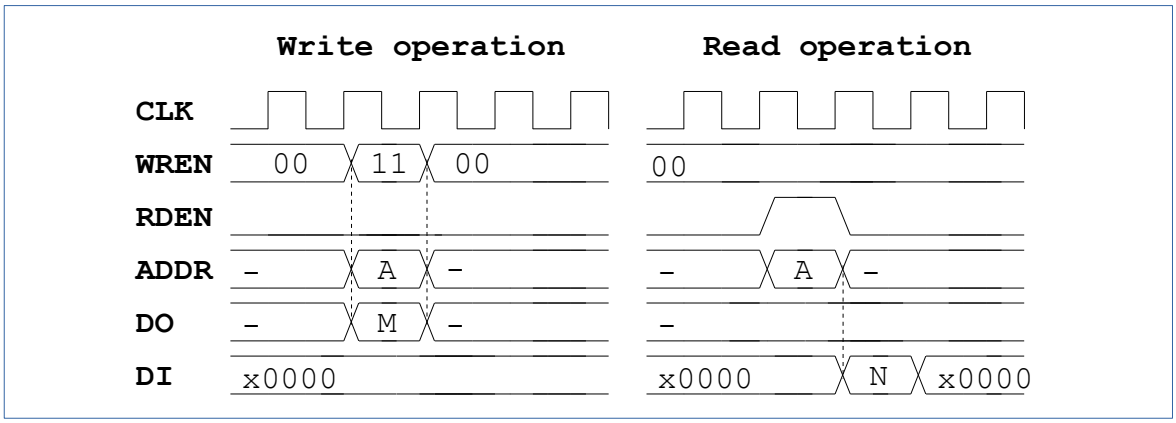

Figure 5: Write and read bus cycles (full word transfers); Write: M → [A]; Read: [A] = N

You can add custom modules to the processor-internal bus, but that requires are very good understanding of the address space layout and the general NEO430 architecture as well. Instead, I encourage you to use the Wishbone bus interface to connect custom logic.

2.2. Internal Instruction Memory (IMEM)

The internal instruction memory (VHDL component **neo430_imem.vhd**) stores the code of the currently executed program. It is located at base address 0x0000 of the address space. The actual IMEM size can be configured via the IMEM_SIZE generic of the processor top entity (see cut-out below). Make sure the IMEM size **does not exceed 32kB**. During run time, the size can be obtained by a program by reading a specific CPUID register from the SYSCONFIG module.

```
IMEM_SIZE => 4*1024, -- internal IMEM size in bytes, max 32kB (default=4kB)
```

The content of the IMEM can either be initialized during synthesis (so it behaves like a 'real' ROM – non-volatile) or during run-time via the bootloader (so it behaves like a RAM – volatile). However, the IMEM is always initiated (even if not used) using the neo430_application_image.vhd file.

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 0x8000 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 28kB**. During run time, the size can be obtained by a program by reading a specific CPUID register from the SYSCONFIG module.

```
DMEM_SIZE => 2*1024, -- internal DMEM size in bytes, max 28kB (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 system reset. It is located at address 0xF000 of the address space. This address is fixed and must not be altered, 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 (generated by the image generator).

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

2.5. Wishbone Bus Interface Adapter (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. Note that these functions are blocking! So a non-responding slave will permanently stall the system!

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 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 (set all byte enable signals to zero).

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 two different transfer cycle modes: *Standard* and *pipelined* transfers (see figures below). Standard cycles keep the strobe signal (`wb_stb_o`) active for the complete cycle, until the slave sends the acknowledge signal. Some slaves might interpret the constantly active strobe signal as additional access cycles, so you might switch to the pipelined mode to access them in a proper way. Pipelined Wishbone transfer cycles only apply the strobe signal during the first clock cycle of the transfer. Thus, only a single access is triggered. Make sure to select the right mode to access your devices. Pipelined transfers are enabled by setting the `WB32_PMODE` bit of the control register.

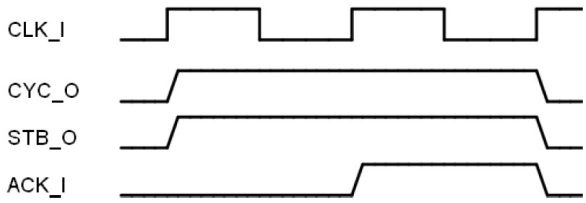

Figure 6: Standard Wishbone cycle

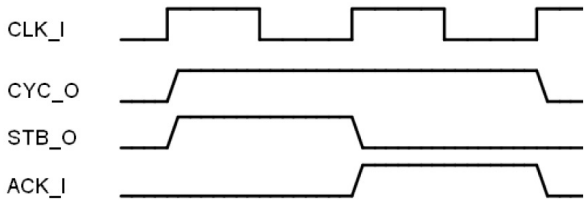

Figure 7: Pipelined Wishbone cycle

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 11: Wishbone bus interface adapter signals

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_LRA	0..15		-/W	Low address word for read transfer
0xFF92	WB32_HRA	0..15		-/W	High address word for read transfer (+trigger)
0xFF94	WB32_LWA	0..15		-/W	Low address word for write transfer
0xFF96	WB32_HWA	0..15		-/W	High address word for write transfer (+trigger)
0xFF98	WB32_LD	0..15		R/W	Low word of read/write data, can be accessed in byte mode
0xFF9A	WB32_HD	0..15		R/R	High word of read/write data, can be accessed in byte mode
0xFF9C	-	0..15		-/-	*Reserved*
0xFF9E	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	WB32_CT_PMODE		0: Standard / 1: Pipelined transfer
		5..14		-/-	*reserved*
		15	WB32_CT_PENDING	R/-	Pending Wishbone transfer flag

Table 12: 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.

Only the WB32_LD and WB32_HD registers should be accessed in byte mode. All other registers should should be accessed in word mode.

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!

2.6. General Purpose IO (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 wish to save some resources, you can exclude the GPIO 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 trigger type for all pins can be globally set to low-active, high-active, rising-edge or falling-edge by writing the according value (see table below) to the `GPIO_CTRL` register. Bit #2 of the control register globally enables the pin-change interrupt when set. If any of the enabled (via the mask register) input pin performs the configured trigger operation, the interrupt request is generated. 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.

Register Map

Address	Name	Bit(s) (Name)	R/W	Function
0xFFB0	GPIO_IN	0..15	R/-	Parallel input port
0xFFB2	GPIO_OUT	0..15	R/W	Parallel output port
0xFFB4	GPIO_CTRL	0..1	-/W	Interrupt trigger type: 00: Low-level, 01: High-level, 10: Falling-edge, 11: Rising-edge
		2	-/W	Global interrupt enable
		3..15	-/-	*Reserved*
0xFFB6	GPIO_IRQMASK	0..15	-/W	Enable according input pin(s) as interrupt trigger

Table 13: GPIO module address map

Do not read from registers, which do not provide the read access feature (e.g. when `R/W = -/W`), since such return undefined data!

2.7. USART / USI

The universal synchronous/asynchronous receiver and transmitter (VHDL component **neo430_usart.vhd**) features standard serial interfaces for chip-external communications. Two independent sub modules are implemented: A **UART** sub module and an **SPI** sub module. Both modules can operate in parallel. In terms of this documentary, the USART is also labeled as USI (universal serial interface).

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFFA0	USI_SPIRTX	0..7		R/W	SPI Rx/Tx data
		8..15		R/-	*Reserved, read as 0*
0xFFA2	USI_UARTRTX	0..7		R/W	UART Rx/Tx data
		8..14		R/-	*Reserved, read as 0*
		15	USI_UARTRTX_UARTRXAVAIL	R/-	Set if UART Rx data available
0xFFA4	USI_BAUD	0..7		R/W	UART baud value
		8..10		R/W	UART baud prescaler select
		10..15		R/-	*Reserved*
0xFFA6	USI_CT	0	USI_CT_EN	R/W	USI enable (SPI & UART)
		1	USI_CT_UARTRXIRQ	R/W	UART Rx available IRQ enable
		2	USI_CT_UARTTXIRQ	R/W	UART Tx done IRQ enable
		3	USI_CT_UARTTXBSY	R/-	UART Tx busy flag
		4	USI_CT_SPICPHA	R/W	SPI clock phase
		5	USI_CT_SPIIRQ	R/W	SPI transmission done IRQ enable
		6	USI_CT_SPIBSY	R/-	SPI transceiver busy flag
		7	USI_CT_SPIPRSC0	R/W	SPI clock select (prescaler value)
		8	USI_CT_SPIPRSC1	R/W	
		9	USI_CT_SPIPRSC2	R/W	
		10	USI_CT_SPICS0	R/W	Dedicated SPI CS0 (high-active)
		11	USI_CT_SPICS1	R/W	Dedicated SPI CS1 (high-active)
		12	USI_CT_SPICS2	R/W	Dedicated SPI CS2 (high-active)
		13	USI_CT_SPICS3	R/W	Dedicated SPI CS3 (high-active)
		14	USI_CT_SPICS4	R/W	Dedicated SPI CS4 (high-active)
		15	USI_CT_SPICS5	R/W	Dedicated SPI CS5 (high-active)

Table 14: USART 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.

Implementation Control

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

```
USART_USE => true, -- implement USART? (default=true)
```

USART/USI Interrupt

The USART features a single interrupt output, which can be used to indicate a *UART RX data available* status and/or a *UART TX done* status and/or an *SPI transfer completed* status. When using more than one possible interrupt sources, you have to check the USART control register by yourself to determine the actual causing event. For simplicity, I suggest to use only one interrupt source and to check the other status option using a generic timer interrupt. If the USART module is excluded from synthesis (disabled), its interrupt request signals are always 0.

2.7.1 Universal Asynchronous Receiver/Transmitter (UART)

In most cases, the UART interface is used to establish a communication channel between the computer/user and an application. 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 not 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 USART must be activated by setting the global `USI_CT_EN` in the USART control register `USI_CT`. Now you can transmit a character by writing it to the `USI_UARTRTX` register. The transfer is in progress if the `USI_CT_UARTTXBSY` bit in the control register is set. A received char is available when bit #15 of the `USI_UARTRTX` 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_USART`)! To make the usage of the UART a little bit easier, the `neo430_usart.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 *infomem* for a final Baud rate of 19200.

UART Baudrate

The actual transfer speed – the Baud rate – can be arbitrarily configured via the `USI_BAUD` register. This register defines a prescaler value (**PRSC**, bits 10:8) and also a direct Baud rate divisor factor (**BAUD**, bits 7:0). According to the selected prescaler, 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 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

Example Baudrate Computation

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

Thus, the USI_BAUD register would look like this: **0b0000.0011.0010.1001**

Actually, you do not have to think about the configuration of the UART Baud rate at all. A function from the `neo430_usart.h` library does all the work for you – just call it once at program start.

2.7.2 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 data size corresponds to the bytes being send during an active chip select (CS) of the connected device. A transmission is started when writing a data byte to the `USI_SPIRTX` register. The `USI_CT_SPIBSY` bit of the control register indicates a transfer being in progress. The received data can be obtained by reading the `USI_SPIRTX` register as soon as the transmission is done and the busy flag is cleared. A “transmission done” interrupt can be activated by setting the `USI_CT_SPIIRQ` bit. Note, that this interrupt also triggers the same interrupt handler as the interrupt sources from the UART 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.

Transmission Configuration

The SPI transmission provides several configuration options. The actual clock phase can be determined via the `USI_CT_SPICPHA` 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. For every transmission, the MSB is send 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 `USI_CT_SPIPRSCx` of the control register. The resulting 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_usart.h` library file you can find elementary functions for performing an SPI transmission and for controlling the dedicated chip-select signals of the USART.SPI module.

Dedicated SPI Chip Select (CS) Lines

The SPI controller features six 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 `USI_CT_SPICSx` bits in the USART control register. Note that these bits need to be set (‘1’) in order to activate (set low) the corresponding chip select pin. Two functions (enable CS and disable CS) from the USART’s driver functions library help to ease the access to those control register bits.

2.8. 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
0xFFC0	TMR_CNT	0..15		R/W	Counter register
0xFFC2	TMR_THRES	0..15		R/W	Threshold register, IRQ on match
0xFFC4	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*

Table 15: High-precision timer 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.

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 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.

2.9. 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_ENABLE 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 bit. If the bit is set, the last reset was generated by a watchdog timeout. If the bit is cleared, the reset was generated via the external reset pin. You can find elementary functions for performing watchdog operations in the neo430_wdt.h library.

Timeout Configuration

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

CLKSELx bits configuration:	000	001	010	011	100	101	110	111
CLK prescaler:	2	4	8	64	128	1024	2048	4096
Timeout period in 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. When disabled, none of the provided functions are available and the system only be reset using the dedicated external reset signal.

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

Register Map

Address	Name	Bit(s)	Name	R/W	Function
0xFFD0	WDT_CTRL	0..2	WDT_CLKSELx	R/W	Timeout interval selection
		3	WDT_ENABLE	R/W	Watchdog enable bit
		4	WDT_RCAUSE	R/-	Cause of last processor reset (0=external reset, 1=watchdog timeout)
		5..7	-	R/-	*Reserved, read as 0*

Table 16: Watchdog timer address map

2.10. System Configuration Module (SYSCONFIG)

The system information module (VHDL component **`neo430_sysconfig.vhd`**) gives access to various system information and allows the configuration of the 4 interrupt vectors. The module is subdivided into two

Register Map Part I – System Information ROM

The first module (lower 8 word-aligned addresses) provides the read-only information memory, which contains various system information. By accessing this component, a program can determine the available RAM space, check if specific instructions or hardware module are present 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. All these parameters are set using the configuration generics of the NEO430 top entity during instantiation.

Address	Name	Bit(s) (Name)		R/W	Function
0xFFE0	CPUID0	0..15: HW_VERSION		R/-	Hardware version
0xFFE2	CPUID1	0	SYS_CFU_EN	R/-	Set if **CFU** 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_USART_EN	R/-	Set if **USART** 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..15: *reserved*		R/-	*Reserved, read as 0*
0xFFE4	CPUID2	0..15: USER_CODE		R/-	Custom user code, defined via top's generic
0xFFE6	CPUID3	0..15: IMEM_SIZE		R/-	Size of IMEM in bytes
0xFFE8	CPUID4	0..15: DMEM_BASE		R/-	Base address of DMEM
0xFFEA	CPUID5	0..15: DMEM_SIZE		R/-	Size of DMEM in bytes
0xFFEC	CPUID6	0..15: CLOCKSPEED_LO		R/-	Low word of clock speed (in Hz)
0xFFEE	CPUID7	0..15: CLOCKSPEED_HI		R/-	High word of clock speed (in Hz)

Table 17: System information memory address 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.

Register Map Part II – Interrupt Vector Configuration Registers

The second module (higher 8 word-aligned addresses) contains the four CPU interrupt vector configuration registers. These are used to specify the according handler addresses for each interrupt channel. The remaining four register addresses also access these interrupt vector configuration register, as they are simply mirrored to these addresses. The following table shows the different addresses and the execution priorities of each interrupt vector.

Address	Name	Priority	Bit(s)	R/W	Function
0xFFF0	*IRQVEC_TIMER*	*1 (highest)*	*0..15*	*R/W*	*Timer match IRQ handler address*
0xFFF2	*IRQVEC_USART*	*2*	*0..15*	*R/W*	*UART Rx/Tx done / SPI done IRQ handler address*
0xFFF4	*IRQVEC_GPIO*	*3*	*0..15*	*R/W*	*GPIO pin change IRQ handler address*
0xFFF6	*IRQVEC_EXT*	*4 (lowest)*	*0..15*	*R/W*	*External interrupt line IRQ handler address*
0xFFF8	IRQVEC_TIMER	1 (highest)	0..15	R/W	Timer match IRQ handler address
0xFFFA	IRQVEC_USART	2	0..15	R/W	UART Rx/Tx done / SPI done IRQ handler address
0xFFFC	IRQVEC_GPIO	3	0..15	R/W	GPIO pin change IRQ handler address
0xFFFE	IRQVEC_EXT	4 (lowest)	0..15	R/W	External interrupt line IRQ handler address

Table 18: Interrupt handler configuration (mirrored 2 times) memory address map

Typically, the interrupt vectors are initialized once at program start. You should assign a “dummy handler” for all interrupt channels, that are not actually used. The cut-out below illustrates this concept. In this example, only a handler for the timer interrupt is defined. All other interrupts will start a dummy handler (`invalid_irq`).

```
// interrupt vector table setup
IRQVEC_TIMER = (uint16_t)(&timer_irq_handler);
IRQVEC_USART = (uint16_t)(&invalid_irq);
IRQVEC_GPIO  = (uint16_t)(&invalid_irq);
IRQVEC_EXT   = (uint16_t)(&invalid_irq);
```

2.11. Custom Functions Unit (CFU)

If you want to add your own hardware accelerator, the custom function unit is the right place (VHDL component **neo430_cfu.vhd**) for it. This unit offers system designers the opportunity to directly integrate custom IP into the processor core. By adding a true *memory-mapped* device the additional access penalties when connecting it via the Wishbone bus are eliminated. Instead, the custom logic can be directly accessed using the CPU's memory access functions.

Implementation Control

By default, the CFU is not synthesized. You can use the `CFU_USE` generic of the processor top entity (see cut-out below) to control implementation.

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

The default CFU provided by this project is only a simple template. It implements eight 16-bit registers, which can also be accessed in single byte mode. The unit already implements the required write and read access logic, so you can directly start to add your custom logic "between" those interface registers. Typical applications might be multipliers (since we don't have one here, right?), dividers, other complex arithmetic (e.g. FFT butterfly accelerator) or more application specific logic like cryptography accelerators. Of course you can also add custom input and output ports to this unit to create your own communication device (maybe a two-wire interface?).

The CFU supports a maximum of 8 accessible 16-bit registers, which can also be accessed using 8-bit transfers. Thus, the address space for this unit is fixed to 16 bytes. If you need more configuration registers you can – for example – implement a FIFO-style access scheme or select one interface register as address buffer for selecting further configuration registers.

Just like the other IO device, the CFU does not feature a dedicated reset signal (of course you can add one). Therefore, you have to either initialize your registers during signal declaration (not so good) or use a single register as control register that performs a reset when set to zero (way better). However, many application like a multiplier may not need a reset at all.

The custom functions unit is accessed using direct register accesses (see table below). It is up to you to write dedicated driver functions for your specialized CFU.

Example Templates

The `rtl\cfu_templates` folder contains some example CFUs (e.g., a multiplay-and-accumulate unit) together with the required C header files to provide the access driver functions. Just replace the default `neo430_cfu.vhd` in the `rtl\core` by the corresponding one from the templates folder and include the header file in your application code.

And don't forget to enable the implementation of the CFU by setting the `CFU_USE` generic to "true" in the top entity of the entire processor ;)

Register Map

Address	Name	Bit(s)	R/W	Function
0xFF80	CFU_REG0	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF80	CFU_REG0_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF81	CFU_REG0_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF82	CFU_REG1	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF82	CFU_REG1_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF83	CFU_REG1_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF84	CFU_REG2	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF84	CFU_REG2_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF85	CFU_REG2_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF86	CFU_REG3	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF86	CFU_REG3_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF87	CFU_REG3_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF88	CFU_REG4	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF88	CFU_REG4_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF89	CFU_REG4_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF8A	CFU_REG5	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF8A	CFU_REG5_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF8B	CFU_REG5_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF8C	CFU_REG6	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF8C	CFU_REG6_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF8D	CFU_REG6_HI	8..15	R/W	Access the high part of the same register in 8-bit mode
0xFF8E	CFU_REG7	0..15	R/W	Simple register, accessed in 16-bit mode
0xFF8E	CFU_REG7_LO	0..7	R/W	Access the low part of the same register in 8-bit mode
0xFF8F	CFU_REG7_HI	8..15	R/W	Access the high part of the same register in 8-bit mode

Table 19: Custom functions unit address map

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. A batch file or a Linux/Cygwin makefile (`compile.bat` or `makefile`) in the `sw/common` folder helps to do this job. Generating an executable is done in several consecutive steps (all done by the compilation scripts):

1. The application start-up code (*crt0.asm*) is assembled into an object file. This start-up codes conducts 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.

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 actual program image 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_pplication_image.vhd` file – no manual copy required.

The last step is done by a small C program, which is located in the `sw/tools/image_gen` folder. A pre-compiled EXE file is available (it was built for a 64-bit Windows machine). If you are using Linux or Cygwin as build-environment, the make process will automatically recompile the image generator.

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 of can be uploaded to the processor to be directly executed and/or programmed into an attached 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:

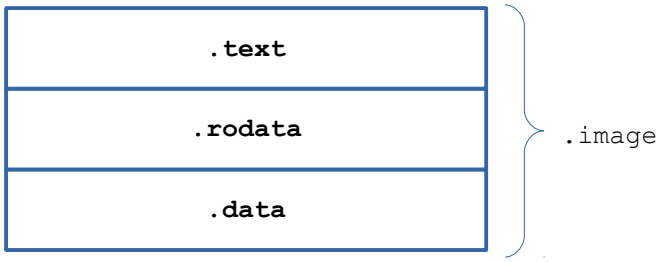

Figure 8: Construction of the final program image

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 – so 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 heap 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!

3.1.2. 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 some basic system setup:

- Setup the stack-pointer according to the memory size/layout configurations from the `CPUID` registers
- Clear the CPU status register: Disable interrupts and disable write access to IMEM
- Clear the control register of the Wishbone bus interface, of the serial interface (UART/SPI), of the GPIO port and of the timer → reset them
- Set all GPIO output ports to zero and deactivate the watchdog timer → reset it
- Set all interrupt vectors to 0x0000
- Clear complete DMEM, including the `.bss` segment
- Copy the `.data` section from IMEM to DMEM

- Initialize all CPU data registers to zero
- Call the application's main function
- If the main function returns, program control comes back to the start-up code, the watchdog timer is deactivated, the CPU SREG is cleared disabling interrupts and the CPU is set to eternal sleep mode.

3.1.3. Executable Image Formats

The specific 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 calling 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 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 USART 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 this image to this device 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 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 V20170716 by Stephan Nolting

HWV: 0x0120
CLK: 0x05F5E100
ROM: 0x4000
RAM: 0x2000
SYS: 0x00FF

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):

- **HWV**: Hardware version
- **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 is started (see next chapter).

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

```
NEO430 Bootloader V20170716 by Stephan Nolting

HWV: 0x0120
CLK: 0x05F5E100
ROM: 0x4000
RAM: 0x2000
SYS: 0x00FF

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

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

NEO430 bootloader console after pressing a key

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 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 launch it. Of course, you can also load it manually using the "Load EEPROM" option.

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 (at 1/1024 of the processor clock) 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 is send to the terminal, the status LED is permanently activated and the system must be manually reset to proceed.

- **Error 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).
- **Error 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.
- **Error 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.
- **Error 04**: Your program is way too big for the internal IMEM. Increase the IMEM size of your NEO430 project or optimize your code to save memory.
- **Error 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 or was built for a very old version of the bootloader.

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. The tutorials are partly written for Windows or Cygwin/Linux users, so make sure to select the right one.

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, only containing the core itself and 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, …).

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. 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 he basic steps are the same when using the core as part of your project.

4. The configuration of the NEO430 processor is done using the generics of the instantiated processor top entity (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 32kB (default=4kB)
  DMEM_SIZE   => 2*1024,        -- internal DMEM size in bytes, max 28kB (default=2kB)
  -- additional configuration --
  USER_CODE   => x"4788",       -- custom user code
  -- module configuration --
  DADD_USE    => true,          -- implement DADD instruction? (default=true)
  CFU_USE     => false,         -- implement custom functions unit? (default=false)
  WB32_USE    => true,          -- implement WB32 unit? (default=true)
  WDT_USE     => true,          -- implement WBT? (default=true)
  GPIO_USE    => true,          -- implement GPIO unit? (default=true)
  TIMER_USE   => true,          -- implement timer? (default=true)
  USART_USE   => true,          -- implement USART? (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)
)
...
```

5. 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.

6. 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;
```

7. 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.

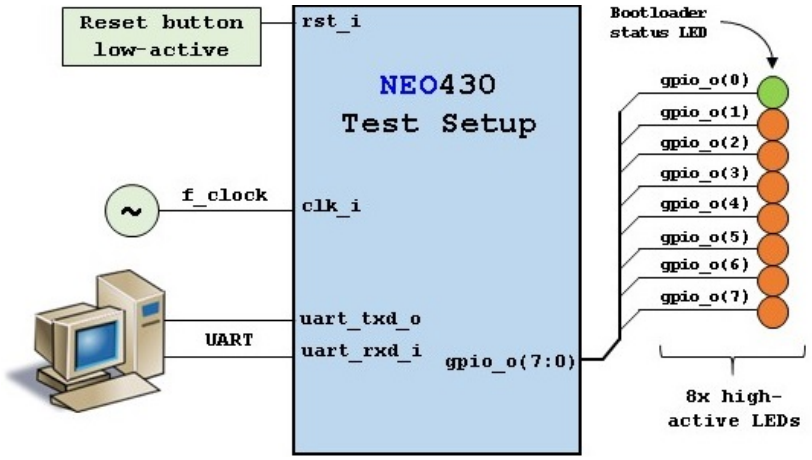

Figure 9: External hardware configuration of the NEO430 test implementation (neo430_test.vhd)

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

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

10. 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. This must be done regardless whether you are using Windows or Linux/Cygwin for the actual application compilation.

1. At first, download the latest version of the **TI msp430-gcc compiler tool chain**. You can downloaded it 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. 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 = 0x8000, LENGTH = 0x0800
}
```

4. 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(s).

4.3. Application Program Compilation using Windows CMD Batch File

Use this guide if you want to compile programs using **Windows**. The compilation script file, which must be executed, is "`make.bat`", which is available in each example project folder.

1. At first, open the common **`compile.bat`** compilation script in the **`sw/common`** folder with a text editor and look for the "USER CONFIGURATION" section.

2. Assign the absolute system path of the compiler's binary folder to the **`BIN_PATH`** variable (remember where you have installed it?). In the example below, the binary compiler sources are located in folder `C:\msp430-gcc-6.2.1.16_win32\bin`.

```
@REM Path of compiler binaries:
@set BIN_PATH=C:\msp430-gcc-6.2.1.16_win32\bin
```

3. That's all for now! Now you can start compiling programs. At first, we will begin with a simple example program. Open a console (`cmd`) and navigate to the `blink_led` folder in the project's software examples folder: `sw\example\blink_led`

4. Execute the actual compilation `make` batch file **`make.bat`** in the current example folder. By this, the "main.c" file is automatically used as main source file.

```
...\sw\example\blink_led>make
```

5. During the compilation process, several messages are generated:

```
...\sw\example\blink_led>make
Memory utilization:
   text    data     bss     dec     hex filename
    726       0       0     726     2d6 main.elf
Installing application image to rtl\core\neo430_application_image.vhd
Final executable size (bytes):
732
```

6. 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. If you re-synthesize your design, the image is ready to boot from the IMEM. Finally, the actual size of the created executable is shown.

7. If you want to use a specific file as main source file, you can pass that file as first argument:

```
...\sw\example\blink_led>make custom_main_file.c
```

8. Congratulations, you have just compiled your first application!

4.4. Application Program Compilation using Cygwin/Linux Makefile

Use this guide if you want to compile programs using **Cygwin** (on Windows) or directly **Linux**. The compilation script file, which must be executed, is "`Makefile`" (available in each example project folder).

1. At first, open the **`Makefile`** main compilation script in the **`sw/common`** folder with a text editor and look for the "USER CONFIGURATION" section.

2. Assign the absolute system path of the compiler's binary folder to the **`BIN_PATH`** variable (remember where you have installed it?). In the example below, the binary compiler sources are located in the folder `/mnt/c/msp430-gcc-6.2.1.16_linux64/bin` .

```
# Path of compiler binaries:
BIN_PATH = /mnt/c/msp430-gcc-6.2.1.16_linux64/bin
```

3. 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/blink_led`

4. Simply execute the actual compilation **`Makefile`** in the current example folder:

```
.../sw/example/blink_led$ make
```

5. During the compilation process, several messages are generated:

```
.../sw/example/blink_led$ make
Memory utilization:
   text    data     bss     dec     hex filename
    732       0       0     732     2dc main.elf
Installing application image to rtl/core/neo430_application_image.vhd
Final executable size (bytes):
738
```

6. 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. If you re-synthesize your design, the image is ready to boot from the IMEM. Finally, the actual size of the created executable is shown.

7. If you want to use a specific file as main source file, you can pass that file via the `MAIN` variable:

```
.../sw/example/blink_led$ make MAIN=custom_main_file.c
```

8. Congratulations, you have just compiled your first application! If you want to clean up your project's workspace again, you can execute a `make clean`.

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

When compiling an application, two final files are generated in the projec tfolder:

- **`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)

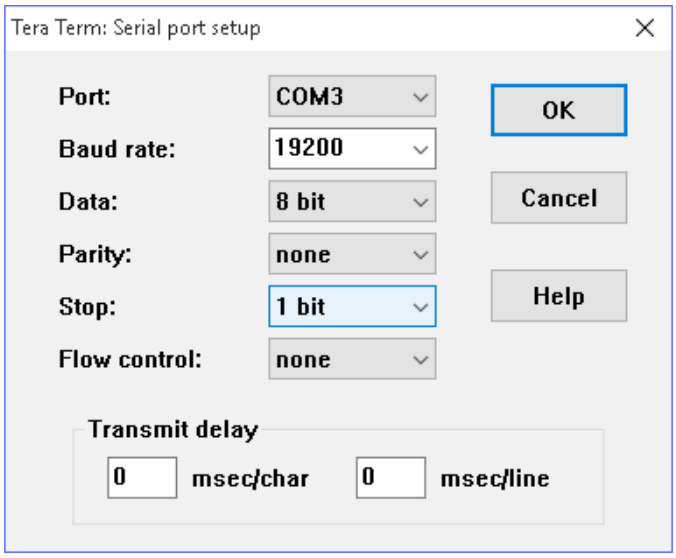

Figure 10: Serial configuration of TeraTerm

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 V20170716 by Stephan Nolting

HWV: 0x0120
CLK: 0x05F5E100
ROM: 0x4000
RAM: 0x2000
SYS: 0x00FF

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

d: Dump
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.

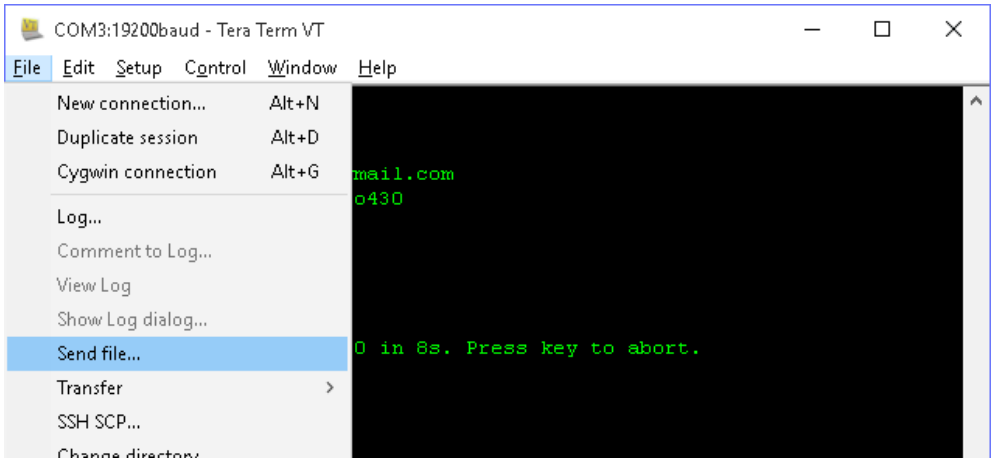

Figure 11: Sending a file using Tera Term

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 12: 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. Congratulations! 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. When your new project folder is located somewhere else (than in the **`sw/example`** folder), you need to adapt the compilation scripts in your project folder:

   ➔ If you are using the Windows-based batch file (**`make.bat`**), open it with a text editor and change the **`COMMON_PATH`** variable. The path given by this variable defines the relative path from the current project folder to the NEO430 **`sw/common`** folder.

   ➔ In case you are using Cygwin/Linux for compilation, the modifications have to be done in the **`makefile`**. Change the **`COMMON_PATH`** variable so it defines the relative part from the new project's folder to the **`sw\common folder`** folder.

   ➔ Finally, you also need to adapt the path of the included **`neo430.h`** library file in your program code.

4. If your project contains additional **`*.c`** files beside the **`main.c`** file, you have to include them into your **`main.c`** file using the C pre-processor:

```
#include "../../lib/neo430/neo430.h"

#include "some_file.c"     // one of your project source files
#include "another_file.c"  // another one of your project source files
```

5. That is all you need to do. Now you can compile your new project for the NEO430. If the simulator issues errors regarding the source file dependencies, you should adapt the order of the included C files. Also make sure to use “include guard” in all your source files.

4.8. Simulating the Processor

Before you do an actual FPGA implementation or if you want to see what's going on, you can do a simulation of the processor core. 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.

If you want to simulate your application code, you should disable the bootloader. Thus, the processor directly boots from the internal IMEM and executes your program without the 8s auto boot sequence (that takes ages when simulating).

To disable the bootloader, set the **`BOOTLD_USE`** generic of the instantiated processor core in the testbench to '**false**':

```
...
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. If you are using **Windows** as build environment, open the main Windows compilation batch file (`sw\common\make.bat`) and configure the **`EFFORT`** variable (in the "USER CONFIGURATION" section) for the required optimization goal:

```
@REM Compiler effort (-Os = optimize for size)
@set EFFORT=-Os
```

2. If you are using **Linux** or **Cygwin** as build environment, open the main Linux makefile (`sw/common/makefile`) and configure the **`EFFORT`** variable (in the "USER CONFIGURATION" section) for the required optimization goal:

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

3. 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**
- **Windows**: Open the **make.bat** file and edit the path to your compiler binaries folder:

```
@REM Path of compiler binaries:
@set BIN_PATH=C:\msp430-gcc-6.2.1.16_win32\bin
```

- **Windows**: Now execute a "make". This will compile the bootloader's sources and will also generate the bootloader VHDL memory initialization file **neo430_bootloader_image.vhd** file in the project's **rtl/core** folder.

```
...\sw\bootloader\default>make
```

- **Linux/Cygwin**: Open the **makefile** file and edit the path to your compiler binaries folder:

```
# Path of compiler binaries:
BIN_PATH = /mnt/c/msp430-gcc-6.2.1.16_linux64/bin
```

- **Linux/Cygwin**: Now execute a "make". This will compile the bootloader's sources and will also generate the bootloader VHDL memory initialization file **neo430_bootloader_image.vhd** file in the project's **rtl\core** folder.

```
.../sw/bootloader/default>make
```

- 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, 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`** 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 & AXI Lite Bus 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, AXI and Avalon are quite popular, especially in terms of Xilinx or 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`** or **`AXI Lite`** 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_std_logic.vhd`	Top entity using only std_logic / std_logic_vector as port types.

4.13. Troubleshooting

- ✔ Have you added all HDL files from the `rtl/core` folder to your project?
- ✔ 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 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!
- ✔ 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 to its binaries in the compilation script(s)? Did you install the correct compiler (TI's msp430-gcc)?
- ✔ If you are communicating with the bootloader via UART, have you configured your terminal with the right settings?
- ✔ Are you uploading the binary executable in raw-byte mode?
- ✔ Was the application compilation process successful?

5. Change Log

Date	Modifications
28.07.2016	New “initial” version – with added change log. Removed Timer SCRATCH register, added GPREG0 to GPREG3. SYSCONFIG is now one 16-entry RAM. Reworked some chapters.
15.08.2016	Added SYSMEM write enable flag to SYSMEM, updated UART baud configuration, changed initial state of processor after bootloader exits (no CPU reg clearing), tiny bootloaders are now both 512B large
19.08.2016	Changed bootloader menu style, added info about cleared register after bootloader termination, new tutorial chapter about simulating the core, added ISIM waveform configuration info
03.09.2016	Deleted SFU from project; added watchdog timer (WDT) instead
04.09.2016	Added manual reset capability to WDT
09.09.2016	Added information/tutorial about building a “bare-metal” application
14.09.2016	Added Linux makefiles for compiling the bootloaders; re-work of first chapters of this document
26.09.2016	Reworked core; new memory architecture; modified a LOT in the documentary
27.09.2016	Added check list to the getting started section
05.11.2016	New hardware version: Arithmetic instructions (excluding the decimal addition) need less cycles to execute; added info about the external interrupt request line; updated implementation results
18.11.2016	Added CPU_ORIGIN identifier to infomem; improved crt0.asm; added bootloader error code for updating the IMEM when it was implemented as true ROM
24.11.2016	Added support for TI’s msp430-gcc tool chain (only for compiling application code, the bootloader cannot be build using msp430-gcc yet!!!)
25.11.2016	Removed ending.asm code from compilation files (not needed at all)
26.11.2016	Removed support for MSPGCC since it is too buggy… The new TI msp430-gcc tool chain sadly does not support the NEO430 hardware multiplier, so I have removed that unit from the project
27.11.2016	Bootloader can now be compiled using msp430-gcc; updated bootloader sources
20.12.2016	Added “neo430_” suffix to all NEO430 rtl files
06.01.2017	Added support files for modelsim simulation; info regarding the missing hardware multiplier added
13.01.2017	Small improvements of the compile example chapters (added stuff regarding the new memory utilization output during the compile process)
07.02.2017	Added link to the neo430 project on *github*
09.02.2017	Added custom user code specifier; updated msp430-gcc name including version; updated bootloader screen shots; removed manual WDT reset functionality
22.02.2017	New hardware version **0x0108**: Moved configuration from package file to top entity generics; heavily reworked documentary; reworked C library files (names of hardware registers and bits); removed GP registers from the sysconfig module; renamed PIO unit to GPIO unit; changed C library constants (removed the ‘_C’ suffix)
23.02.2017	Updated Xilinx Virtex-6 implementation results
20.04.2017	Corrected typos
22.04.2017	New hardware version (**0x0109**): Added password error reset to WDT; added IRQ acknowledge signals (only relevant for ext. IRQ); reworked interrupt chapter; added info regarding dynamic memory

Date	Modifications
07.06.2017	New hardware version (**0x0111**): Re-added GPIO input IRQ mask register; added information regarding write-only configuration registers
20.06.2017	New hardware version (**0x0113**): Updated control state machine – in general one cycle less required for instruction execution. Updated implementation results (utilization by entity only).
19.07.2017	New hardware version (**0x0120**): Updated Wishbone controller (no more timeout counter) and drivers (to support correct byte addressing – thanks to Edward!); added custom functions unit and example application; updated documentary; modified bootloader; minor code fixes
19.07.2017	New hardware version (**0x0121**): Debugged and optimized Wishbone adapter; added CFU templates
20.07.2017	New hardware version (**0x0122**): Changed SPI chipselect bits in the USART control register to be high-active (set bit to one to set corresponding CS line to low)
15.08.2017	New hardware version (**0x0123**): Deactivating the global interrupt enable flag keps the queuing of further interrupt requests; added Avalon master and resolved port type top entities