NEO430 Processor

by Dipl.-Ing. Stephan Nolting

Processor Hardware Version: 0x0101

Proprietary Notice

License / Disclaimer

The **NEO430 Project**
© 2015-2017, 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://www.opencores.org/project,neo430
https://github.com/stnolting/neo430

Table of Content

1. Introduction

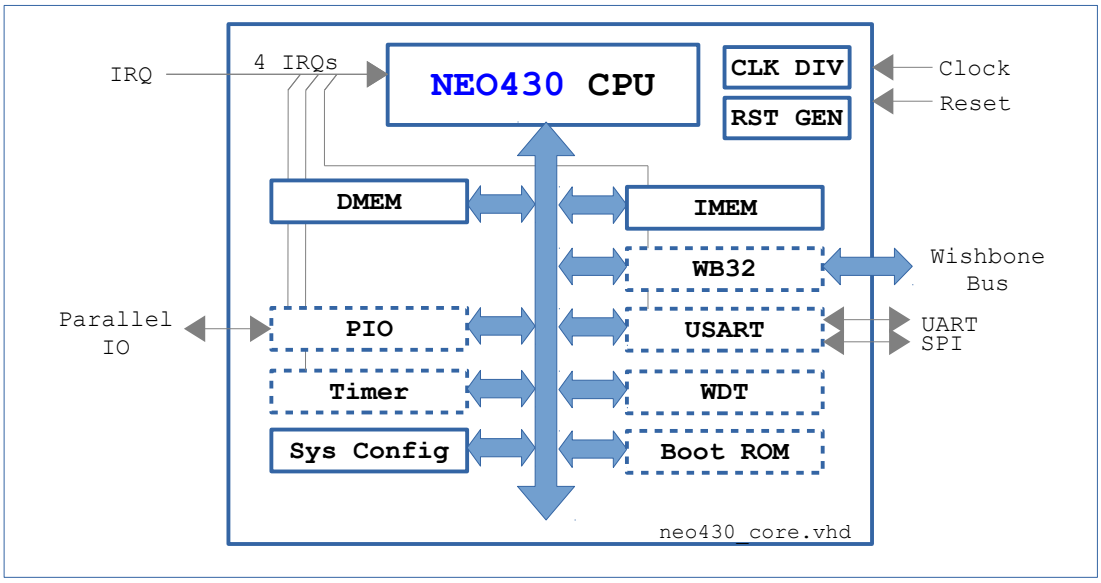

Figure 1: NEO430 processor block diagram, optional modules printed using dashed lines

Hello and welcome to the **NEO430 Processor** project!

This processor project is the latest one in my list of open-source soft core projects. It is my newest attempt to provide a small and nevertheless powerful processing platform for embedded applications. I designed the CPU of this project completely from scratch, based only on the original TI MSP430 specifications. Most of the IO devices – like the UART or the timer – came from my previous projects (STORM and Atlas2k). Finally, the NEO430 processor came out. This processor is 100% ISA-compatible (!) to the famous TI(TM) MSP430 instruction set architecture. Further information about the instruction set, the available addressing modes and general CPU operation can be found in the "*MSP430xl xx Family User's Guide*", available at www.ti.com/lit/ug/slau049f/slau049f.pdf.

The NEO430 features a very small outline, only implementing elementary modules like a single timer and basic peripherals and communication modules. Any additional IP blocks, which make a more customized system, can be connected via a Wishbone-compatible bus interface. Also, most of the processor-internal modules can be excluded from implementation, if their features are not required. By this, you can build a system-on-chip, that perfectly fits your needs without any unnecessary hardware overhead.

The software development for this processor system is based on the free **TI msp430-gcc compiler tool chain**. Make sure to get a copy of the sources to make this project run. 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". The default system setup can be synthesized and uploaded to your FPGA of choice. Application installation work by executing a single "make" command. Jump to the "**Let's Get It Started**" chapter to get your system running right now! That's all you need to do to get your first NEO430 project started. So, have fun with this project! ;)

1.1. Processor Features

- ✔ 16-bit RISC open source soft-core processor
- ✔ Full support of the original MSP430 instruction set architecture
- ✔ Tool chain based on free TI msp430-gcc compiler (C / C++)
- ✔ Application compilation scripts for Windows and Linux/Cygwin
- ✔ Completely described in behavioral, platform-independent VHDL93, no “exotic” VHDL packages used
- ✔ **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
- ✔ Fully customizable hardware outline
- ✔ Optional high-precision timer (TIMER)
- ✔ Optional USART interface; UART and SPI (USART)
- ✔ Optional parallel IO port (16 inputs, 16 outputs) with pin-change interrupt (PIO)
- ✔ 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

1.2. Main Differences to TI's Original MSP430 Architecture

Since the NEO430 is not intended as MSP430 clone, there are several differences to TI's original architecture. Existing programs must be modified and re-compiled to successfully run on the NEO430.

The *main* differences are:

- ✗ Completely different processor modules with different functionality
- ✗ No hardware multiplier support yet (but emulated in software)
- ✗ Maximum of 32kB instruction memory and 28kB data memory
- ✗ Specific memory map – included NEO430 linker script and compilation script required
- ✗ Just 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
- ✗ Wishbone-compatible interface to attach custom IP (e.g. modules from opencores.org)
- ✗ Internal bootloader with text interface (via UART serial port)

1.3. Project Folder Structure

The project contains several folders and sub-folders, which are about to be briefly explained.

```
├─doc            This folder contains a copy of the implemented Wishbone specifications
│                as well as the processor documentary (the document you are currently
│                reading).
│
├─rtl            All the rtl files of the project can be found here.
│ │
│ ├─core         This folder contains all the rtl (VHDL) files of the NEO430 processor.
│ │              Make sure to add ALL of them to your FPGA EDA project.
│ │
│ └─wishbone     This folder contains Wishbone network components. For instance, you can
│                find a compatible Wishbone boot ROM here.
│
├─sim            The sim folder contains a simple VHDL testbench for simulation and
│ │              additional simulation files (waveform configurations).
│ │
│ ├─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 with.
  │
  ├─bootloader   Sources and compilation scripts of the NEO430-internal bootloader.
  │
  ├─common       Compile and linker script for normal application program generation.
  │              Also, the ASM start-up code is located here.
  │
  ├─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.
```

Table 1: Project's folder structure

1.4. Processor VHDL File Hierarchy

All necessary hardware description files are located in the project's **`rtl/core`** folder. The top entity of the processor is `neo430_core.vhd`, but since this entity also provides interfaces (like the Wishbone bus) for chip-internal component, it is 'encapsulated' into the `neo430_top.vhd` file. Here, you can add your additional custom IP blocks. So the actual **top entity of the complete processor system** is **`neo430_top.vhd`**.

neo430_top.vhd	Complete system's top entity
└─**neo430_core.vhd**	Processor core top entity
├─**neo430application_image.vhd**	IMEM (ROM) initialization image for your custom application. During synthesis, the initialization image is moved to the boot ROM. This file is automatically generated and copied when compiling an application.
├─**neo430_boot_rom.vhd**	Bootloader ROM
├─**neo430_bootloader_image.vhd**	Boot ROM initialization image for the bootloader. During synthesis, the init image is moved to the IMEM. This file is automatically generated and copied when re-compiling the bootloader's sources.
├─**neo430_dmem.vhd**	Internal RAM for storing data
├─**neo430_imem.vhd**	Internal RAM or ROM for the application code
├─**neo430_package.vhd**	Processor VHDL package file
├─**neo430_parallel_io.vhd**	General purpose parallel IO port
├─**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

Table 2: RTL file hierarchy (`/rtl/core` folder)

1.5. System Top Entity

The following table shows all interface ports of the system's top entity (`neo430_top.vhd`). **Use this file as top entity for your first implementation.** The type of all signals is **std_ulogic** or **std_ulogic_vector**, respectively.

Signal name	Width (#bits)	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, synchronous, **low-active**	all
Parallel IO				
pio_o	16	Output	General purpose parallel output	PIO
pio_i	16	Input	General purpose parallel input	PIO
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[1] (boot EEPROM)	USART.SPI

Table 3: neo430_top.vhd - system's top entity interface ports

Of course you can encapsulate the system top entity `neo430_top.vhd` into another entity and only connect the specific signals, that you actually need, to the outer world. Obviously, the clock and reset signals are mandatory. The UART signals are required if you want to use the UART itself or the bootloader. The SPI signals are obviously needed for using the SPI module and also if you want to load program images (via the bootloader) from an external memory.

Bit #0 of the parallel output port is used by the bootloader to drive a status LED (high-active). This fact is only important when the bootloader is actually being executed. During normal program execution this bit can be used as general purpose output pin. However, you should not use this pin as data signal for any kind of external logic, which might go crazy by the bootloader's usage of this signal pin.

Chip select 0 (bit #0 of the `spi_cs0_o` signal) is used by the bootloader to access an SPI EEPROM for loading/storing a boot image. Hence, this chip select should not be used for any other purpose if you wish to use the “boot from EEPROM” feature of the bootloader.

1 Chip select 0 (bit #0) is used by the bootloader to access the boot SPI EEPROM.

1.6. Processor Top Entity

The following table shows all interface ports of the processor's top entity (`neo430_core.vhd`). The type of all signals is **std_ulogic** or **std_ulogic_vector**, respectively.

You should not select this file as top entity for implementation. The processor's top entity is meant for being instantiated within a larger, more complex design, that also implements your customlogic. You can use the `neo430_top.vhd` as starting point for your very own SoC design.

Signal name	Width (#bits)	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, synchronous, low-active	all
Parallel IO				
pio_o	16	Output	General purpose parallel output	PIO
pio_i	16	Input	General purpose parallel input	PIO
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	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
Interrupt Request Lines				
irq_i	1	Input	Interrupt request signal, high-active	CPU

Table 4: neo430_core.vhd - processor's top entity generics and interface ports

1.7. Implementation Results

This chapter shows some example implementation of the NEO430 processor for different FPGA platforms and for different configurations.

1.7.1. Full Default Implementation

Hardware configuration:

- Hardware version: 0x0100
- Internal IMEM: 4kB
- Internal DMEM: 2kB
- Boot ROM: 2kB (default bootloader)
- Wishbone bus adapter (WB32): Included
- Parallel IO port (PIO): Included
- Serial interface (USART): Included
- High-precision timer (TIMER): Included
- Watchdog timer (WDT): Included
- DADD instruction: Included

Altera Cyclone IV EP4CE22F17C6N	
Resource	**Utilization**
`Total logic elements:`	`1188 / 22320 = 5%`
`Dedicated logic registers:`	`556 / 22320 = 2%`
`Total memory bits:`	`70016 / 608256 = 11%`
`9-bit multiplier:`	`0 / 132 = 0%`
`Maximum Frequency:`	`116 MHz (slow 1200mV 0°C model)`

Table 5: Intel Quartus Prime 16.1 fitter results ("balanced implementation")

Xilinx Virtex-6 XC6VLX240T-2FF1156	
Resource	**Utilization**
`Number of 6-input LUTs:`	`811 / 150720 = >1%`
`Number of Slice Registers:`	`676 / 301440 = >1%`
`Number of Block RAM:`	`4 / 832 = >1%`
`Number of DSP48E1s:`	`0 / 768 = 0%`
`Maximum Frequency:`	`174 MHz (synthesis result)`

Table 6: Xilinx ISE 14.7 P&R results ("speed-optimized")

1.7.2. Custom Implementation

Hardware configuration:

- Hardware version: 0x0091
- Internal IMEM: 2kB
- Internal DMEM: 1kB
- Boot ROM: 2kB (default bootloader)
- Wishbone bus adapter (WB32): Included
- Parallel IO port (PIO): Included
- Serial interface (USART): Included
- High-precision timer (TIMER): Included
- Watchdog timer (WDT): Included
- DADD instruction: Included
- Additional features: MachXO2 EFB interface, Wishbone switching fabric

Lattice MachXO2 MACHXO2-1200ZE	
Resource	**Utilization**
Number of LUT4s:	1096 / 1280 = 86%
Number of registers:	519 / 1595 = 33%
Number of block RAMs:	6 / 7 = 86%
Maximum Frequency:	12.09 MHz (constrained)

Table 7: Lattice Diamond 3.7.1.502 mapping results

1.6.3. Resource Utilization by Entity

This table shows the required resources for each entity of the processor system. Logic functions of different modules might be merged into single logic elements (LE), so the total number of required resources might slightly change from the table above. Implemented configuration: Default configuration (all optional modules included, 4kB IMEM, 2kB DMEM, 2kB boot ROM), hardware version: 0x0091.

Altera Cyclone IV EP4CE22F17C6N					
Entity/Module	**Function**	**LEs**	**FFs**	**MEM bits**	**DSPs**
CPU	Central processing unit	578	189	256	0
IMEM (4kB)	Instruction memory (RAM)	5	1	32768	0
DMEM (2kB)	Data memory (RAM)	5	1	16384	0
Boot ROM (2kB)	Bootloader ROM	1	1	20480	0
Sysconfig	System information & IRQ config	8	9	128	0
PIO	GPIO parallel in/out ports	27	44	0	0
WDT	Watchdog timer	46	34	0	0
TIMER	High-precision timer	82	55	0	0
USART	Serial interfaces (UART + SPI)	160	124	0	0
WB32	Wishbone bus interface	60	76	0	0

Table 8: Intel Quartus Primes 16.1 fitter results

2. Hardware Architecture

The NEO430 processor system is constructed from several different modules. This chapter takes a closer look at all of these modules and their specific functionality.

Address Space

Since the NEO430 is a complete new processor system - implementing the full MSP430 ISA - the included hardware modules and thus, the provided functions and the resulting memory layout are completely new. The complete address space is 64kB deep (16 bit address).

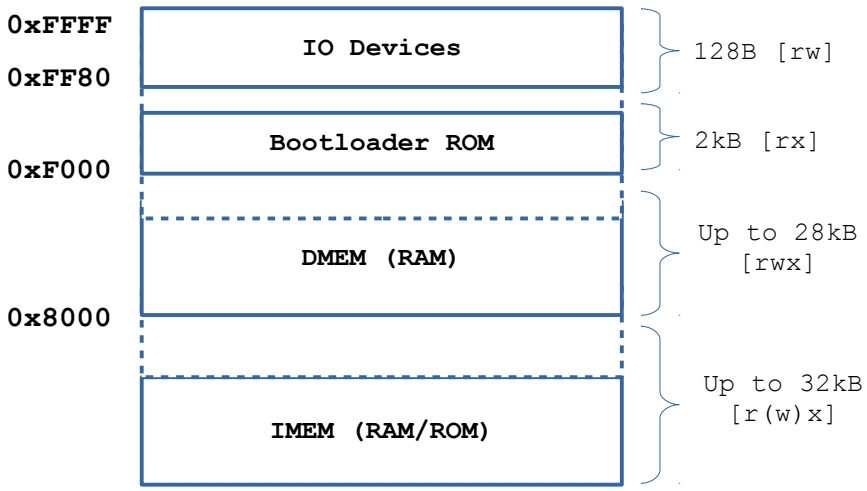

Figure 2: NEO430 address space

Peripheral/IO Devices

In contrast to the original MSP430, the NEO430 does not have any special function registers at the beginning of the memory space. Instead, all 'special functions' - like peripheral/IO devices and interrupt enable configurations - are conducted by configuration bits 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 deep. 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 MSP, the NEO uses (*now)* 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 this RAM 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 a configuration parameter). 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 of the peripheral IO devices in the IO region can only be accesses using 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 (e.g. clearing the corresponding control register before using the unit). The application's start-up code will perform the minimal-required system initialization. The hardware reset signal of the processor can either be triggered via the external reset pin or via the internal watchdog timer (if implemented).

Internal Clock Generator

An internal clock divider generates 8 selectable clocks for the different processor modules. These clocks are derived from the main processor input clock signal. If none of the connected devices require an active clock, the clock divider is automatically deactivated to save dynamic power.

External Interrupt Request Signal

The NEO430 processor (neo430_core.vhd) features a single external interrupt request signal (irq_i). The interrupt is triggered by a high level applied to this signal (and synchronous to the processor clock). For some applications, a single IRQ signal is enough, but other design may require several interrupt signals. If you need more than one, you can attach a custom interrupt – connected via the Wishbone bus – to the processor's external interrupt line. It is the task of this interrupt controller to schedule all arriving interrupt and to trigger the processor's IRQ line only once until the interrupt is acknowledges. For instance, the interrupt can be acknowledged by reading the controller's "interrupt source" register (depends on the controller's implementation). After reading such a register, the interrupt handler of the NEO430 external IRQ can call the appropriate software handler.

Hardware Multiplier Support

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 tool chain, therefor a compiler-supported hardware multiplier is not available yet. However, you could connect a multiplier (tailored to your needs regarding precision) as co-processor to the Wishbone interface to have a feasible work-around. ;)

2.1. NEO430 CPU

The CPU is the heart of the NEO430 processor. It implements all of the instructions, emulated instructions and addressing modes of the original TI MSP430 instruction set architecture (ISA). However, there are many differences especially when it comes to memory layout and attached peripheral devices. Also, the required cycles for instruction execution of the NEO430 are increased due to the fully registered architecture. However, this is no major drawback, since the NEO430 processor can operate at very high clock rates.

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 and 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 quite complex finite state machine (FSM). This FSM generates the control signals for the data path, that actually processes the data. The image below shows the simplified architecture of this data path.

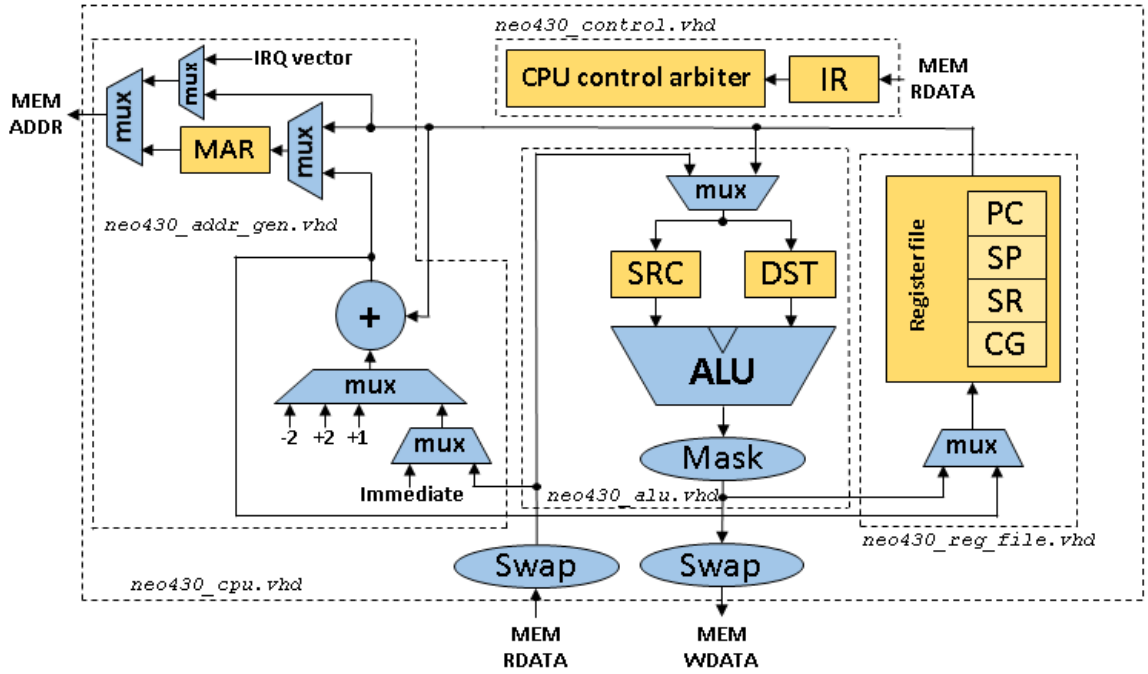

Figure 3: Simplified data path of the NEO430 CPU

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 results from the last ALU operation. Via the **I** flag, interrupts can be globally activated or deactivated. If this flag is disabled, all pending interrupt requests in the CPU are deleted. 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 IO devices – 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, when cleared, only read accesses to the IMEM are possible. 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.

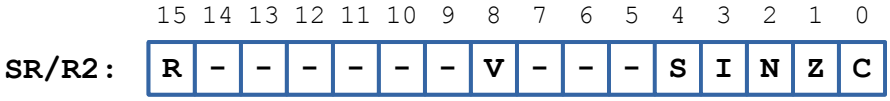

Figure 4: Processor status register

Bit#	Name	R/W	Function
0	**C_FLAG_C**	R/W	**C**arry flag
1	**Z_FLAG_C**	R/W	**Z**ero flag
2	**N_FLAG_C**	R/W	**N**egative flag
3	**I_FLAG_C**	R/W	Global **I**nterrupt enable
4	**S_FLAG_C**	R/W	**S**leep mode (CPU off)
5..7	-	-	*Reserved, read as 0*
8	**V_FLAG_C**	R/W	O**V**erflow flag
9..14	-	-	*Reserved, read as 0*
15	**R_FLAG_C**	R/W	Allow write access to IMEM "**R**OM"

Table 9: Bits of the status register

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

If the global interrupt enable flag is deactivated (cleared), all pending interrupt requests are deleted. Also, no new requests can en-queue until the flag is reactivated.

2.1.2. Interrupts

The NEO430 features 4 independent interrupt request lines. When triggered, each of this requests start a unique interrupt handler. The base address 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.

IRQ 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_PIO	3	PIO input pin change.
IRQVEC_EXT	4 (lowest)	External interrupt request via the `irq_i` top entity signal.

Table 10: Interrupt sources, priorities and handler base address configuration register names

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. These enable flags are located in the according interrupt-causing modules (e.g. the timer).

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 stored. When the handler of the interrupt with the highest priority exits, the handler of the interrupt with the next priority stage is started afterward. Of course, you can reactivate the global interrupt enable flag inside an interrupt handler to implement a nested interrupt behavior.

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.

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

2.1.3. Instruction Set

The instruction set of the NEO430 CPU is fully compatible to the original TI MSP430 (c) instruction set architecture (ISA). This chapter gives a brief overview of the core instruction set – the pseudo-instruction are not listed in this chapter. The full data sheet and the according user guide can be found at www.ti.com/lit/ug/slau049f/slau049f.pdf.

Notation

As	Source addressing mode.
Ad	Destination addressing mode.
d/s	Regarding to the **d**estination or the **s**ource.
n, reg, src, dst	Any register from the register file (if not further specified).
x	A 16-bit word.
PC	The program counter (r0).
SP	The stack pointer (r1).
SP	The status register (r2).
CG	The constant generator (r3).
B/W	**B**yte ('1') or **W**ord ('0') operation.

Addressing Modes

As	Ad	d/s	Register	ASM Syntax	Description
00	0	ds	n != 3	**Rn**	Register direct: The operand is the content of Register Rn.
01	1	ds	n != 0,2,3	**x(Rn)**	Indexed: The operand is located in memory at address Rn+x.
10	-	s	n != 0,2,3	**@Rn**	Register indirect: The operand is in memory located t the address stored in Rn.
11	-	s	n != 0,2,3	**@Rn+**	Indirect auto-increment: As register indirect, after wards the register Rn is incremented by one or two.
01	1	ds	0 (=PC)	**Label**	Symbolic: The operand is located in memory at address PC+x.
11	-	s	0 (=PC)	**#x**	Immediate: The operand is the next word in the instruction stream.
01	1	ds	2 (=SR)	**&Label**	Absolute: The operand is in memory located at address x.
10	-	s	2 (=SR)	**#4**	Constant: The operand is “4”.
11	-	s	2 (=SR)	**#8**	Constant: The operand is “8”.
00	-	s	3 (=CG)	**#0**	Constant: The operand is “0”.
01	-	s	3 (=CG)	**#1**	Constant: The operand is “1”.
10	-	s	3 (=CG)	**#2**	Constant: The operand is “2”.
11	-	s	3 (=CG)	**#-1**	Constant: The operand is “-1”.

Format I instructions (double-operand)

Opcode				Source reg				Ad	B/W	As		Destination reg				Mnemonic	Description
15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
0	1	0	0	src				Ad	B/W	As		dst				**MOV**	Copy src to dst.
0	1	0	1	src				Ad	B/W	As		dst				**ADD**	Add src to dst.
0	1	1	0	src				Ad	B/W	As		dst				**ADDC**	Add src to dst with carry.
0	1	1	1	src				Ad	B/W	As		dst				**SUBC**	Subtract src from dst with carry.
1	0	0	0	src				Ad	B/W	As		dst				**SUB**	Subtract src from dst.
1	0	0	1	src				Ad	B/W	As		dst				**CMP**	Compare src with carry (no result).
1	0	1	0	src				Ad	B/W	As		dst				**DADD**	BCD addition of src and dst with carry.
1	0	1	1	src				Ad	B/W	As		dst				**BIT**	Test bits (AND) of src and dst (no result).
1	1	0	0	src				Ad	B/W	As		dst				**BIC**	Bit clear (AND NOT) src and dst.
1	1	0	1	src				Ad	B/W	As		dst				**BIS**	Bit set (OR) of src and dst.
1	1	1	0	src				Ad	B/W	As		dst				**XOR**	Exclusive or of src and dst.
1	1	1	1	src				Ad	B/W	As		dst				**AND**	Logical AND of src and dst.

Format II instruction (single-operand)

						Opcode			B/W	As		Register				Mnemonic	Description
15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
0	0	0	1	0	0	0	0	0	B/W	As		reg				**RRC**	Rotate right through carry.
0	0	0	1	0	0	0	0	1	0	As		reg				**SWPB**	Swap bytes.
0	0	0	1	0	0	0	1	0	B/W	As		reg				**RRA**	Rotate right arithmetically.
0	0	0	1	0	0	0	1	1	0	As		reg				**SXT**	Sign extend byte to word.
0	0	0	1	0	0	1	0	0	B/W	As		reg				**PUSH**	Push data on the stack.
0	0	0	1	0	0	1	0	1	0	As		reg				**CALL**	Call a subroutine.
0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	**RETI**	Return from interrupt handler.

Branches

			Condition			Offset									Mnemonic	Description
15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
0	0	1	0	0	0	10-bit offset (signed)										**JNE/JNZ**	Jump if not equal / not zero.
0	0	1	0	0	1	10-bit offset (signed)										**JEQ/JZ**	Jump if equal / zero.
0	0	1	0	1	0	10-bit offset (signed)										**JNG/JLO**	Jump if no carry / lower.
0	0	1	0	1	1	10-bit offset (signed)										**JC/JHS**	Jump if carry / higher or same.
0	0	1	1	0	0	10-bit offset (signed)										**JN**	Jump if negative.
0	0	1	1	0	1	10-bit offset (signed)										**JGE**	Jump if greater or equal.
0	0	1	1	1	0	10-bit offset (signed)										**JL**	Jump if less.
0	0	1	1	1	1	10-bit offset (signed)										**JMP**	Jump always (unconditional).

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**	7	10	8	8
	Indexed [R+n]	10	11	11	11

*Table 11: Double-operand (**format I**) instruction execution cycles*

The double-operand (format I) "decimal addition instruction" (`DADD`) requires an additional execution cycle to complete.

		SRC = DST			
		Register direct R	**Indexed [R+n]**	**Indirect [R]**	**Indirect auto inc [R++]**
Operation	**CALL**	9	12	10	10
	PUSH	8	11	9	9
	Others	7	10	8	8

*Table 12: Single-operand (**format II**) instruction execution cycles (except `RETI`)*

Instruction / Operation	**Branches**	4
	RETI	9
	Interrupt	7

Table 13: Special instructions / operations execution cycles

The "*decimal coded binary addition with carry*" instruction (`DADD`) requires a lot of hardware and may also represents the critical path of the CPU. 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 `synth_dadd_c` constant in the `neo430_package.vhd` file.

2.1.5. System Bus

All components of the NEO430 processor – including main memory and bootloader ROM – are connected to the CPU via the main system bus. Since the connected devices are accessed using a memory-mapped approach, the actual selection of a specific device must be done inside the modules by checking the applied address.

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 14: 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 IO device, the actual 16-bit address and the data, that shall be written, are applied together with the write enable signal(s). For single byte transmission only the corresponding bit of the WREN signal is set. All in all, a write transaction needs only a single cycle to complete. Read operations do 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.

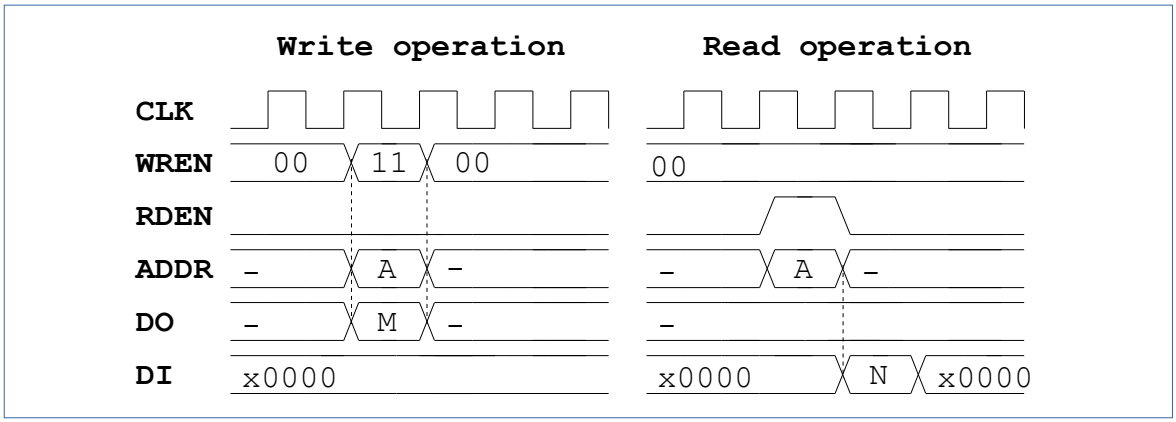

Figure 5: Write and read bus cycles (full word transfers); Write: M → [A]; Read: [A] = N

To prevent a multiplexer-style implementation of the data read-back path to the processor, which might create additional propagation times, a different approach has been implemented: 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 may 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 it's data output port in the *next cycle*. In any other situation, the data output of that module has to be set to 0x0000.
Of course you can add custom modules to this 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 in the neo430_package.vhd file (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 *infomem* module.

```
constant imem_size_c : natural := 4*1024; -- internal IMEM size in bytes, max 32kB
```

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 in the neo430_package.vhd file (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 *infomem* module.

```
constant dmem_size_c : natural := 2*1024; -- internal DMEM size in bytes, max 28kB
```

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. ROM size can be configured in the neo430_package.vhd file (see cut-out below), but **must not exceed 2kB**. During synthesis, the boot ROM is initialized ussing the neo430_bootloader_image.vhd file.

```
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 switch from the neo430_package.vhd file (see cut-out below) to exclude it. If the bootloader implementation is deactivated, the CPU starts booting from address 0x0000 instead from the base address of the boot ROM at 0xF000.

```
constant bootld_use_c : boolean := false; -- implement and use bootloader?
```

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. But - if required - only a subsection of the address and/or data buses can be actually connected to create a Wishbone bus with 32-bit address and 16-bit data, 16-bit address and 32-bit data or the minimal version with 16-bit data and address buses. The `neo430_wishbone.h` library file in the `sw/lib/neo430` folder already implements the most common Wishbone operations. A detailed description of the implemented Wishbone bus protocol and the according interface signals can be found in opencores' documentation data sheet "*Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*". A copy of this document is located in the `doc` folder of this project.

Implementation Control

Use the `wb32_use_c` switch from the `neo430_package.vhd` file (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.

```
constant wb32_use_c : boolean := true; -- implement WB32 unit?
```

Wishbone Transactions

To perform a Wishbone transaction, several tiny steps are required. At first, the according byte enable signals (`WB32_CT_WBSELx_C`) and the transfer cycle type (see later) must be configured in the control register (see table below). Additional, the module must be activated by setting the `WB32_CT_EN_C` bit in the control register.

In case of a write transfer, the data, which shall be written, must be loaded into the `WB32_HDO` (high part) and `WB32_LDO` (low part) registers. These registers directly drive the `wb_dat_o` data output bus. Next, the byte write enable signals for the `wb_sel_o` bus must be configured in the control register (only one time for a specific configuration) These configuration bits directly drive the `wb_sel` bus signal. To start the actual transfer, the address is written to the `WB32_HA` and **`WB32_LAW`** register. The actual store access to the low address word register initiates the transfer.

For a read transfer, the high part of the address is also stored to the `WB32_HA` register, but for this scenario, the low part must be written to **`WB32_LAR`** to trigger a read transaction. As soon as the transaction is started, the `WB32_CT_PENDING_C` 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. Also, a pending transfer can be automatically terminated if the accessed devices is not responding (no ACK signal). Set the `WB32_CT_TO_EN_C` bit in the control register to enable the auto-timeout feature. In this case, a bus access can take up to **256 cycles**, before it is automatically canceled. This timeout error occurs can be checked via `WB32_CT_TIMOEUT_C` bit of the control register. If you wish to abort a pending transfer, you have to disable the device by clearing the `WB32_CT_EN_C` bit in the control register.

The Wishbone bus interface of the NEO430 processor **cannot** be used for instruction fetch or direct data access via the standard memory-access operations. 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 are enabled by setting the `WB32_PMODE_C` bit of the control register.

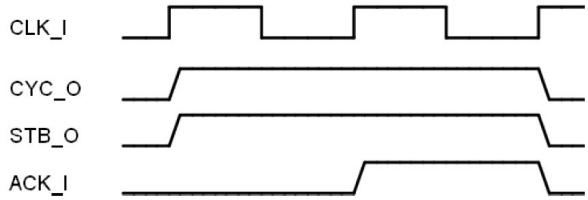

Figure 6: Standard Wishbone cycle

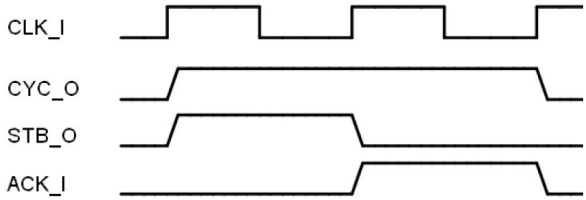

Figure 7: Pipelined Wishbone cycle

Only a the very elementary signals of the Wishbone interface protocol are implemented by this adapter unit. These signals are sufficient to provide classic and/or pipelined single-access transfer. If additional signals are required (LOCK, ERR, …) you need to implement a custom logic for them.

Wishbone 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 15: Wishbone bus interface adapter signals

The Wishbone bus operates using the processor clock.

Register Map

Address	Name	Bit(s) (Name)		R/W	Function
0xFF90	WB32_LAR	0..15		W	Low address word for read transfer (+trigger)
0xFF92	WB32_LAW	0..15		W	Low address word for write transfer (+trigger)
0xFF94	WB32_HA	0..15		W	High address word for read/write transfer
0xFF96	WB32_LDO	0..15		W	Low word of write data
0xFF98	WB32_HDO	0..15		W	High word of write data
0xFF9A	WB32_LDI	0..15		R	Low word of read data
0xFF9C	WB32_HDI	0..15		R	High word of read data
0xFF9E	WB32_CT	0	WB32_CT_WBSEL0_C	R/W	Byte 0 transfer enable → wb_sel_o(0)
		1	WB32_CT_WBSEL1_C	R/W	Byte 1 transfer enable → wb_sel_o(1)
		2	WB32_CT_WBSEL2_C	R/W	Byte 2 transfer enable → wb_sel_o(2)
		3	WB32_CT_WBSEL3_C	R/W	Byte 3 transfer enable → wb_sel_o(3)
		4..10		-	*reserved*
		11	WB32_CT_TO_EN_C	R/W	Enable auto-timeout when 1
		12	WB32_CT_EN_C	R/W	Enable Wishbone interface adapter when 1
		13	WB32_CT_PMODE_C	R/W	0: Standard / 1: Pipelined transfer
		14	WB32_CT_TIMEOUT_C	R	Timeout occurred during bus access
		15	WB32_CT_PENDING_C	R	Pending Wishbone transfer flag

Table 16: 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.

2.6. Parallel IO (PIO)

The parallel IO controller (VHDL component **neo430_parallel_io.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 or as control signals for other chips) or system-internally to provide control signals for other IP modules.

Pin-Change Interrupt

The parallel input port features a **single** pin-change interrupt. The trigger type for these pins can be set to low-active, high-active, rising-edge or falling-edge by writing the according value (see table below) to the `PIO_CTRL` register. Bit #2 enables the pin-.change functionality when set. If any 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_pio.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	PIO_IN	0..15	R	Parallel input port
0xFFB2	PIO_OUT	0..15	R/W	Parallel output port
0xFFB4	PIO_CTRL	0..1	W	Interrupt trigger: 00: Low-level 01: High-level 10: Falling-edge 11: Rising-edge
		2	W	Interrupt enable
		3..15	-	*Reserved*

Table 17: Parallel IO module address map

An LED connected to bit #0 of the parallel output port is used by the bootloader as status indicator. This output pin should only be used for driving a status LED. Nevertheless, the pin operates as general purpose output during normal program operation.

Implementation Control

I know, hardware resources are precious. The PIO 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 PIO module from synthesis. Use the `pio_use_c` switch from the `neo430_package.vhd` file (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.

```
constant pio_use_c : boolean := true; -- implement parallel IO unit?
```

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_C	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_C	R/W	USI enable (SPI & UART)
		1	USI_CT_UARTRXIRQ_C	R/W	UART Rx available IRQ enable
		2	USI_CT_UARTTXIRQ_C	R/W	UART Tx done IRQ enable
		3	USI_CT_UARTTXBSY_C	R	UART Tx busy flag
		4	USI_CT_SPICPHA_C	R/W	SPI clock phase
		5	USI_CT_SPIIRQ_C	R/W	SPI transmission done IRQ enable
		6	USI_CT_SPIBSY_C	R	SPI transceiver busy flag
		7	USI_CT_SPIPRSC0_C	R/W	SPI clock select (prescaler value)
		8	USI_CT_SPIPRSC1_C	R/W	
		9	USI_CT_SPIPRSC2_C	R/W	
		10	USI_CT_SPICS0_C	R/W	Dedicated SPI CS0 pin control
		11	USI_CT_SPICS1_C	R/W	Dedicated SPI CS1 pin control
		12	USI_CT_SPICS2_C	R/W	Dedicated SPI CS2 pin control
		13	USI_CT_SPICS3_C	R/W	Dedicated SPI CS3 pin control
		14	USI_CT_SPICS4_C	R/W	Dedicated SPI CS4 pin control
		15	USI_CT_SPICS5_C	R/W	Dedicated SPI CS5 pin control

Table 18: 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_c` switch from the `neo430_package.vhd` file (see cut-out below) to control implementation. But keep in mind, that this communication unit is required to access the console of the default standard bootloader!

```
constant usart_use_c : boolean := true; -- implement USART?
```

USART/USI Interrupt

The USART features a single interrupt output, which can be used to indicate an UART Rx data available status and/or an UART Tx done status and/or an SPI transfer completed status. Whe using more than one possible interrupt sources, you have to check the USART control register by yourself to determine the actual causing event. If the USART module is excluded from synthesis (disabled), its interrupt request signals is 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_C` 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 as long as the `USI_CT_UARTTXBSY_C` 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.

Example:

Clock frequency: **50MHz**
Desired Baudrate: **19200**
Prescaler: **64 → 011**
BAUD value: **41**

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

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

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_C` 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_C` 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 PIO controller or a specific Wishbone device.

The SPI transmission provides several configuration options. The actual clock phase can be determined via the `USI_CT_SPICPHA_C` 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_C` 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:

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

Int 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.

The intrinsic chip-select line #0 (CS0) of the USART is dedicated for the use of a boot EEPROM. Hence, it is used by the bootloader to access an EEPROM for program storage. Therefore, you should not use this signal for any other devices.

2.8. High-Precision 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_c` switch from the `neo430_package.vhd` file (see cut-out below) to control implementation. But keep in mind, that this unit is required by the default standard bootloader to operate properly!

```
constant timer_use_c : boolean := true; -- implement TIMER?
```

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_C	R/W	Timer enable
		1	TMR_CT_ARST_C	R/W	Automatic reset on timer match
		2	TMR_CT_IRQ_C	R/W	IRQ enable
		3	TMR_CT_PRSC0_C	R/W	Timer counter increment clock prescaler PRSC
		4	TMR_CT_PRSC1_C	R/W	
		5	TMR_CT_PRSC2_C	R/W	
		7..15		R	*Reserved, read as 0*

Table 19: 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_C` in the control register `TMR_CT` and setting a timer threshold `TMR_THRES`. Corrseponding 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_C` 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_C` bit). If the auto-reset bit `TMR_CT_ARST_C` 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_C` 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, the access is ignored. A user can determine the causeof 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. A manual hardware reset can also be triggered at any time by an application program by setting the WDT_SYSRST bit.

To control the timeout period, one can select 1 of 8 different timeout period 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_c switch from the neo430_package.vhd file (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.

```
constant wdt_use_c : boolean := true; -- implement WDT?
```

Address	Name	Bit(s)	Name	R/W	Function
0xFFD0	WDT_CTRL	0..2	WDT_CLKSELx_C	R/W	Timeout interval selection
		3	WDT_ENABLE_C	R/W	Watchdog enable bit
		4	WDT_RCAUSE_C	R/-	Cause of last processor reset (0=external reset, 1=watchdog timeout)
		5	WDT_SYSRST_C	-/W	Force hardware reset when set to '1'
		8..15	WDT_PASSWORD_C	-/W	Watchdog access password (0x47)

Table 20: Watchdog timer address map

2.10. System Configuration Module

The main system information module (VHDL component **`neo430_sysconfig.vhd`**) gives access to various system information and the interrupt vector configuration. Therefore, the module is subdivided into two consecutive modules with 8 word-aligned addresses each.

System Info – System Info

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. All these parameters are specified in the main processor configuration package file: `neo430_package.vhd`.

Address	Name	Bit(s) (Name)		R/W	Function
0xFFE0	CPUID0	0..15: HW_VERSION		R	Hardware version
0xFFE2	CPUID1	0	*reserved*	R	*Reserved, read as 0*
		1	SYS_WB32_EN_C	R	Set if **WB32** is implemented
		2	SYS_WDT_EN_C	R	Set if **WDT** is implemented
		3	SYS_PIO_EN_C	R	Set if **PIO** is implemented
		4	SYS_TIMER_EN_C	R	Set if **TIMER** is implemented
		5	SYS_USART_EN_C	R	Set if **USART** is implemented
		6	SYS_DADD_EN_C	R	Set if **DADD** instruction (CPU) is implemented
		7	SYS_BTLD_EN_C	R	Set if bootloader is implemented and used
		8	SYS_IROM_EN_C	R	Set if IMEM is implemented as true ROM
		9..15: *reserved*		R	*Reserved, read as 0*
0xFFE4	CPUID2	0..15: CPU_ORIGIN		R	Always read as 0x4E053
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 21: System information memory address map

Interrupt Vectors Configuration – Register Map

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 are used to implement 4 general purpose registers. These register do not have any kind of interrupt functionality. The following table shows the different addresses and the execution priorities of each interrupt vector.

Address	Name	Priority	Bit(s)	R/W	Function
0xFFF0	GPREG0	-	0..15	R/W	General purpose register 0, **no interrupt function!**
0xFFF2	GPREG1	-	0..15	R/W	General purpose register 1, **no interrupt function!**
0xFFF4	GPREG2	-	0..15	R/W	General purpose register 2, **no interrupt function!**
0xFFF6	GPREG3	-	0..15	R/W	General purpose register 3, **no interrupt function!**
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_PIO	3	0..15	R/W	PIO pin change IRQ handler address
0xFFFE	IRQVEC_EXT	4 (lowest)	0..15	R/W	External interrupt line IRQ handler address

Table 22: Interrupt handler configuration and general purpose registers 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_PIO   = (uint16_t)(&invalid_irq);
IRQVEC_EXT   = (uint16_t)(&invalid_irq);
```

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 script(s)):

1. The application start-up code[2] (*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`).

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 included 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 like the stack and the head (actual size depends on the application program).

If you have several source files, that shall be included into your project, you should create a single main.c file and include all other sources using the "include" pre-processor instruction.

2 The start-up code calls the application's main function. If the application returns, program control returns to the crt0.asm code, which deactivates the global interrupt flag and sets the CPU into permanent power down mode (sleep).

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:

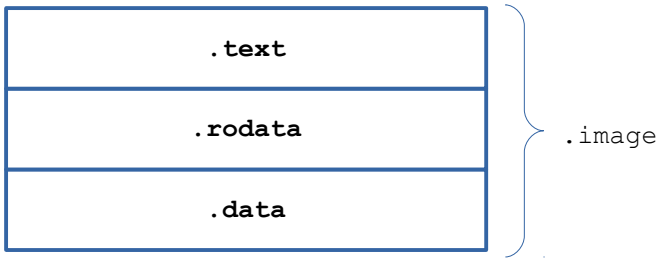

Figure 8: Construction of the final program image

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. 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 → reset it
- Clear the control register of the serial interface (UART/SPI) → reset it
- Clear the control register of the parallel IO port → reset it
- Set all PIO output ports to zero
- Clear the control register of the timer → reset it
- 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 3x2 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. The NEO430 Bootloader

The bootloader requires at least the TIMER and the USART units to be included into the design!

The included bootloader (only 2kB!) 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 it and the system can directly boot it atsystem start 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 PIO 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 V20161130
by Stephan Nolting

HWV: 0x0100
ROM: 0x1000
RAM: 0x0800

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 (`CPUID0`)
- **ROM**: Size of internal IMEM in bytes (`CPUID3`)
- **RAM**: Size of internal DMEM in bytes (`CPUID5`)

Now you have 8 seconds to press any key. Otherwise, the bootloader starts the auto boot sequence and will try to load a program image from an external SPI EEPROM. This EEPROM must be connected to the SPI port using the special SPI.CS0 as chip select signal. When the transfer process is completed, the loaded program is automatically started. If the loading attempt fails, the bootloader freezes and the status LED is permanently activated.

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

```
NEO430 Bootloader V20161130
by Stephan Nolting

HWV: 0x0100
ROM: 0x1000
RAM: 0x0800

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 new
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 (*.bin file) via UART

A new program is uploaded to the NEO430 by using the upload function. The compile script of this project generates a compatible binary executable (*.bin format), which must be transmitted by your terminal program without using any kind of protocol – just raw data mode. When the image is completely uploaded, it resides in the IMEM and you can launch it using the "`Start app`" option. If you want to take at look at the data in IMEM and DMEM, 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 to EEPROM`"). The bootloader can copy an image from the EEPROM at start up and automatically launch it. Of course, you can also load it manually using the "`Load from EEPROM`" option.

3.2.1. Auto Boot Sequence

When you reset the NEO430 processor, the bootloader waits 8 seconds for a user input before it starts the automatic boot sequence. This sequence tries to fetch a valid boot image from the external SPI EERPOM, which must be connected to the SPI chip select port #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 in it, the bootloader freezes and the status LED is permanently activated.

3.2.2. Error Codes

Sometimes things go wrong. In this case, an error message will show up. Most of the errors are “resume-able”, but in some cases a hardware reset of the processor is required.

- **ErrX**: If you try to load 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.
- **ErrS**: 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.
- **ErrC**: 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 build for an older version of the bootloader.
- **ErrE**: 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 (CS0!).
- **ErrM**: 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_rom_c` configuration flag in the core’s package file to ‘false’ to implement the IMEM as writable RAM.

If an error occurs during the auto boot sequence, the according error code is send via the UART interface. Additionally, the status LED stops flashing and remains active.

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 more. Chose the one according to you operating system environment (Windows or Linus/Cygwin).

Check the following check list to make sure, that you have not forgotten anything crucial.

Check List

- ✔ Have you added all HDL files from the `/rtl/core` folder to your project?
- ✔ Have you selected the `neo430_top.vhd` file as top entity?
- ✔ Have you assigned at least the signals for the status LED, reset signal and the UART communication lines?
- ✔ Does the reset button has the correct polarity (active low)?
- ✔ Have you specified the correct clock frequency in the `neo430_package.vhd` package file?
- ✔ Have you configured the IMEM an DMEM sizes there as well?
- ✔ Have you configured the same memory sizes in the appl linker script `neo430_linker_script.x`?
- ✔ Have you disabled all peripheral modules in the package files, that you do not want to implement?
- ✔ Do you want to directly execute you application from the IMEM or do you want to use the bootloader as well and execute that at system start?
- ✔ Have you installed your compiler correctly and have you configured the path to its binaries in the compilation script(s)? Did you select 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?

Any Problems?

If you have any problems setting up your first project, feel free to contact me ;)

4.1. General Hardware Setup

Follow these steps to prepare the FPGA hardware of your NEO430 project. A little note: The order of these step might be a little different for your specific EDA tool.

1. Create a new project with your FPGA EDA tool of choice (Xilinx ISE / Vivado, Altera Quartus, …).

2. Add all VHDL files from the project's `rtl/core` folder to your project.

3. Select the **`neo430_top.vhd`** file as top entity.

4. Assign the IO pins of the processor system to the corresponding pins of your FPGA board. The most important signals are the clock (**`clk_i`**) and reset (**`rst_i`**) signals and the UART communications lines (**`uart_txd_o, uart_rxd_i`**). Connect the reset line to a button of your FPGA board and check whether it is low-active or high-active – the reset signal of he processor is **low-active**. If possible, connected bit #0 of the parallel output port (**`pio_o`**) to a high-active LED (invert the signal if your LEDs are low-active). All remaining signals (other parallel IO lines, SPI signals, ...) are not further important for a minimalist setup, so you can assign them to "*any uncritical*" FPGA pins. If you want to use the example programs from the tutorial below, you should assign at least the lowest 8 bit (bits 7 downto 0) of the output port to high-active LEDs. The image below illustrates this minimal hardware configuration.

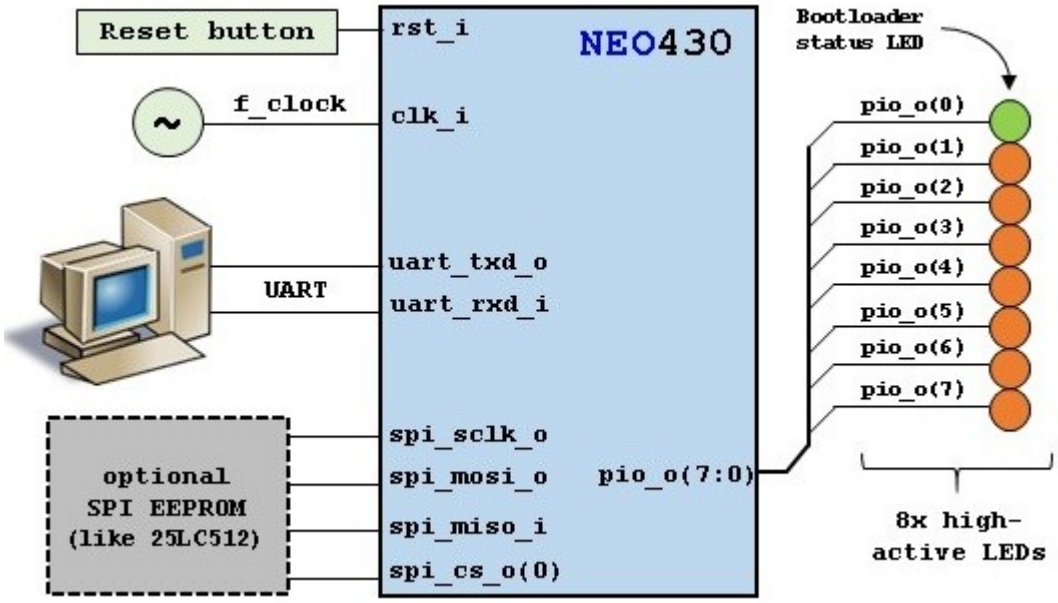

Figure 9: Minimal external hardware configuration of the NEO430 FPGA implementation (neo430_top.vhd)

5. Open the **`neo430_package.vhd`** file from the project's `rtl/core` folder. At the beginning of the file, you will find the "System Configuration" section. There is are some things to edit there:

   ➔ At first, you need to tell the hardware the clock speed of the clock input signal. This step is crucial, since the bootloader uses this declaration to compute the Baud value for the UART connection. Specify the clock speed in Hz:

```
constant clock_speed_c : natural := 100000000; -- main clock in Hz
```

➔ Optionally, you can change the size of the internal IMEM and DMEM. But for now, let's use the default configuration: IMEM = 4kB, DMEM = 2kB

```
constant imem_size_c : natural := 4*1024; -- internal IMEM size in bytes, max 32kB
constant dmem_size_c : natural := 2*1024; -- internal DMEM size in bytes, max 28kB
```

➔ There is one thing left: You can specify which optional components shall be included into the synthesis process. Let's keep things simple for now and use the default configuration:

```
...
constant synth_dadd_c : boolean := true; -- implement DADD instruction?
constant wb32_use_c   : boolean := true; -- implement WB32 unit?
constant wdt_use_c    : boolean := true; -- implement WBT?
constant pio_use_c    : boolean := true; -- implement parallel IO unit?
constant timer_use_c  : boolean := true; -- implement timer?
constant usart_use_c  : boolean := true; -- implement USART?
...
```

6. Perform the project HDL compilation (synthesis, mapping, …, bitstream generation).
7. Download the generated bitstream into your FPGA ("program" it).
8. Press the reset button (just to make sure everything is sync).
9. Done! If you have assigned the status LED, it should be flashing now.

4.2. General Software Setup

OK, 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 **TI msp430-gcc compiler tool chain**. It can be downloaded without registration (select the "compiler only" package in the download list) from http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPGCC/latest/index_FDS.html. Extract/install all files into a folder somewhere in your file system. For instance, you can name the folder "msp430-gg". Remember where you have installed msp430-gcc, since this will be important for the setup of compilation scripts in the next chapter(s).

2. You need to tell the linker the size of the internal RAM of the NEO430 (you defined that during the previous tutorial via configuring the ram_size_c constant). 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 and the RAM memory section according to the previously configured DMEM size (hexadecimal representation).

```
MEMORY
{
  rom  (rx) : ORIGIN = 0x0000, LENGTH = 0x1000
  ram (rwx) : ORIGIN = 0x8000, LENGTH = 0x0800
}
```

3. Well, this is all you need to do for the general software setup. The next step is to hit the actual compilation flow for your specific build environment.

4.3. Application Program Compilation using Windows CMD Batch File

Use this guide if you want to compile applications by using Windows. The compilation script (available in each example project folder) is "`make.bat`".

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 (you did that during the general software setup). In the example below, the (**msp430-gcc**) compiler sources are located in `C:\msp430-gcc` and the binaries in the sub-folder `C:\msp430-gcc\bin`.

```
@REM Path to compiler binaries:
@set BIN_PATH=C:\msp430-gcc\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. 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
```

8. Done! Several files are generated and moved to the current working folder (`blink_led`):

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

9. Additionally, the according IMEM VHDL memory initialization image of this project is directly installed into the `neo430_application_image.vhd` file in the rtl folder. If you resynthesize your design, the image is ready to boot from the IMEM.

4.4. Application Program Compilation using Cygwin/Linux Makefile

Use this guide if you want to compile applications by using Cygwin (on Windows) or directly Linux. The compilation script (available in each example project folder) is "`Makefile`" (no file type suffix).

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 (you did that during the general software setup). In the example below, the (**msp430-gcc**) compiler sources are located in `C:/msp430-gcc` and the binaries in the sub-folder `C:/msp430-gcc/bin`.

```
# Path to compiler binaries:
BIN_PATH = C:/msp430-gcc/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. 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
    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. Finally, the actual size of the created executable is shown.

7. The default configuration of the makefile passes a relative path from the `sw/common` folder to the current example folder and the default main application source file (= main.c) to the common makefile in the `sw/common` folder. If you wish to specify a different file than "main.c" as main source file, pass this file name as MAIN argument:

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

8. Done! Several files are generated and copied to the current working folder (`blink_led`):

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

9. Additionally, the according IMEM VHDL memory initialization image of this project is directly installed into the `neo430_application_image.vhd` file in the rtl folder. If you resynthesize your design, the image is ready to boot from the IMEM.

10. If you want to clean-up your current workspace, you can delete all the previously generated files by making a clean:

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

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

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

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

2. Start a terminal program. I recommend the following two (when using Windows):

   - **Tera Term**: https://ttssh2.osdn.jp/index.html.en → **this program is used in this tutorial**
   - **HTerm**: https://www.der-hammer.info/terminal/

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)

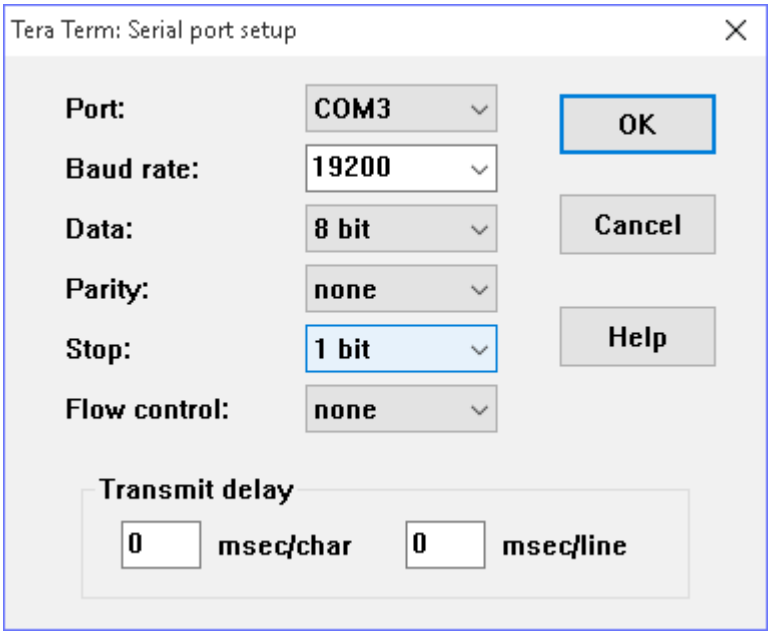

Figure 10: Configuration of TeraTerm

In the next step, you will need to send characters via the terminal to the NEO430. These characters need to transferred in "raw" mode, so no ENTER (carriage return or line feed or something like that) characters must be send afterward.

4. Press the NEO430's reset button (you have assigned one, right?). The status LED starts blinking and the bootloader intro appears in your console. Hurry up and press any key (hit space! :D) to abort the automatic boot sequence and to start the actual bootloader user interface console:

```
NEO430 Bootloader V20161130
by Stephan Nolting

HWV: 0x0100
ROM: 0x1000
RAM: 0x0800

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 new
CMD:>
```

5. Execute the “Upload” command by typing **u**. Now, the bootloader is waiting for a binary executable to be send:

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

6. 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.

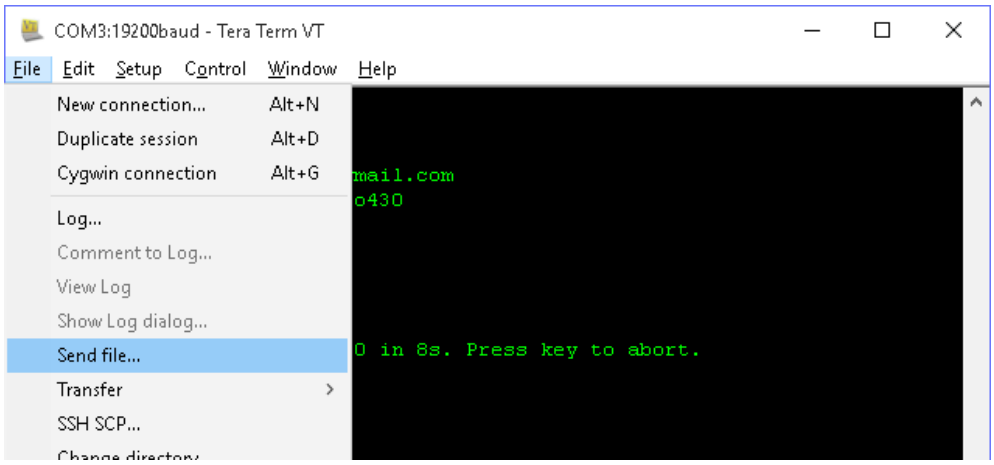

Figure 11: Sending a file using Tera Term

7. 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 (see image below).

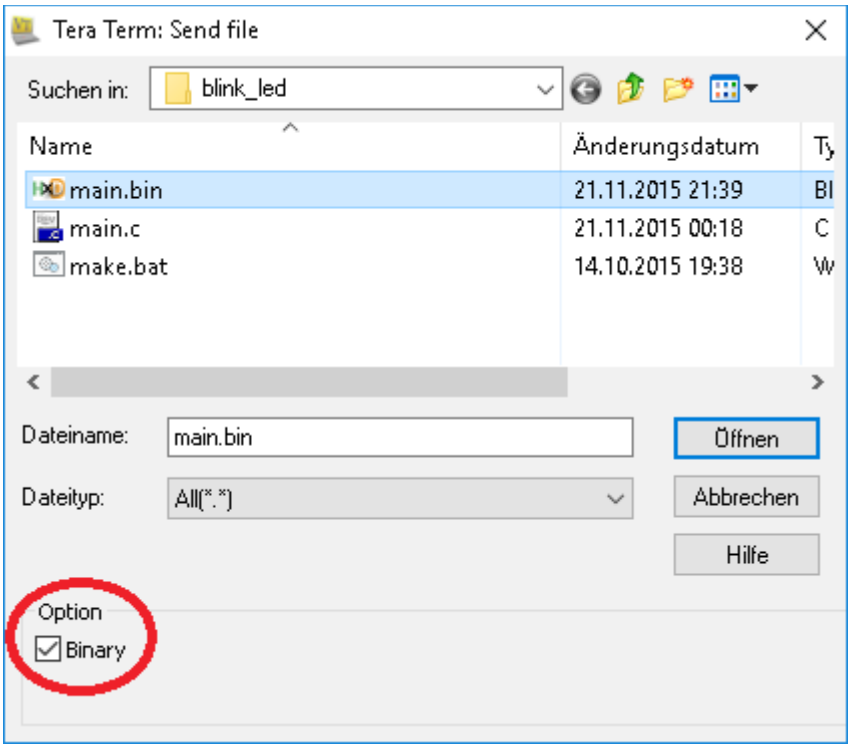

Figure 12: Transfer executable in binary mode (German version of TeraTerm)

8. If everything went fine during the transmission, `OK` will appear in your terminal:

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

9. The program image now resides in the internal IMEM of your NEO430. Optionally, you can store it to an attached SPI EEPROM (if available). To execute the program now, launch 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 output port (you have assigned them as well, right?).

```
CMD:> s
Booting...

Blinking LED demo program
```

10. Congratulations! Now you are prepared to start your own project! ;)

4.6. Programming an External SPI Boot EEPROM

If you want the NEO430 to automatically fetch and execute an application image at start-up, you can store it to an external SPI EEPROM. The advantage of the external EEPROM is the possibility to re-program it at any time just by executing some bootloader commands. So no FPGA bitstream recompilation is required at all. Of course, the external EEPROM adds some additional electrical components to your system.

You need an EEPROM, that is compatible to Microchip ® SPI EEPROM (like 25LC512), with 16-bit addresses and a 32-bit wide SPI transfer frame. The EEPROM has to be at least as big as the internal RAM. Make sure to connect the EEPROM's chip select signal to the chip select #0 signal (`spi_cs0_o(0)`) of the NEO430's SPI port.

This tutorial explains how to program the external SPI EEPROM assuming it is already connected properly to the NEO430's SPI port.

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 for now to prevent changes in the original RAM-residing 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, depending on the size of the NEO430's IMEM.

```
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.

4.7. Setup of a New Application Project

Done with all those example programs? Then it is time to start your own application project. 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. Just copy the complete folder and delete all unnecessary files. You need to keep the Windows compilation batch file **`make.bat`** or the Linux/Cygwin **`makefile`**, depending on your operating system. If your project's application folder is also located in the `sw/example` folder, everything is fine and the compilation scripts will work properly.

When your new project folder is located somewhere else in your file system, you need to adapt the compilation scripts. 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 the `common` folder of the NEO430 software folder. Additionally, modify the `COMMON_PATH` variable to represent the relative part from `common` folder to the project's folder.

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

4.8. Simulating the Processor

Before you do an actual FPGA implementation in case you want see what's going on, you can do a simulation of the processor system. For this purpose, I have implemented a simple testbench (`neo430_tb.vhd`, located in the project's `sim` folder). This testbench instantiates the top entity of the processor system (`neo430_top`) and also includes a serial UART receiver unit, connected to the processor's `uart_txd_o` pin.

When simulating this testbench, the receiver unit outputs all chars send by the processor to the simulator's console. Also, the output is printed to a simple txt file (→ `uart_rx_dump.txt`), which is generated in the simulator project home folder.

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 simulation 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 generate 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 a reduced code size. If you want to get more performance (with an increased code size) you can change the compilation effort / optimization goal (`-O1, -O2, -O3, -Os`).

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.

4.10. Rebuilding the Bootloader

If you want to customize the internal bootloader, you need to re-compile and re-install it. This tutorial will show you how to do this.

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

Follow the upcoming steps to install the modified bootloader code 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 the msp430-gcc binaries folder:

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

- **Windows**: Now execute a "make". This will compile the bootloader's main function and will also install the bootloader ROM VHDL memory initialization into the `neo430_bootloader_image.vhd` file in the project's `rtl` folder.

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

- **Linux/Cygwin**: Open the `makefile` file and edit the path to the msp430-gcc binaries folder:

```
# Path to compiler binaries:
BIN_PATH = C:/msp430-gcc/bin
```

- **Linux/Cygwin**: Now execute a "make". This will compile the bootloader's main function and will also install the bootloader ROM VHDL memory initialization into the `neo430_bootloader_image.vhd` file in the project's `rtl` 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 Fix in IMEM)

The purpose of the bootloaders is to load your application code at any time via UART of from an external SPI EEPROM into the internal IMEM. This provides a lot of flexibility, especially during development. But when you have completed your software, the bootloader is no longer necessary. Thus, you can disable it to save hardware resources and to directly boot your application 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. Open the `neo430_package.vhd` file and and set the bootloader use parameter to ”false”:

```
constant bootld_use_c : boolean := false; -- implement and use bootloader?
```

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.

4. The IMEM could be still modified by setting the R flag in the CPU’s status register allowing write accesses. To prevent this and to implement the IMEM as true ROM, deactivate this feature by assigning “true” to the IMEM ROM paramter:

```
constant imem_rom_c : boolean := true; -- implement IMEM as true ROM?
```

5. And we are done now! Perform a synthesis and upload your new bitstream. You application code resides now unchangeable in the processor’s IMEM and is directly executed.

5. Change Log

Date	Modification
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 codefrom compilation files 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 (yet), so I have removed that unit completely 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