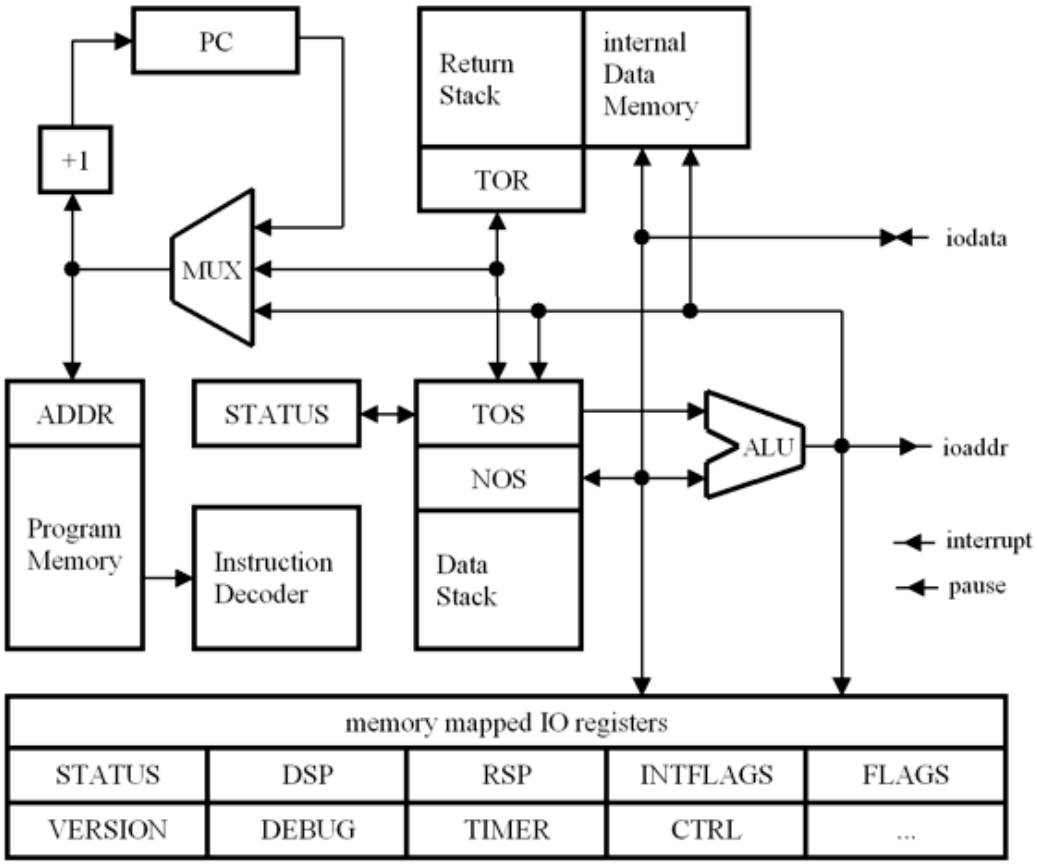
1 Introduction....2
1.1 Design Philosophy....2
2 Design Flow....3
3 VHDL Code Structure....4
4 uBus....6
5 Booting....6
5.1 Boot Loader....6
6 Exceptions....7
6.1 Interrupt....7
6.2 Pause....8
7 Adding Registers....10
8 Adding Instructions....11
8.1 Semantic Possibilities....11
9 VHDL Coding Standard....13
9.1 Readability....13
9.2 Understandability....16
9.3 Portability....18



1 Introduction

µCore is a high-performance, scalable, deterministic, dual stack Harvard computing engine for embedded control that fits into FPGAs. Development started in 2000 out of frustration about silicon manufacturers' reluctance to openly discuss processor bugs and the need to program around rather then fixing them. In addition, with the advent of the XC4000 series, FPGAs seemed to be large enough for an entire processor.

Prior to 2000 I had been the architect of two Forth ASICs: At first, I took Charles Moores[1] NC4000 as a model and enhanced it to become the FRP1600 with an external Motorola bus and vectored interrupts, which never made it beyond second silicon and a remaining interrupt bug. A fresh approach was the realization of the "Fieldbus Processor IX1" in the 1990s that sold in industrial automation. The IX1 was already a Harvard Architecture, and its return stack was located in the data memory area.

Using an FPGA, a choice can be made as to whether a needed functionality should be implemented in hardware or software. The least complex, most energy efficient solution can be realized while working on a specific application. µCore will make you a "core aided programmer" giving you full control of a system.

1.1 Design Philosophy

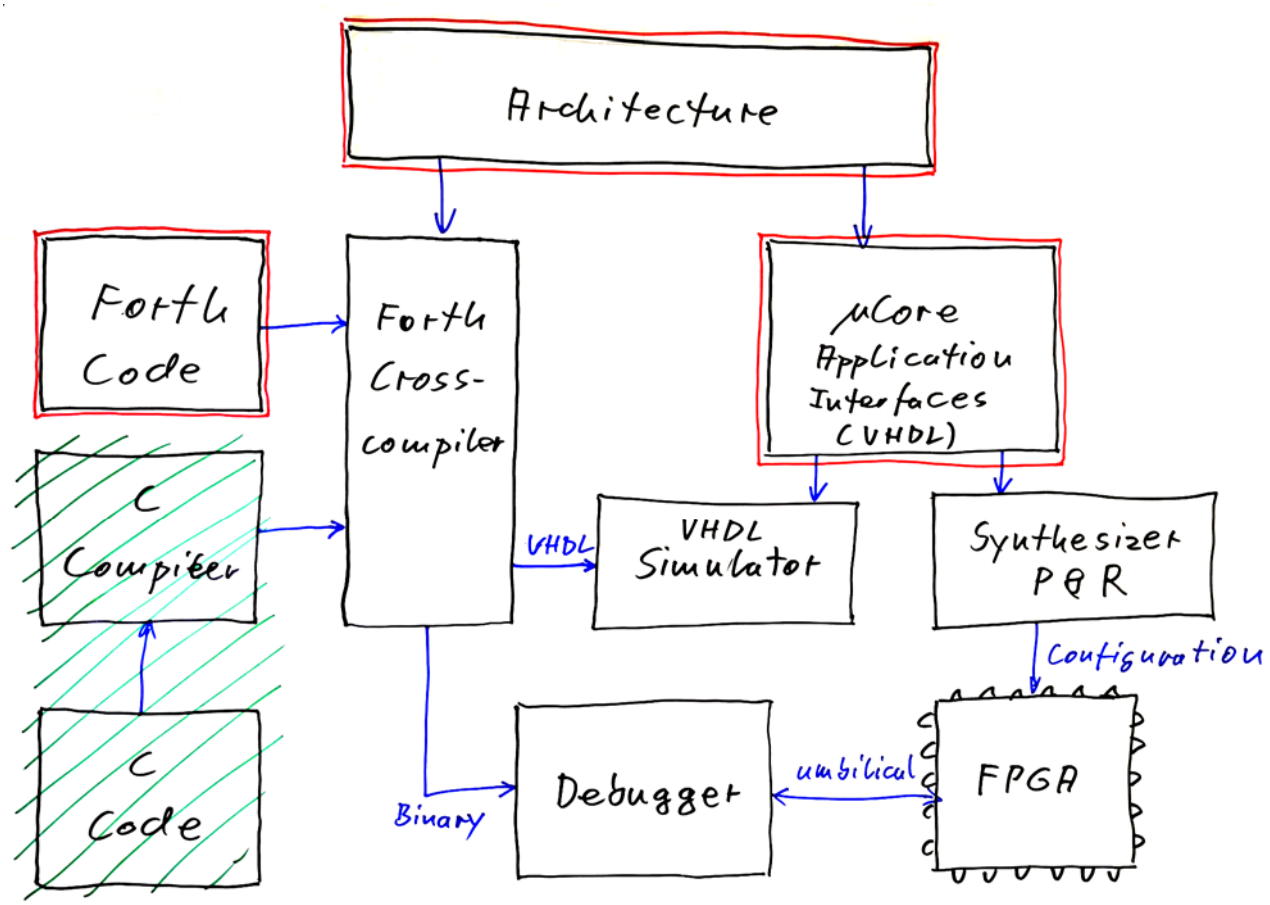
µCore's top priorities are

- simplicity
- portability
- scalability
- extensibility
- low interrupt latency.

Its design approach has been different from most other processors: Its "assembler" was first (Forth), and it realizes 47 core, 24 extended and 5 floating point instructions. Whereas most other processors attempt to give you the utmost in instruction diversity from a minimum of hardware, you will find some very specialised instructions of high semantic content in µCore, because 50 years of Forth experience have proven that they are needed. Nevertheless, most instructions execute in one cycle with the exception of data memory reads, which need two. And µCore is interruptible between any two instructions.

[1] Charles "Chuck" Moore is the inventor of the Forth programming language, which he developed in the 1960s and early 1970s while working on real time control applications using mini computers.

2 Design Flow



Both the hardware and the software aspect of µCore are based on the **Architecture** definition in **architecture_pkg.vhd**. It is written in such a way that both the **VHDL Simulator**, the **Synthesizer**, as well as the µForth **cross compiler** are able to interpret it. Therefore, the cross compiler will always be in-sync with the hardware.

The cross compiler is able to generate either binary code to feed the **Debugger**, or it may generate an ASCII file to feed the **VHDL Simulator** for program memory initialization.

The **Debugger** connects to the **FPGA** via a 2-wire UART allowing interactive control of the target system.

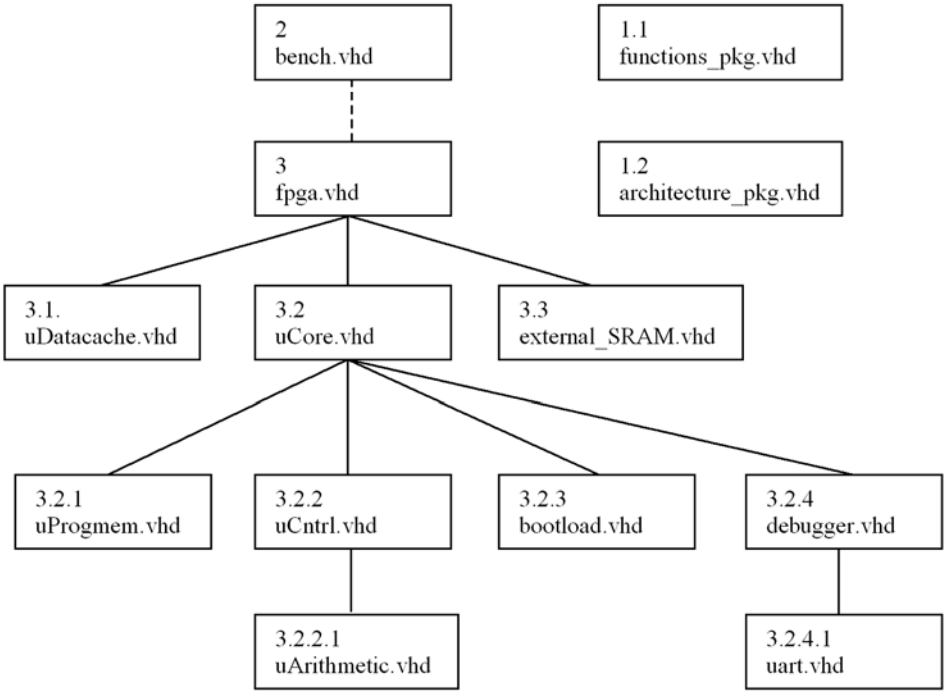
A prototype C compiler has been realized based on LCC generating µForth source code. It does not only have a µCore back end, but also an advanced algorithm for stack allocation instead of register allocation.

µCore is written in VHDL-93 in order to synthesis for legacy FPGAs.

ModelSim has been used for simulation, and Synplify has been used for synthesis.

µCore has been ported to these FPGA families: Xilinx (XC2S), Lattice (XP2), Altera (EP2), and Actel/Microsemi (A3PE).

3 VHDL Code Structure



1. Packages

1.1 functions_pkg.vhd

Functions and components which are used throughout µCore.
It always has to be compiled first, because all subsequent files use it. Constant `async_reset` can be set either true or false depending whether reset should be asynchronous or synchronous throughout the design.

1.2 architecture_pkg.vhd | architecture_pkg_sim.vhd

Defining all µCore parameters and records, as well as the opcode values. This file is also used by the cross compiler and therefore, it has been written in such a way that both the VHDL compiler as well as the cross compiler only see specific sections of the code. There are two versions, one for synthesis, the other one for simulation with slight differences.

2 bench.vhd

General simulation test bench for µCore with RS232 umbilical interface. Immediately following the ARCHITECTURE declaration, one out of a set of constants can be set to '1' in order to test specific aspects of **debugger.vhd**. When all of the constants are set to '0', **bootload_sim.vhd** will be executed. The program memory itself will be initialized at the start of simulation when `MEM_FILE := "../software/program.mem"` in **architecture_pkg_sim.vhd**. `program.mem is` generated by the cross compiler using `'MEM-file program.mem'` when cross compilation has finished.

3 fpga.vhd

Top level µCore entity. This file should be edited for technology specific additions like e.g. pin assignments, and it is the source of `uBus`.
Negative logic I/O (GND = true) will always be handled on this level so that the rest of the design strictly uses positive logic (VCC = true).

`CTRL_REG` as well as `uBus` are defined on this level. All signals that are to be included in `uBus` must be brought up to **fpga.vhd** from its generating entity. An external data memory interface `external_SRAM` is instantiated here as well when present.

3.1 uDatacache.vhd

Definition of the internal data memory. Here FPGA specific dual port memory IP can be included if the inferred blockRAM does not meet specifications. If dma capability is not needed, a single port memory suffices.

3.2 uCore.vhd

The µCore processor kernel. It interconnects `microcontrol`, the `boot_loader`, and `program memory` using blockRAMs, and the `debugger`.

3.2.1 uProgmem.vhd

Definition of the program memory. Here FPGA specific single port memory IP can be included. Program memory will always use internal blockRAMs.

3.2.2 uCntrl.vhd

The µCore engine and instruction decoder including interrupt and pause processing. Here all internal registers are defined: `tos`, `nos`, `tor`, `status`, `dsp`, `rsp`, and the `pc`. The asynchronous instruction decoder defaults to `noop`.

3.2.2.1 uArithmetic.vhd

Application specific adder IP in case the inferred one is too slow.
And the definition of the hardware multiplier if constant **`with_mult`** is true in **architecture_pkg.vhd**. An FPGA specific multiplier IP can be included if no hardware multiplier is available. When **`with_mult`** is false, µCore will use a multiply step instruction that needs data_width+2 cycles to execute.

3.2.3 bootload.vhd | bootload_sim.vhd

Program memory ROM implemented using a case statement. Its content is generated by the µForth cross compiler. It will be statically synthesized into the design as cold boot loader. There are two versions, one for synthesis, the other one for simulation. There are slight differences and they have been compiled from **bootload.fs** resp. **bootload_sim.fs**.

3.2.4 debugger.vhd

The debugger connects to the host via an RS232 interface and allows interactive control while µCore is running. The debugger does not use the processor itself, it is a specific set of state machines. When it accesses µCore's data memory during up- or download, it does so by cycle stealing.

3.2.4.1 uart.vhd

Serial UART - 8N2 format with receive fifo. `umbilical_rate` defines the baud rate in **architecture_pkg.vhd**. You have to make sure that the intended baud rate can be generated "precise enough" from the system clock's `clk_frequency`.

3.3 external_SRAM.vhd

Connecting external SRAM memories of configurable data word width. µCore will always access `data_width` number of bits, which will take multiple SRAM accesses when data_width > SRAM data width. For slow SRAMs, each access may take several clk cycles. This can be configured using the `ram_...` constants in **architecture_pkg.vhd**.
`data_addr_width` has to be adjusted to take the address space for the external memory into account. The external memory logic will only be generated when `data_addr_width > cache_addr_width`. See: WITH_EXTMEM.

4 uBus

uBus is a resource that is centrally defined in **fpga.vhd**. From there on, it is fed into almost every entity of μCore's design as input signal.

The record is defined as uBus_port in **architecture_pkg.vhd**:

```
TYPE  uBus_port  IS RECORD
   reset       : STD_LOGIC; -- synchronous, positive logic reset signal
   clk         : STD_LOGIC; -- clock signal
   clk_en      : STD_LOGIC; -- enable at the end of a uCore cycle
   chain       : STD_LOGIC; -- true when executing multicycle instructions
   pause       : STD_LOGIC; -- pause exception
   delay       : STD_LOGIC; -- extend uCore's cycle to wait for slow peripherals
   tick        : STD_LOGIC; -- produces apulse every "ticks_per_ms"
   sources     : data_sources; -- array of register outputs
   reg_en      : STD_LOGIC; -- enable signal for register access
   mem_en      : STD_LOGIC; -- enable signal for dcache and return stack
   ext_en      : STD_LOGIC; -- enable signal for external memory and IO
   write       : STD_LOGIC; -- 1 => write, 0 => read
   addr        : data_addr; -- I/O address
   wdata       : data_bus;  -- data to registers and memory
   rdata       : data_bus;  -- data from memory
END RECORD;
```

Please note that all of μCore's uBus signals have to be brought up to **fpga.vhd**!

5 Booting

Given μCore's hardware architecture, this is simple:

The reset signal initializes the pc to 0, instruction to noop, and chain to '1'. Therefore, μCore will execute a noop and fetch the next instruction from address zero, which happens to be the location of the reset trap vector.

5.1 Boot Loader

After FPGA configuration, internal flip-flop warmboot will be set to '0', and the program **bootload.vhd** will be executed. It may e.g. copy the image of the application program from an external flashROM into the program memory blockRAM. **bootload.vhd** is generated by cross compiling **bootload.fs** (don't forget to do this before synthesis!).

Once program memory initialization has finished, an absolute branch to zero (the reset vector location) will set warmboot to '1' and hence instructions will be fetched from the program memory proper.

Depending on the length of the boot loader, boot_addr_width has to be set appropriately in **architecture_pkg.vhd**. The boot loader will be statically synthesized into the FPGA as a ROM.

6 Exceptions

6.1 Interrupt

An event did happen that was NOT expected by the software.

An interrupt condition exists as long as any flag in `INT_REG` is set whose corresponding bit had been set in the interrupt enable register using `int-enable`. Interrupt processing will start as soon as interrupts are globally enabled using `ei`, and the processor is not already executing an interrupt (`s_iis` status bit not set). Then `STATUS_REG` is pushed on the data stack, the `s_iis` bit is set, and a call to the interrupt trap vector is made.

At the beginning of interrupt processing `intflags` will read the interrupt flags. Each interrupt source has an associated flag. They are defined after `INT_REG` in **architecture_pkg.vhd**. Writing a bitmask to `INT_REG` will activate (`int-enable`) or deactivate (`int-disable`) specific interrupts. Interrupts are static and therefore, it is the responsibility of each specific interrupt server to reset "its" interrupt signal.

Constant `interrupts` in **architecture_pkg.vhd** defines the number of interrupt sources. If set to 0, the interrupt logic and `INT_REG` will not be instantiated at all. The interrupt flags are located in the least significant bits of `INT_REG` and `FLAG_REG` respectively[2]. When adding an interrupt source's flag, constants `interrupts` and `status_width` have to be adjusted accordingly.

6.1.1 Example from uCore

In **architecture_pkg.vhd** an external interrupt `i_ext` is defined. External signal `int_n` is inverted, synchronized, and connected to `flags(i_ext)` in **fpga.vhd** using

```
synch_interrupt: synchronize_n PORT MAP(clk, int_n, flags(i_ext));
```

Let's assume that somewhere in the µForth code we defined interrupt server `ext_int_server`. And the phrase '`#i_ext int-enable ei`' has been added to the system initialization code.

The global interrupt server is usually defined in the top level **load/sim.fs** file immediately prior to defining the traps:

```
: interrupts  ( -- )  intflags
   dup #i_ext = IF  ext_int_server  THEN
   <additional specific interrupt servers in priority order>
   drop
;
```

Please note that interrupt priority is defined by the sequence, in which interrupt flags are probed.

Finally, we can define the interrupt trap vector handler:

```
#isr TRAP: isr ( -- ) interrupts IRET ;
```

Don't forget: At first, external interrupts have to be synchronized or else you might run into metastability troubles! To put it a different way: Whenever experiencing strange system stalls, it is a good idea to look for unsynchronized external signals.

[2] When reading `FLAG_REG` (`flags`) instead of `INT_REG` (`intflags`), the interrupt signals' actual values will be shown irrespective of their enable status.

6.2 Pause

An event did NOT happen that was expected by the software.

A pause is triggered by reading or writing a peripheral register that is not "ready" yet as e.g. reading a UART's data register, which did not yet receive a character, or writing to a peripheral, which is still busy serving the previous request.

Each peripheral itself "knows" whether it is ready or not. Therefore, it can generate a specific pause signal, which eventually feeds `uBus.pause` in **fpga.vhd**. At that point, all the specific pause signals are ored together.

When a pause is raised during an instruction execution cycle, the instruction currently executed is aborted, i.e. µCore's internal registers will not be updated. Instead, `op_PAUSE` is inserted in the instruction stream for forced execution in the next cycle, which will make a call to the Pause Service Routine (`psr`) trap.

When the code at `psr` just does an `EXIT`, the processor will return to the aborted instruction in the following cycle, stalled in this tight loop until eventually the condition has vanished, which caused the pause signal to be raised.

In the case of a multitasking system, the code at `psr` will execute the multitasker's `pause` routine. This will call the scheduler and give other tasks a chance to run. Eventually, the paused task will be activated again to re-execute the aborted instruction.

Given this mechanism, mutual exclusion of shared resources can be realized in hardware without the need for software semaphors.

6.2.1 Example from uCore

Lets look at `sema_proc` and `flags_pause` in **fpga.vhd**.

Flag bit `f_sema` of the `FLAG_REG` is used as software semaphor. Macros `pass` and `release` may be applied to it.

`#f_sema release` will unconditionally set `f_sema` to zero.

`#f_sema pass` will store a one into it. When it was one already, `flags_pause` will be raised until an interrupt service routine or another task stores a zero into it.

6.2.2 Hardware Semaphors

For the sake of discussion, let's assume that an AD-converter with an eight channel input multiplexer and a serial interface is used (e.g. ADC128S102). An interface register `AD_REG` is used to set the multiplexer and start conversion when storing a channel number into it. When conversion has finished, `AD_REG` can be read to fetch the sample. Thereafter, the AD-converter will accept another conversion request without raising a pause.

To implement this scenario, we have to

1. Design the AD-converter interface to produce an `AD_busy` signal. In the case of the ADC128S102 this is identical to its chip select signal.
2. Declare signal `AD_sema` for use as semaphor flip-flop.
3. Define a bit for it in the `FLAG_REG`, e.g. `f_adc`.
4. Assign `flags(f_adc) <= AD_sema;` This makes the semaphor readable by software.
5. Instantiate a semaphor entity defined in **architecture_pkg.vhd** as follows:
   ```
   sema_AD: semaphor PORT MAP(uBus, AD_REG, f_adc, AD_sema);
   ```
6. Define `AD_pause` to be or-included into `uBus.pause`:
   ```
   AD_pause <= '1' WHEN (AD_sema = '1' AND uReg_write(uBus, AD_REG))
                     OR (AD_busy = '1' AND uReg_read(uBus, AD_REG))
               ELSE '0';
   ```

As a consequence, a pause will be raised when another task tries to use the AD-converter while the result of the previous conversion has not been read yet (mutual exclusion). When the task that initiated the conversion tries to read the sample, a pause will be raised until conversion has finished. A task can not "steal" the conversion result from another task as long as its read request will always be done after having started the conversion with a write to AD_REG.

Given this hardware setup, the AD128S102 can be used in a multitasking environment in the following manner, pushing the sample on the data stack:

```
... <channel> AD_REG !   AD_REG @ ...
```

No need to probe semaphors and ready-flags any more on the software level!

Integrating the semaphor into `FLAG_REG` allows to

1. Read the semaphor using `#f_adc flag?` in order to read the interface state for e.g. the detection of a potential deadlock.
2. Force the semaphor to zero by executing `#f_adc release` if needed.

7 Adding Registers

All register outputs are included in the `uBus.sources` array and they have unique index numbers, which are defined in **architecture_pkg.vhd**. All register addresses are located at the upper end of the address space. For the sake of readability and independence from `data_width`, they are expressed as negative indices, because in a 16 bit system, -1 will be treated as address $FFFF by the fetch and store instructions.

The `uBus.sources` array's upper limit is `max_registers(-1)`, its lower limit is `min_registers`, which has to be adjusted when adding or removing registers.

Let's assume we want to add `NEW_REG` to the existing set of registers.

First of all, `NEW_REG`s index has to be defined as an integer constant, placed immediately ahead of the definition of `min_registers`. In addition, `min_registers` has to be decremented by 1 so as to make room for `NEW_REG` in the array. This concludes the modifications that need to be made in **architecture_pkg.vhd**.

Somewhere in the your code will be the actual definition of the register itself. Let's assume it is defined as `SIGNAL newreg : UNSIGNED(6 DOWNTO 0);`

As a next step, `newreg` has to be brought up to `fpga.vhd` in order to assign `newreg` to `uBus`' sources array:

```
uBus.sources(NEW_REG) <= resize(newreg, data_width);
```

Don't forget to include `newreg` in both the `ENTITY` and the matching `COMPONENT` definitions when bringing it up to **fpga.vhd**!

As a last step, `NEW_REG` has to be integrated into the cross compiler. This is done in **constants.fs** by adding one line at the end of the file:

```
H NEW_REG T Register New-reg
```

when we want to stick to μCore's naming convention. Any other name would be fine as well in the μForth environment.

This concludes the creation and integration of `NEW_REG`.

Depending on the application, we may want to read or write `newreg` using `New-reg @` or `New-reg !` in μForth. To this end, two functions have been defined in **architecture_pkg.vhd**:

`uReg_read` and `uReg_write`

The following phrase has to be included in the sequential process that controls `newreg` in order to store NOS into `newreg`:

```
IF  uReg_write(uBus, NEW_REG)  THEN
   newreg <= uBus.wdata(newreg'high DOWNTO 0);
END IF;
```

Executing `New-reg @` will push the content of `newreg` on the stack, because it has been assigned to `uBus.sources(NEW_REG)`.

If `something` extra as e.g. resetting a semaphor flag needs to be done when reading, `uReg_read` can be used:

```
IF  uReg_read(uBus, NEW_REG)  THEN  something;  END IF;
```

8 Adding Instructions

Adding a new instruction to μCore is rather simple and consists of three steps:

1. Define the binary code for `op_NEWNAME` as a constant in **architecture_pkg.vhd** and **architecture_pkg_sim.vhd**. The list of opcode constants has an empty line when no opcode has been defined for a specific numerical value.
2. Add a new `WHEN` clause to the instruction decoder's `CASE` statement in **uCntrl.vhd** to define the semantics of `op_NEWNAME`. Please note the default behaviour for opcodes, i.e. when the resp. `WHEN` clause does not exist or has been defined as `NULL`:
   $00 .. $5F : noop
   $60 .. $7F : software trap
3. Define a name for `op_NEWNAME` in **opcodes.fs** to make it known to μForth, e.g.

   ```
   op_NEWNAME  Op: newname   ( n1 -- n2 )   don't
   ```

   The defining word Op: needs two names and a stack comment:

   1. The first name (`newname`) is the name for `op_NEWNAME` in the μForth context.
   2. The stack comment is required or else the compiler will bark.
   3. The second name (`don't`) is a word in the host context, which will be executed when `newname` is executed during target compilation. `don't` means that `newname` can not be executed during target compilation. See `Op: dup` in **opcodes.fs** for an example that can be executed.

8.1 Semantic Possibilities

All that can be done in one μCore clock cycle [3]. That excludes data memory read access, because it needs two uninterruptible cycles. See `r.chain` and `r.inst` in the next table, which form a mechanism for chaining un-interruptible instructions.

Instruction semantics are defined in the long asynchronous process `uCore_control` in **uCntrl.vhd**. At first, default states are defined for all output signals. Followed by a large case statement with a `WHEN` clause for each instruction.

Sequential process `uReg_proc` updates all internal registers and handles `pause`.

[3] A **μCore clock cycle** is defined by the (processed) input clock `clk` and the `clk_en` signal. Constant `cycles` in **architecture_pkg.vhd** specifies the number of `clk` cycles per `clk_en` signal.

These are the possible input and output signals that can be used in `uCore_control`:

Input	Output	Remark
uBus		This includes all registers in `uBus.sources`.
r.tos	r_in.tos	Top of Stack
r.nos	r_in.nos	Next of Stack
ds_rdata	ds_wdata	The stack memory location register `DSP_REG` points to.
r.tor	r_in.tor	Top of Return Stack
r.status	r_in.status	Status register
r.dsp	r_in.dsp	Data Stack Pointer
r.rsp	r_in.rsp	Return Stack Pointer
r.pc	r_in.pc	Program Counter
r.inst	r_in.inst	INSTruction register, which holds the instruction to be executed when r.chain = '1'. When r.chain = '0', `prog_rdata` will be executed.
r.chain	r_in.chain	Set to '1' for an instruction that follows the current instruction immediately without interrupts in between. Use procedure `set_opcode` for instruction chaining. Used for memory reads and read-modify-write instructions.
	progmem	Program memory control signals.
prog_addr	paddr	`paddr` is the Program memory ADDRess of the next instruction and will be latched by blockRAM memory.`Prog_addr` is the address of the current instruction (`prog_rdata`).
	datamem	Data memory control and wdata signals.
mem_rdata		Data memory output at the address that had been latched into blockRAM memory in the preceding cycle. External Memory assumes asynchronous memory and therefore, both read and write can be done in a singel µCore cycle without chaining.

Re-entrant instructions can operate on up to four input and output registers, namely

`TOS`, `NOS`, `ds_rdata/ds_wdata`, and `TOR`,

which are the top items of the stacks. In the present instruction set, all instructions are re-entrant. Non re-entrant instructions can use special purpose registers in addition.

9 VHDL Coding Standard

The purpose of the coding standard is to aid in creating reliable code. To this end it helps when the code is written in such a way that it is easy to read and understand by the initial programmer himself, his colleagues, and by system engineers (as far as possible).

9.1 Readability

Readability deals with the layout of the source code on a display. The code's structure should be easy to recognize. Identifier names should be descriptive and easy to remember.

Place all submodules, which are needed for a functional module as close together as possible. Ideally, one functional module and its submodules can be seen on one editor screen whithout having to scroll.

9.1.1 Character Case

All VHDL keywords are written in `UPPER CASE`.

Identifiers, i.e. names of signals, variables, procedures, and processes are written in lower case.

9.1.2 Spacing and Parenthesis

Always put a space between identifiers, constants and operators.

```
IF  reset = '1'  THEN
```

If in doubt, include a complex expression in parenthesis, even if they may be superfluous according to operator precedence rules.

```
cache_en <= internal AND ( (clk_en AND datamem.write) OR
                           (read_en AND NOT datamem.write)
                         );
```

9.1.3 Indentation

Each nesting level should be indented by at least an additional three spaces from the previous one. Do not use tabs instead of spaces, because different editors will interpret the tabs differently and therefore, your carefully crafted text layout may look funny.

```
syn_select: PROCESS (reset, clk)
BEGIN
   IF  rising_edge(clk)  THEN
      IF  reset = '1'  THEN
         io_select <= '0';
      ELSE
         io_select <= internal;
      END IF;
   END IF;
END PROCESS syn_select;
```

9.1.4 Declarations

In the signal declaration section, always use one line for one signal, variable, or constant declaration. This gives the opportunity to comment each single signal.

Group the signals by process and functional block.

Do not initialize signals or variables when declaring them. Explicitly initialize them in the process that uses them.

9.1.5 Literals

Restrict the use of based literals to bases 2, 8, 10, or 16. In long binary, octal and hexadecimal literals the _ character should be used for better readability. Hexadecimal numbers above 9 should use upper case letters, e.g. **16#9ABC#**.

9.1.6 Integers

When using integers or naturals, always restrict their range to the values actually used.

```
SIGNAL baud_ctr : NATURAL RANGE 0 TO 12;
```

9.1.7 Entities

Put every I/O port or generic declaration on a single line along with a comment about the port/generic if the name is not self-explanatory. Do NOT insert extra lines for comments, because that makes an entity's list of signals difficult to read.

If possible, all output signals should be registered. Asynchronous outputs are a potential source for combinatorial loops, which are always difficult to locate after synthesis.

9.1.8 Instantiating Components

If a component has up to four interface signals, it may be instantiated by just using a list of the binding signals on one line.

```
synch_rxd: synchronize PORT MAP (clk, rxd, rxd_s);
```

If it has more than four signals, it should be instantiated using `entity-name => actual-name`, one line for each signal.

```
data_stack: internal_ram
GENERIC MAP (data_width, r.dsp'length, "rw_check")
PORT MAP (
   clk   => clk,
   en    => core_en,
   we    => ds_wr,
   addr  => ds_addr,
   di    => ds_wdata,
   do    => ds_rdata
);
```

Do keep the order of an entity's interface signal names in component declarations and instantiations. Re-shuffling the signals just constitutes a brain twister, because it breaks the entity's template.

Explicitly use `OPEN` for unconnected output signals.

Use `FOR ... GENERATE` statements for multiple instantiations of the same component if possible.

9.1.9 Architectures

Use meaningful names to describe architectures: "rtl", "behavioural", "structural", "test_bench", etc.

9.1.10 Sequential Processes

Use `rising_edge` or `falling_edge` for clocking. `Rising_edge` should be used consistently in the design. Limit the use of `falling_edge` to e.g. instantiate DDR IO blocks as much as possible if supported by the technology. Because the use of `falling_edge` imposes a

strict requirement on the clock's duty cycle or else static timing analysis will not return meaningful results.

9.1.11 User Defined Types

Do avoid user defined types with the following exceptions:

1. States of an FSM,
2. Arrays,
3. Subtypes for re-occurring data structures. Subtypes remain compatible with the original type inheriting the functionality of the parent type.

9.1.12 Naming Conventions

Identifiers (signals, variables, procedures, and process names) should describe their function. If more than one word is needed to make up a functional name (compound name), the words are concatenated using the _ character. Example: `data_addr_width`.

There are a number of suffixes for identifiers, which further denote the type by convention:

Identifier	Semantics
`<name>`	Signals just have a <name> in lowercase letters. Instead of "camel casing" long names, use the _ character as a separator.
`<name>_n`	Signals with negative logic, i.e. '0' = high level, '1' = low level. Only use in the top-level module.
`<name>_addr`	Address
`<name>_ctr`	Counter
`<name>_cnt`	A count value, which is used to compare against a `_ctr` signal. Most of the time this will be a constant, perhaps computed during compilation.
`<name>_reg`	Register
`<name>_en`	Enable signal
`<name>_a`	Asynchronous signal `<name>`
`<name>_i`	"internal" signal for an entity's OUT signal <name> that can be used as an input in the body of the entity.
`<name>_s`	Synchronized signal `<name>`
`<name>_d`	Signal `<name>` delayed by one clock cycle
`<name>_dd`	Signal `<name>` delayed by two clock cycles
`<name>_proc`	Process `<name>`

9.1.13 File Names

As a rule, one file will define a single VHDL entity and an RTL file name will be identical to its entity name. As an exception, packages may contain several entities.

`<entity>.vhd`	VHDL definition for component `<entity>`
`<entity>_tb.vhd`	Test bench definition for `<entity>`
`<name>_pkg.vhd`	Package definition
`<name>_sim.vhd`	A version of file `<name>.vhd` to be used for simulation

9.2 Understandability

Understandability deals with the program logic itself and how processes are described in VHDL. As a general rule: reduce the amount of implicit context information needed to understand the code.

Write the code in such a way that you do not need to twist your brain unnecessarily in order to understand it!

9.2.1 Logic Polarity

As a strict rule, use positive logic for all signals and variables. I.e. '1' = high level, '0' = low level.

If an FPGA's pin to the outside world requires negative logic, the inversion should be made in the topmost entity that defines the FPGA's external pins.

Example: Let `ram_ce_n` be the FPGA's output pin that drives the ce_n input of an external SRAM. Inside the FPGA, signal `ram_ce` will be defined with the needed functionality and only in the topmost entity the assignment `ram_ce_n <= NOT ram_ce;` will be used to drive the output pin.

9.2.2 Signal Direction

`<name>_in` or `<name>_out` have to be avoided with the exception of pin names, which relate to peripheral chip signals of the same name. If used inside the design, `_in` or `_out` are always ambiguous, because their implicit direction depends on the point of view. A widely used name for memory data input or output is `din` or `dout`. But what does it actually mean?

On the memory entity itself the situation is clear: `din` is the data that is fed into the memory, and `dout` is the data that is fed from the memory into our design.

Inside the design, rather use `wdata` for the data that is written into the memory, and `rdata` for the data that is read from the memory. Therefore, when the memory is instantiated, its parameter list should e.g. read as follows:

```
...
din  <= wdata,
dout <= rdata,
...
```

Similar confusions can arise when using receive and transmit.

9.2.3 Record Types

Records are an elegant way to make the entity interface lists shorter and more comprehensible. As an added benefit, if you need to add a signal to an interface that uses records, you just have to add it to the record definition rather than threading it through a hierarchy of Entity and Component definitions and their instantiations.

Unfortunately, there is a severe limitation: Records of type INOUT are not standardized. They will work on some simulators and synthesizers, but not on others. Therefore, if you have an interface that consists of IN and OUT signals, you can not combine them in one record, but you have to use a separate record for the IN and another one for the OUT signals.

9.2.4 Using an Entity's OUT Signals as Input Arguments

Since microCore limits itself to VHDL-93 for legacy reasons, an entity's OUT signals can not be used as input arguments in an entity's body. Instead, an additional signal with the suffix _i must be declared that serves as IO signal in the entity's body. Eventually, the phrase

```
outname <= outname_i;
```

will set outname on the entity interface.

9.2.5 Factorisation

Unfortunately, this topic is fuzzy. So is the current practice of VHDL programming in that respect. Factorisation in VHDL takes place on two levels: Entities and Processes. Entities are modules in the context of the complete project. Processes are modules in the context of an entity. Getting the factorisation "right" has a major impact on understandability.

The complete project should be broken down into entities in a natural way to reflect the major building blocks of an application. A major building block is a module, which has a succinct interface and closely interrelated internal signals. An entity itself may contain component instantiations if these components either serve a complex specific or a basic (general) function. Parameter lists getting "too long" are an indication for bad factoring.

An entity may consist of several processes rather then promoting every process to an entity. Keeping a flat hierarchy of component instantiations makes it more readable, because otherwise you have to flip back-and-forth between different entities and files in order to understand the code.

A project has one top-level module and several sub-level modules. The top-level module should only contain instantiations of the sub-level modules besides pin assignment (if done on the VHDL level), signal inversion for negative logic pins, definition of tristate pins, and synchronisation of external signals.

Processes should be "self confined" in the same way that entities have succinct parameter lists and closely interrelated internal signals. If you are forced to introduce synchronisation signals in order to synchronise two different processes within the same entity you are probably better off to amalgamate these tow processes into one, even if it becomes longer than a single screen. This is, of course, only possible when these processes use the same clock.

Group processes that are longer than a single screen into commented, functional sections as much as possible.

When implementing complex state machines it is often advantageous to break the state machine up into an asynchronous and a short synchronous process. The asynchronous process is used to compute the next state(s) using variables. The synchronous process just updates the state flip-flops, whose next state is transferred from the asynchronous process via signals.

Write your code in a parameterized style using generics.

9.2.6 System Wide Definitions

Define Constants, Types, Subtypes, and Records that are used throughout the system in a package and one single file **constants_pkg.vhd**.

Define Functions, Procedures, and often re-used Entities in a package and one single file **functions_pkg.vhd**.

If a VHDL phrase re-occurs over and over again, it should be included in the **constants_pkg.vhd** or the **functions_pkg.vhd** package as a procedure or function definition.

9.2.7 Avoiding Latches

For long **IF ... THEN ... ELSE** or **CASE** statements assign default values to signals prior to entering the conditional statements to avoid missing any assignment.

9.3 Portability

9.3.1 Information Hiding

Avoid as much as possible the use of numerical values in the RTL code. Instead, define constants.

9.3.2 Vector Subscripts

Do not use numbers for vector subscripts. Instead, use attributes. That makes the code independent from changes in vector width.

```
shift_reg <= shift_reg(shift_reg'high-1 DOWNTO 0) & rxd;
```

9.3.3 Libraries

Always use `IEEE.std_logic_1164.` If arithmetic has to be done on logic vectors, use `IEEE.NUMERIC_STD`. As a consequence, use UNSIGNED rather than STD_LOGIC_VECTOR.

9.3.4 Logic Block Inference

When possible write the RTL code to infer components such as memory blocks (RAM, ROM), multiplication blocks (DSP blocks), etc. Refer to the synthesizer documentation to write the code to properly infer the desired components.

9.3.5 Technology Specific Instantiations

Sometimes, technology (vendor) specific IP instantiations can not be avoided for the sake of performance or functionality.

Use `IF ... GENERATE` statements controlled by generic values (e.g. from **architecture_pkg.vdh**) in order to separate portable simulation code from technology specific synthesizable or debug only code.

9.3.6 Subdirectory Structure

For the sake of portability w.r.t. different FPGA technologies the following sub-directory structure will be used for VHDL designs:

<project name>

documents	top level directory for all <project> related documents
<technology>	top level directory for technology specific files (place&route) including technology specific constraint files
<synthesis>	top level directory for synthesis and synthesis constraint files
<simulation>	top level directory for simulations
software	top level directory for µForth programs
vhdl	top level directory for µCore and its testbenches

This overrides the directory structures forced on the user when technology specific design environments are used, because those are different for different technologies.

Alas, it is not always easy to separately start the simulator, the synthesizer, and the place&route tools outside of the vendor's design environment. In that case, use the <technology> directory for the vendors software suite.