OpenAVR - introduction

OpenAVR is a simple 8-bit CPU compatible with Atmel AVR architecture. It supports full instruction set of ATTiny48 MCU, however, does not support interrupts. The OpenAVR bundle consists of the following components:

- A sample firmware that repeatedly sets LED register (=PORTB) to 0x55 and 0xAA.
- A VisualHDL project with THDL++ source code. The project has 2 configurations:
  - A simulator-based configuration. Run it then look at **uut\LEDs** signal.
  - An FPGA-targeted configuration. You can build & program it to a Xilinx Spartan3 evaluation board (XC3S700AN FPGA), or use any other board and FPGA by customizing the **synthesize_hardware** statement in **MCU.thp**.

The OpenAVR example demonstrates the use of policy classes to design flexible hardware. The policy classes are used to:

- List the peripherals of the microcontroller and specify register locations
- Define CPU instructions
- Define ALU operations

THDL++

THDL++ is a language that combines VHDL semantics and C++ syntax. Essentially, you can take a VHDL design, replace begin/end with { }, move “architecture” contents inside “entity”, replace VHDL operations like “or” to C operations like “|” and you’ll get a valid THDL++ entity.

However, unlike VHDL, THDL++ allows to use inheritance, “foreach” semantics, enhanced generics and many more features improving user experience. You can read more about THDL++ here:

http://visualhdl.sysprogs.org/tutorial/

Policy classes

A policy class is essentially a collection of constants, functions and typedefs. It extends the concept of generics used in VHDL and other similar languages. We will explain the policy class concept on an example from the OpenAVR project.

Consider 2 different AVR-based MCUs: ATTiny26 and ATTiny48. Each of them has a different RAM starting address, and set of common hardware registers:

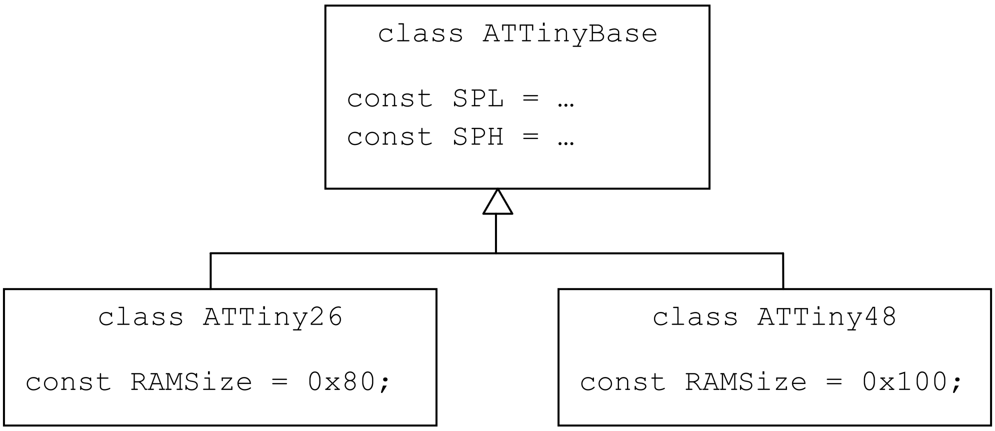
Parameter	ATTiny26	ATTiny48
RAM starting address	0x60	0x100
RAM size	0x80	0x100
ROM size	0x100	0x100
SPL address	0x5D	
SPH address	0x5E	
SREG address	0x5F	
PORTB address	0x25	

If we want to avoid copy-pasting the code to switch between ATTiny26 and ATTiny48 emulation, classical VHDL provides 2 ways of doing this:

- Making 7 generic parameters
    - Advantage: we specify parameters (e.g. RAM size) when MCU entity is instantiated. Thus, can use one inside a simulator testbench, another in a simple FPGA testbench, and third one in production.
    - Disadvantage: propagating generics to a nested entity is cumbersome. Adding new generics requires modifying tons of code.
- Making a package with 7 constants
    - Advantage: no explicit generic propagation (nested generic maps).
    - Disadvantage: package name is specified in entity declaration, not in instantiation. I.e. having one testbench for ATTiny26 and another for ATTiny48 requires constantly switching package name in VHDL sources.

There is one more general disadvantage: VHDL offers no way of expressing the hierarchical structure of the MCU definitions (e.g. ATTiny26 and ATTiny48 both share some common properties of ATTiny family). Thus, extra copy-pasting is required.

As mentioned before, a policy class is a collection of constants and definitions. It's similar to a C++ class consisting only of static members. E.g. the information about MCU types can be represented in the following way:



Here we have combined all "configuration" information about one MCU into a single policy class (see **MCUs.thp** for more details). The MCU entity can use the class as a template argument (i.e. generic):

```
template <any _MCUType> entity MCUWithROM
{
	...
	ComplexRAMWrapper<_MCUType.RAMStart, _MCUType.RAMSize> ...
}
```

When the entity is instantiated, we specify which MCU policy class to use. E.g.:

```
MCUWithROM<ATTiny26> uut(...);
```

To use ATTiny48 instead of ATTiny26 in another testbench, simply specify it during instantiation:

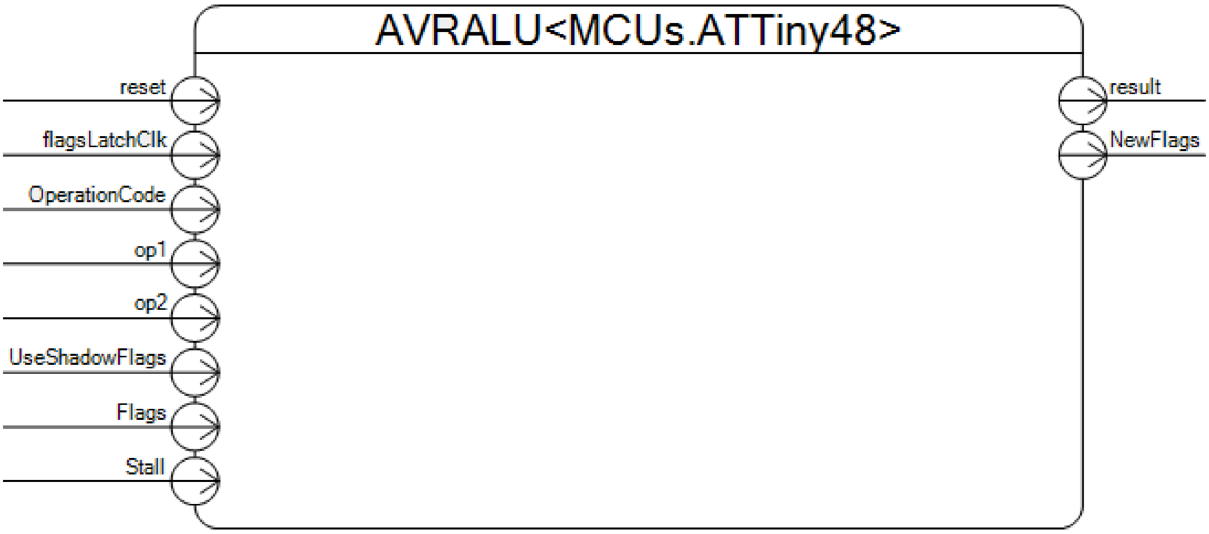
```
MCUWithROM<ATTiny48> uut2(...);
```

This is similar to specifying the values for 7 VHDL generics, however, the form is much more compact and a lot of copy-pasting is avoided.

The rest of this document explains how policy classes are used in other parts of OpenAVR.

The generic ALU

The ALU is implemented inside **ALU.thp**. The input/output ports of the ALU entity are shown below:



The most important inputs are **OperationCode**, **op1**, **op2** and **flags**. In turn, the outputs are **result** and **NewFlags**. The information about ALU operations is defined in a policy class tree originating from **OperationBase**. Every policy class defines the following members:

- **OperationCode** constant.
- **SetFlags()** function**.**
- **ComputeResult()** function.

The ALU operations are defined using inheritance. E.g. "**LogicalOperation**" class defines how flags are set for logical operations and classes like **And**, **Eor** and **Or** inherit it and only define the **ComputeResult** function.

The core of the ALU is the **operation** process (THDL++ processes have VHDL semantics) that is defined in a generic way:

```
For each supported ALU operation
    If (OperationCode == code defined by the current operation)
        Set result and flags, as defined by operation.
```

Adding new ALU operations, or modifying their internal behavior is done by modifying **supportedOps** list and does not require changing the processes and signals.

The generic instruction decoder

Instruction decoding is implemented similarly to ALU. Instructions are defined in a tree of policy classes (**InstructionDefinitions.thp**) and the actual decoding is done inside the **decode** process (**CPU.thp**).

Every instruction is represented by a class that defines a **Match()** function and various helper functions (e.g. **IsRAMWritten()**) that describe the instruction behavior.

Multi-cycle instructions do not define the helper constants and functions. Instead they contain a list called **Stages** that lists stage policy classes. A stage policy class acts like a "sub-instruction" and defines all helper functions inside itself.

Thus, a single-stage instruction class does not contain the **Stages** definition. Instead, it is treated as a stage class itself (i.e. defines helper functions and constants). This is captured inside the **InstructionStages** function:

```
any InstructionStages(any insn)
{
      if (__defined(insn.Stages))
            return insn.Stages;
      else
            return insn;
}
```

The **decode** process implements the following functionality:

```
For each supported instruction
    If currently loaded instruction matches it
        If it's a single-stage instruction, select first stage
        If it's a multi-stage instruction, select stage based on CurrentStage signal
            Set all decoded signals as defined in the stage class
```

Adding or removing supported instructions simply requires changing the **Instructions** list.

Note that THDL++ uses VHDL semantics. Thus, any subsequent signal writes in a process will overwrite its value. This is used to handle "unknown instruction" condition:

- A special **UnknownInstruction** class is defined. Its **Match()** always returns true.
- When iterating through supported instructions, **UnknownInstruction** is always checked first.

Thus, if no instruction matches the currently loaded instruction word, the default behavior defined in **UnknownInstruction** will take place.

Instruction stages are efficiently reused. E.g. the **PushPopInstruction** instruction class has a template argument "_ReplacePCH" that specifies whether the high-order byte of PC register is replaced with the popped value. It is always **false** for the actual push/pop instruction. However, the **RetInstruction** reuses those stage definitions setting **_ReplacePCH** to true. This reflects the fact that "ret" is essentially a set of two "pop" instructions where the result output is routed to PC.