

HW support in TASTE : preliminary (very promising!) results...

As explained in the documentation¹, TASTE takes a “first-design-your-interfaces” point of view. You start with “boxes” of subsystems, and proceed to specify their interfaces in very simple terms:

1. what input parameters go in, what output results come out. This is done in terms of “abstract” content, e.g. an INTEGER, or a floating point, or a structure containing a number of types, etc. ASN.1 is used for specifying the parameter types.
2. what type of interface this is, and what timing information it obeys – e.g. periodic with a period of 100ms, or sporadic with a minimum interarrival time of 500ms, a worst case execution time of 150us (WCET), etc
3. what language/tool this subsystem is built on (e.g. manual C code, or Simulink/RTW generated, or SCADE generated, etc)

A growing set of parsers and code generators then reads this spec – we call these tools the TASTE toolchain. The tools parse the specification, and generate, completely automatically:

- skeleton projects, that contain fully specified, type-wise, interface specifications. The users of TASTE just “fill-in the blanks” of these skeletons.
- runtime “glue”, that uses ASN.1 encoders and decoders to allow code generated by different tools and languages to speak to one-another

Having the above executive summary in mind, what do we have to do to support HW components, if we are to follow the TASTE mentality?

The same things!

- We automatically generate the component skeletons - which means we generate the complete specifications of (a) the VHDL “component” section (input signals, output signals, etc) and (b) SystemC header and implementation file for the component.
- We also generate runtime “glue”, that will allow the other components to speak to the HW component at runtime – which means that we will generate (completely automatically) the device driver that will speak to the component.

Let’s look at an example, which was demonstrated on a Xilinx Spartan3 board in ESTEC, in late April.

Say we want to create a simple circuit, that checks whether a number is prime or not.

What would this circuit (component) have as input and output?

¹ <http://www.semantix.gr/assert/assert/node7.html>, <http://www.semantix.gr/assert/assert/node8.html>

Well, INTEGERS, of course:

```
DataView DEFINITIONS AUTOMATIC TAGS ::= BEGIN

T-INTEGER ::= INTEGER (0..9223372036854775807) -- 2^63-1

END
```

This is the ASN.1 grammar – it describes the types of messages that will be exchanged in our interfaces.

Then, what is this component’s interface?

Simple: one input (the input number), one output (the first factor that divides this number). If it is equal to the input, then the number is prime, of course:

```
SUBPROGRAM compute
FEATURES
    in_tocheck:IN PARAMETER DataView::T_INTEGER {encoding=>UPER;};
    out_factor:OUT PARAMETER DataView::T_INTEGER {encoding=>Native;};
END compute;

SUBPROGRAM IMPLEMENTATION compute.VHDL
PROPERTIES
    FV_Name => "work_func";
    Source_Language => VHDL;
END compute.VHDL;
```

So, we feed this simple spec to the TASTE skeleton generators and glue generators. What do we get?

The VHDL skeleton/glue

We said before that the generated skeletons of TASTE subsystem have complete input/ output specifications, including their type info. Indeed, look at this section of the generated “TASTE.vhd”:

```
-- Circuits for the existing PIs
component compute is
port (
    in_tocheck : in std_logic_vector(63 downto 0);
    out_factor : out std_logic_vector(63 downto 0);
    start_compute : in std_logic;
    finish_compute : out std_logic;
    clock_compute : in std_logic
);
end component;
```

You see that the interface params (the two integers) have been mapped to corresponding VHDL entities. The skeleton also includes signals “start”, “finish” and “clock”:

- “clock” is the obvious
- “start” is the signal raised by the circuit’s user, as soon as the “in_tocheck” param has been written – it tells the circuit: “go on, your input data are there”
- “finish” is the signal raised by the circuit, as soon as the computation is completed – it tells its user: “I am done, go read “out_factor”.

Obviously, that’s not all. This is just the core declaration of the circuit. So far, one might say: we could write this manually, it would not be a big deal.

If however, the parameter type is more complex, then the input mapping becomes... a real task. e.g. what would happen if we used type T_POS from the grammar below?

```
ttsiod@elrond: /home/ttsiod/ASSERT/Codegen-A/VHDL/passive_function/output
TypeEnumerated ::= ENUMERATED {
    red(0),
    green(1),
    blue(2)
}

TypeNested ::= SEQUENCE {
    intVal      INTEGER(0..10),
    int2Val     INTEGER(-10..10),
    int3Val     MyInt (10..12),
    intArray    SEQUENCE (SIZE (10)) OF INTEGER (0..3),
    realArray   SEQUENCE (SIZE (10)) OF REAL (0.1 .. 3.14),
    octStrArray SEQUENCE (SIZE (10)) OF OCTET STRING (SIZE(1..10)),
    boolArray   SEQUENCE (SIZE (10)) OF T-BOOL,
    enumArray   SEQUENCE (SIZE (10)) OF TypeEnumerated,
    enumValue   TypeEnumerated,
    enumValue2  ENUMERATED {
        truism(0),
        falsism(1)
    },
    label       OCTET STRING (SIZE(10..40)),
    bAlpha      T-BOOL,
    bBeta       BOOLEAN,
    sString     T-STRING,
    arr         T-ARR,
    arr2        T-ARR2
}

-- A more realistic definition
T-POS ::= CHOICE {
    longitude   REAL(-180.0..180.0),
    latitude    REAL(-90.0..90.0),
    height      REAL(30000.0..45000.0),
    subTypeArray SEQUENCE (SIZE(10..15)) OF TypeNested,
    label       OCTET STRING (SIZE(50)),
    intArray    T-ARR,
    myIntSet    T-SET,
    myIntSetOf  T-SETOF,
    anInt       My2ndInt
}
```

13,0-1 24%

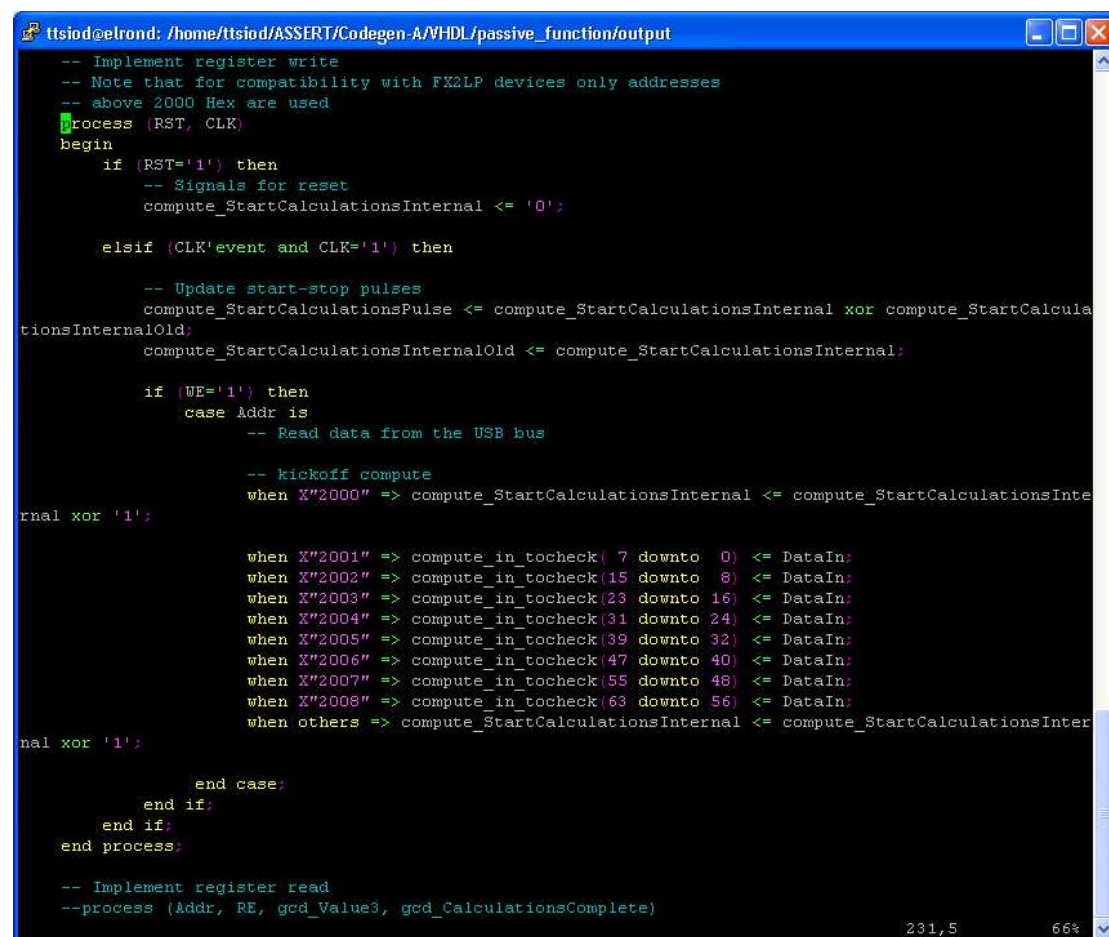
Suddenly, mapping this type to VHDL is not as easy. With our TASTE mapper, this is just as easy to handle, as a simple INTEGER!

And that's just the beginning.

The TASTE mapper knows what the target FPGA architecture is: What bus the FPGA is operating over (PCI? USB? Etc), what FPGA type this is (Spartan? Virtex?), etc. So it can generate ALL THE CODE necessary for “speaking” – at runtime – to the chip, intercepting write accesses (over the bus) and responding to read access (over the bus).

In our case, we used a ZestSC1 board – a simple Spartan3-based board that is attached through a USB port, and allows 8bit communications with the FPGA over the USB bus.

The automatically generated VHDL skeleton includes code like this:



```
ttsiod@eltrond: /home/ttsiod/ASSERT/Codegen-A/VHDL/passive_function/output
-- Implement register write
-- Note that for compatibility with FX2LP devices only addresses
-- above 2000 Hex are used
process (RST, CLK)
begin
    if (RST='1') then
        -- Signals for reset
        compute_StartCalculationsInternal <= '0';

    elsif (CLK'event and CLK='1') then

        -- Update start-stop pulses
        compute_StartCalculationsPulse <= compute_StartCalculationsInternal xor compute_StartCalculationsInternalOld;
        compute_StartCalculationsInternalOld <= compute_StartCalculationsInternal;

        if (WE='1') then
            case Addr is
                -- Read data from the USB bus

                -- kickoff compute
                when X"2000" => compute_StartCalculationsInternal <= compute_StartCalculationsInternal xor '1';

                when X"2001" => compute_in_tocheck( 7 downto 0) <= DataIn;
                when X"2002" => compute_in_tocheck(15 downto 8) <= DataIn;
                when X"2003" => compute_in_tocheck(23 downto 16) <= DataIn;
                when X"2004" => compute_in_tocheck(31 downto 24) <= DataIn;
                when X"2005" => compute_in_tocheck(39 downto 32) <= DataIn;
                when X"2006" => compute_in_tocheck(47 downto 40) <= DataIn;
                when X"2007" => compute_in_tocheck(55 downto 48) <= DataIn;
                when X"2008" => compute_in_tocheck(63 downto 56) <= DataIn;
                when others => compute_StartCalculationsInternal <= compute_StartCalculationsInternal xor '1';

            end case;
        end if;
    end process;

-- Implement register read
--process (Addr, RE, gcd_Value3, gcd_CalculationsComplete)
```

If you look at this VHDL code close, you'll see that it intercepts write accesses over the bus, and ***knows* how to map them to the appropriate input registers!**

This is an important part of designing chips, and with TASTE, it is done completely automatically!

Notice that the code generator knew that this is an 8-bit bus, so it mapped the input parameter (the 64bit value that will pass over the bus) to 8 bus addresses, from 0x2001 to 0x2009.

0x2000 is reserved for the “kick-off” signal – when someone (who? Keep reading) writes to this address, the chip’s “start” signal will be raised.

There is corresponding code for the reverse direction: the reading of the response over the USB bus:

```
-- Implement register read
--process (Addr, RE, gcd_Value3, gcd_CalculationsComplete)
process (Addr, RE, compute_out_factor, compute_CalculationsComplete)
begin
    if (RE='1') then
        case Addr is
            -- Write data to the USB bus

            -- result calculated flag compute
            when X"2000" => DataOut <= "0000000" & compute_CalculationsComplete;

            when X"2009" => DataOut <= compute_out_factor( 7 downto 0);
            when X"200a" => DataOut <= compute_out_factor(15 downto 8);
            when X"200b" => DataOut <= compute_out_factor(23 downto 16);
            when X"200c" => DataOut <= compute_out_factor(31 downto 24);
            when X"200d" => DataOut <= compute_out_factor(39 downto 32);
            when X"200e" => DataOut <= compute_out_factor(47 downto 40);
            when X"200f" => DataOut <= compute_out_factor(55 downto 48);
            when X"2010" => DataOut <= compute_out_factor(63 downto 56);

            when others => DataOut <= X"00";
        end case;
    else
        -- avoid latches
        DataOut <= X"00";
    end if;
end process;
```

Notice that the read accesses were automatically mapped to different offsets – this depends of course, on many things, including whether the FPGA board accepts bidirectional register access or not – **but the point is, you don’t have to be involved with these parts, they are written for you, automatically.**

The device driver

So OK, we have a ready-to-use VHDL skeleton/glue, all we have to do is write the implementation of the “compute” component, and it will take over “speaking” to the chip.

But what about communications with the SW world? There will be others, SW components most likely, that will be speaking to this component. These others will most probably run inside CPUs (Leons, or x86 Linux, so far). How will they “speak” to the chip?

Well, since the VHDL “bridge” was written automatically by TASTE code generators, the same code generators – who knew the register offsets they allocated to each parameter – can also write a COMPLETE device driver!

Here's a part of the automatically generated driver code, for our ZestSC1 board:

```
int Convert_From_uper_To_T_INTEGER_In_compute_VHDL_in_tocheck(void *pBuffer, size_t iBufferSize)
{
    STATIC asn1SccT_INTEGER var_T_INTEGER;
    flag errorCode;
    STATIC BitStream strm;
    BitStream_AttachBuffer(&strm, pBuffer, iBufferSize);

    if (asn1SccT_INTEGER_Decode(&var_T_INTEGER, &strm, &errorCode)) {
        /* Decoding succeeded */
        {
            unsigned char tmp, i;
            asn1SccSint val = var_T_INTEGER;
            for(i=0; i<sizeof(asn1SccSint); i++) {
                tmp = val & 0xFF;
                ZestSC1WriteRegister(g_Handle, BASE_ADDR + 0x1 + i, tmp);
                val >>= 8;
            }
        }
        return 0;
    } else {
        fprintf(stderr, "Could not decode T-INTEGER (at %s, %d), error code was %d\n", __FILE__, __LINE__, errorCode);
        return -1;
    }
}
```

ZestSC1 offers a simple API to “speak” to the chip, over the USB bus. Notice the two last parameters that the generated code passes to the “ZestSC1WriteRegister” function (above): a register offset, and a value. BASE_ADDR is in fact, 0x2000 – so you see, this function:

- Obtains an incoming INTEGER value – sent, presumably, from other TASTE subsystems that are curious whether this number is a prime or not
- Writes the incoming value over the USB bus, one byte at a time, in the appropriate target offsets.

So, since our code generator created the “receiving” code of the VHDL side, it knows how to write the corresponding “sending” side, in the driver code!

And it knows to “kick-off” the chip, as soon as all input parameters are in...

```
void Execute_compute_VHDL()
{
    unsigned char flag = 0;

    // Now that the parameters are passed inside the FPGA, run the processing logic
    ZestSC1WriteRegister(g_Handle, BASE_ADDR + 0x0, (unsigned char)1);
    while (!flag) {
        // Wait for processing logic to complete
        ZestSC1ReadRegister(g_Handle, BASE_ADDR + 0x0, &flag);
    }
}
```

... and wait for the result to be calculated! (the “finish” flag to be raised).

Imagine writing all this code, but not for the simple case of an INTEGER – no, imagine writing it for a complex type like the T_POS we met before – or even better, imagine you are adding yet another field in your type definition, which shifts all your register addresses by some offset. Imagine having to update all the necessary code, in the driver, in the VHDL side... TASTE *completely* solves this problem!

What about SystemC?

Sure, sure, all these are nice things to have. But what about... the state of the art? What if we want to implement the “compute” component on our FPGA, not by writing tedious VHDL code, but in ... SystemC?

Well, you can!

Important note: Some people seem to think that SystemC code will execute in a CPU - that's not correct: the plan with SystemC, is (a) write your code in good old C++ (b) use a C++ compiler to compile it (c) run it, and thus simulate the chip, and verify it works correctly and FINALLY (d) use a SystemC compiler to compile your design to VHDL, so you can download the design to your FPGA!

So, this is the automatically generated SystemC header that TASTE creates:

```
class compute : public sc_module
{
public:
    sc_in<sc_uint<64> > in_tocheck;
    sc_out<sc_uint<64> > out_factor;

    sc_in<bool>          start_compute;
    sc_out<bool>         finish_compute;
    sc_in<bool>          clock_compute;

    void do_compute ();

    SC_CTOR (compute)
    {
        SC_THREAD(do_compute);
        sensitive_pos << clock_compute;
    }
};
```

And this is the automatically generated SystemC skeleton:

```
void compute::do_compute()
{
    // Declare your variables here
    finish_compute = 0;
    while (1) {
        do {
            wait();
        } while (!start_compute.read());
        finish_compute = 0;

        // Write your processing logic here
        // Read data from in_tocheck
        // ...

        // Write result for out_factor
        finish_compute = 1;
        wait();
    }
}
```

Now, look at this code:

- it waits until someone (the device driver) raises the “start” signal
- the user-written code will then be executed (notice that automatically generated comments tell the user what to do: “read data from in_tocheck”, “write result to “out_factor”)
- when the user code is finished, the chip will raise the ‘finish’ flag

So what do we see?

We see all the parts that comprise the interfacing aspects of the chip – that is, the TASTE users never have to write a single line in terms of interfacing, they are written for them, by TASTE.

Conclusion

With the HW-specific extensions that the TASTE experts are currently introducing in TASTE tools, developing HW components will be just as clear as it is for SW components. Given the added complexity factor that is automatically solved (device driver code, VHDL code, SystemC code) the TASTE benefits are even more clear in the case of HW designs.