Assembler with VHDL User-defined Commands (AVUC)

AVUC 1.0

Date: 18. Jul. 2009

Contents

1. Introduction.

Here is presented *avuc.pl*, a Perl script that creates a VHDL entity that is able to execute an assembler program written on an input file (in our example, the input file is *max_mem.usm*) with the particularity being that the assembler commands are defined by the user.

The mentioned input file contains not only the program but also the definitions of the assembler commands, written in VHDL language by the user, with which the program is composed. There are two predefined commands (*jump* and *nop*) and two possible types of commands defined by the user: conditional jump commands and ordinary commands.

Every ordinary assembler command is defined as a series of timed VHDL language commands that modify one or more VHDL signals. In that way, the number of clock cycles that takes the execution of each of these commands is determined by the user.

Once the user has defined all the desired assembler commands, a program using all these commands can be written.

The user should also declare the internal and the input-output signals so that the VHDL entity could be built.

In addition, some extra code can be included in order to better complete the function of the program.

In summary, the user writes an input file with:

- user-defined VHDL assembler commands.
- an assembler program using these commands.
- signal declarations: internal signals and signals for the entity interface (input and output signals).
- extra VHDL code.

This made, the Perl program *avuc.pl* takes the input file and creates a VHDL entity that can run the assembler program.

The program is implemented as a case-when VHDL process, which results in a manufacturer independent and fully synthesizable ROM (the ROM bus width grows with the base 2 logarithm of the number of code lines).

The program is started by a rising edge at the interface VHDL input signal *avuc_start* and can be ended (standby) by a rising edge at the interface VHDL input signal *avuc_rst*.

The entity also includes an automatic start-up reset, that brings the program to the end/standby after power-up (after 32 clocks if all the FPGA flip-flops initialize to zero).

The end of the program is implemented as an infinite loop, so that, to run the program again, a rising edge on the mentioned input signal *avuc_start* should be applied.

The state of the program (if it is running or has been finished) can be obtained by another interface signal, *avuc_state* (an output signal of the entity). If the program has finished, the value of *avuc_state* is *AVUC_STATE_STOPPED*; if not, its value is *AVUC_STATE_RUNNING*.

These two constants are defined in a separate package *avuc_pkg* inside the file *avuc_pkg.vhd*.

If these constants are not to be used by the user, they could be copied to the signal declaration block and, then, the *avuc_pkg* should not be used.

This system has been developed to implement complex state machines with many iterations or conditional jumps, that are easily implemented by means of simple structured programming.

The operations performed by the program are usually not as time-critical as the rest of the VHDL code and, so, it can be considered for readier analysis that the program is 'slow' with respect to the 'fast' signals that are tested or driven by the program (i. e. the rest of the circuit is 'fast' with respect to the *avuc* microprocessor).

2. Sintax of the input file.

The input file is composed of several blocks defined by keywords. Each keyword begins with the double symbol *&$*.

2. 1. User-defined commands.

Most of the commands used by the assembler program are user-defined commands. Out of these, there are only two predefined commands:

- *nop*: No operation. Do nothing, it is a clock cycle delay.
- *jump label*: Jump to the entry point marked by *label*.

Therefore, commands inside the assembler program can only be one of the predefined commands (*nop* and *jump*) or one of the user-defined commands.

There are two types of user-defined commands: ordinary user-defined commands and conditional jump user-defined commands.

For the first case, the user has to supply the actions that define the command. For the second case, the user has to supply the conditions under which the program will jump to the specified label.

2. 1. 1. Ordinary user-defined commands.

Each ordinary user-defined command changes the value of one or more signals (i. e. VHDL processes are generated that drive these signals).

All the commands that modify a particular VHDL signal (signal drivers) should be grouped together into a block called *opcode_def*.

This block should be enclosed by the keywords *&$opcode_def* and *&$end_opcode_def*. The names of the signals to be modified should be declared in the same line after the keyword *&$opcode_def*.

For example, if the signals *s1* and *s2* are to be modified by a set of assembler commands, the set of commands that modify them should be defined inside the block:

Declaration of an *opcode_def*:

```
&$opcode_def s1 s2
... [set of definitions of assembler commands driving s1 and s2]
&$end_opcode_def
```

Every user-defined command inside the o*pcode_def* block should be preceded by the identifier *&$* (i. e., a user-defined command named *my_command* should be declared as *&$my_command*, included inside an *opcode_def* block).

After the user-defined command declaration, every action over the signals is preceded by the keyword *&$cycle_def* plus a number indicating the clock cycle number at which the action take place (i. e. 1 for the first clock cycle, 2 for the second clock cycle and so on).

Then, the VHDL code should be written after the *&$cycle_def* declaration.

Declaration of a user-defined command:

```
&$user_command                              [name of the command, not a keyword]
   &$cycle_def 1
   ... [VHDL actions taking place during the the first clock]
   &$cycle_def 2
   ... [VHDL actions taking place during the the second clock]
   &$cycle_def ...
   ... [VHDL actions taking place during the the following clocks]
```

So, the definition for a command called *ff_pulse* that change the signal *ff* to 0, then to 1, and then again to 0, will be the following:

```
&$ff_pulse
    &$cycle_def 1
        ff <= '0';
    &$cycle_def 2
        ff <= '1';
```

```
    &$cycle_def 3
        ff <= '0';
```

(indentation is only used for better understanding).

Of course, the duration of a particular command (the number of clock cycles) depends on how many *&cycle_def* declarations it is divided.

The usual case is that several commands operate over the same VHDL signal or signals. So, if the user wants to define *user_command1*, *user_command2*, *user_command3*, ..., commands that drive the VHDL signals *s1* and *s2,* they should be declared in the following way:

Declaration of o*pcode_def* plus several user commands:

```
&$opcode_def s1 s2
      &$user_command1
      ... [&$cycle_def and VHDL definitions]
      &$user_command2
      ... [&$cycle_def and VHDL definitions]
      &$user_command3
      ... [&$cycle_def and VHDL definitions]
      &$default
      ... [only VHDL definitions]
&$end_opcode_def
```

The user could also define what is going to do the processor when no command is executed. That is done by means of the keyword *&$default*, as shown in the example above. The VHDL definitions after the *&$default* declaration would be executed permanently when none of the other user commands for the corresponding VHDL signals is called.

As an example for a complete *&$opcode_def* declaration along with commands, here is presented the definition of two commands that operate over a signal called *mem_addr.* The first command is named m*em_addr_init*, designed to initialize the signal *mem_addr*, and the second command is named *mem_addr_inc*, defined to increment *mem_addr* by 1.

The definition of these two commands will be:

```
&$opcode_def mem_addr
    &$mem_addr_init
        &$cycle_def 1
            mem_addr <= (others => '0');
    &$mem_addr_inc
        &$cycle_def 1
            mem_addr <= mem_addr + 1;
&$end_opcode_def
```

As both commands only need a clock cycle to complete, there is only a *&$cycle_def* identifier for each one.

The user can include VHDL conditions or loops in the definition of the user-defined commands, but it should be understood that this VHDL code will be used inside a VHDL *process* and, therefore, the suitable sintax for processes should be used (*if* should be used instead of *when* and *case-when* clauses instead of *with-select-when* ones).

For example:

```
&$data_assign
    &$cycle_def 1
        for i in DATA_WIDTH-1 downto 1 loop
            if flag = '1' then
                data(i) <= data_a(i-1);
            else
                data(i) <= data_a(i);
            end if;
        end loop;
```

An ordinary user-defined command could also make use of the autogenerated VHDL signal *usm_data* (this signal is also used to carry the jump address in case of a jump command). This signal is set to the value that is given following an ordinary user-defined command.

For example, if we have a command named *set_s3*, and we write in the program:

```
set_s3 12
```

this means that the internal VHDL signal usm_data will have the value 12 when this command is executed.

This fact allow us to define the command depending on the u*sm_data* value. Following the example, we could define set_s3 as:

```
&$opcode_def s3
    &$set_s3
        &$cycle_def 1
            s3 <= usm_data (3 downto 0);
&$end_opcode_def
```

so that the signal s*3* take the value following the command *set_s3* in the program, i. e., 12 in the example above, so *s3* will be 12.

2. 1. 2. Conditional jump commands.

The name of the conditional jump commands is as the name of ordinary commands determined by the user.

The natural name of these commands could be *jump_if_...*. So, if the user want to define a jump [to some other place of the program] if some signal *s1* is 1, a suitable name for the command could be j*ump_if_s1_eq_1*.

Obviously, the conditional jump commands, when invoked inside a program, always carry an indication to the point to jump as long as the condition is met. That point is marked by a label, as is usually done in any language,.

There are two implicit predefined labels (so, these special labels are keywords and should not be used by the user):

- *begin*: point to the first line of the program.
- *end*: point to an internal line that is included automatically after the last line of the program. This line implements an infinite loop. (i. e. it is an additional line with the command *jump end*).

All the conditional jump commands defined by the user should be grouped in a block headed by the keyword *&$jump_opcode_def*.

Following the keyword *&$jump_opcode_def*, all the user-defined conditional jump commands preceded by the keyword *&$condition* and its corresponding jump conditions are to be declared:

```
&$jump_opcode_def
   &$condition jump_user_command1
      ... [VHDL boolean condition 1]
   &$condition jump_user_command2
      ... [VHDL boolean condition 2]
   &$condition jump_user_command3
      ... [VHDL boolean condition 3]
   ...
```

In this way, if the *VHDL boolean condition 1* is met, at the moment in wich the command *jump_user_command1* is executed, there will be a jump to the specified label.

Here an example is presented:

```
&$jump_opcode_def
   &$condition jump_if_s1_eq_0
      s1 = '0'
   &$condition jump_if_counter_eq_end
      counter = COUNTER_END
```

where *s1* and *counter* are VHDL signals and *COUNTER_END* is a VHDL constant defined in the signals declaration or in a package accessible to the entity.

Being made these definitions, we can use through the program the user-defined conditional jump assembler commands *jump_if_counter_eq_end* and *jump_if_s1_eq_0*.

2. 2. Remaining definitions.

In order to generate the VHDL entity properly, some parameters such as the name of the entity and others should be given. These are defined after the keywords explained through this section.

The lines after a keyword belong to the keyword, finishing when the next keyword is found. As before, keywords begin with *&$*.

2. 2. 1. Constants and identifiers.

The general sintax for the constant and identifiers keywords is:

```
&$keyword value
```

The keywords are *&$clock*, *&$entity* and *&$data_bus_min_width*. For instance:

```
&$clock my_clock
&$entity my_entity
&$data_bus_min_width 11
```

&$clock: Name of the clock used for the program operation, the clock is an input signal of the entity.

&$entity: Name of the generated VHDL entity.

&$data_bus_min_width: Minimum data bus (*usm_data*) width.

The data bus is the autogenerated signal named *usm_data* and is used for two purposes already explained:

- As a line address for a jump command.
- As a piece of data for a user-defined ordinary command.

If *data_bus_min_width* is set to 0, the *usm_data* width is adjusted to the maximum line address width in order to carry the address for the jump commands.

So, if you want to use the bus to pass data in user-defined ordinary commands, the maximum width of the the data you want to pass should be here specified as the minimum data bus width.

2. 2. 2. VHDL signals declaration, headers and extra code.

The format of the lines after each of the keywords presented in this chapter is that of a VHDL code.

&$header: VHDL comment information you want to put at the beginning of the created file

&$include: Lines of VHDL include and use libraries before the entity declaration. For example:

```
&$include
   library work;
   use work.avuc_pkg.all;
```

&$generic: List of generic parameters for the entity. For example:

```
&$generic
   max_delay: integer := 100;
   is_fast: boolean := true;
```

&$port: List of input/output signals of the entity interface. The autogenerated input or output signals (like the clock, *avuc_start* or *avuc_rst*) should not be included. For example:

```
&$port
   data_in: in std_logic_vector(7 downto 0);
   data_out: out std_logic;
```

&$sig_declaration: Internal VHDL signals or constants declaration, declared inside the architecture section of the created entity. For example:

```
&$sig_declaration
   signal data_middle: std_logic_vector(7 downto 0);
   constant key_something: std_logic_vector(7 downto 0) := x"3B";
```

&$extra_code: VHDL extra code to perform additional functions.

2. 3. Program.

The assembler commands that form the executable program should begin after the keyword *&$prog_code*. The sintax of each line of the program should be one of the following two, depending on if it is a jump command or not:

```
[label:] not_jump_command [usm_data_value]
[label:] jump_command label_to_jump
```

where *not_jump_command* can be a user-defined ordinary command or the predefined command *nop* and *jump_comman*d can be a user-defined conditional jump command or the predefined command *jump*. In any case, assembler commands used in the program are written without the prefixing symbols &$, used to define them.

There are two predefined commands: *jump* and *nop* and two predefined labels: *begin* and *end.*

3. Call to the Perl script.

Finally, the Perl script *avuc.pl* should be invoked to process the input file containing the commands definitions, the program, etc., to give a synthesizable VHDL entity. Its usage is as follows:

```
avuc.pl [flags] inputfilename
```

where the flags can be one or more of the following:

-o output filename
-h show this help
-v verbose
-d debugging information

For example:

```
avuc.pl -o /home/ferblanco/example/ts.vhd -v /home/ferblanco/example/max_mem.usm
```

If no output filename is supplied, the output filename is formed with the input filename, with the extension changed to .vhd.