Abstract
The current problems in adopting VHDL as a Hardware Description Language for a microcontroller design project are the design methodology and generating VHDL code which will map to appropriate hardware. This paper describes the design and validation approach used to develop a general purpose microcontroller. In addition, it describes VHDL code writing styles, which will map to desirable hardware. Also, it will discuss various optimal synthesis strategies.

Introduction
A typical microcontroller consists of the following components:
1. CPU Core.
2. Bus controller.
3. Peripherals.
Figure 1 shows a block diagram of a microcontroller.

<table>
<thead>
<tr>
<th>CPU Core</th>
<th>Bus Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peripheral 1</td>
<td></td>
</tr>
<tr>
<td>Peripheral n</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Microcontroller Block Diagram
The CPU core is designed using following criteria:

1. Performance.
2. Power dissipation.
3. Area.
To meet the above objectives, one has to design the CPU core data path using custom circuits since synthesizing the data path does not produce the best results. The control logic of the CPU core, bus controller and peripherals can be synthesized. Hence this approach of using custom and standard cells poses following problems:
1. Mixing transistor level net list with standard cell net list.
2. Performing full-chip simulation.
3. Full-chip verification.
The mixing of transistor level netlist and gate level netlist poses full chip verification problem. All the simulators cannot handle both transistor level and gate level netlist. The gate level netlist has to be converted to transistors and then merged with the transistor level netlist. The simulation of transistor level netlist is slower compared to gate level netlist. In addition, the CPU core can be described in a different language, such as C, since the core might be purchased from a vendor. Also, time-to-market and price are important factors in releasing an IC to the market. Hence using appropriate tools, which can provide optimum speed, area and power, plays a significant role in the overall design methodology.

This paper describes an approach used for design tool evaluation, overall design methodology, optimum hardware inference coding styles, and use of VHDL to describe the hardware elements and performance monitoring approach.
VHDL Choice

We choose VHDL as our language to describe all the peripherals for the following reasons:
1. Availability of internal/external simulation and synthesis tools.
2. Internal expertise in VHDL.
3. Customer's preference.

Design Methodology

After making hardware description language choice, the focus was to evaluate simulation and synthesis tools. The simulation tool choice was made based on the following factors:
1. Simulation speed.
2. User Interface.
3. Ease of use.
4. Test vector generation capability.
5. Test command language features.
6. Interface to other languages such as C.
7. Price.

The simulation speed of a tool enables a user to quickly evaluate the model's functionality and make appropriate changes. In addition, the simulation speed increases productivity and reduces the number of licenses required for a given project. A good user interface aids user to quickly adapt to the tool. The test vector generation capability allows a user to capture test vectors from the simulation tests to verify the functionality. The test command language facilitates a user in test development. The interface to the languages such as C allows user to interface to models written in language other than VHDL.

The choice of synthesis tool was based on the following factors:
1. Ease of use.
2. Area/speed optimization.
3. Accurate report generation.

The synthesis tool's ease of use facilitates user in evaluating circuit's area/performance for different architectures. The synthesis tools use different algorithms to optimize a given behavioral model. While evaluating a synthesis tool, one should compare the area numbers without the routing factors since each routing tool has its own characteristics. Hence area numbers for combinatorial and non-combinatorial gates should be used for comparison purposes. The report generation facility of a synthesis tool enables user to evaluate different architectures with respect to speed and area. The following reporting features should be available from a synthesis tool:
1. Area
2. Maximum frequency
3. Critical path
4. Gate usage statistics.
5. Logic inference by the compiler.
6. Point to point timing on a given path.

The area and maximum frequency number helps a user to evaluate different architectures. The critical path analysis enables user to decide on cycle time and change any attributes in the circuit to evaluate that path faster. The gate usage statistics helps determine if any modifications need to be done in the standard cell library to reduce unused gates within a macro or a multi-gate cell. The logic inference reporting capability gives feedback to the user as to type of components inferred by the compiler. This feedback can be used to change the source code to infer requisite components. The point to point timing report can be used to determine timing on a given path to make early architecture decisions.

With some idea on the design methodology and deciding on the simulation and synthesis tool, one can come up with a complete design methodology for the project. The design process begins with the architectural specifications, which are then translated into a VHDL behavioral model. This behavioral model is tested stand-alone either using a VHDL test bench or command file using test command language of the simulator. After satisfactory functional simulation, the behavioral model is plugged into the overall chip model for chip level testing. At this point it is possible that the CPU model is written in C. To integrate CPU model written in C with the VHDL model, a VHDL wrapper is written using the foreign language interface of the simulator. This enables the user to communicate to the peripherals via CPU. All the target model tests can be then written in the microcontroller assembly language. After satisfactory verification
of the models, the behavioral code is synthesized to obtain gate level net list to prepare for timing verification and layout. If synthesis does not produce satisfactory results, one has to either change synthesis constraints or change the source code. After synthesis, a gate level VHDL can be generated and plugged into the chip level model to verify the functionality of the netlist. The netlist generated from synthesis is used by the layout tool for circuit layout. The layout parasitics can be extracted and back annotated to the VHDL netlist to verify timing. The flowchart in figure 2 depicts the design flow described.

VHDL Coding Style
While writing VHDL code, the following fact should be kept in mind:
1. Target hardware.
2. Chip Structure.
The feedback on the target hardware can be obtained from the synthesis tool during preliminary stages of behavioral code compilation. The final check can be performed after the gate level netlist is generated.

Knowledge of chip, block and sub block structure aids the user in writing VHDL code with appropriate hierarchy. In addition, hierarchical code facilitates physical design process and
Figure 3. An Example Chip Structure

reuse of code by another block/sub block. Figure 3 shows an example chip structure. The chip structure consists of top level chip structure TL and block BA, BB, BC and BD within the chip structure. Each block uses common sub block SB. Using this structure, the following VHDL code can be written:

```
--SUB Block code
--Library use statements
Entity SB is
  port (  
    -- Sub block inputs/outputs here
  );
end SB;

Architecture SD_BEH of SB is
--signal/attribute statements here
begin
  --code body here
end SB_BEH;

--Block A Code
--Library statements here
Entity BA is
  port (  
    --input/output statements here
  );
end BA;

Architecture RA_BEH of RA is
--signal/attributes here

Component SB
  port (  
    --subblock input/output here
  );
end component;
begin
  --code body here
  --Instantiate SUBBLOCK l_SB: SB
  port map (  
    --map input/outputs to the subblocks here
  );
end BLOCK_BEH;

--similarly BLOCKB, BLOCKC and BLOCKD code is written

-- Top level code begins
-- Library statements here
Entity TL is
  Port (  
    --top level input/output
  );
end TL;

Architecture TL_BEH of TL is
--signals/attributes here
Component BA
  port (  
    --Block A inputs/outputs here
  );
end BA;
```
Component BB
port (  
-- Block B inputs/outputs here
)
end BLOCKB;
Component BC
port (  
--Block C inputs/outputs here
);
end BC;
Component BD
port (  
--BLOCK D inputs/outputs here
);
end BD;
begin
--Top level code here
--Instantiate BLOCKA here
L_BA: BA
port map (  
--Block A input/output map
);
L_BB: BB
port map (  
--Block B input/output map
);
L_BC: BC
port map (  
--Block C input/output map
);
L_BD: BD
port map (  
--Block D input/output map
);
end TL_BEH;

The above coding style can be extended to any levels of hierarchy. In addition, hierarchical coding style aids in layout of a block which exhibits data flow. Figure 4 shows an example of logic which exhibits data flow properties. Again, in this case hierarchical coding style can be adopted to meet the data flow structure in the design.

In a microcontroller design, it is common to have bidirectional data bus. The tri-state buffer are inserted in a tri-state bus. For example, register file is typically read/written from bidirectional data bus. It is natural to put state read/write control statement within entity. If the register file consists of smaller data bits, it will result in inserting tri-state buffer for each data bit. Hence, this will generate state buffer for each register bit. The following example illustrates the problem:

--Register read code
--Library statements here
Entity REG is
Port (  
--Register input/outputs here
);
end REG;
Architecture REG_BEH of REG is
--signal/attribute statements here
begin
Process (CLK, READ, REG_DATA)
begin
if ((CLK'event and CLK = '1') and READ = '1' and DECOD = '1') then
DATA_BUS <= REG_DATA;

Figure 4. An Example of Logic Exhibiting Data Flow
else
DATA_BUS <= "ZZZZZZZ"
end if;
end process;
--Register write process here
end REG_BEH;

The above code register will result in the hardware shown in figure 5.

![Figure 5. Register Entity](image)

If this register entity is instantiated multiple times, it will generate tri-state buffers for each register bit as shown in figure 6. Hence, as shown in figure 6, for n 8-bit register structure, the code generates 8n tri-state buffers. To avoid this problem and generate 8 tri-state buffers for a n 8-bit register structure, write a common read process at the next level of hierarchy as shown below:

--Common read process example
--Library Statements here
Entity REG_FILE is
Port (  
 --Register file inputs/outputs here  
);
end REG_FILE;
Architecture REG_FILE_BEH of REG_FILE is
--signal/attribute statements here
Component REG --8 bit register
port (  
 --Register inputs  
);
end component;
begin
I_REGA: REG
port map (  
 --Register input/output map here  
);
I_REGB: REG
port map (  
 --Register input/output map here  
);
--Similarly instantiate all the registers
Process (READ, CLK)
begin
if ((CLK'event and CLK='1') and READ = '1' and DECODA = '1') then

![Figure 6. Multiple Register Instantiation](image)

end if;
end process;
end REG_BEH;
Figure 7. Multiple Register Instantiation Optimized

```
DATA_BUS <= REG_A_DATA;
elsif ((CLK'event and CLK='1') and READ = '1' and DECODEB = '1') then
  DATA_BUS <= REG_B_DATA;
--Similarly write elsif statements for other registers
else
  DATA_BUS <= "ZZZZZZZZZ";
end if;
end process;
end REG_FILE_BEH;
The above code generates a structure shown in figure 7:

Portability

The VHDL source code is generally not completely portable. The source file has to be changed to move from one synthesis tool to another. The following are reasons for not being able to port source code from one synthesis tool to another:
2. Specification of attributes to infer certain components.
Hence, while writing VHDL code, one has to keep the target synthesis tool in mind. For example, the following code can be used to infer a latch with asynchronous reset using Synopsys synthesis tool[2]:
--Library statements
Entity ASYNC_RESET_LATCH is
port (reset, clk, input: in std_logic;
data: out std_logic);
end ASYNC_RESET_LATCH;
architecture BEHAV of ASYNC_RESET_LATCH is
attribute async_set_reset of reset: signal is "true"
begins
process (reset, clk, input)
begin
if (reset = '1') then
  data <= '0';
elif (clk = '1') then
  data <= input;
end if;
end process;
end BEHAV;
The multiple asynchronous reset latches can be inferred for the above data bits using for generate statements as shown below:
--Library statements
Entity ASYNC_RESET_LATCH is
port (reset, clk, input: in std_logic;
data: out std_logic_vector(7 downto 0));
end ASYNC_RESET_LATCH;
architecture BEHAV of ASYNC_RESET_LATCH is
attribute async_set_reset of reset: signal is "true"
begins
GD: for D in 7 downto 0 generate
process (reset, clk, input)
begin
    if (reset = '1') then
        data <= '0';
    elsif (clk = '1') then
        data(D) <= input(D);
    end if;
end process;
end generate GD;
end BEHAV;

The above code does not work with Synopsys synthesis tool, since the data is a std_logic_vector. The correct inference will occur by replacing data with std_logic instead of std_logic_vector. The following example illustrates the concept:

Entity ASYNC_RESET_LATCH is
    port (reset, clk, input: in std_logic;
          data: out std_logic);
end ASYNC_RESET_LATCH;

architecture BEHAV of ASYNC_RESET_LATCH is
attribute async_set_reset of reset: signal is "true";
begin
    process (reset, clk, input)
    begin
        if (reset = '1') then
            data <= '0';
        elsif (clk = '1') then
            data <= input;
        end if;
    end process;
end BEHAV;

Entity REG is
    port (reset, clk: in std_logic;
          input_bus: in std_logic_vector(7 downto 0);
          data_bus: out std_logic_vector(7 downto 0));

architecture REG_BEHAV of REG is
    Component REG
    port (reset, clk, input: in std_logic;
          data: out std_logic);
    begin
        GD: for D in 7 downto 0 generate
            REG_GEN: REG
            port map (reset => reset,
                      clk => clk,
                      input => input(D),
                      data => data(D));
        end generate GD;
    end architecture REG_BEHAV;
end REG;

Hence, looking at the above code, one can see that the VHDL source can vary from one synthesis tool to another.

Performance Monitor

The modular nature of VHDL can be used to develop performance monitor. In a microcontroller development project, it is interesting to determine number of read/write cycles to the external memory. Consider a microcontroller with address, data and control signals. The control signals consist of:
1. Memory chip select signal CS.
2. Read/write signal.

Using these signals, one can write a model to determine number of read/write cycles. The following figure illustrates the application:

![Microcontroller Model Diagram]

**Figure 8. Microcontroller Model**
The code for the above configuration can be written as follows:

```vhdl
--Library statements here
Entity Perf_Monitor is
Port (clk, rw, cs, reset: in std_logic);
end Per_Monitor;
Architecture BEHAV Perf_Monitor is
signal program_end: boolean;
signal No_of_clk_cycles: integer;
signal No_of_read_cycles: integer;
signal No_of_write_cycles: integer;
file PERF_FILE: TEXT is out "path_name/file_name";
begin
    process (clk, cs, rw, reset)
    variable buf: LINE;
    begin
        --Program_end logic here
        if (reset = '1') then
            No_of_clk_cycles <= 0;
        elsif (clk'event and clk = '1') then
            No_of_clk_cycles <= No_of_clk_cycles + 1;
        end if;
    end process;
end architecture BEHAV;
end Perf_Monitor;
```

3.8
end if;
if (reset = '1') then
  No_of_read_cycles <= 0;
elsif (cs'event and cs = '0' and rw = '1') then
  No_of_read_cycles <= No_of_read_cycles +1;
end if;
if (reset = '1') then
  No_of_write_cycles <= 0;
elsif (cs'event and cs = '0' and rw = '1') then
  No_of_write_cycles <= No_of_write_cycles +1;
end if;
if (program_end) then
  write (buf, string("No. of Clock Cycles ="));
  write (buf, No_of_clk_cycles);
  writeln(PERF_FILE, buf);
  write (buf, string("No. of Read Cycles ="));
  write (buf, No_of_read_cycles);
  writeln (PERF_FILE, buf);
  write (buf, string("No. of Write Cycles ="));
  write (buf, No_of_write_cycles);
  writeln (PERF_FILE, buf);
end if;
end process;
end BEHAV;
Entity Microcontroller is
  Port (  
    --Microcontroller I/O here
  );
end Microcontroller;
Architecture MICRO_BEHAV of Microcontroller is
  Component Perf_Monitor
    port (clk, rw, cs, reset: in std_logic);
  end component;
  begin
    --Microcontroller code here
    IPP: Perf_Monitor
      port map (  
      clk => clk,
      cs => cs,
      rw => rw,
      reset => reset  
    );
  end MICRO_BEHAV;
In the above code, all the data pertaining to number of read/write cycles will be written in a file.

Conclusions
To develop a design methodology, a tool set has to be chosen to accomplish the overall project goals. In addition appropriate coding style has to be adopted to infer required hardware structures. Also, the source code one tool can be different from the other tool.

To make VHDL code portable from synthesis tool to another, an attempt should be made to standardize libraries. Also, the tool should be capable of inferring appropriate hardware without additional attributes in source code.

Acknowledgments
I would like to thank Pramod Arg, Technical Manager at AT&T Bell Labs reviewing this paper. I would also like to thank Jayaram Bhasker at AT&T Bell Labs for helping me with the VHDL coding.

References