Using HDL Coder for Complex Algorithm Deployment

Steve Hamilton, James Hui & Ian Brown
Image and signal processing in firmware

♦ Is complex!

♦ Requires significant algorithm re-structuring
  - Conversion from frame-based to serialised architecture

♦ Considerable data throughput
  - High data rates
  - High resolution sensors
  - Multiple sensor feeds processed by one device
  - Parallel processing chains

♦ Memory intensive
  - line & frame buffers

♦ Fixed-point considerations

♦ Requires supporting infrastructure
  - Sensor interfaces
  - External memory controller / interface
  - Communication peripherals
  - A System-On-Chip design philosophy
Traditional Workflow

- Algorithm specifications are passed to F/W developers at onset and this may be the only juncture of information exchange
  - Communication can be *ad hoc* – with no process steps to enforce review
  - Can lead to functional divergence between specification and implementation
  - Algorithm developers are often not involved after the specification is defined
    - Regardless of the suitability for FPGA deployment

- Quality of verification relies on F/W developers comprehension of the algorithm
  - Co-simulation can trap differences, but is dependant on the quality of the test vectors and how lengthy the time is to execute the simulation
  - It is difficult for F/W developers to determine the functional correctness of low level units

- Algorithm bug fixes / updates are only applied to the F/W
  - This decouples the implementation from the specification
  - Another source of divergence
  - Compromises traceability to specification

- Estimating process can be difficult
  - It is often difficult for F/W developers to estimate the implementation effort for complex algorithms
HDL Coder based workflow
Algorithm specification is fundamental

- Specification consists of
  - Algorithm Description Document (functionality & rationale)
  - Executable Reference Specification (MATLAB / C++)
  - Test vectors / expected outputs
- Should be explicit and detailed
- Modifications to the specification must be handled formally
- Must be reviewed from a firmware/hardware implementation perspective
  - Challenge the specification if it seems excessive / unnecessary!

The interaction between the algorithm and infrastructure firmware is critical

- The processing timeline and clocking scheme must be designed to meet the requirements of algorithms, infrastructure and the targeted device
- Interface definitions
- Memory access patterns must be considered

Careful consideration must be given to partitioning of functionality

- Algorithmic functionality between FPGA and other processors
  - Not all algorithmic operations are well suited to firmware
- Between algorithms firmware and infrastructure firmware
Algorithm implementation stages

- **Reference algorithm implementation**: MATLAB, floating-point, toolbox calls, frame-based, no frame blanking.
- **Functional representation**: Simulink, floating-point, code generation compatible, frame-based, frame blanking.
- **Serialized implementation**: Simulink, floating-point, primitive operations, serialized, frame blanking.
- **Target-optimised implementation**: Simulink, fixed-point, primitive operations, serialized, frame blanking.
- **VHDL implementation**: VHDL, fixed-point, auto-generated, serialized, frame blanking.

Process breakdown (effort)

- **Requirements Capture**
- **Code-generation**
- **Implementation**
- **Verification**

Detailed development & verification process

[Diagram showing the process breakdown with detailed steps and timelines for each phase.]
Algorithm implementation process stages

Reference algorithm implementation:
- MATLAB
- floating-point
- toolbox calls
- frame-based
- no frame blanking

Functional representation:
- Simulink
- floating-point
- code generation compatible
- frame-based
- frame blanking

Serialised implementation:
- Simulink
- floating-point
- primitive operations
- serialised
- frame blanking

Target-optimised implementation:
- Simulink
- fixed-point
- primitive operations
- serialised
- frame blanking

VHDL implementation:
- VHDL
- fixed-point
- auto-generated
- serialised
- frame blanking
Detailed development & verification process
Approximate relative effort
Collaboration between engineering disciplines is critical

- This process requires a combination of Algorithms, Firmware/Electronics and Systems Engineering expertise and resource
  - Effective communication is key to success
  - Establish a common language
  - Partition tasks appropriately (play to existing strengths while developing capability)

- The diagram below outlines a typical breakdown of effort between engineering disciplines

The importance of firmware domain expert consultancy through the process cannot be overestimated!
The Tools
Generates VHDL (or Verilog) from Simulink (and/or MATLAB)

- Algorithms are expressed at a similar level as in hand-coded VHDL from library blocks
  - Arithmetic, logical and relational operators
  - Counters, registers, shift-register, RAM blocks
- Allows maintenance of IP in the modelling domain
- MATLAB / Simulink integration
  - MATLAB function blocks can be used inside Simulink
    - It is often easier or more intuitive to express logic using M-code
    - Care must be taken regarding coding style to ensure efficient generated HDL!
- Code generation is a small automatic part of an otherwise manual process
  - Various tools in MATLAB/Simulink can help with other stages
    - Fixed-Point Advisor
    - Verification/debug blocks excluded from code-generation
- Integrates well with external firmware tools to streamline synthesis and resource/timing analysis
  - This is very helpful and encourages users to synthesise generated code on a module-by-module basis
HDL Verifier is a key component of our verification workflow

- **Modelsim co-simulation**
  - Generates a co-simulation model and all of the scripts required to coordinate simulation of auto-generated HDL in an external simulator
  - Co-simulation is a critical component in our workflow
    - Verifies the generated HDL against the Simulink implementation

- **FPGA-in-the-loop**
  - Can accelerate throughput of test cases by hardware acceleration
    - Though performance is limited by any co-simulation in Simulink
  - Capability limited by interface considerations
  - Limited support for current Ethernet protocols (restricts FPGA board selection)
  - Successfully used to deploy algorithmic modules to hardware for confidence testing

- **‘Black-boxing’ existing VHDL**
  - Allows hand-coded VHDL to be integrated with Simulink models for code generation
  - A behavioural model is created in Simulink, but no code is generated for the black-box
  - R2013a has poor support for VHDL Generic lists (requiring workaround)
Case studies
In-place bubble sort

- A timing bottleneck was alleviated by changing the output behaviour of the algorithm (see diagrams)
  - The selectable output was replaced by shift-register like behaviour outputting from only one register
  - This removed a large multiplexer simplifying routing
  - Very rapid turnaround for this modification

*Analyse the output of synthesis tools and optimise the model accordingly*

Parallel bubble sort

- A pipelined parallel bubble sort was implemented using HDL Coder and compared with an optimised hand-coded VHDL implementation
  - The post-synthesis resource utilisations were identical
  - No bloat was introduced by auto-coding process
  - While this is a simple algorithm it is an encouraging result!
**Coding style is key to efficient HDL**

- **Persistent variables must be initialised with ‘if (isempty(…))’**
  - This is a compile-time operation, there should be no trace of this in the schematic
  - If this isempty(…) is logically OR’d with a signal as shown in the 1st listing an extra register and logic gate are produced

- **Ensuring that state initialisation is separate from algorithmic reset logic avoids this extra logic**
  - Otherwise, the code listings below are functionally equivalent
  - Over a large design this may result in a significant resource saving

```matlab
function output = func_combined_reset(input, reset)
    %#codegen
    persistent prev_val; % Declare memory element
    % Use a common reset clause for system reset and soft reset
    if (isempty(prev_val) || reset)
        % Define initial value (and class) of memory element
        prev_val = uint8(0);
    end
    output = prev_val; % Output registered value
    prev_val = input; % Update register value for next cycle
end
```

```matlab
function output = func_separate_reset(input, reset)
    %#codegen
    persistent prev_val; % Declare memory element
    % Clause for system reset
    if (isempty(prev_val))
        % Define initial value (and class) of memory element
        prev_val = uint8(0);
    end
    % Separate clause for soft reset
    if (reset)
        prev_val = uint8(0); % Sub-optimal replication of in 0
    end
    output = prev_val; % Output registered value
    prev_val = input; % Update register value for next cycle
end
```
Our experiences
General observations

- **Expectation management**
  - Auto-coders will not generally be as efficient as hand-coded optimised implementations
    - They aim to improve the development process, not the efficiency of the HDL
  - Some are quick to judge workflows of this sort without sufficient understanding of the process

- **Adopting this workflow is enabling, but disruptive**
  - Enables Algorithms / Systems Engineers to contribute to algorithm deployment more effectively
  - Enables larger more complex algorithmic systems to be deployed on FPGA
  - May change the focus of Firmware Engineers effort

- **This process moves a complex subset of hardware design into modelling domain**

- **Simulink / HDL Coder tools are permissive**
  - Care must be taken to avoid overly naïve modelling of algorithm constructs
  - This is the responsibility of the modeller – not the tools!

- **Generated HDL is neat and well structured**

- **There is a steep learning curve due to the number of process steps and tools involved**
  - Don’t expect massive improvements in delivery time on the first attempt!
HDL Coder workflow benefits

- **Graphical design entry in Simulink**
  - Helps visualise the processing architecture

- **Integrates well with existing firmware design tools**
  - Simplifies obtaining resource and timing estimates
  - Facilitates monitoring and early reporting of device utilisation

- **Complements existing ‘traditional’ Firmware design workflow**
  - The generated VHDL integrates well with hand-coded firmware
  - There is no need for an ‘all or nothing’ approach to HDL autocoding!

- **Rapid turnaround on minor design iterations**

- **A rich suite of verification & visualisation modules are available**
  - Developers can easily create and integrate bespoke debugging tools for simulation
    - I.e. frame synchronous image displays, RAM content monitors, data logging etc.

- **Enables more task-parallelism in Firmware development**
  - Algorithm Engineers can focus on algorithm implementation while Firmware Engineers concentrate on architecture design, sensor & memory interfaces, data management etc.
  - Remember that Firmware consultancy / supervision is required throughout the process
Limitations of HDL Coder (Alg Eng/User perspective)

- **MATLAB to HDL workflow is less mature and stable than Simulink to HDL**
  - We tried this flow for various algorithms and encountered several failures

- **Error messages frequently fail to reach user in large complex models**
  - Sometimes errors such as ‘Code generation failed due to above error’ would be shown in dialog windows
  - Generating code on a module-by-module basis minimised these occurrences but became tedious and time-consuming
  - Could a verbose text log file be provided in the event of failure / crash?

- **Launching external firmware tool processes freezes Simulink**
  - Can we tell HDL Workflow Advisor that we have performed synthesis etc. externally?

- **Some instabilities in recent releases (R2012b, R2013a)**
  - Opening documentation frequently closes Simulink

- **Support for reusing library subsystems is a little clunky**
  - MATLAB & Simulink do this well but there are limitations for code-generation

- **Support for Simulink buses and matrices is limited (particularly on interfaces)**

- **Standard maths functions are code-gen compatible, but perform poorly**
  - We were unable to distribute registers through division & square-root operations and black-boxed existing VHDL IP

- **Occasionally fails to recognise equivalent data types**
  - Boolean / ufix1, uint8 / ufix8

- **Documentation for some HDL Coder options/_settings is a little sparse**
HDL Coder limitations (Firmware Eng perspective)

- **Hidden hardware signals (clock_enable & reset)**
  - Don’t appear in Simulink model and control of them is minimal
  - Prevents some optimisation techniques which would be implemented by hand
  - Often become heavily loaded resulting in timing bottlenecks

- **Generated code is very sensitive to coding style**
  - Great care is required in MATLAB function blocks (in Simulink)
  - It is important to understand the implications of various coding constructs for HDL generation

- **Default behaviour for registers is to include a reset pin**
  - This is not clear from the Simulink model (see first bullet)
  - Can prevent SRLC’s being inferred for shift registers in the FW tools resulting in increased resource utilisation

- **No obvious warnings are provided when accidentally including non-synthesisisable elements in a model**
  - Forgetting to change the data type of a constant from ‘double’ can waste quite a bit of time!

- **It isn’t currently possible to attach VHDL attributes to signals**
  - This complicates optimisations such as signal preservation

- **No clear way to preset contents of RAM modules**
Auto-coding automates one part of a complex process
- This should be viewed as a development & verification process – not a big red button!

Our experience has been quite positive overall
- However we still have much to learn

We have never had any issues with the functional correctness of the generated VHDL
- Algorithms considered are complex image and signal processing functions
- Our design iterations were typically driven by timing and/or area optimisation

Significant investment is required to become productive
- The correct mix of complimentary skills are key
- Initially this workflow should be used as complimentary to existing development processes
- It takes time to become familiar with the nuances of HDL Coder to avoid common pitfalls

We used Mathworks consultancy throughout our ramp-up phase
- It was very useful – but the cost should be factored in to the decision to adopt these tools

Communication between Algorithms and Firmware Engineers has been improved by this workflow
- This has effectively been accomplished by hybrid engineers acting as conduits between Algorithms and Firmware teams