Software Studio for Indigenous Processor Powered Data Acquisition and Signal Processing System

Syamlal L.S.,
Avionics
Vikram Sarabhai Space Centre
Trivandrum, India
syamlal_ls@vssc.gov.in

Abey Rose J.,
Avionics
Vikram Sarabhai Space Centre
Trivandrum, India
abey_rose@vssc.gov.in

Anjana S.J.,
Avionics,
Vikram Sarabhai Space Centre
Trivandrum, India
sj_anjana@vssc.gov.in

Jayalekshmy L.,
Avionics
Vikram Sarabhai Space Centre
Trivandrum, India
l_jayalekshmy@vssc.gov.in

doi: https://doi.org/10.37285/bsp.SACAD2025.40

***Abstract*—An indigenous processor powered data acquisition and signal processing system (IPDS) is being developed as an ASIC catering to different avionics applications. Multiple functional modules with analog interfaces and digital configurability are included in the system designed for applications such as data acquisition, digital signal processing, control actuation and telemetry.**

Software development for IPDS requires support for different hardware modules, from a high level programming language. The software studio for IPDS is conceived around specifying hardware configuration for the functional modules declaratively in a domain specific language (DSL). These specifications are translated to assembly programs for configuring, operating and monitoring various hardware modules. A DSL Compiler has been designed and implemented to perform this translation. An indigenously developed system software tool suite comprising Cross Compilers, Software Integrator, Assembler, Linker and Simulators is already operational in ISRO, for the 32-bit indigenous processor module within IPDS. This toolset will convert the generated IPDS peripheral libraries to machine code.

The peripheral libraries produced by the DSL Compiler have been extensively used for the hardware development and validations. A graphical user interface (GUI) based Integrated Development Environment (IDE) is also developed, with focus on the formulation of hardware specification. It facilitates configuring each hardware module and specifying the flow of data and control across modules.

This paper describes the software ecosystem for IPDS, formulation of DSL, architecture of DSL Compiler and development of IDE. The validations performed on the software development tools and future scope are also included.

***Keywords*—DSL, Compiler, IDE, IPDS**

I. Introduction

IPDS is a mixed signal ASIC catering to different avionics requirements. Various analog modules for data acquisition, filtering, sampling, digital signal processing, communication etc. are provided with software configurability. An indigenous 32-bit processor core is also included in the ASIC. The software interface is provided as a set of configuration registers which are memory mapped to the processor address space. A block schematic of IPDS is shown in Fig. 1.

IPDS is designed to acquire inputs, process those and transmit outputs in a seamless manner with minimal involvement of processor. For onboard applications, after initiating the various peripheral operations, the processor can go ahead with other Navigation, Guidance and Control (NGC) computations, with periodic status monitoring of the functional modules as required.

II. Related Works

In the past, an indigenous microcontroller development project was initiated, for which, the software ecosystem was designed as an extension to the Cross Compiler and System Library. Compiler-specific constructs were developed in the high level language and System Library routines were developed in assembly language. The programming model was an imperative one. For IPDS, a different approach with a declarative DSL is chosen, which simplifies the application software development. Even though the final generated peripheral library is a procedural program, the DSL allowed the application developer to specify the mode of operation of each hardware module as higher level specifications.

Analog Device's CodeFusion Studio is a recent project which also supports programming hardware modules. This was developed as extensions in the VSCode IDE [5]. The peripheral level and register level configurations were supported. Hardware pin configurations were also controllable [6]. This work served as an inspiration and reference for IPDS IDE development. Unlike CodeFusion Studio, IPDS IDE is developed as standalone software, not as extensions. However, it supports similar features.

Microchip's MPLAB Artificial Intelligence (AI) coding assistant is another project which is also provided as an extension to VSCode IDE [5]. It makes use of proprietary Large Language Model (LLM) to assist in the code generation [3, 4]. The relevant information is retrieved from multiple design documents in natural language [7]. This approach seems to be apt for the IPDS IDE, as it also involves a large number of modules with different configurations

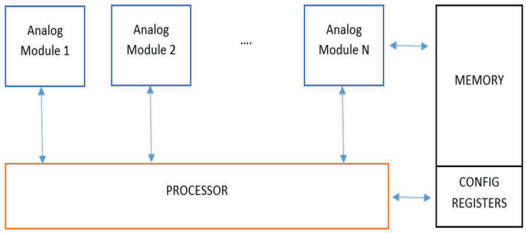

Fig. 1. Block Schematic of IPDS

III. Design and Implementation

The software development for onboard computers is done conventionally using imperative programming methodologies. The other paradigms like functional programming, object oriented programming etc. are not very popular, especially when dealing with hardware peripherals. For IPDS, auto-code generation is envisaged by specifying the configuration of different hardware modules. The configuration of different hardware modules encompasses assigning specific values, in a pre-determined sequence. Our design is based on the vision that this can be implemented through a declarative DSL. The overall architecture of IPDS software platform is shown in Fig. 2

There are 21 hardware modules, 197 register types and 880 field types in IPDS. Since the software development scheme is similar for different modules, code generation was automated using "Django Templates" [12]. The hardware design documents in Microsoft Office Word format is parsed and stored into a database. This is used to generate the specification of DSL [1,2], code for DSL compiler, IDE, memory map and User Manual. User Manual is generated in Markdown format, with hardware module overviews, configuration register details, input DSL specifications, illustrations and output assembly programs.

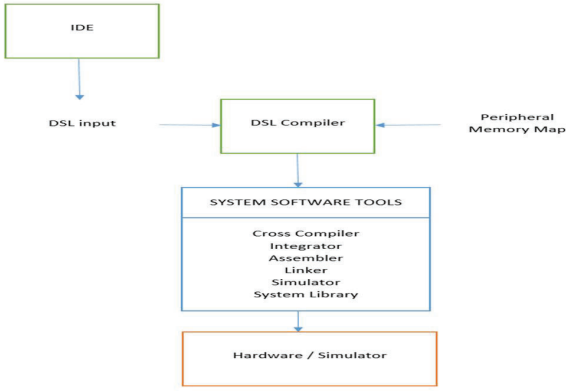

Fig. 2. IPDS Software Platform Architecture

As the hardware design evolved, this design approach proved to be useful in seamlessly incorporating the hardware design changes. It helps in scaling up the platform to any number of modules or registers. This design strategy enables inclusion of any additional hardware modules which may be designed in future too. By separating memory mapping of the configuration registers to a separate memory map file, the physical memory configurations could be modified easily without affecting the user programs and DSL compiler implementation.

In the current design, modules are developed for analog data acquisition (AAQ), comparator (AVGCOMP), digital filters (PDF), Digital to Analog Convertors (DAC), Fast Fourier Transform (FFT), Relay Drivers (RDIF), Serial Peripheral Interface (SPI), Universal Asynchronous Receiver Transmitter (UART), Stepper Motor Controller (STP_MTR), Chip Debug module (DEBUG), Cyclic Redundancy Checks (CRC), Sensor Excitation Generator (PWMEXG), Capture Compare Module (CMPWM), Timer module (TIMER), Digital I/O (DIO), Event Acquisition (EVACQ), Storage module (TRG), Quad-SPI module for interfacing Flash devices (QSPI) and Auxiliary module (TOP_AUX). The DSL specification for each module is designed as per the specific domain.

Indigenous system software tools for the processor core, which are already operational in ISRO, will convert the DSL compiler output to the machine code for executing in the processor core. The DSL compiler and IDE are designed to be compatible with this toolset.

A. DSL Specification

The DSL specification for each configuration registers within each hardware module is predefined. A unique name is assigned to hardware modules, configuration registers and fields in each configuration register. Register names are defined to start with "$" character and the field names are defined to start with "#" character

For each module, there might be multiple instances. Each instance is given a unique integer value. This is suffixed with the configuration register name. Each configuration register might occupy multiple memory locations. This is suffixed with the field name. A sample DSL specification for FFT module is shown in Fig. 3.

```
` Configuration Register in FFT Module
$FFT_CONFIG_REG
` Mode – 0 for FFT & 1 for IFFT
#MDOE 0
` Number of points in FFT (1024/512/256/128)
#POINTS 256
` Number of channels (1 to 24)
#NCHANNELS 4
` Averaged outputs (0) or Complex outputs (1)
#OUTMODE 0
` Count for averaging (1 to 32) the output
#AVGCOUNT 20
```

Fig. 3. DSL specification for FFT module

For each field, a value can be assigned. It can be either an integer or real type, whose range is defined as per hardware design document. Categorical values are assigned a unique numerical value in the DSL specification. The human understandable descriptive names are supported in the IDE, which will generate the equivalent numerical values.

Single line comments are also supported. The characters subsequent to single quote character are treated as comments. The tokens and grammar of DSL specifications are chosen so as to simplify the DSL compiler implementation. The parsing & generation of DSL specification is also supported in the IDE.

B. DSL Compiler

The DSL compiler translates the hardware specification into assembly language routines to configure, operate and monitor different hardware modules. It is developed in C++ language. The memory mapped register locations are used to generate the assembly routines. Validation of values permitted for each field and packing of different fields are also handled in the DSL compiler. Lexical analysis, syntax analysis, semantic analysis and template based code generation are the different phases in DSL compiler implementation.

```
.NAME FFT_CONFIGURE
    LOAD    A, [FFT_CFG_VAL]
    STORE   [ADDR+0], A
    LOAD    A, [FFT_WINDOW_COEFF]
    STORE   [ADDR+1], A
    LOAD    A, [FFT_IP_SEL_VAL]
    STORE   [ADDR+2], A
    RETURN
    CONSTANT ADDR, H`10000`
    CONST    FFT_CFG_VAL H`0420`
    CONST    FFT_WWINDOW_COEFF f`2.0`
    CONST    FFT_IP_SEL_VAL H`0420`
.END
.NAME FFT_START
    LOAD      A, 1
    BIT_SET   [ADDR], A
    RETURN
    CONSTANT ADDR, H`10020`
.END
.NAME FFT_STOP
    LOAD      A, 1
    BIT_RESET [ADDR], A
    RETURN
    CONSTANT ADDR, H`10020`
.END
.NAME FFT_STATUS
    LOAD    [ADDR], A
    RETURN
    CONSTANT ADDR, H`10021`
END
```

The generated assembly code for the same DSL specification is shown in Fig. 4. Assembly routines to configure the FFT module, start the FFT operation, stop the operation and read the status, are shown here. It also generates routines to populate the input buffer and read from the output buffer. The configurability offered by the DSL varies as per the hardware module, as illustrated in Table I.

Typically, the outputs from a previous module within IPDS will be fed as inputs to the next module. Nevertheless, input/output routines are provided for all modules to enable the application software interventions. Status Registers available for module interfaces and those status registers internal to each module are both supported by the DSL Compiler.

The code generation for certain modules like digital filters is comparatively more challenging that the other modules. The filter hardware design in IPDS follows a different scheme from the rest of the modules. In this case the configuration registers are not accessible from the processor memory map. A custom protocol needs to be implemented for reading and writing to filter internal memory.

The output of DSL compiler is used as a peripheral library, which can be used in the application software modules. A driver program is required to be developed by the user to invoke these library routines.

C. IDE

The IDE provides a Graphical User Interface for IPDS software development. The IDE allows the user to create, edit, list or delete any register specification. For each field, meaningful descriptions understandable by the user are displayed and validations of input values are supported. This will be automatically converted to DSL.

Functionalities such as search, sort and pagination for organizing content into separate pages etc. are also provided. The overall architecture and different use cases are depicted graphically too. The configuration of each hardware module can be exported in DSL format. Similarly, DSL format inputs can be imported to the IDE.

TABLE I. CONFIGURABLE FEATURES IN DSL - EXAMPLE

Module Name	Features supported
FFT	Forward & Inverse FFT computation
	Real & complex outputs
	Configurable number /selection of input channels
	Averaging of outputs
	Scaling & normalization of output values
Digital Filter	Configurable filter stages
	Configurable decimation rates
	Programmable gain
	Integer or fixed point operation etc.

The IDE is developed using "React JS" framework [8]. The state management is implemented using "Redux Toolkit" [10], styling is performed using "Bootstrap" framework [11], bundling is done using "Electron JS" [14], routing is implemented with "React Router" [9] and data flow and control flow visualizations are implemented using "xyflow" [13]. Sample screenshots of IPDS IDE are shown in Fig. 5 and Fig. 6.

In addition to the generic functionalities detailed earlier, specific functionalities are also provided. For Analog Acquisition Module, the hardware provides a schedule in terms of time delays, whereas the users commonly specify the requirements in terms of sampling frequency. An algorithm required for the conversion is also implemented as part of the software platform. For filter module, the required filter coefficient generation is also supported by interfacing with Matlab libraries.

D. Application Deployment

The DSL compiler can be deployed independently as a command line executable. It can also be embedded in the IDE, which is developed using Web technologies. IDE is also bundled and distributed as a GUI executable, which can invoke the DSL compiler and other system software tools. Electron JS makes use of Chromium web rendering engine to convert the web based GUI to standalone executable [14].

IV. Future Scope

In future, the DSL specification and DSL compiler can be extended to provide a runtime environment in IPDS. This implies that when a hardware module needs to be dynamically reconfigured at runtime, the platform should autonomously support changes in its operating modes. This becomes challenging since DSL compiler would have generated code for the operation modes of the target hardware prior to the start of application execution. Even though such requirements might be rare in the NGC domain, such a feature could be useful for highly dynamic applications.

AI-assisted code generation using LLM is also planned in the IDE, wherein the LLM model has to generate correct DSL outputs based on the natural language inputs. The custom syntax and semantic information needs to be considered for such code generation. The appropriate design documents require to be referred for retrieving the hardware specifications. Approaches like Retrieval Augmented Generation (RAG) are to employed. The relevant hardware specification needs to be fed in such scenario via a prompt. The generated outputs shall also be structured as per DSL specification. Invocation of software development tools in Agentic workflow can also be considered further.

Support for on-chip debugging of the applications through built-in DEBUG module and UART interface can also be included in future release of the product. The processor states like the execution status, internal register values, memory content etc. can be read or written into using the studio, with the help of hardware debugging features.

V. Conclusion

Software studio for IPDS is designed and developed in-house with a declarative, DSL specification. A DSL compiler is also developed, reviewed and verified. The generated peripheral libraries can be translated to machine code using processor system software toolset. This was used for validating the implementation of hardware modules in VHDL simulator before delivering the design to the semiconductor fabrication agency. An IDE was also developed in parallel to help with the generation of DSL from hardware specification.

The IPDS is currently being fabricated and is planned to be used for future Avionics system designs. The software platform will be useful in the software development for such systems, where users need not re-invent the wheel and may instead focus on the real system design. Auto code generation for the IPDS peripherals by the DSL compiler reduces the development time and minimizes errors.

Acknowledgment

The authors gratefully acknowledge all their colleagues and management, for their encouragements, suggestions and support in this activity. The authors also wish to thank Deputy Director, VSSC (Avionics) and Director, VSSC for giving permission to publish this article.

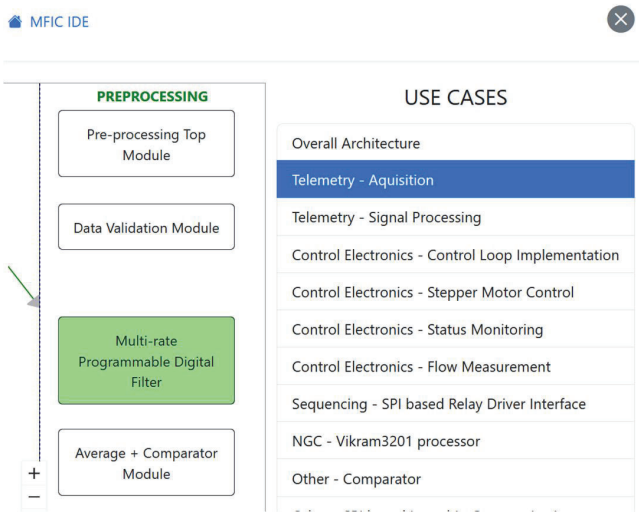

Fig. 5. IPDS IDE Home Page

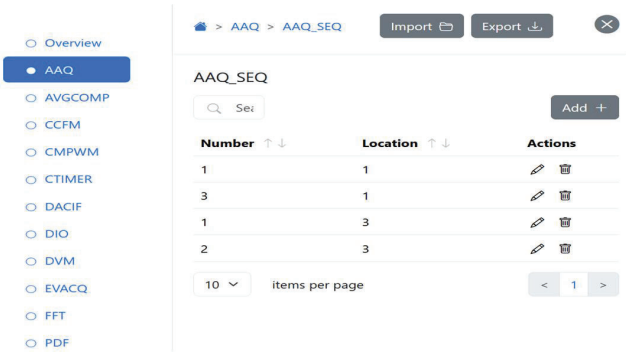

Fig. 6. IPDS IDE Screen for Analog Acquisition Module

REFERENCES

[1] Marjan Mernik, Jan Heering, and Anthony M. Sloane, "When and how to develop domain-specific languages", ACM Computing Surveys, 37(4):316–344, 2005.doi:10.1145/1118890.1118892

[2] Diomidis Spinellis, "Notable design patterns for domain specific languages", Journal of Systems and Software, 56(1):91–99, February 2001. doi:10.1016/S0164-1212(00)00089-3

[3] Vaswani Ashish, Shazeer Noam, Parmar Niki, Uszkoreit Jakob, Jones Llion, Gomez Aidan N, Kaiser Łukasz, Polosukhin Illia, "Attention is All you Need", Advances in Neural Information Processing Systems

[4] Brown Tom B., Mann Benjamin et.al., "Language Models are Few-Shot Learners", Advances in Neural Information Processing Systems

[5] VSCode [Online]. Available: https://code.visualstudio.com/

[6] Analog Device's CodeFusion Studio [Online]. Available: https://www.analog.com/en/resources/evaluation-hardware-and-software/embedded-development-software/codefusion-studio.html

[7] Microchip's MPLAB AI coding assistant [Online]. Available: https://developerhelp.microchip.com/xwiki/bin/view/software-tools/ides/extensions/ai-coding-assistant/

[8] React JS [Online]. Available: https://react.dev/

[9] React Router [Online]. Available: https://reactrouter.com/

[10] Redux Toolkit [Online]. Available: https://redux-toolkit.js.org/

[11] Bootstrap [Online]. Available: https://getbootstrap.com/

[12] Django [Online]. Available: https://www.djangoproject.com/

[13] xyflow [Online]. Available: https://xyflow.com

Electron JS [Online[. Available: https://www.electronjs.org/