RH(+): THE MODEL FOR HIGH-LEVEL EMBEDDED SYSTEM DESIGN ON RUN-TIME RECONFIGURABLE HARDWARE

by

Bayram Kurumahmut

B.S., Computer Engineering, Istanbul Technical University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2007

ACKNOWLEDGEMENTS

Firstly, I would like to thank Assoc. Prof. Arda Yurdakul for her supervising the thesis work and her guidance to get the best solutions. Also, I would like to thank Prof. Oğuz Tosun and Assist. Prof. Feza Buzluca for their attendance to the examining commitee and their feedback.

Secondly, I want to thank my family for their support in all steps of my life as in the thesis work. I want to thank my father, Selahattin Kurumahmut, for his insistence me on applying Boğaziçi University to start master program. I thank my mother, Rukiye Kurumahmut, for her endless affection to me. Hakkı Kurumahmut, who is my brother, helped in many steps during the thesis. I want to thank him for his relation about the details of the thesis. Samet Kurumahmut is my little brother, and I want to thank him for giving me moral support during the thesis.

Thirdly, I want to thank Reyhan Aydoğan and Mehmet Yunus Dönmez for their format review of the thesis.

Lastly, and especially, I want to thank Hande Zırtıloğlu for her support in every steps of the thesis work. She helped me for many time consuming jobs in the thesis like reviews. She is the most important person who has helped me to bring the thesis up against the thesis deadlines.

ABSTRACT

RH(+): THE MODEL FOR HIGH-LEVEL EMBEDDED SYSTEM DESIGN ON RUN-TIME RECONFIGURABLE HARDWARE

The process of embedded system design on reconfigurable architectures needs smart solutions to reduce the cost of development life-cycle and to use resources efficiently at run-time. However, the current solutions (SystemC/xtUML), which are extended from the traditional languages (C++/UML), are insufficient for that. Inefficiency occurs due to: detailed operator definition requirement, forcing user to pay attention low-level design problems at higher levels, complex hardware abstraction procedures, misguiding user during mapping software to hardware, not permitting user to define constraints at the level having software intermediate representation, and outputs lacking of performance from high levels to lower levels. Therefore, the traditional methods must only be used for what they are designed, in order to benefit from them efficiently.

In this thesis, we propose: (1) RH(+); a brand new high level embedded system design model for run-time reconfigurable architectures, solving the aforementioned inefficiency problems, (2) LRH(+); a brand new design language which is not extended from any traditional languages, (3) FRH(+); the framework meeting RH(+) requirements. In our work, we have the tools for developing board support package, defining miscellaneous operators, generating graphs for user interactions, profiling, resource scheduling, finding possible paths with their execution delays, and run-time emulation of reconfigurable hardware.

ÖZET

RH(+): ÇALIŞMA ANINDA YENİDEN BETİMLENEBİLEN DONANIMLAR ÜZERİNDE YÜKSEK SEVİYEDE SİSTEM TASARIMI MODELİ

Yeniden betimlenebilen mimariler üzerinde gömülü sistem tasarımı süreci, geliştirme süresindeki giderleri azaltmak için ve çalışma anında kaynakları verimli kullanabilmek için akıllı çözümlere ihtiyaç duymaktadır. Fakat, varolan dillerin (C++/UML) uzantısı olan şu anki çözümler (SystemC/xtUML) bunun için yeterli değildir. Oluşan verimsizliğin sebepleri şunlardır: duyulan ayrıntılı operatör tanımı gereksinimi, yüksek seviyelerde kullanıcıyı alt seviye tasarım problemleriyle karşı karşıya bırakmak, karışık donanım soyutlama yöntemleri, yazılımın donanıma atanması sırasında kullanıcıyı yanlış yönlendirme, yazılımın ara gösterimi üzerinde kullanıcının kısıtlar girmesine izin vermeme, ve yüksek seviyeden alt seviyeye başarım eksikliği olan çıkışlar üretme. Bu nedenlerden ötürü, varolan yöntemlerlerden verimli bir şekilde yararlanmak için bu yöntemler sadece tasarlandıkları amaçlarla kullanılmalıdırlar.

Bu tez çalışmasında, şunları önermekteyiz: (1) RH(+); çalışma anında yeniden betimlenebilen mimariler için sözü edilen verimsizlik problemlerini aşan yeni marka bir yüksek seviyede gömülü sistem tasarımı modeli, (2) LRH(+); varolan dillerin bir uzantısı olmayan yeni marka bir tasarım dili, (3) FRH(+); RH(+) gereksinimlerini karşılayan çerçeve. Çalışmamızda, hedef mimari destek paketi geliştirmek, çeşitli operatörler tanımlamak, kullanıcı etkilişimli grafikler üretmek, yazılım profili çıkarmak, kaynak yönetimi sağlamak, ve çalışma anı öykünüm için araçlar bulunmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADC	Analog-to-Digital Converter
ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
BB	Basic Block
BSP	Board Support Package
CAN	Controller Area Network
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CH	Custom Hardware
CLB	Configurable Logic Block
CRC	Cyclic Redundancy Check
DFG	Data Flow Graph
DMA	Direct Memory Access
DSM	Deep-Sub-Micron Technology
EEPROM	Electronically Erasable Programmable Read Only Memory
FFT	Fast Fourier Transform
FLASH	Electrically Erasable Reprogrammable Nonvolatile Memory
FPGA	Field-Programmable Gate Array
FRH(+)	Framework for RH(+) Model
GUI	Graphical User Interface
HLS	High Level System
HW	Hardware
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IR	Intermediate Representation
ISR	Interrupt Service Routine
I/O	Input/Output
LCD	Liquid Crystal Display
LED	Light Emitting Diode

LIN	Local Interconnect Network
LISA	The Language of Instruction Set Architecture
LLS	Low Level System
LRH(+)	Language for RH(+) Model
MD5	Message-Digest Algorithm 5
MIMD	Multiple Instruction Multiple Data
MIMOLA	Machine Independent Microprogramming Language
MISD	Multiple Instruction Single Data
NEC	NEC Electronics Corporation
nML	not Machine Language
NoC	Network on Chip
OPDEF	Operator Definition
PIM	Platform Independent Model
PSM	Platform Specific Model
PWM	Pulse Width Modulation
RAM	Random Access Memory
RH(+)	Hardware Software Co-design on Reconfigurable Hardware
ROM	Read Only Memory
RTL	Register Transfer Level
SDL	Specification and Description Language
SHA1	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SW	Software
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
xtUML	Executable and Translatable Unified Modeling Language
xUML	Executable Unified Modeling Language

1. INTRODUCTION

In this thesis, we consider high level embedded system design solutions in the literature and commercial area. However, to be able to understand why we focus on this area, we must understand the historical development of the embedded systems architecture. Thus, in this chapter, we explain this architectural progress at first. Then, we introduce run-time reconfiguration methods. After that, we list the causes of our motivation. The last section has the organization of thesis book.

1.1. Embedded System Target Architecture History

Embedded systems target architecture has a variable structure due to improvements added to the current deep-sub-micron (DSM) technology [1]. Figure 1.1 shows architectural development in timeline. Three types of architectures are seen: nonconfigurable, design-time reconfigurable, and run-time reconfigurable.

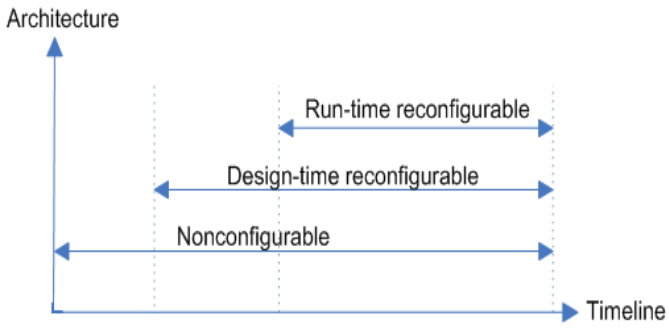


Figure 1.1. Architectural development

The nonconfigurable architectures, which include only the static elements like hard processor, packaged memory, buses, etc. are mounted to their surface at fabrication time. Therefore, when target architecture to be used in a project is decided, the decision to change it later in the project is much more costly. For example, if the designer selects Local Interconnect Network (LIN) bus [2] to connect the boards, each of which has its own microcontroller and circuit, he must buy the microcontrollers from the family supporting LIN to be able to design the system. After a period of

time, the designer may need a more secure and faster bus, so selects Controller Area Network (CAN) [3] whose nodes are his boards. In the above scenario, the designer should buy microcontrollers from the family supporting CAN to be able to design this system. However, this leads to inevitable costs in terms of time and money. Therefore, the tendency from the traditional methodology to the configurable architectures [4] has appeared thanks to the encouragement that the developed DSM technology gives.

The design-time configurable architectures can solve these kind of problems as given by the aforementioned scenario. These architectures can be configured at design-time. Because they can be reconfigured after configured once, they are called as (design-time) reconfigurable architectures in the literature [4]. The remaining steps after this reconfiguration are similar to nonconfigurable ones. Moreover, reconfiguration at design time provides the designers with rapid prototyping of the system by variable architecture models on the same board, shown in Figure 1.2.

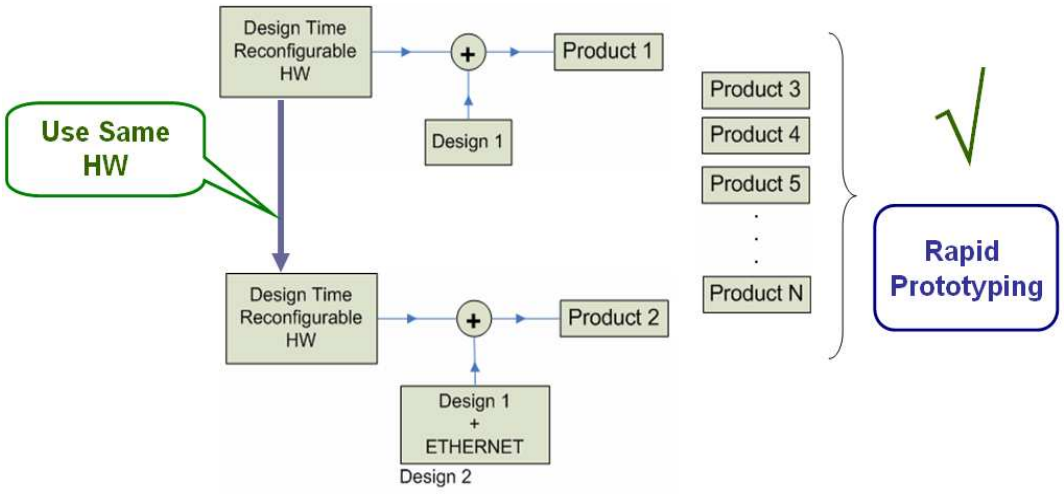


Figure 1.2. Solve design change problem

Furthermore, the duration of assurance for microcontrollers is not short. For example, minimum six weeks are required to obtain the microcontrollers from NEC Electronics [5]. This is a proof that the configurable architectures have more effective solutions than the traditional ones in rapid prototyping. Also, reconfiguration provides performance increase compared to the nonconfigurable hardware. However, it does not let the embedded software use the target architecture efficiently. For example, let us

assume that the designer has an embedded software having two modules. The software of them are implemented using a number of operators (like addition, subtraction, multiplication, division, etc.). To be able to run software on reconfigurable hardware, all operators used by two modules must be mapped to the hardware at design-time. This results in that; when one of the modules is running, there may be irrelative operators mapped to the hardware, which can cause inefficient use of the board. As a second scenario, a partial location may be damaged on the board. In such cases, the design-time reconfigurable models do not have a solution. Thus, the board is rubbish at run-time following the partial hazard. Therefore, the methodology of reconfiguration at run-time [4] has arisen to remove these limitations.

Run-time reconfiguration is the extreme point reached, owing to the current DSM technology. According to this method, target hardware can be reconfigured at run-time. It solves the efficiency problem encountered in the priorly mentioned scenario. For that, it maps the operators used by the active module to the hardware and removes the ones belonging to the sleeping one from the hardware at the time of module context switching. The surface that is used by the removed operators can be used for the active module. Thus, the number of configured operators for the active module can be increased compared to design-time reconfiguration case. Because we have more operators on the surface for the active module, the execution delay (like interrupt latency) of the active module is decreased. Moreover, partial reconfiguration [4] is used to overcome the damage encountered in the latter scenario. In partial reconfiguration, the functionality of the broken surface is moved to a safe part of the target hardware. Also, it can be guaranteed that the broken surface will not be used in further reconfigurations. Another advantage of partial reconfiguration is that it can be applied while the active module is running on safe surface.

1.2. Run-time Reconfiguration

Run-time reconfiguration is the ability to configure target architecture at run-time. The decisions about the reconfiguration can be made at design-time or run-time. These decisions are applied on both software and hardware of the design. Software can

be defined as a collection of Basic Block (BB) each of which is seperated by the control structures programming languages provide.

In design-time reconfiguration decisions, High Level System (HLS) designer must decide the number of required hardware blocks for each BB. These values must be assigned to each BB. To be able to use this type of solutions, there must be a configuration algorithm in Low Level System (LLS) design. The algorithm must look at HLS designer entries to extract the number of required hardware blocks for the current BB at run-time. Then the algorithm must configure this number of hardware blocks to the target architecture. This algorithm can be applied to all software solutions because its job is common to all software solutions.

In run-time reconfiguration decisions, we can have more effective solutions with some overhead. At run-time, running conditions can be changed and some BB blocks can run more frequently than others. In this case, different configuring strategies can be applied by HLS or LLS designer. These strategies are embedded to the configuration algorithm. The algorithm must have a counter for each BB. The corresponding counter must be incremented when a BB is visited. When a counter value is reached to different threshold values different decision algorithms can be applied.

1.3. Motivation

In previous section, we examined history of the embedded system target architectures. Meanwhile, there has been application development method proposals for nonconfigurable, design-time reconfigurable, and run-time reconfigurable architectures. The most of the current solutions are derived from the same point of view, shown in Figure 1.3. For the programmer, they have the same model that results in efficiency problems. This is the main motivation of this thesis. We will explain this in more detail in this section.

The process of the embedded system development on (run-time) reconfigurable hardware cares about software and hardware simultaneously. This is called as hard-

ware/software co-design. Therefore, reconfigurable architectures must have a different programmer model from nonconfigurable hardware. This new model must have systematic series of actions originated from hardware/software co-design. This is an obligation to reduce the cost of development life-cycle and to use target hardware resources efficiently (at run-time). In the following paragraphs, we examine whether the available solutions are complete for the most recent embedded systems architectures, the ones reconfigurable at run-time.

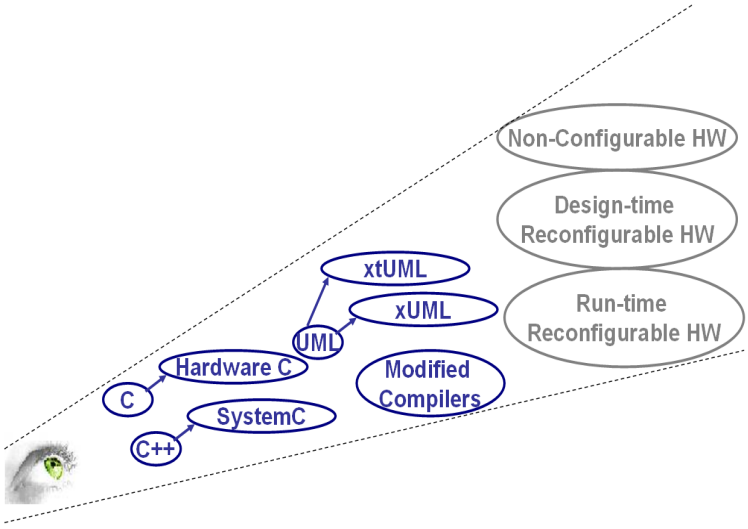


Figure 1.3. Same point of view problem

First of all, the current programming languages like C, C++, Java, etc. have been proposed for the nonconfigurable embedded systems target architecture. To use them with the (run-time) reconfigurable architectures, they have been tailored or extended. Results are languages like SystemC [6], HardwareC [7], etc. It is also questionable that these extended languages are sufficient. For example, SystemC does not provide representation for resource constraints [8]. Besides, it has not the ability to have architectural model as input [9]. Because SystemC is extension of C++, it can be misunderstood that SystemC has inhereted all capabilities of C++. This is not the case. Many important properties of C++ are not supported by SystemC [10]. These are local and nested class declarations, dynamic memory allocation, exeception handling, function recursion, overloading, file I/O, inheritence, pointers, floating-point data types, and user-defined templates. Namely, it is not completely correct that the power and

functionality of C++ for software design are provided by SystemC. Another disadvantage of SystemC is that it uses different data types for software algorithm specification and hardware inputs and outputs specification [10]. Another proof showing insufficieny of the extended languages is that the extended ones are also extended in some works [11, 12, 13] to provide better solutions. Also, only extension of the current languages are not adequate. Standard compilers provided for them must also be modified to be able to partition the hardware and the software. This modification is to import new syntax and semantic information into compilers. Because languages and compilers are modified, we have double work-effort here.

Secondly, the modeling languages have been proposed during hardware technology improvement. Specification and Description Language (SDL) [10, 14], Unified Modeling Language (UML) [10, 14, 15, 16], etc. are well-known examples. However, the cores of them are inappropriate to use them as solutions for the new programmer model. Thus, the community has proposed extensions to adapt them to the applications on reconfigurable hardware. Instead, the modeling languages must only be used as representation languages to express the design visually. They should not be used to implement the design completely. Thus, the generality of the modeling languages will not be lost.

In other words, when we are using modelling languages, complete implementation must be postponed to the programming languages, to which the modeled design will be converted. On the other hand, in itself, this idea has the problem of double work-effort due to extension requirements (e.g. xUML [17], xtUML [18, 19]) in the modeling language and in the programming language (e.g. SystemC) it is converted to.

Furthermore, there are approaches converting UML design to byte-codes. This method decreases work-effort to one by eliminating programming language extension. However, this method cause UML to diverge from its aim of birth that is the modeling the application not programming the application. This concludes that using modeling languages as direct solutions are not convenient for the models of reconfigurable hardware.

We want to note that the use of the modeling languages is meaningful if the model is converted to a programming language requiring no tailoring/extension. This means that only tailoring/extension for the modeling language is required, resulting in one work-effort. In other words, porting the modeling language to the programming one is sufficient in this case. It must be noted that one work-effort is unavoidable.

As a consequence, there are tailoring and extension methods used to conform the current programming and modeling languages to the embedded system design on the reconfigurable architectures. In contrast, their usefulness for the reconfigurable hardware programmer model is questionable due to additional work-effort.

In the literature, extending the available standard tools is called as *retargetting* them. Because target hardware may change, it is wanted that tool cores are sufficient to produce new tools for new target hardware. To retarget existing tool suite, architecture description languages (ADL) are used. However, they need low-level details which are irrelevant for software development level. This is the second main motivation of this thesis. In the following paragraphs, we consider available ADLs with their problems.

MIMOLA is one of the ADLs. It models target hardware in a similar manner to VHDL [20, 21]. Thus, it explicitly [22] requires definition of hardware modules with their behavioral algorithm and detailed interconnection scheme between modules. This results in that MIMOLA has problems with complex instructions. We understand this from the quaility of produced code by the tools using MIMOLA as ADL. They have poor quality because it is difficult to extract complex instructions from MIMOLA descriptions [23, 24].

nML [22] is another example of ADLs. The main disadvantage is that it requires low-level details to define target architecture. These are behaviour, assembly language mnemonic, and binary code of the modules (instructions). New instructions are constructed by using predefined operators. Predifned operators occurence is a result of module behaviour requirement. Therefore, they are also overhead for high-level software development life-cycle.

LISA [25] is an enhanced ADL compared to nML. However, it has the disadvantage that it requires behavioral description of instructions as nML.

In both nML and LISA, new operators (instructions) can be created in two ways. Firstly, predefined operators (+, -, *, etc.) are used to create new operators. Secondly, these created operators can be used to create more complex operators. In an application, both types of operators for the same functionality can be created. Software designer can choose this to decide which of them provides more efficient solution. However, if software designer uses nML or LISA, he will have a huge work overhead while switching between these two types of implementation. The cause of the overhead is that nML and LISA force the user to give different operator names for each type of implementation. Thus, designer must scan all lines in software, find each references to the operator, and replace it with new name.

MIMOLA, nML, and LISA requires bit lengths of the operands for the operators. However, this data is not relevant for software designer. He must be free to enter any width operand for an operator. Bit width decisions must be postponed to the lower levels, even to the run-time. However, these languages do not provide designer with flexible wordlengths.

1.4. End Markets

In this thesis, we propose a model and a framework to provide efficient design solutions for run-time reconfigurable architectures. Also, it is possible to use nonconfigurable and design-time configurable architectures with our proposals.

The selection between the architectures depends on the market. However, preferences may be changed from project to project in the same market. In some projects, time cost may be the most important measure. In other projects, this choice may be changed and hardware cost may be the most important measure for system design team. If the latter is true, run-time reconfigurable architectures provide the best solutions. HLS designer can select a small-sized target architecture to decrease hardware

cost. If HLS designer wants both using architecture efficiently in time and decreasing development life-cycle, the proposed model and framework provide efficient solutions for him.

In our proposals, we consider HLS design. This concludes that our proposed works can be used for all product fans produced by different companies. Our work does not depend on any specific target architecture. It provides a front-end (platform-independent) design. Back-end (platform-dependent) design is postponed to the target architecture specific solutions.

1.5. Thesis Organization

In Chapter 2, we introduce our solutions for the problems stated in Motivation section. We propose a new model. Then, we implement it thanks to a framework. In this framework, we use a new brand design language. Framework can be configured by the designer via provided tools. These tools enable designer creating a collection of abstracted hardware, defining his board, and defining operators that will be used in application software. In Chapter 3, we implement our proposals. We present our Integration Development Environment (IDE). In Chapter 4, we develop simple examples to understand how our IDE works. In Chapter 5, we concentrate on an industrial example, Traffic Controller. In Chapter 6, the class organization, which is implemented to develop our IDE, is explained. In Chapter 7, we conclude thesis book with future work. In Appendix A, we develop a step by step example. In Appendix B, an example output file for our IDE is presented. In Appendix C, we give software details for our IDE.

2. RH(+) SUITE

In previous chapter, we give the target architecture development history. At the end of history, we concluded that the run-time reconfigurable ones are the most flexible embedded system target architectures. Then, we details the problems with current approaches to these target architectures.

In this chapter, we define our new approach, RH(+). It is a model for embedded system design on run-time reconfigurable hardware. We have assumed that (-) means only hardware or only software design. However, (+) means that hardware/software co-design so (+) is an ehnanced mode of (-). As a result, RH(+) stands for Hardware Software Co-design on Reconfigurable Hardware.

2.1. RH(+)

There are three main requirements of RH(+) Model. Firstly, Low-Level System (LLS) details must be limited for High-Level System (HLS) design. Secondly, hardware must be abstracted by using these limited set of details. Thirdly, software must be mapped to the hardware efficiently. We explain them in more detail in this section.

2.1.1. LLS Details Minimization

The key idea in HLS design is to minimize LLS details from HLS designer because these details are irrelevant for the HLS design. To achieve minimization, importing LLS details into the application must be postponed to the LLS design steps. However, to be able to analyze HLS design, some low-level data is required. These are the number of resources (CLB, I/O pins) the hardware consumed, and execution delay the hardware requires to finish its job. Resources are shown in Figure 2.1. Moreover, HLS designer must be capable of analyzing his system by using the low-level details. However, he may not know the exact values of them. Therefore, these values must be inputs from LLS to HLS.

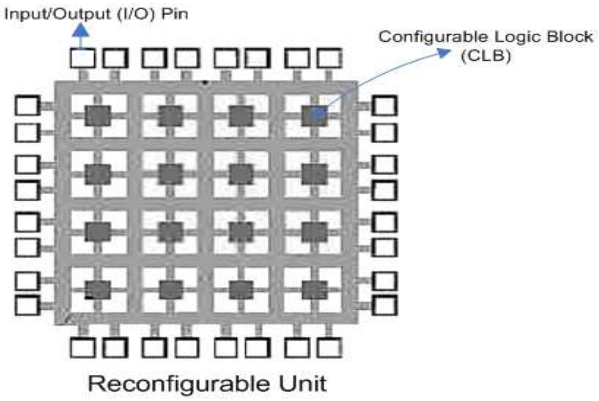


Figure 2.1. LLS details: resources

2.1.2. Hardware Abstraction

There are two types of hardware on target architectures. These are called as operators and block hardwares. Operators are the unique data. Block hardwares are components like UART, DMA, I2C, etc.

Each operator has a name. This name provides abstraction of the operators. In operator definition, it is not necessary that HLS designer enters the following parameters. Instead, partial definition of the operator is adequate. This partial definition includes only minimum low-level details. The decision about some parameters must be made in the LLS design. These are operator's opcode, behaviour of the operator, assembly syntax of the operator, register files at which operands are located, wires defining input and output nets, and physical addresses for memory locations.

As operators, partial definition for a block hardware is sufficient but it has one important additional parameter. This is the collection of interface functions. This collection provides abstraction of the block hardwares. A block hardware can be accessed via interface functions. It is not necessary to know behaviour of these functions.

2.1.3. Efficient Mapping of Software to Architecture

HLS designer must be guided correctly when it is time to map the software to the hardware. Developed software can be optimized via well-known compiler techniques. Also, software can be reorganized due to constraints (in terms of time and resource) applied by the designer. Therefore, it must be possible that HLS designer can analyze his design after these optimizations and reorganization. Thus, the designer can make another software mapping decision after seeing the optimized and reorganized software. This strategy will lead efficient use of target architecture.

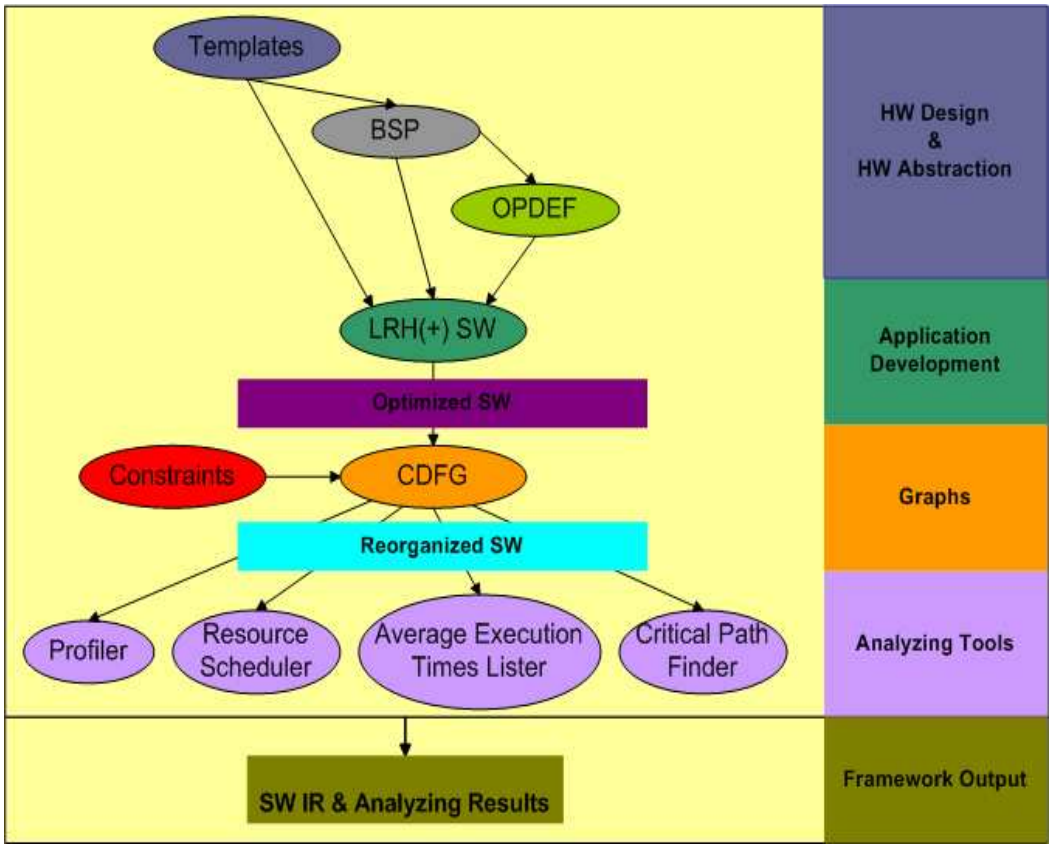


Figure 2.2. FRH(+) design flow

2.2. FRH(+)

FRH(+) is the framework for RH(+). It is our solution for RH(+) requirements. In Figure 2.2, general structure of the framework is presented. Before starting HLS design, three configuration data must be ready. They are prepared by HLS designer in the

first level. These are Templates, Board Support Package (BSP), and Operator Definitions (OPDEF). They include hardware abstraction required for RH(+), Section 2.1.2. During definition of these data, it is sufficient that HLS designer gives minimized set of LLS details explained in Section 2.1.1. In the second level, HLS designer develops his system by using three configuration data, and proposed LRH(+) language. Third level meets efficient mapping of software to architecture requirement of RH(+), Section 2.1.3. This level has Control Data Flow Graph (CDFG). It has optimized version of LRH(+) application. HLS designer can insert constraints into CDFG. In the fourth level, there are application analyzing tools. Analyzes are applied on CDFG that is reorganized according to constraints. *Profiler* shows resource references at each basic block (BB). *Resource Scheduler*, schedules the resources at the BB level. Resources are operators, operands and functions. *Critical Path Finder* finds the path having longest execution delay. *Average Execution Times Lister* lists all possible paths and their average execution times. The fifth level has the output of the framework. Output has intermediate representation (IR) of the application, and analyzing results.

Section 2.3 has proposed language details. In Section 2.4, we will present our OPDEF procedure. Templates are explained in Section 2.5. Section 2.6 contains BSP management steps. Application analyzing tools and output format can be changed according to strategy of framework implementation. We implement our proposed framework, FRH(+). We present our implementation in Chapter 3. We also give details of analyzing tools in this chapter.

2.3. LRH(+)

After BSP, which is described in Section 2.6, and Templates, which is described in Section 2.5, are ready, application development can be started via proposed design language. Its structure includes traditional programming conventions and the ones special to Custom Hardware (CH) which has reconfigurable and nonconfigurable hardware at the same time. In addition, it includes structures FRH(+) proposes. The structure of design language is shown in Table 2.1. You can follow the table with the following instructions:

- If SubSubTags column has entries, there are additional data for them in the Tag column.
- If SubSubTags column has no entry but SubTag has entries which starts with capital letters, there are additional data for SubTag entries in the Tag column.
- If there is an entry beginning with lower case in SubTags column, this entry is primitive data and there are no more entries for it in this table.

Table 2.1. LRH(+) components

Tag	SubTags	SubSubTags	Templates (Y/N)?
Application	Shared Data	General	N
		Array	N
	Functions	Function	N
	Tasks	Task	N
	other Tags	-	Y
General	nameOfData	-	N
	size (bits)	-	N
	initialValue	-	N
	location	-	Y
Array	nameOfData	-	N
	arraySize	-	N
	sizeOfElements (bits)	-	N
	initialValues	-	Y
	location	-	Y
Function	Parameters	General	N
		Array	N
	Scope	-	N
Task	Scope	-	N
Scope	ScopeData	General	N
		Array	N
	Statements	-	N
	If/ElseIf/Else	-	N

Table 2.1. (continued)

Tag	SubTags	SubSubTags	Templates (Y/N)?
	Loop	-	N
	FunctionCall	-	N
	StateMachine	-	N
	Methods like TimerStart()	-	Y
If/ElseIf/Loop	condition, Scope	-	N
Else	Scope	-	N
StateMachine	initialState	-	N
	States	State	N
State	Scope	-	N
	NextStates	NextState	N
NextState	condition	-	N
	nameOfNextState	-	N
	Scope	-	N
Delay	hour	-	N
	minute	-	N
	seconds	-	N
	miliseconds	-	N
	microseconds	-	N
	nanoseconds	-	N
FunctionCall	nameOfFunction	-	N
	Parameters	General	N
		Array	N
Interrupt	source	-	Y
	nameOfFunction	-	N
I2C	-	-	Y
UART	-	-	Y
CAN	-	-	Y
ADC	-	-	Y

Table 2.1. (continued)

Tag	SubTags	SubSubTags	Templates (Y/N)?
Port	-	-	Y
Timer	-	-	Y
DMA	-	-	Y
Processor	-	-	Y
Location	-	-	Y
Clock	-	-	Y
DataBus	-	-	Y

An application is developed under *application* tag which includes *SharedData*, *Functions*, *Tasks*, *Interrupts*, and tags learnt from Templates. General and Array are the data types and data under *SharedData*, *ScopeData*, *Parameters* of *Function*, and *Parameters* of *FunctionCall* must have type that is one of them. To be able to define *General* data, its name, size in terms of bits, location and initial value of it must be given. *Location* information is retrieved from Templates or definitions in application level since HLS designer can also add locations definitions in application level. *Array* tag requires size of array and initial values of its elements whose sizes are also given in terms of bits.

Table 2.2. Statement and condition formats

Tag	Format 1	Format 2
Statement	c = a + b;	.=(c, .+(a, b));
	d = func1(c);	.=(d, .func1(c));
	e = d/c*4;	.=(e, .*(./(d, c), 4));
Condition	((a > b) && (c < d)) || (a + b < 10)	.||(.&&(.>(a,b), .<(c,d)), .<(.+(a, b), 10))

Common tag in many tags is *Scope* which is parent of many tags. *Function*, *Task*, *If*, *ElseIf*, *Else*, *Loop*, *State*, and *NextState* has *Scope* tag. Each *Scope* has its data which is valid till the end of it. *Statements* can be written under *Scope*. *If*,

ElseIf, and *Loop* are also tags under *Scope* and have condition. In Table 2.2, there are statement and condition examples. We have two formats. Format 1 is the traditional method to write expressions. Format 2 is another way to write expressions. Format 2 has an advantage that operators can have many operands. Operators in Format 1 can have at most two operands.

Else tag has only Scope as *Task* tag. *StateMachine* can be defined under *Scope* and it is meaningful for only a task because it is attached to a processor. In practical, HLS designer prefers to define a state machine if he wants to construct a state controller on a processor. *StateMachine* may have *States*; HLS designer initializes it by selecting *initialState* between *States*. When *StateMachine* is in a *State*, *Scope* of *State* is executed; however, there may be triggers to go to one of the *NextStates*. These triggers are given with conditions. If condition is true, transition to *NextState* is realized and before applying transition *Scope* of *NextState* is executed; it generally includes operations for the release of the current state and operations for the initialization of the next state.

Delay tag is used to give a delay to the application at any instruction point. *Interrupt* tag is used to define an interrupt whose available sources are given with Templates. Interrupt Service Routine (ISR) is a function defined by HLS designer and it is also selected while defining an interrupt.

Templates include possible ingredients any board may have, and examples of them are: I2C, UART, CAN, ADC, Port, Timer, DMA, Processor, Location, Clock, and DataBus. They may be built in with the board; in other words, they may be nonconfigurable. For example, processor may be a hard processor, not soft processor. These are defined in BSP of the board. In addition to BSP, HLS designer may want to create hardware on reconfigurable part of board. The framework provides HLS designer with this flexibility; for that, it extracts all the data required to define the hardware from Templates. The framework explores the target board resources if sufficient resources are available to add new hardware to the application.

2.4. OPDEF

In traditional programming, addition, subtraction, etc. are traditional operators and programmer does not know about how it is executed. Generally, it is executed in ALU of hard processors. However, when CH is the case, all of the operators should be defined by the application developer for efficient use of hardware. Here, efficient means fast and minimized cost.

Table 2.3. OPDEF example

Operator	Input1	Input2	Output1	CLB	I/O	Cycle	Comment
+	ram	ram	ram	4	2	4	8-bit adder 5
							ram is third party so I/O is required
+	register	register	register	1	0	4	2-bit serial adder
[ ]	eeprom	ram	register	0	0	6	Input1: array
							Input2: array index
							Output1: array element
/	ram	ram	ram	24	0	4	ram is internal so no I/O is required

Operators have inputs and outputs; these can be located anywhere on CH like RAM, register, etc. The locations must be defined before OPDEF. Operator execution cycle and the number of resources used by it must be known by the design environment to produce resource scheduling and CH mapping in time. In this way, operators can be overloaded via input count, output count, inputs locations or outputs locations. For example, an operator whose name is "+" may sum two or three numbers. They have same name but different signatures. In Table 2.3, "+" operation is overloaded via locations. First "+" operator has inputs and output from RAM; however second "+" operator has inputs and output from register. Secondly, "[ ]" operator refers to an array in EEPROM and its index value is restored in a RAM location; the element at this index is read into a register. "[ ]" has no area overhead in this case because it uses existing hardware overhead: data buses, and EEPROM, RAM, register locations.

While developing application, HLS designer applies to operator at the time of writing expression for conditions or statements. HLS designer cannot refer to an operator which is not defined before or an operator which is not be able to be expanded.

2.4.1. Expand Procedure

Expand Procedure is applied if an operator used in original code is not defined before by HLS designer. In Table 2.4, there is an example, in which original code and its expanded code are given. X, Y, and Z are variables are located in ram. Reg1, reg2, and reg3 are registers. In this example, original code has a (+) operator. Operands of this operator are located in ram. However, let us assume that HLS designer has not an operator whose operand locations are ram, and has an operator whose operand locations are register. In this case, original code is useless so Expand Procedure must be applied. In other words, original code must be rewritten by using existing operators. There is a disadvantage of this procedure. The original code has one line of code. After Expand Procedure, the expanded code has four lines of code. Therefore, execution time is increased so HLS designer must be warned about expansion. If there is no way to apply an expansion by using defined operators, an error must be raised.

Table 2.4. Expand Procedure example

Original Code	Expanded Code

ram (X) = ram (Y) + ram (Z);	reg (reg1) = ram (Y);
	reg (reg2) = ram (Z);
	reg (reg3) = reg (reg1) + reg (reg2);
	ram (X) = reg (reg3);

2.4.2. Complex Operators

In Table 2.3, traditional operators are customized; however, complex operators can also be described via using operator definition procedure. For example, a butterfly operation used in Fast Fourier Transform (FFT) can be defined. Hardware blocks

like Cyclic Redundancy Check (CRC) generator, hash value producer, etc. can also be considered as complex customized operators. They can be accessed from the application software by using its name. Defining complex operators in style of operator definition, results in that they are black box for the application design level, and in the lower levels, detailed implementation of them exists.

2.5. Templates

Template collection is the module that must be ready before starting application development. It provides procedures for new hardware block definitions. Commonly, all of them have number of used CLBs and I/O pins to implement it and functions assigned to it. For example, a timer may be implemented into 5 CLB with 2 I/O pins and it may serve as a counter which raises an interrupt when counter value is equal to comparison value. Also, a timer may produce a pulse on one of its outputs; in this case, a third I/O pin is required.

Templates must also be ready before defining BSP. For example, locations defined by HLS designer are grouped under Locations tab and procedure for adding a new location under it must be known. This template may include size, interface its connected to board, the number of I/O pins and CLB to implement it. This information extracted from template is used while adding a location at BSP or application level.

If location is connected to board via a special interface not only via I/O pins, this interface must also be implemented on CH. In other words, a template for this interface must be defined; this obligation reflects a nested approach being a modular solution. In the case of interface is I2C Controller [26], some circuitry, signal lines and registers will be required. However, these hardware details are not required by the FRH(+); it is sufficient to be able to configure functional settings and to know quantity of resources (the number of CLB and I/O pins) used to implement it. Bus clock is the unique functional data that HLS designer must be able to set via this template, because communication line is serial, and data bus size is known, 1. Examples of templates are

shown in Table 2.5. As seen, templates may have random module information. This flexibility solves the problems different third party CH may have different HW blocks on it, and HLS designer wants to create a variety of HW blocks.

Table 2.5. Template example

Module	Type	CLB	I/O	Data
I2C	Core	4	3	Frequency
UART	Core	12	4	WordLength Number of StopBits Parity BaudRate FlowControl
	Rx	3	1	-
	Tx	3	1	-
CAN	Core	16	-	-
	Transmitter	2	2	-
ADC	2	1	-	-
Port	Core	1	8	Type (PushPull, OpenCollector) Direction (Input, Output)
	Digital	1	8	-
	Analog	5	8	-
Timer	Core	5	3	
	PWM	-	1	Frequency PositiveEdgePercent (ex. 80%) NegativeEdgePercent (ex. 20%)
	Input Capture	1	1	Frequency Selected edge (Falling / Rising)
	Counter	-	-	Period
DMA	Core	10	-	SourceLocation DestinationLocation
Processor	Type1	28	6	InternalClock

Table 2.5. (continued)

Module	Type	CLB	I/O	Data
	Type2	38	6	InternalClock
Location	Type1	1	1	-
	Type2	2	1	-
	Type3	3	1	-
	Type4	4	1	-
Clock	Type1	2	1	-
	Type2	2	1	-

Templates not only serve as a base data for BSP and application hardware definitions but also let HLS designer define methods to access hardware and properties of hardware. This information can be taught as software of driver function prototypes for hardware. The symbolic names provided by them are sufficient for the FRH(+) because low level details change according to third parties. Examples of template methods are shown in Table 2.6. In this table, methods start with template module name they belong to, and random modules are accessed via random methods.

For a proprietary product, BSP and driver software for it are distributed via commercial tools. Therefore, every company creates its application development framework and develops its Integrated Development Environment (IDE) or contracts with a third party company to develop an IDE for its CH. FRH(+) can be used for any CH produced by different companies. And, output of FRH(+) can be assembled to a specific processor via a compiler devoted to this processor whose type may be user implemented soft processor, commercial soft processor or commercial hard processor. In other words, via FRH(+), a Platform Independent Model (PIM) solution is obtained, and Platform Specific Model (PSM) is postponed.

Table 2.6. Hardware abstraction for Template example

Method	CLB	I/O	Cycle
I2CReceiveData()	1	-	3
I2CTransmitData()	1	-	3
UARTReceiveData()	1	1	4
UARTTransmitData()	1	1	4
CANReceiveData()	1	1	2
CANTransmitData()	1	1	2
PortSetPinLevel()	1	-	1
PortTogglePinLevel()	1	-	1
TimerStart()	1	-	1
TimerStop()	1	-	1
TimerPWMGetLevel()	1	-	1
TimerInputCaptureGetTickCount()	1	-	1
TimerCounterGetValue()	1	-	1
HashValueGeneratorSHA1Calculate()	2	-	5
HashValueGeneratorMD5Calculate()	2	-	5
RandomModuleMethod1()	3	-	6
RandomModuleMethod2()	4	-	8

2.6. BSP

CH may have logic circuits ready to be configured, RAM, FLASH, connection controllers which co-processors can be attached to, and buses which third party hardware blocks can be attached to. BSP is used to define the structure of any CH via templates.

CH may have more than one processor, so number of processors must be known. In application level, a task for each of them can be written. In this case, tasks must access available resources mutually exclusive. Operators can be defined for each processor, if they are different or it may be assumed that same operator information can be used. If they are same type of processor, same operator information is valid for all of them.

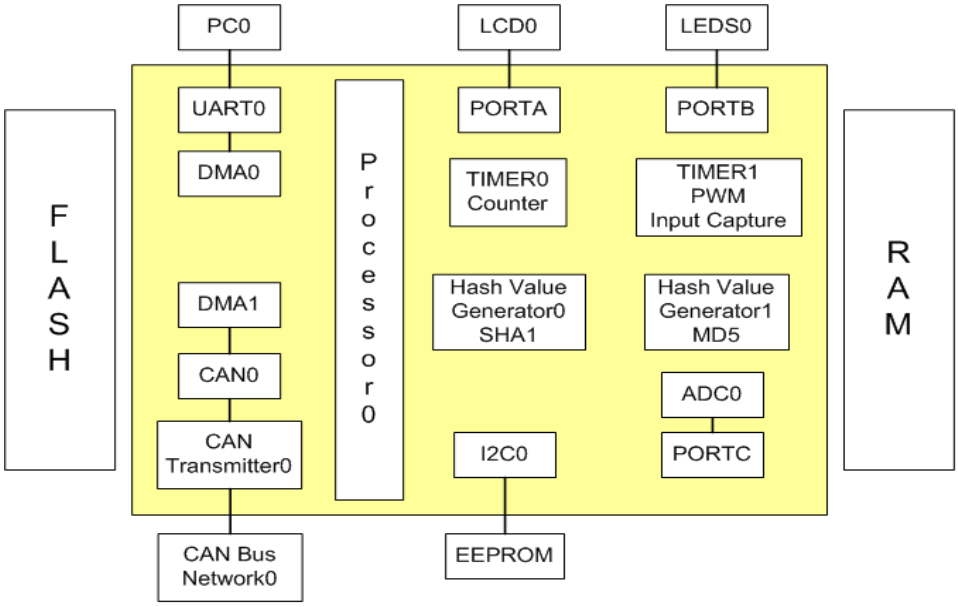


Figure 2.3. The board for Template and BSP examples

CH has configurable areas called as CLB and I/O pins so total number of them must be remarked in BSP. These values are used when HLS designer wants to create his hardware on CH; if there are not enough resources, this is an error. Besides, there may be already configured CLBs and implemented I/O pins on board so the number of used CLBs and I/O pins must be remarked. For example, a LCD may have been connected to some I/O pins or an UART controller can be configured on CH. This information is meaningful if these resources are devoted to these implemented components (LCD, UART controller) and reconfiguration of them is disabled.

Until buying target board, its ingredients are not known so all components must be collected under abstract groups whose methods and properties are decided before via Templates. If this is provided, while adding an ingredient to BSP, only these methods and properties are overridden. Because templates are also defined by HLS designer, FRH(+) is independent from board structure.

In fact, it can be assumed that CH has more than one location in many cases. In other words, CH has one of the programming models Single Instruction Multiple Data (SIMD) [27] or Multiple Instruction Multiple Data (MIMD) [27] because CH has generally two types of data location: block RAM surroundings and distributed RAM

in it. However, this is only a practical result and it is not a limitation for FRH(+). In other words, other programming models which are Single Instruction Single Data (SISD) [27] and Multiple Instruction Single Data (MISD) [27] can be modeled via FRH(+). BSP for the board, which is shown in Figure 2.3, is shown in Table 2.7.

Table 2.7. BSP example

Class	Instance	Type/Data Values/SubInstances
Clocks	Clock1	25 MHz
	Clock2	4 MHz
	Clock3	8 MHz
	Clock4	32 KHz
Locations	RAM	Type1
		4 K
	FLASH	Type2
		64 K
	EEPROM	Type3
		16 K
		I2C0
	Register	Type4
		32 bit-width
I2C	I2C0	-
UART	UART0	Rx and Tx
CAN	CAN0	CanTransmitter0
CAN-Transmitter	CanTransmitter0	-
ADC	ADC0	-
Port	PORTA	Digital
	PORTB	Digital
	PORTC	Analog

2.7. Constraints

In Section 1.2, we consider two run-time reconfiguration decision methods: design-time and run-time reconfiguration. We select the former in FRH(+). We apply this selection by defining constraints on LRH(+) software. These constraints determine the

number of required hardware blocks for each BB. We write BB by using keywords of *condition* and *Statements*, which are shown in Table 2.1. These blocks are separated by using control structures, which are *If*, *ElseIf*, *Else*, and *Loop*.

Table 2.8. Constraints example

BB	Type of BB	+	-	*	/	<	>
1	Statements;	2	3	4	1	0	0
2	Loop(condition)	0	2	0	0	1	0
3	If (condition)	1	0	1	0	0	1
4	Statements;	1	2	0	2	0	0
5	Statements;	0	1	3	1	0	0
6	Statements;	1	0	0	1	0	0

The number of the required operators for a hardware block unit depends on performance decisions made by the HLS designer. He can attach minimum number of operators to a BB not having critical response time. However, if BB must end at a minimum time, HLS designer can attach maximum number of operators to the BB. In Table 2.8, we present an example about deciding constraints on each BB of an example of LRH(+) software. The first column shows the BB number. The second column has LRH(+) software. All lines in this software correspond to a BB. The remaining columns are the operator names which are required in the corresponding BB. In other words, these operators must be ready before visiting the corresponding BB. Software starts with a list of stamtents, which is BB1. We want two "+", three "-", four "*", and "/" operators to be ready at the beginning of BB1. After these statements, loop condition will be executed. We assign two "-" and one "<" operators for BB2. When this condition is true, BB3 will be executed. Therefore, the required hardware blocks for BB3 must be configured on the target architecture. These cases are also same for BB4, BB5, and BB6.

We want to note some LLS design issues about constraints. We see that there are transitions between two blocks. For example, there is a transition from BB2 to BB3 in Table 2.8 when loop condition is true. Let us assume that our target architecture

has no empty surface to configure the hardware blocks required by BB3. In this case, LLS configuration algorithm must decide which surface used by other hardware blocks will be reconfigured. After this decision, configuration of the hardware blocks of BB3 can be started. It is also possible that there is enough space for the hardware blocks required by BB3 at the time of transition from BB2 to BB3. In this case, empty surface can be used for run-time configuration.

3. RH(+) IDE

In Chapter 2, we have introduced RH(+) suite having components for our new approach for embedded system design on run-time reconfigurable hardware. RH(+) Model presents the requirements which a framework must meet. FRH(+) is presented for a solution for this new model. Then, basic components-first two levels-, LRH(+), OPDEF, Templates, and BSP, of this layered structure has been detailed.

In this chapter, the next two levels' components will be detailed because they are common in the framework development area. The differences and enhancements will be noticed in the related contexts. In this chapter, the implementation of an Integrated Development Environment (IDE) for FRH(+), will be introduced.

3.1. Design Flow

RH(+) IDE has a front-end compiler. The output of this front-end design will be input for back-end design. Design flow for our IDE is shown in Figure 3.1. In HLS design, LLS details are stored in *Hardware Library*. Before starting HSL design, these values are imported into the design. They are used in *BSP* and *OD* design. After designing these files, *RHPlus* design can be started.

LRH(+) is used to implement software. This software can accept *Constraints*. Software with constraints are used to get analysis graphs. HLS designer can analyze these graphs and change constraints. After these changes, constraints applied to the software can be changed and new analysiz graphs can be obtained. In this strategy, we enable HLS designer interact with graphs with given constraints. The graphs collection is used to construct *CDFG*. If the results are suitable to pass to the LLS design, HLS designer may end HLS design.

LLS design is black-box for HLS design. The output of the HLS design is used to reach new decisions. If these decisions are appropriate for the LLS designer, he may

end the project. Also, he may want to change values of LLS details. For that, he can change these values for the HLS design. After that, a new software analysis process can be started on the same software with different LLS details. LLS designer can decide to end the project. In this case, because a new project ends, there may be new hardware library items obtained from analyses. If so, *Hardware Library* can be updated to use in further projects.

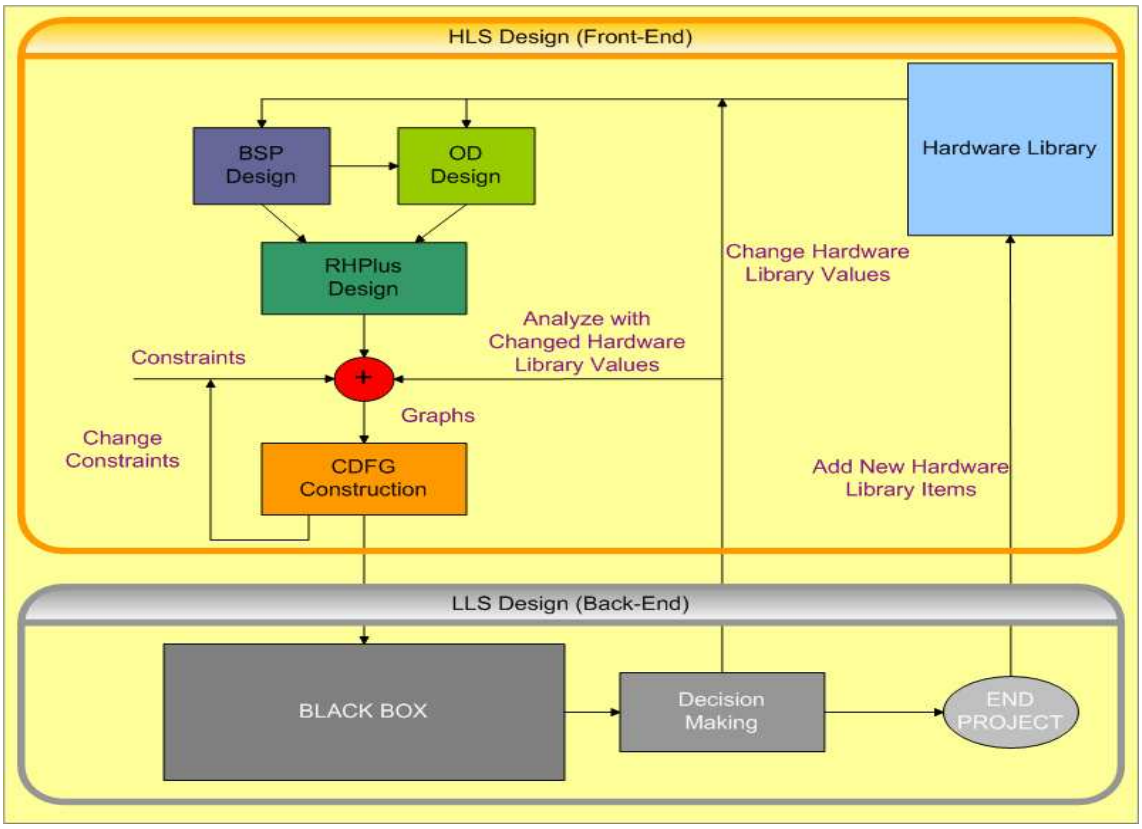


Figure 3.1. RH(+) IDE design flow

3.2. Technology

We should state that the eXtensible Markup Language (XML) [28] technologies capabilities are noticed by many works as in [12, 29, 30, 31]. XML provides us with storing data structurally. Therefore, we select the base technology used in the development of our IDE as XML. We have constructed our design language over the XML.

Also, the templates, the operators, the board definition, the application, the graphs, the constraints, the IR of the application, and the map of the embedded software to the hardware are stored in XML format. Because, outputs of RH(+) IDE is in XML format, it can moved to another application easily via deserialization of them.

Besides, thanks to eXtensible Stylesheet Language (XSL) [28], data transformation from XML to the other representations can be achieved via a set of XSL translation rules. However, XSL is not used in the current work but it has an important part in our future work.

3.3. Capabilities

Board support packages, operator definitions, application development, deciding resource constraints, profiling the number of resource references, finding critical path, and scheduling resources to basic blocks can be achieved by using RH(+) IDE. RH(+) IDE provide HLS designer with developing application visually in CDFG style. HLS designer is able to drag and drop control blocks and data blocks to the application file.

Graphs are used to analyze the design. According to analysis results, HLS designer can change design parameters. RH(+) IDE can generate three types of graphs: Resource Scheduling, Possible Paths and Execution Times, and Resource References. Firstly, HLS designer can operate on each BB, which are expressions for statement lists and conditions. HLS designer can add resource constraints by using Ref Collection Editor to each block. This concludes that we know what will be mapped to the hardware for each BB.

While switching from one BB to another BB, run-time reconfiguration on target architecture will be applied. HLS designer decides hardware map in time by using Ref Collection Editor. After these constraints, RH(+) IDE can generate Resource Scheduling graph via calcultating data dependencies in the current basic block. Secondly, RH(+) IDE is able to generate possible paths and their total execution times for the selected block.

HLS designer is able to enter probability being true for the conditional expressions. Also, the possible number of loop counts can decided via Properties Window. By caring about these probability and possible iteration values, RH(+) IDE can decide worst and best cases between the possible paths. As a result, worst case is the critical path of the selected block. Thirdly, the number of resource references can be graphed for each basic block which is under the selected block. In this case, the result graph is a collection of resource references graphs.

Output file of RH(+) IDE includes the control and data flow data. Control flow is extracted from **.rhplus* files. Data flow is the merge of outputs for each Resource Scheduling of each basic block. Output file also includes the results of the Possible Paths and Execution Times, and Resource References. In addition, output file has expressions used in basic blocks. The output file is **.cdfg* under project directory. This file can be used in LLS design to get configurations and machine codes for the target hardware.

Lastly, RH(+) IDE provides three file types: **.bsp*, **.od*, and **.rhplus* files. **.bsp* files include board support package data which is designed visually by the HLS designer. **.od* files include operator definitions that will be used in the application. **.rhplus* files include the application, there are references to **.bsp* files and **.od* files.

3.4. Components

RH(+) IDE has a Toolbox having components used in design of three files types. It must be stated that Template and BSP design is merged via RH(+) IDE. There are ready templates that will be used in BSP design. Template properties will be modified/enhanced in **.bsp* files.

3.4.1. BSP Components

- Port: Core component will be required when defining a port **.bsp files*. Same core can be shared between the following components. Digital and Analog are types

of ports. After Core component is located on **.bsp* files, then Digital and Analog components can be located on **.bsp* files, and integrated with Core component.

- ADC: Core component is used to construct Analog Digital Converter on **.bsp* files. To be able to activate this component, it must be integrated with a Port Analog component.
- Timer: There are different modes for a timer, however, core of each are similar. This similar hardware and software properties are collected in Core component. To be able to define a timer with a mode, Core component must be located on **.bsp* files at first. Same core can be shared between Counter, Input Capture, and PWM components. Counter, Input Capture and PWM components provide functionality for counting up or down, input capture, and PWM, respectively. To activate their functionality, they must be integrated to a Core component.
- UART: If HLS designer wants to establish a serial connection, he may use Core component on **.bsp* files. This component provides core hardware and software requirements for communication line. These requirements are customized by HLS designer. Same core can be shared between RX and TX components. To activate data transfer from outside into Core components, RX component must be used. To activate data transfer from Core component to outside, TX component must be used. To activate functionality of RX and TX components, they must be integrated to a Core component.
- I2C: To be able to construct an I2C network, Core component must be located on **.bsp* files. Then, network clients can be integrated to Core component to attach them to network.
- DMA: Direct Memory Access can be selected to prevent processor susspending by waiting end of time consuming data transfer operations. In this case, Core component must be located on **.bsp* files to provide data transfer between two addressable locations. First of all, Core component is attached to a location like UART Core, or Location Core component. Then, the other addressable components are integrated with DMA Core. As a result, DMA can transfer data from/to one location to/from many locations. DMA Core is the bridge for the component it is attached. However, it is an interface for the components integrated to which.

- CAN: To be able to construct an CAN network, Core component must be located on **.bsp* files. Then, network clients can be integrated with Core component to attach them to network. Transmitter component has both transmission and reception ability. To activate Core component, a Transmitter component must be integrated with it.
- Processor: HLS designer can add a processor on **.bsp* files via Core component.
- Clock: HLS designer can define a clock on **.bsp* files via Core component. Core component can be integrated to another component later.
- Location: HLS designer can define a memory location on **.bsp* files via Core component. Core component can be integrated with DMA Core or Data Bus Core.
- Data Bus: HLS designer can define a data bus between two memory locations on **.bsp* files via Core component. There may be maximum eight bus memory locations attached to data bus.

3.4.2. OD Components

- Operator Definition: If HLS designer wants to define a new operator that will be used in application, he must add an Operator component on **.od* files.

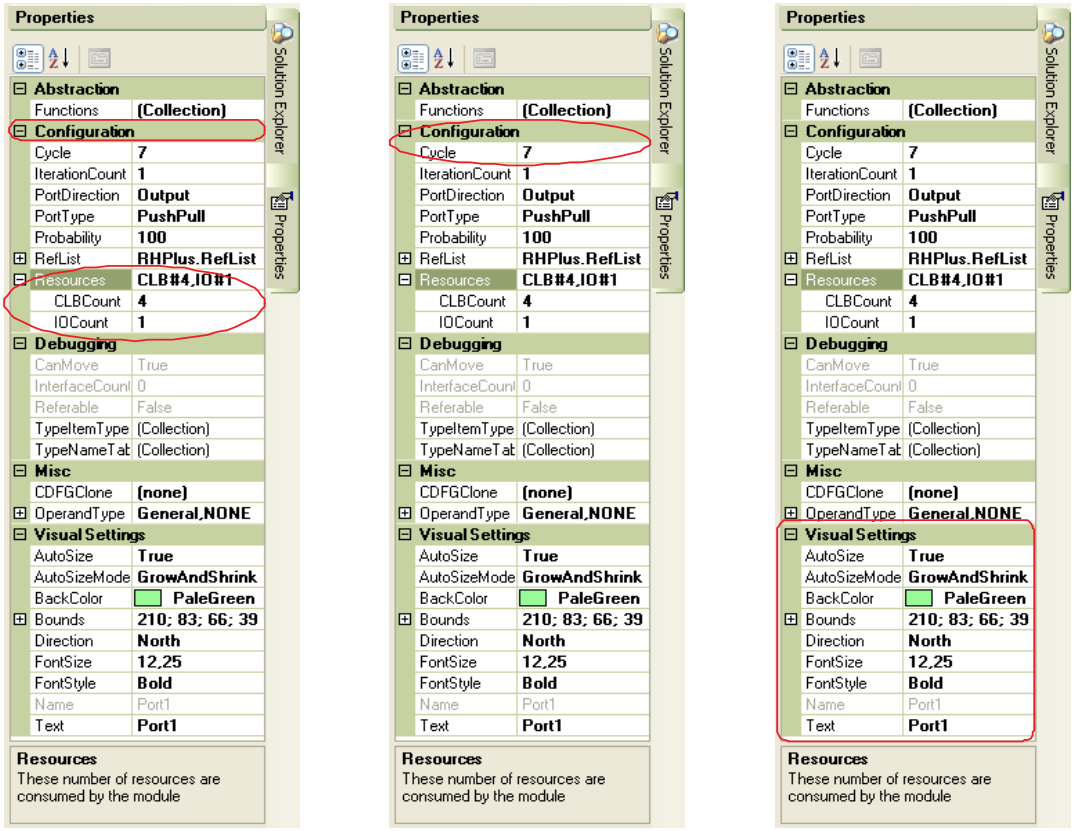
3.4.3. RHPlus Components

- General/Array Data: A new data in the current scope can be defined via adding General or Array component on **.rhplus* files. General component is used to define variables including the well-know traditional types (int, byte, short, long, double, float, etc.) in any bit-width. An array of General data can be added to the current scope by using Array component.
- Global Blocks: Functions can be added to the global scope on **.rhplus* files via Function component.
- Conditional Blocks: Loop, If, ElseIf, and Else blocks can be added on **.rhplus* files via Loop, If, ElseIf, and Else components, respectively.
- Expressions: A list of statements can be added to the current scope via adding

Statements component on **.rhplus* files.

3.5. Properties Window

Components are the ones selected from Toolbox. Then they are dragged towards to the related files at the right of Toolbox. The components accepted by these files are considered above. For example, an UART component cannot be dragged towards and dropped onto the an **.od* file. As a result, we have three types of components: BSP elements, operators, and RHPlus elements.



(a) Resources (b) Cycle (c) Visual settings

Figure 3.2. Common properties

After components are dropped to the related file, they can be selected via by clicking them. After a component is clicked, Properties Window will show the properties values for the selected component. RH(+) IDE will get HLS designer entries about these components via the Properties Window.

3.5.1. Common

Components, which are BSP elements, operators, and RHPlus elements, have common properties. Resources property is one of them. This property includes two subproperties: CLBCount, and IOCount. CBLCount is the required number of configuration blocks. Also, component may need input output locations, namely ports. The number of the required ports is the IOCount. Resources property is used when it is time to map the component to the hardware. The values of these properties can be changed by using Properties Window at the right side of RH(+) IDE. It is under Configuration category.

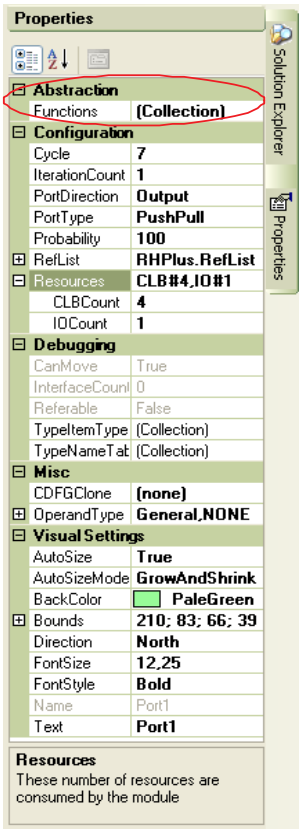


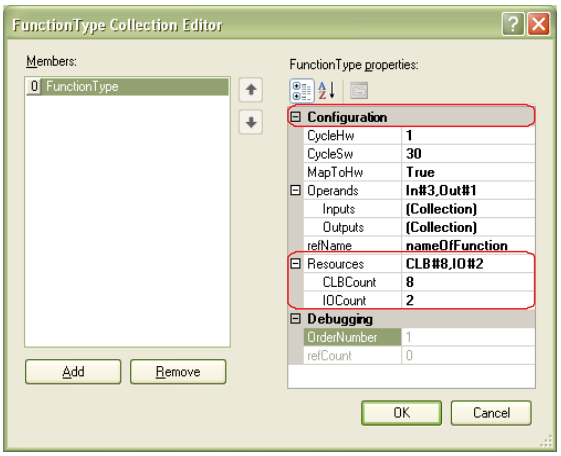
Figure 3.3. Functions

Cycle is another common property. It shows the number of cycles required for a block to finish its jobs. Its value can changed by using Properties Window at the right side of RH(+) IDE. It is under Configuration category. The remaining common properties are for visual settings. For them, there is a category in Properties Window.

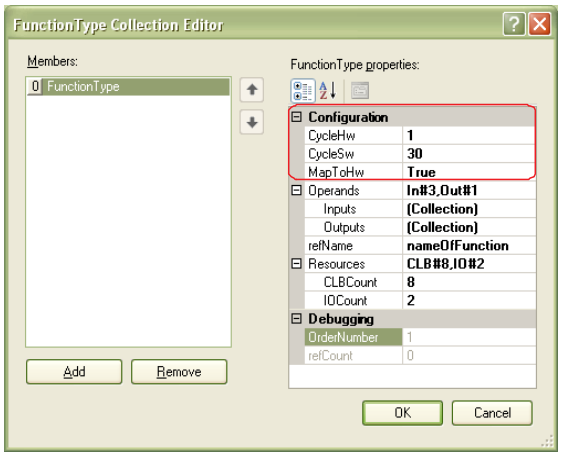
The name of the category is Visual Settings. In Figure 3.2, (a), (b), (c) shows Resources, Cycle, and Visual Settings, respectively.

3.5.2. BSP Elements

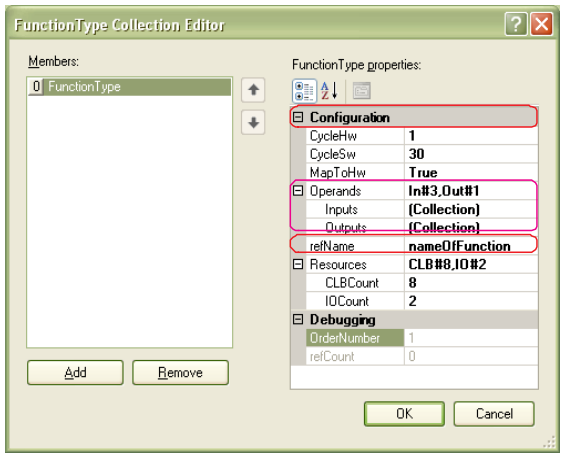
In addition to common properties explained in Section 3.5.1, Functions property for the BSP elements exists. Functions are common for all each BSP elements. Functions mean the prototype of the functionalty for the component. BSP elements may have Functions to access them. These Functions provide hardware abstraction. The Functions have the same structure with the functions considered in **.rhplus* files. New Functions can be entered for a component by using Properties Window.



(a) Resources



(b) Cycles and Mapping



(c) Operands and refName

Figure 3.4. Function type collection editor for functions

Properties Window has a category whose name is Abstraction. In this category, there is a Functions collection. It can be seen in Figure 3.3. By clicking this collection, a new collection editor for Functions is opened. New Functions can be defined by using the Add button, and existing Functions can be deleted by using the Remove button. Each Functions has the common properties Resources explained in Section 3.5.1. In Figure 3.4, (a) shows Resources.

Functions have two types of cycle data: CycleHw and CycleSw. CycleHw shows the number of cycles required for a Function to finish its jobs if it is mapped to the hardware. CycleSw shows the number of cycles required for a Function to finish its jobs if it is not mapped to the hardware. One of them will be used in Resource Scheduling. MaptoHw property shows which of them will be used. True means CycleHw will be used; False means CycleSw will be used. In Figure 3.4, (b) shows these properties.

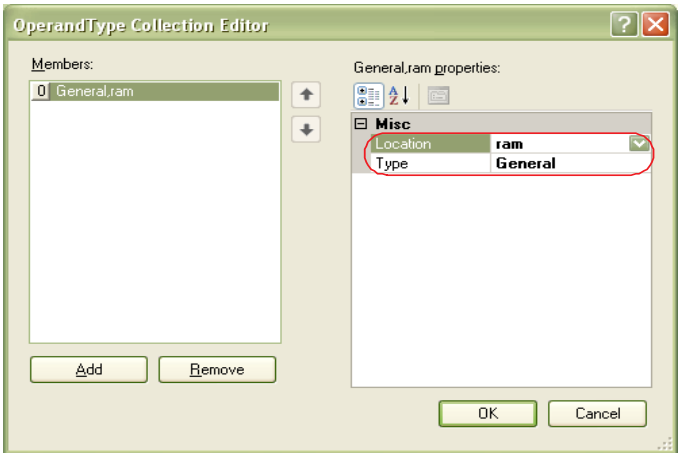
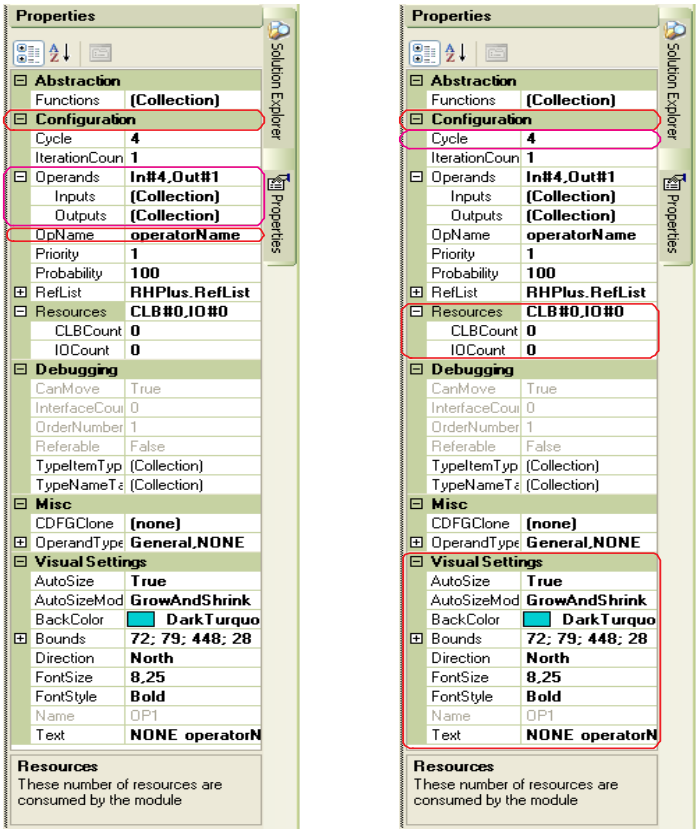


Figure 3.5. Location and type

Functions have refName and Operands properties. In Figure 3.4, (c) shows these properties. refName is used to access/reference the Functions from **.rhplus* files. Operands may inputs and outputs. Both inputs and outputs have their Collection Editor. New operands can be defined via using these Collection Editors. Each operand has Location and Type properties. They are shown in Figure 3.5. Possible locations are inhereted from the locations on **.bsp* files. Types are General and Array.

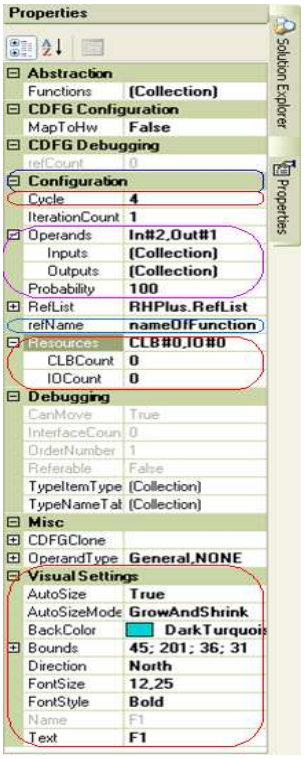
3.5.3. Operators

Operators have OpName property under Configuration Category in Properties Window. This name is used to refer to the operator from **.rhplus* files. Also, Operators can have operands: inputs and outputs. New operands can be defined via using OperandType Collection Editor. Both inputs and outputs have their Collection Editor. Each operand has Location and Type properties. They are shown in Figure 3.5. Possible locations are inhereted from the locations on **.bsp* files. Types are General and Array. In Figure 3.6, (a) shows OpName and Operands properties. Also, each Operator has the common properties Resources, Cycle and Visual Settings explained in Section 3.5.1. In Figure 3.6, (b) shows operator resources.

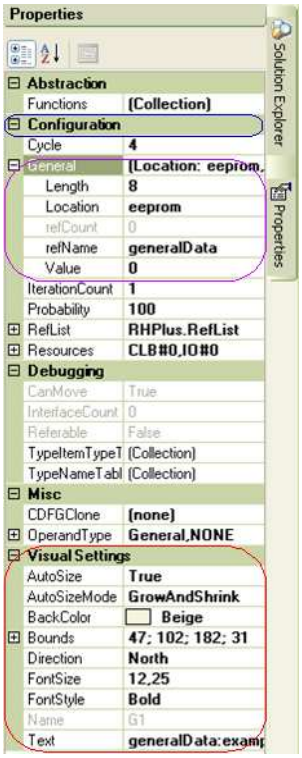


(a) Operands and OpName (b) Resources

Figure 3.6. Operator properties

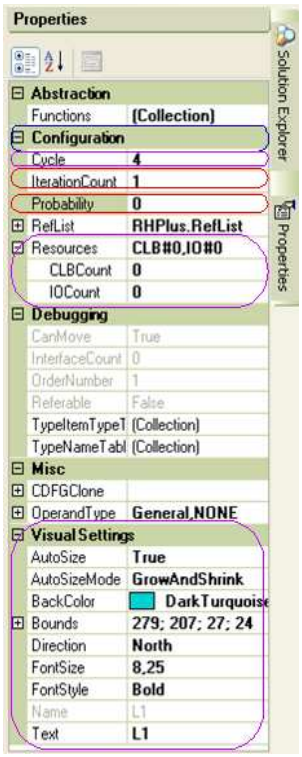


(a) Function

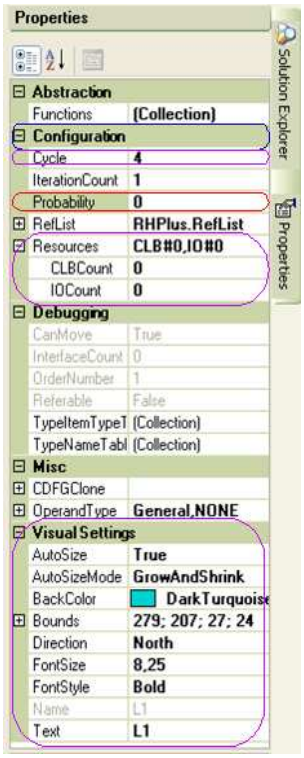


(b) General

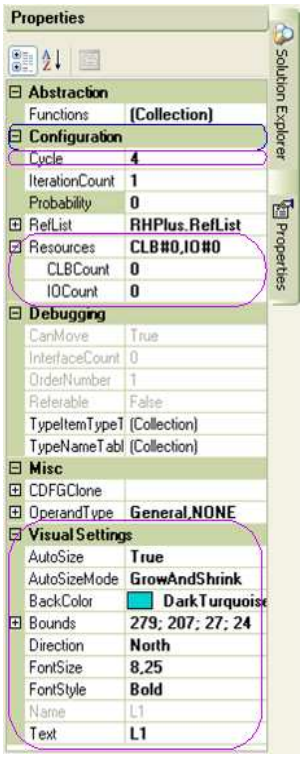
(c) Array



(d) Loop



(e) IfElif



(f) El and ExpList

Figure 3.7. RHPlus elements properties

3.5.4. RHPlus Elements

Functions have common properties: Cycle, Resources, and Visual Settings. HLS designer does not enter any value for Cycle and Resources. They are updated by RH(+) IDE. refName property is used to give the function's name. This name is used to refer to the function. In Figure 3.7, (a) shows these properties. General has Visual Settings as a common property. General property under Configuration Category on Properties window is used to enter properties. Length property is in terms of bits.

Location property shows where it is located. Possible locations are inhereted from the locations on **.bsp* files. refName is the name of the variable. Initial value for this variable is Value property. In Figure 3.7, (b) shows these properties.

Array has Visual Settings as a common property. Array property under Configuration Category on Properties window is used to enter properties of this data. Array data has Length property. It is the length of the array. Location property shows where array is located. Possible locations are inhereted from the locations on **.bsp* files. refName is the name of the array. In Figure 3.7, (c) shows these properties.

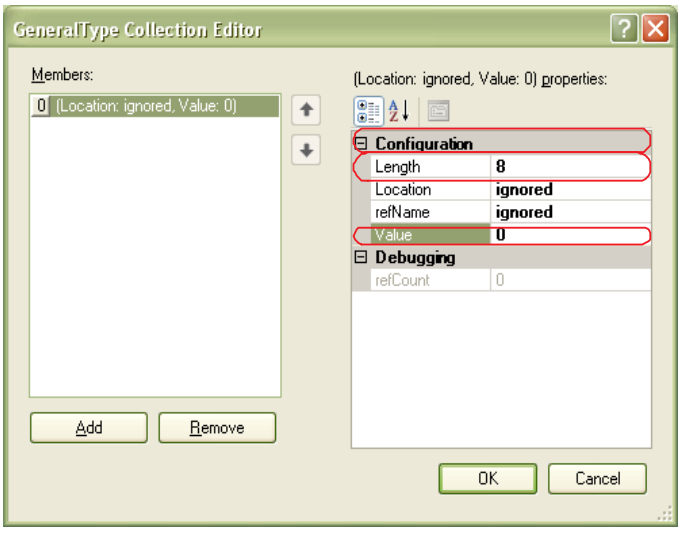


Figure 3.8. Length and value

All elements in array are General Data. They are entered via GeneralType Collection Editor. Array elements have Length and Value properties, shown in Figure 3.8.

Loops have common properties: Cycle, Resources, and Visual Settings. HLS designer does not enter any value for Cycle and Resources. They are updated by RH(+) IDE. IterationCount property is the possible number of the iterations for the loop. Probability property is the probability loop condition is true. IterationCount and Probability values will be used while calculating execution times of the possible paths. In Figure 3.7, (d) shows these properties.

If and Elif have common properties: Cycle, Resources, and Visual Settings. HLS designer does not enter any value for Cycle and Resources. They are updated by RH(+) IDE. Probability property is the probability condition is true. Probability value will be used while calculating execution times of the possible paths. In Figure 3.7, (e) shows these properties.

Each El have common properties: Cycle, Resources, and Visual Settings. HLS designer does not enter any value for Cycle and Resources. They are updated by RH(+) IDE. In Figure 3.7, (f) shows these properties.

Each expression list have common properties: Cycle, Resources, and Visual Settings. HLS designer does not enter any value for Cycle and Resources. They are updated by RH(+) IDE. In Figure 3.7, (f) shows these properties.

3.5.5. Interrupts

Interrupts are managed by HLS designer using **.bsp* and **.rhplus* files. All BSP components may have interrupt list. These entries are entered by using Interrupt Collection Editor, shown in Figure 3.9. This editor is launch by clicking the InterruptList entry under Configuration category in Properties Window. All interrupt needs a name as identifier. In the figure, we define DataReceivedInt and DataTransmittedInt interrupts.

The defined interrupts in **.bsp* files are accessed from **.rhplus* files to attach them as a source of an ISR. After clicking a function, its content is attached to Properties

Window. For each function we have Interrupt Settings category, shown in Figure 3.10. We can attach a function as an ISR by setting true for value of IsISR.

Moreover, we can assign source interrupts which trigger the execution of the function. For that, we activate InterruptSource Collection Editor by clicking InterruptSources. In this editor, we can add new entries. For each entry, we assign a source whose values are inhereted from **.bsp* files. In the figure, we see the interrupts we define in the selected BSP file: DataReceivedInt and DataTransmittedInt. All possible source interrupts are listed while determining ISR interrupt trigger sources.

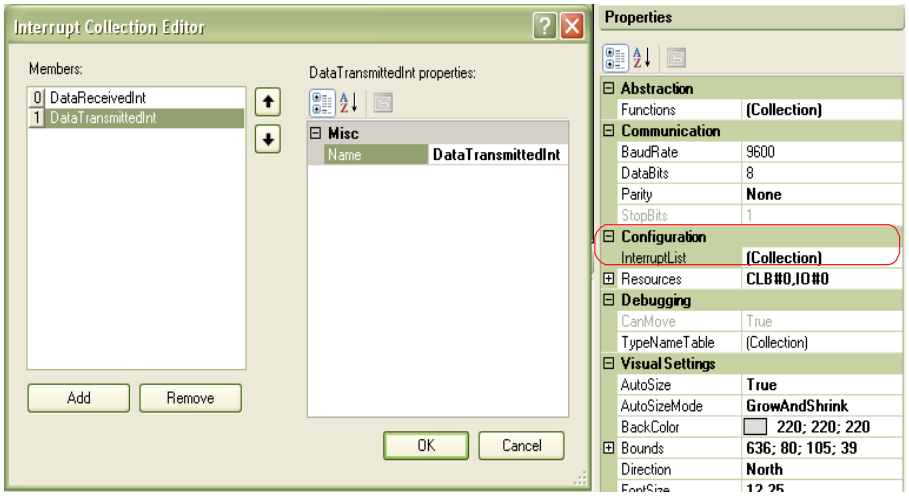


Figure 3.9. Interrupt list

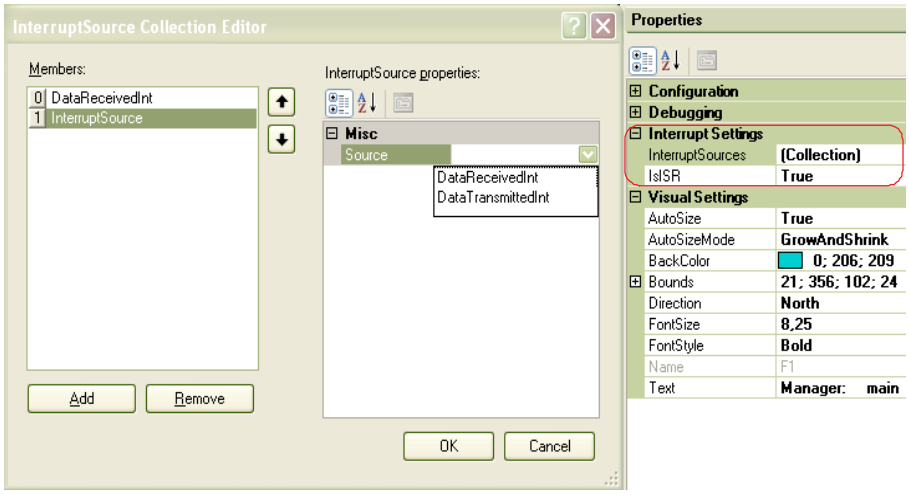


Figure 3.10. Interrupt sources

3.6. Editor Window

Editor Window is used to illustrute the current settings of the components to the HLS designer in text format. In **.rhplus* files, it is used as code editor. HLS designer can enter conditions for Loop, If, and Elif components from Editor Window. ExpList has a list of expressions. These expressions are also entered by using Editor Window. To activate Editor Window for a component, component must be clicked. For example, if HLS designer wants to write condition for a loop, he must click the loop component. Thus, the component will be selected. Then, Editor Window will show the contents of the selected component.

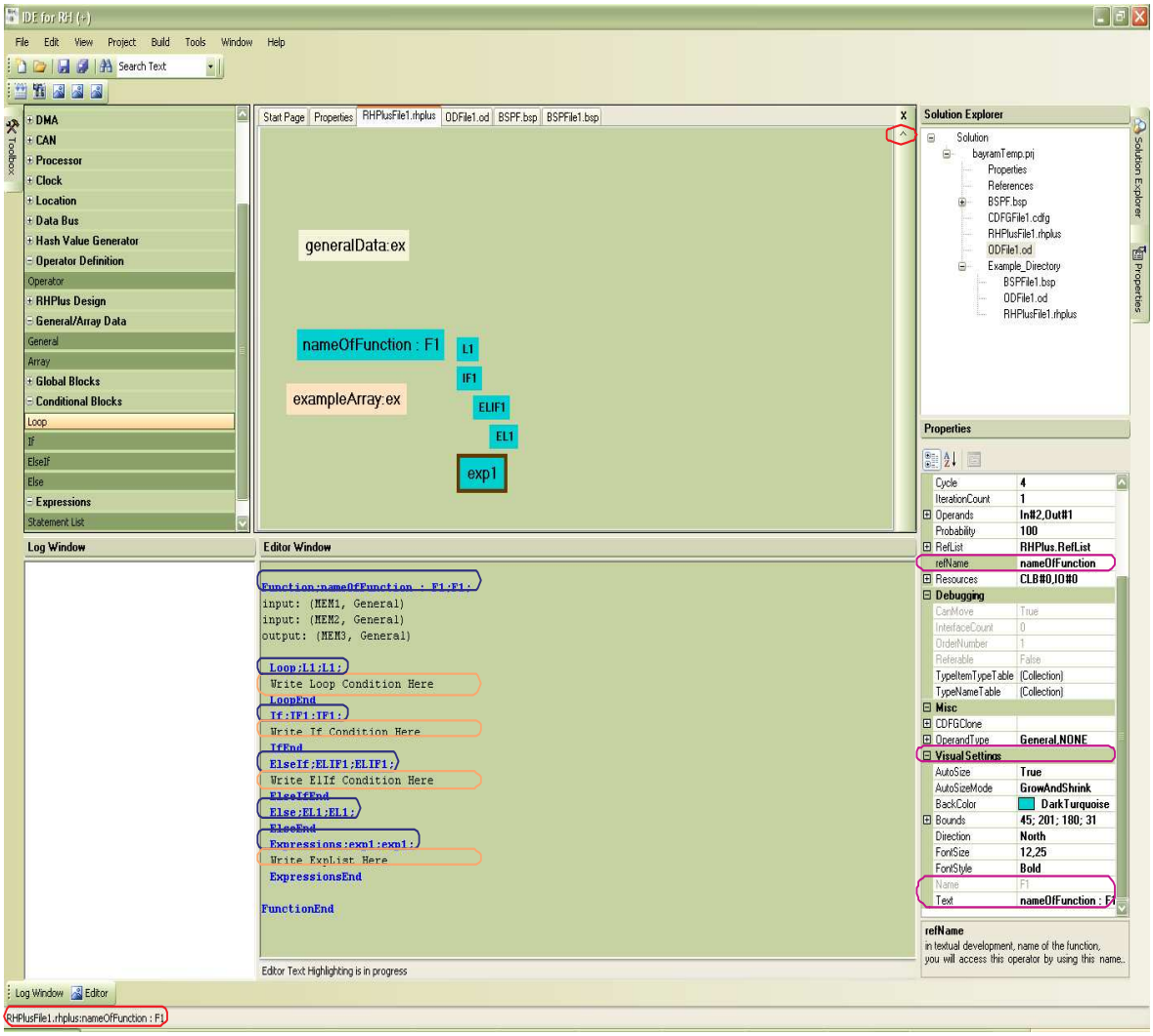


Figure 3.11. Editor window

Each body has a colorful text for its body start and body end. These colorful texts are keywords for the Editor Window. HLS designer cannot edit these keywords. Editing of them is prevented by RH(+) IDE. Keyword format is same for all blocks. It is concatenation of component type, component text, and component type. These are keyword members. They are seperated with ';' character in the keyword. Also, keywords end with ';' character. Component type is the type of block. Types are listed in Toolbox items, in Section 3.4. Component text can be changed by using Text property under Visual Styles category on the Properties Window. Component name is attached by the RH(+) IDE, and it is used to access to the block.

Furthermore, conditions include only one expression. However, ExpList is a list of expressions. Expressions in ExpList is seperated with ';' character. Expression format is Format 2, which is explained in Section 2.3.

In Figure 3.11, there is a program code to show the keywords on Editor Window. In this example program, function has Loop, If, Elif, El, and ExpList. Bodies of Loop, If, Elif, and El are empty. The lines to where HLS designer will write conditions and expressions are also shown. In addition, keyword members on Properties window are shown in this figure.

3.7. OD Design

In the active **.od* file, HLS designer can add a new operator to it. For that, HLS designer selects the Operator under Operator Definition Category from Toolbox. He drags it towards to the file, and drops it onto the file. Like this, many operators can be added to the current scope. This just added operator's content is empty so Operator Text is "->" as shown in Figure 3.12. Operator properties can be setted by the HLS designer as explained in Section 3.5.3. After that, Text will be updated with operand locations and operator name. This is also shown in Figure 3.12.

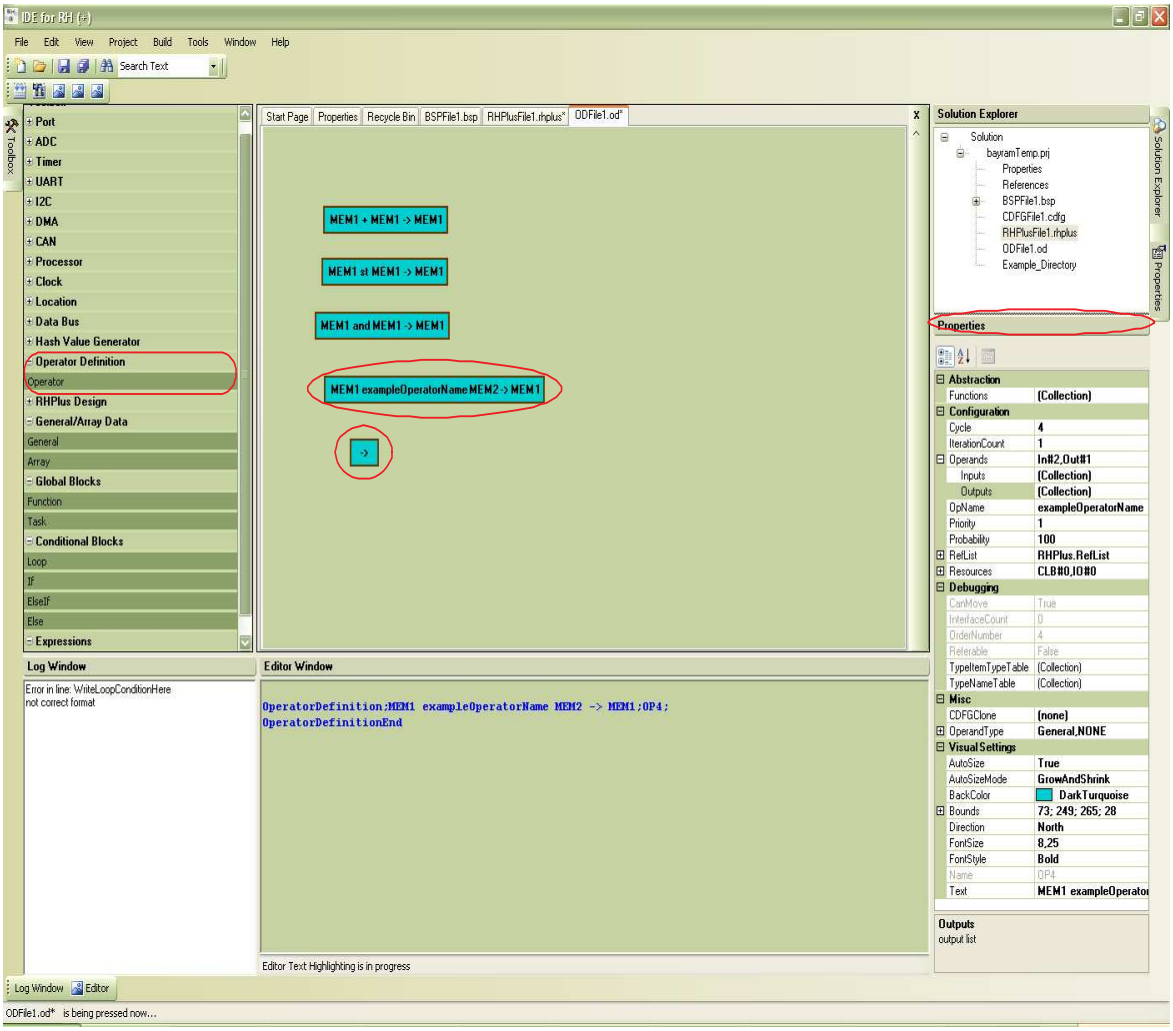


Figure 3.12. Operator empty and filled states

3.8. RHPlus Design

3.8.1. Navigation

Navigation is possible for **.rhplus* files. This files include many blocks like Function, Loop, If, Elif, and El. An example can be seen in Figure 3.11. In this figure, contents of nameOfFunction function is displayed. In this content, HLS designer cannot add a function definition to the *RHPlusFile1.rhplus* file because current scope is not valid for function definition. Current scope is shown in the status strip located bottom-left of RH(+) IDE. To navigate towards the global scope, HLS designer must use ‘ ^ ’ button which is also shown in the figure. If HLS designer wants to enter a

new scope, he must click the component with pressed SHIFT key. Except ExpList, all components are expandable.

3.8.2. Program Flow Organization

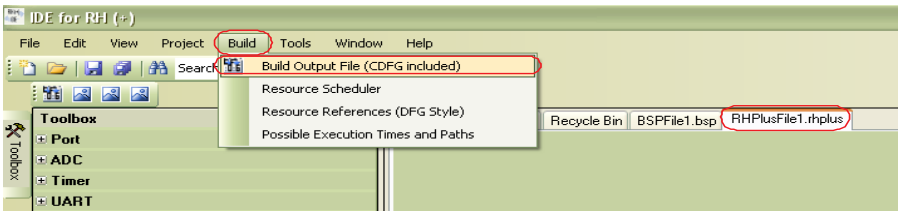
In the active scope, HLS designer can add a new component to it. For that, HLS designer selects the component from Toolbox. He drags it towards to the file, and drops it onto the file. Like this, many components can be added to the current scope. To activate the components, they must be integrated to the program flow. First component of the current scope is decided via clicking with pressed 'L' key. Then, another component may be appended to the first component. For that, HLS designer selects another component existing in the current scope. He drags it towards the first component, and drops it onto the first component with pressed 'F' key. Appended component is follower of the first component.

Furthermore, both Elif and El can be appended to If and Elif components. It is not possible to add Elif and El to other components. Elif and El are called as steppers for If component. El is also a stepper for Elif. To append Elif or El to If and Elif HLS designer drags and drops them onto latter ones. Steppers are indented according to its owner: If or Elseif, which can be seen in Figure 3.11.

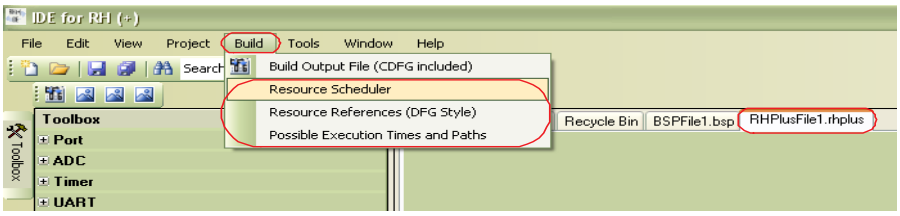
Moreover, exp1 follows IF1. For that, we have clicked to exp1 and dragged it towards If, then dropped it onto If with pressed 'F' key. exp1 is follower of IF1. Elif and El do not accept any follower. Therefore, if we try dropping exp1 onto the ELIF1 or EL1, it will not be accepted. Lastly, "nameOfFunction: F1" is expanded. Current scope is body of this function. All visible components in this scope are in the same level/depth. Current scope components are L1, IF1, ELIF1, EL1, and exp1 for the figure. If we click to L1 with pressed SHIFT key, its contents will be shown at the right of L1. Then, if we click IF1 with pressed SHIFT key, L1 contents will be invisible and IF1 contents will be visible.

3.8.3. Key Operators

There are three operators that are keywords for RH(+) IDE: indicator for function and operator references ".", assignment operator "=", and exact time indicator "?". These operators must not be used as operator or function names. "." is added to the start of operators and functions when they are called. "=" is used assign output of operation to another variable. As a result, "=" operator has two operands. "?" is appended to the end of operator or function names. When calling operators or functions from **.rhplus* files, we may want them execute in certain time intervals. For example, "secondDelay" operator has an exact time to start operation in example shown in Section 5.3.



(a) Build output file

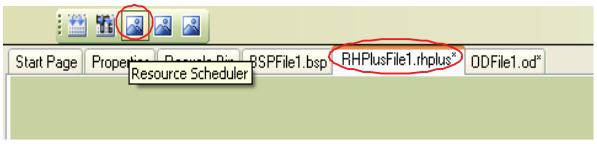


(b) Analysis tools

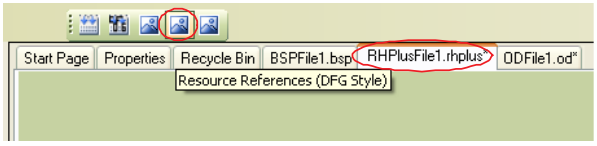
Figure 3.13. Menu tabs

3.9. Menu Items

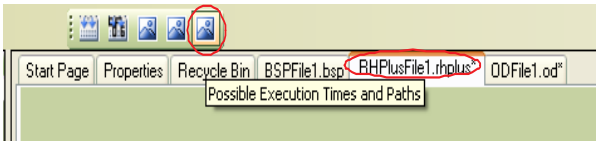
Main capabilities of RH(+) IDE are listed in Section 3.3. These functionalities can be activated via shortcuts. All of these capabilities can be used if the current file is a **.rhplus* file. In main menu, under Build tab, there are lines to access these capabilities. (a) and (b) show them in Figure 3.13. (a), (b), (c), and (d) in Figure 3.14 show short cuts for Resource Scheduling, Resource References, Possible Paths and Execution Times, and Output File (CDFG Included), respectively.



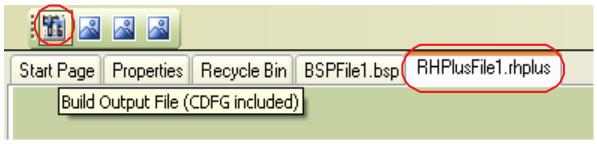
(a) Resource scheduler



(b) Resource references



(c) Possible paths



(d) Output file

Figure 3.14. Menu short cuts

4. RH(+) IDE: DESIGN WITH EXAMPLES

4.1. BSP Design with Example

We designed an example board support package by using Toolbox items, dragging and dropping them on **.bsp* file. It includes possible hardware components that will be programmed on reconfigurable hardware at run-time, when they are needed. All components are included in Figure 4.1. Their connections are listed below. The numbers located to the left or bottom of the components are the interfaces they have. Each interface has a corresponding interface with the same number.

- Port1: This is a Port Core. PD1 is connected to it, *interface0*, and it is attached to Processor1, *interface25*.
- PD1: This is a Port Digital and connected to Port1, *interface0*.
- Port2: This is a Port Core. PA1 is connected to it, *interface1*, and it is attached to Processor1, *interface26*.
- PA1: This is a Port Analog and connected to Port2, *interface1*. This component is used by PA1 via *interface2*.
- ADC: This is an ADC Core and connected to PA1, *interface2*, and it is attached to Processor1, *interface18*.
- UART1: This is a UART Core. TX1 and RX1 are used with only this core, and it is attached to Processor1, *interface22*. Also, data can be received from MEM1 to this core via *interface8*. DMA1 is attached to this core, in other words, DMA1 is the bridge for this component. Data from this core to DMA1 destination location, and data from DMA source location to this core can be transfered.
- DMA1: This is a DMA Core, and bridge for UART1. MEM1, MEM3, MEM4 data can be used in transfers, *interface7*, *interface9*, *interface10*, respectively.
- CAN1: This is a CAN Core. CT1 is used with only this core, and it is attached to Processor1, *interface23*. DMA2 is attached to this core, in other words, DMA3 is the bridge for this component. Data from this core to DMA2 destination location, and data from DMA source location to this core can be transfered.

- DMA2: This is a DMA Core, and bridge for CAN1. MEM1, MEM5, MEM6 data can be used in transfers, *interface12*, *interface11*, *interface17*, respectively.
- MEM6: This is a Location. DMA3 is attached to this core, in other words, DMA3 is the bridge for this component. Data from this core to DMA3 destination location, and data from DMA source location to this core can be transfered. Data of this component can be used data transfer for DMA2.
- DMA3: This is a DMA Core, and bridge for MEM6. MEM1, MEM2, MEM5 data can be used in transfers, *interface14*, *interface15*, *interface16*, respectively.
- Data Bus1: This is a Data Bus Core. It has two bus clients: MEM3, MEM4.
- MEM1: This is a Location. Its data can be used in data transfers of DMA1, DMA2, DMA3 via *interface7*, *interface12*, *interface14*, respectively. MEM1 is used by Processor1 via *interface27*.
- MEM2: This is a Location. Its data can be used in data transfers of DMA3 via *interface15*. Also, data transfer can be applied directly between this component and UART1 via *interface8*. MEM2 is used by Processor1 via *interface28*.
- MEM3: This is a Location. Its data can be used in data transfers of DMA1 via *interface9*.
- MEM4: This is a Location. Its data can be used in data transfers of DMA1 via *interface10*.
- MEM5: This is a Location. Its data can be used in data transfers of DMA2, DMA3 via *interface11*, *interface16*, respectively. MEM5 is used by Processor1 via *interface29*.
- Data Bus 1: This is a Data Bus Core. It has two clients, MEM3 and MEM4. This component is used by Processor1 via *interface24*.
- Timer Core 1: This is a Timer Core. It is used by Counter Component, Input Capture 1, and PWM Component via *interface3*, *interface4*, *interface5*.
- Counter: This is a Counter Core. It uses Timer Core 1 via *interface3*.
- Input Capture 1: This is an Input Capture Core. It uses Timer Core 1 via *interface4*.
- PWM: This is a PWM Core. It uses Timer Core 1 via *interface5*.
- Timer Core 2: This is a Timer Core. It is used by Input Capture 2 via *interface6*. This component uses Clock1 via *interface13*, and this is used by Processor1 via

interface20.

- Input Capture 2: This is an Input Capture Core. It uses Timer Core 2 via *interface6*.
- Clock1: This is a Clock Core. It is used by Timer Core 2 via *interface6*.
- I2C1: This is a I2C Core. It is used by Processor1 via *interface19*.
- Processor1: This is a Processor Core. This component can use the remaining components via *interface18*, *interface19*, *interface20*, *interface21*, *interface22*, *interface23*, *interface24*, *interface25*, *interface26*, *interface27*, *interface28*, *interface29*.

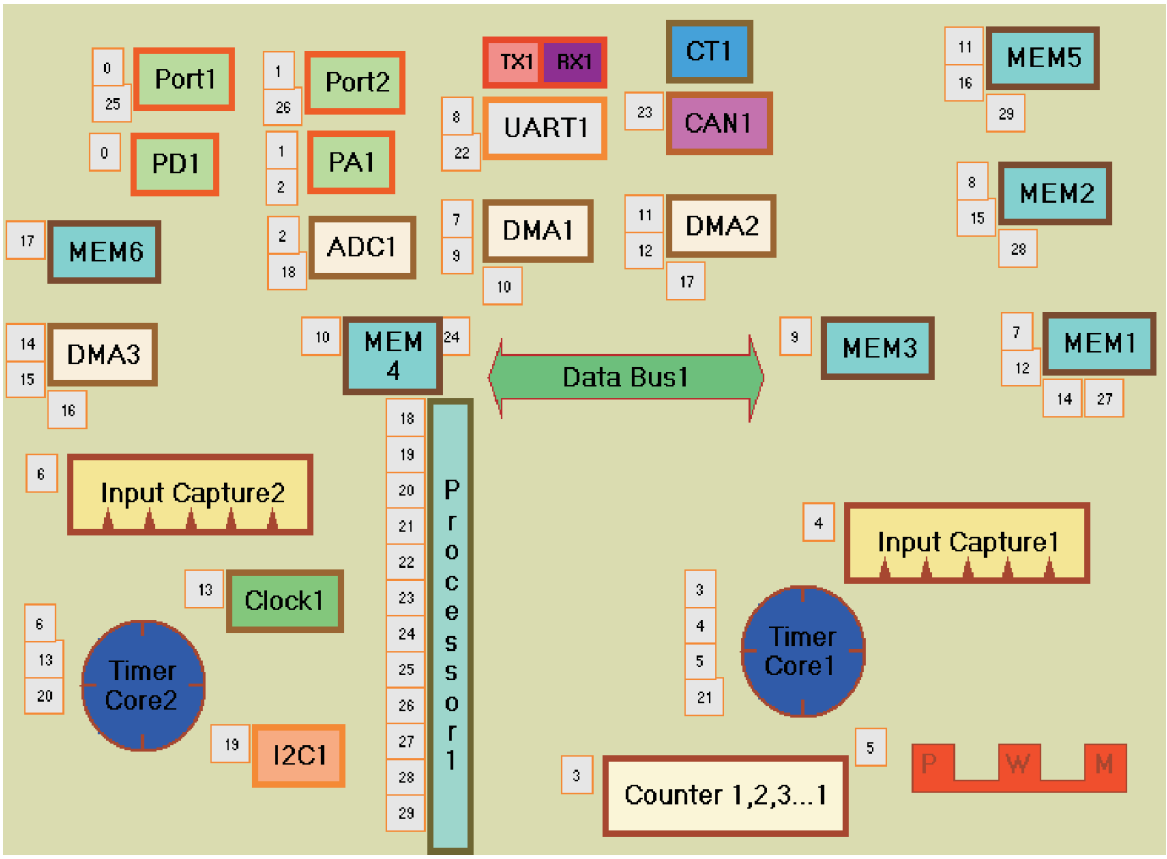


Figure 4.1. Project BSP file

Properties Window is used to set the properties of the BSP components. To activate BSP file, at the right side of the Figure 4.1, there is a context menu including option *Set As Project's BSP File*. By using this option, the **.bsp* files can be assigned as project BSP file. Namely, all components are possible to be referenced from the application, after now.

4.2. OD Design with Example

An **.od* file must be ready to be able to develop an application. Operator Definition category on the Toolbox has Operator subentry. This entry is dragged over the **.od* file, and then dropped to define an operator. While creating these operators, memory locations from selected BSP file are referred. Figure 4.2, there are four operator whose names are different. Also, each of them is overloaded three times. There are totally 12 operators in this example.

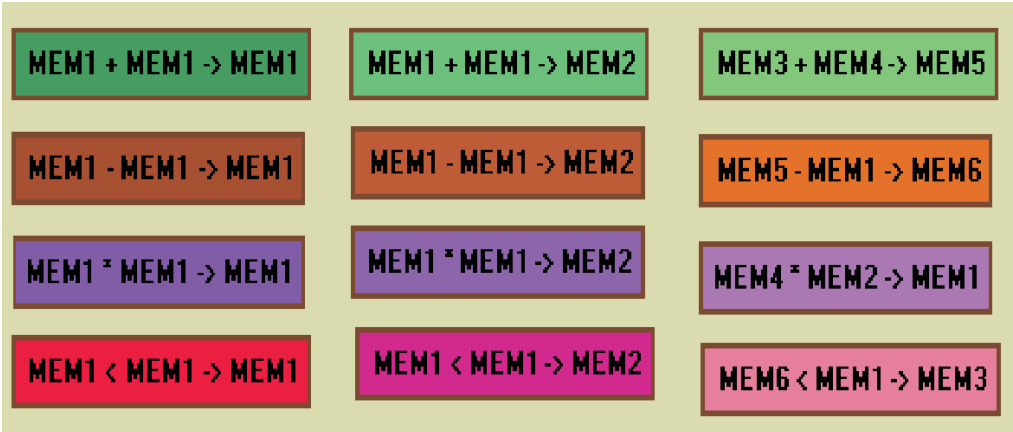


Figure 4.2. Project OD file

- "+": Three types of addition are defined. First one is "MEM1 + MEM1 -> MEM1". Inputs are located in MEM1. Output is located in MEM1. Second one is "MEM1 + MEM1 -> MEM2". Inputs are located in MEM1. Output is located in MEM2. Third one is "MEM3 + MEM4 -> MEM5". Inputs are located in MEM1 and MEM4. Output is located in MEM5.
- "-": Three types of substraction are defined. First one is "MEM1 - MEM1 -> MEM1". Inputs are located in MEM1. Output is located in MEM1. Second one is "MEM1 - MEM1 -> MEM2". Inputs are located in MEM1. Output is located in MEM2. Third one is "MEM5 - MEM1 -> MEM6". Inputs are located in MEM5 and MEM1. Output is located in MEM6.
- "*": Three types of multiplication are defined. First one is "MEM1 * MEM1 -> MEM1". Inputs are located in MEM1. Output is located in MEM1. Second one is "MEM1 * MEM1 -> MEM2". Inputs are located in MEM1. Output is

located in MEM2. Third one is "MEM4 * MEM2 -> MEM1". Inputs are located in MEM4 and MEM2. Output is located in MEM1.

- "<": Three types of smaller than are defined. First one is "MEM1 < MEM1 -> MEM1". Inputs are located in MEM1. Output is located in MEM1. Second one is "MEM1 < MEM1 -> MEM2". Inputs are located in MEM1. Output is located in MEM2. Third one is "MEM6 < MEM1 -> MEM3". Inputs are located in MEM6 and MEM1. Output is located in MEM3.

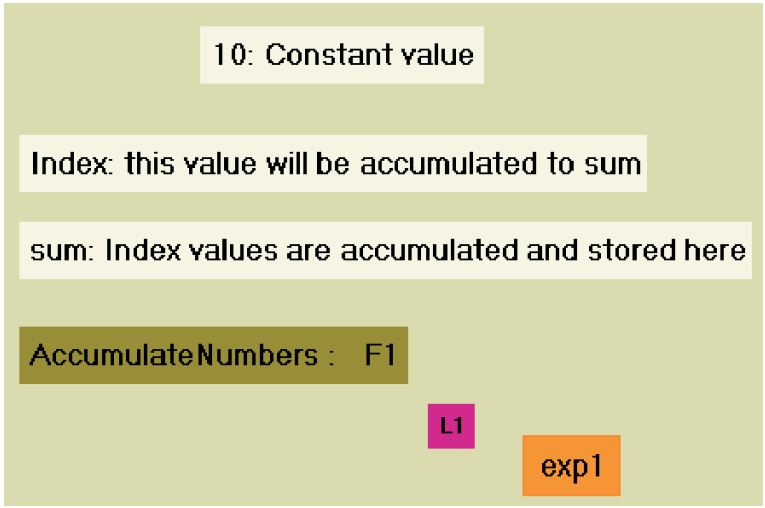
Properties Window is used to set the properties of the operators. To activate OD file, at the right side of the Figure 4.2, there is a context menu including option *Set As Project's OD File*. By using this option, the **.od* files can be assigned as project OD file. Namely, all operators are possible to be referenced from the application, after now. After this option is selected, these operators are loaded to the Toolbox at the left side. HLS designer can verify the correctness of them by examining the operators category in the Toolbox.

4.3. RHPlus Design with Examples

In this section we present examples to be able to analyze RHPlus designs. We have three simple examples. They introduce our analysis tools from simple aspects to more complex ones. Our aim is to provide the base capability for HLS designer for further Traffic Controller example shown in Chapter 5, which is a more complex design example than these simple ones.

4.3.1. Example 1

In Figure 4.3, there is a simple example to introduce rhplus design. We see visual design and text mode design together in this figure. There are three global General data: constant 10, Index, and sum. AccumulateNumbers function has a loop. This loop has a condition and a statement. As a result we have two BB which on we apply our analyses.




```
Function;AccumulateNumbers :   F1;F1;
input: (MEM1, General)
input: (MEM1, General)
output: (MEM1, General)

 Loop;L1;L1;
 .<(Index,10);

  Expressions;exp1;exp1;
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  ExpressionsEnd

 LoopEnd

FunctionEnd
```

Figure 4.3. Program for Example 1

We have a simple example so the results of the analysis are not so complex. In Figure 4.4, possible execution paths for this simple program can be seen. Best case and worst case are also shown. Critital path is the worst case path, which is *L1/exp1*. Timing data do not reflect real values, we only show paths in this example.

In Figure 4.5, the number of resource references for each basic block in AccumulateNumbers function can be seen. L1 is the loop condition, and it refers to Index, "<", and 10 for only once. exp1 is the body of the loop, and it refers to Index, sum, "+", and "*" for three, two, two, and one, respectively. "=" operator is common for all application files, it can be ignored in the analysis when it is shown on graphs.

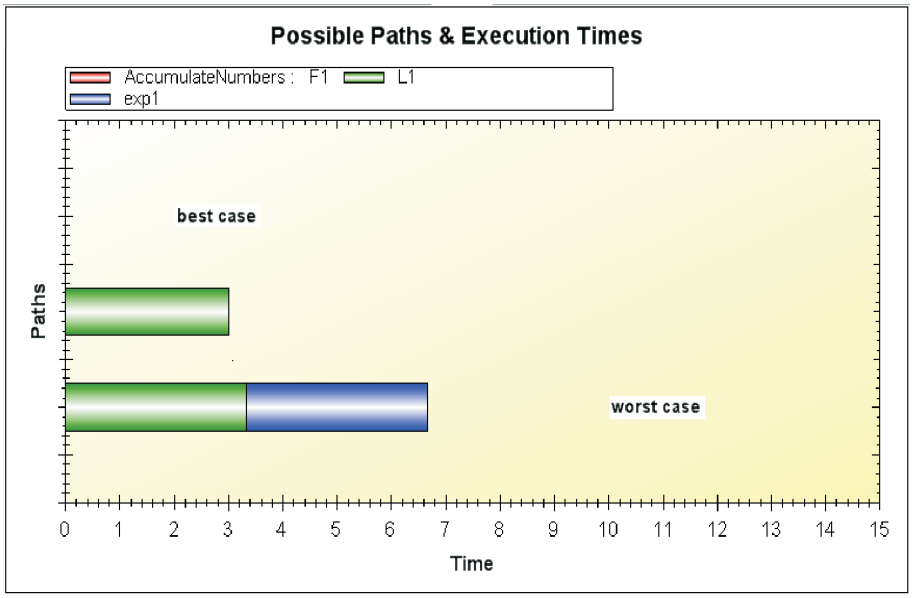


Figure 4.4. Possible execution paths for Example 1

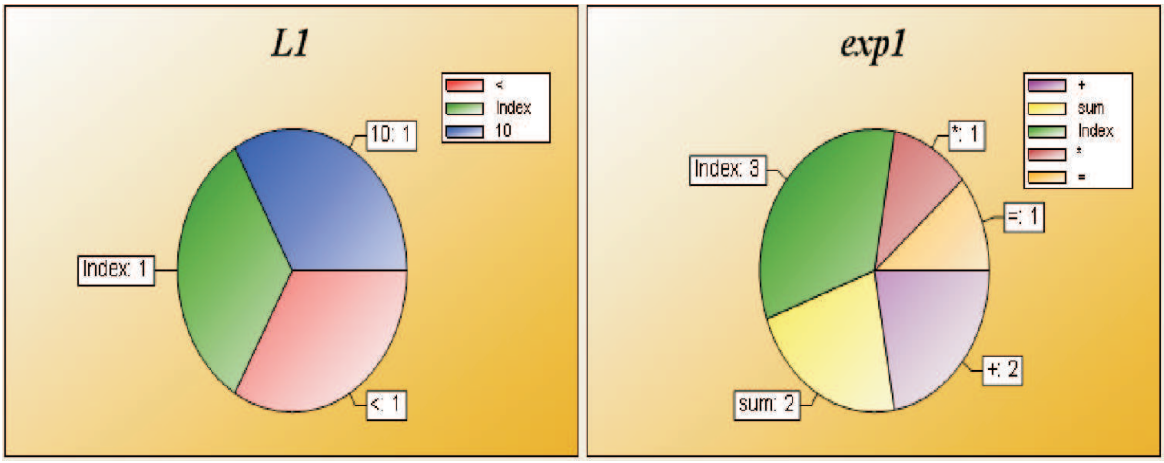


Figure 4.5. References for Example 1

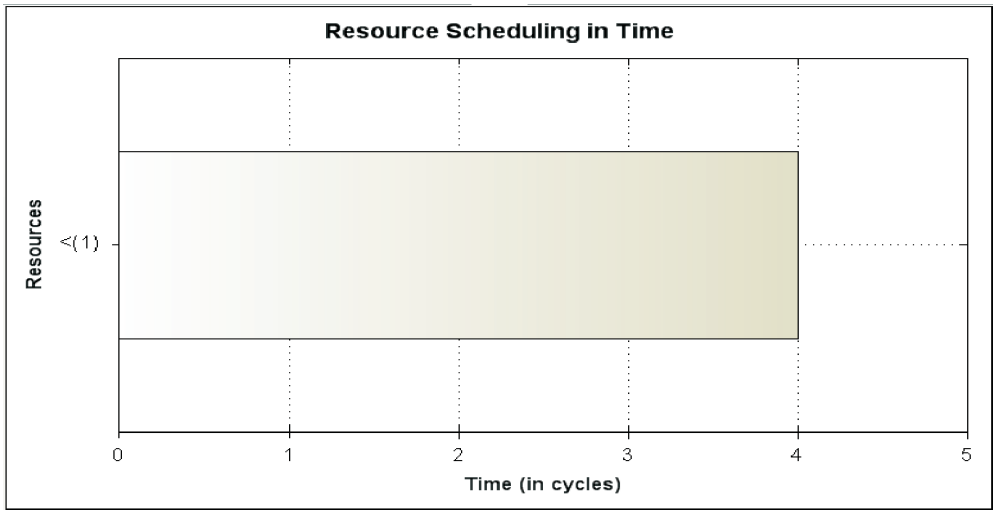


Figure 4.6. Scheduling for loop condition in Example 1

In Figure 4.6, how resources are scheduled to the loop condition can be seen. Since only "<" operator is used in this basic block, only this resource is assigned to the condition.

In Figure 4.7, how resources are scheduled to the loop body can be seen. Since "+" and "*" operators are used in this basic block, these resources are assigned to the loop body.

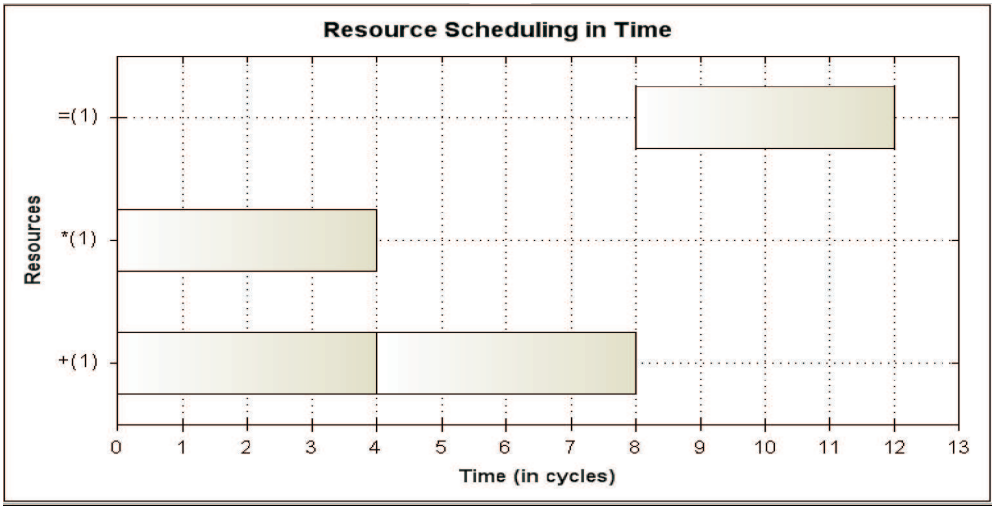


Figure 4.7. Scheduling for loop body in Example 1

4.3.2. Example 2

In Figure 4.8, we edit the program presented in Figure 4.3. There is no meaningful difference in visual design. We only add some BB whose functionality is not considered. Therefore, we do not need to show visual design here. The main difference between two programs is that we copy the same statement line into some BB. In this example, we want to show how we can decrease execution delay of a BB by incrementing the number of the instances for the required resources for the BB.

In Figure 4.9, possible execution paths for the program shown in Figure 4.8. Best case and worst case has been shown. Critital path is the worst case path, which is *L1/exp1/if1/elif1/el1/exp25/exp41*. In other words, if program flow follows this path, the execution delay for the program will be the longest.

```
Function;AccumulateNumbers :    F1;F1;
input:  (MEM1, General)
input:  (MEM1, General)
output:  (MEM1, General)

 Loop;L1;L1;
 .<(Index,10);

  Expressions;exp1;exp1;
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  ExpressionsEnd

 LoopEnd
 If;IF1;IF1;
  .<(Index,10);
  Expressions;exp9;exp9;
  ExpressionsEnd

 IfEnd
 ElseIf;ELIF1;ELIF1;
  .<(Index,10);
  Expressions;exp17;exp17;
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  ExpressionsEnd

 ElseIfEnd
 Else;EL1;EL1;

  Expressions;exp25;exp25;
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  .=(Index,.+(.+(sum,Index),.*(sum,Index)));
  ExpressionsEnd

 ElseEnd
 Expressions;exp41;exp41;
 .=(Index,.+(.+(sum,Index),.*(sum,Index)));
 ExpressionsEnd

FunctionEnd
```

Figure 4.8. Program for Example 2

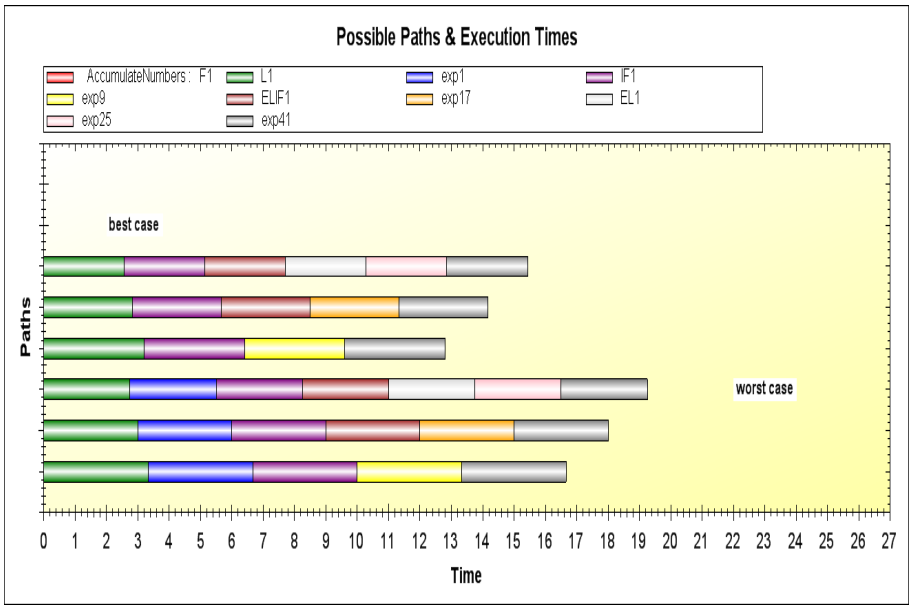


Figure 4.9. Possible execution times for Example 2

The number of resources references for Loop condition, Loop body, If condition, Elif condition, El body, and the last expressions of function body can be shown in Figure 4.10, in Figure 4.11, in Figure 4.12, in Figure 4.13, in Figure 4.14, and in Figure 4.15. Because El has no condition, it has no resource scheduling graph.

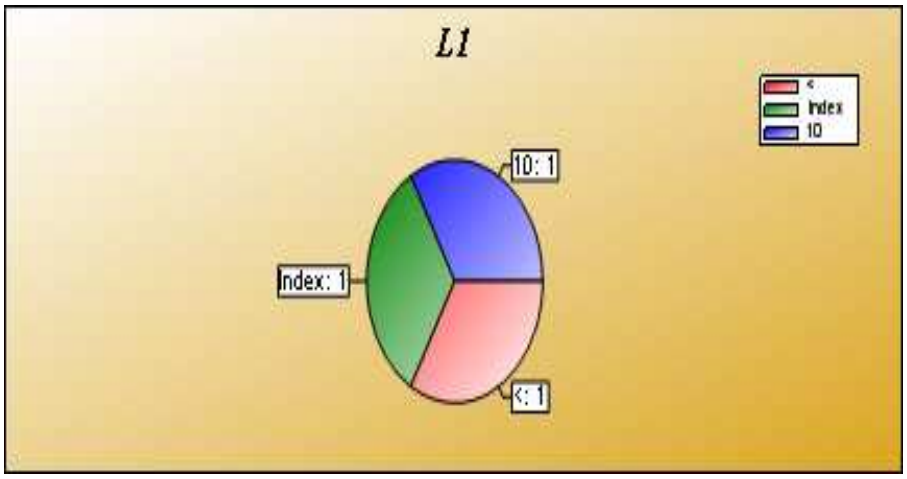


Figure 4.10. References in loop condition in Example 2

As an example, in Figure 4.10, we have three resources which are shown. We can see from the pie centered in the figure. On the upper-right side in the figure, we see a square which shows the assigned colors for each resources. Index, 10, and "<" have different toned pie items, respectively. By examing this graph, we see that Index, 10, and "<" are referred for only once in the Loop condition.

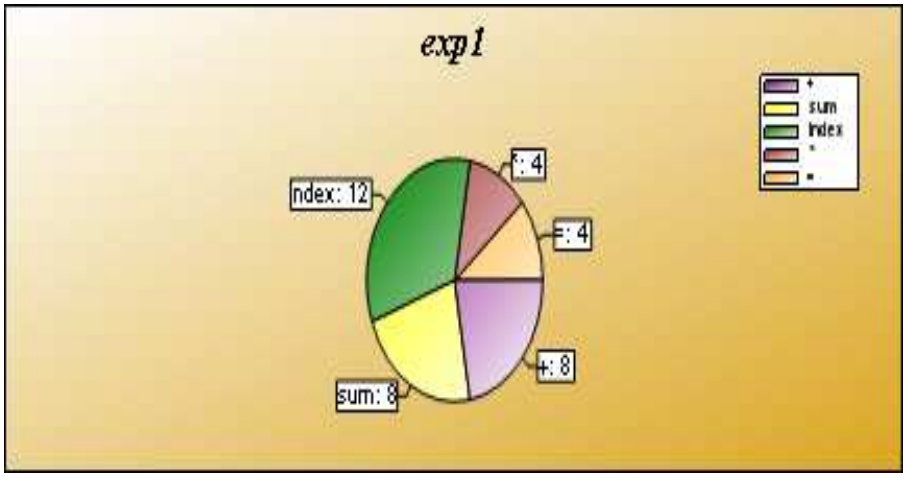


Figure 4.11. References in loop body in Example 2

As an another example, we see many references in Figure 4.11 for loop body. It is shown that Index, sum, *, +, and = are referred for 12, eight, four, four, and eight times, respectively. The remaining resource reference graphs show the similar data about the remaining BB.

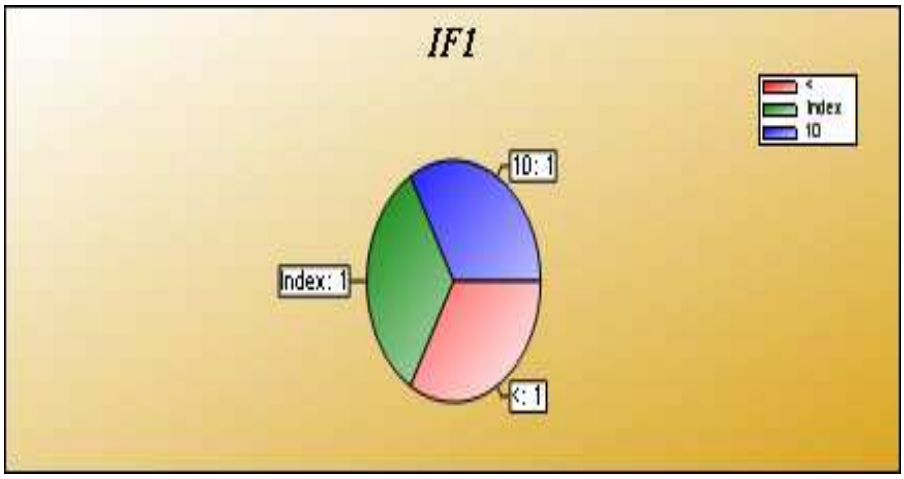


Figure 4.12. References in If condition in Example 2

The resource reference graphs can be used by HLS designer as decision points. He can decide the new values for the number of the instances for the corresponding BB. According to these new values, he can restart the scheduling analysis.

The number of the reference graphs are equal to the number of the BB included in the LRH(+) software. HLS designer can analyse the references for any block. When he wants to analyze the block which has many BB, it may be difficult to follow the all BB. Therefore, HLS designer may select to show each BB separately.

By using Ref Collection Editor of Properties Window, the number of resources for each BB can be decided. It is shown in Figure 4.16. Deciding about the number of the required hardware blocks is useful when HLS designer wants critical sections be executed with short delay times. In this case, he assign more hardware block instances for the critical software part. Ref Collection Editor shows HLS designer the number of the references to the selected resource in the selected scope. HLS designer use this value as an entry for his decision.

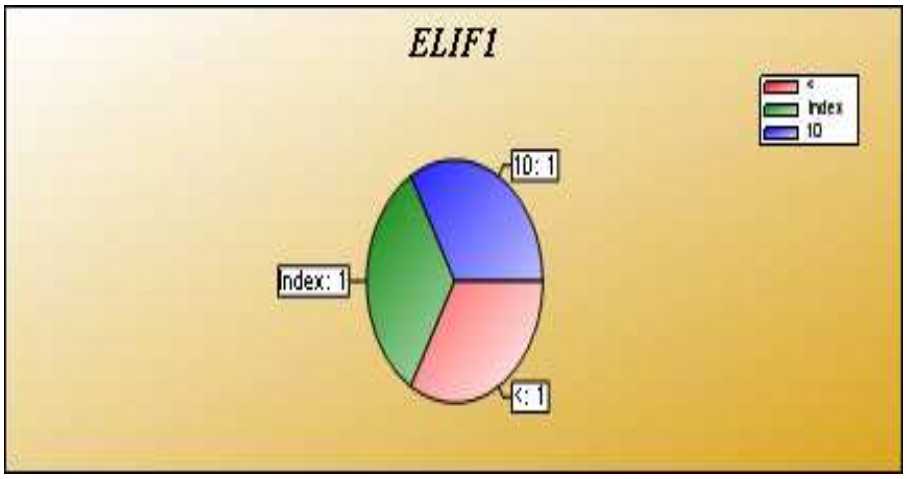


Figure 4.13. References in Elif condition in Example 2

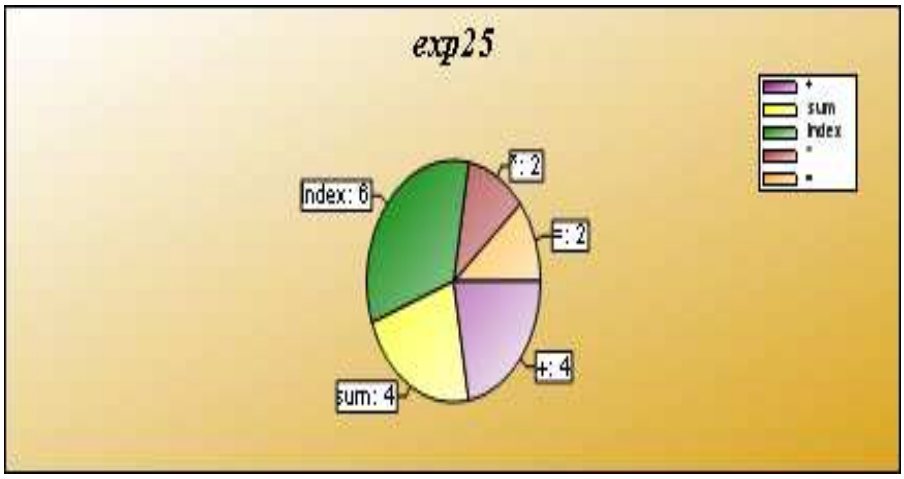


Figure 4.14. References in El body in Example 2

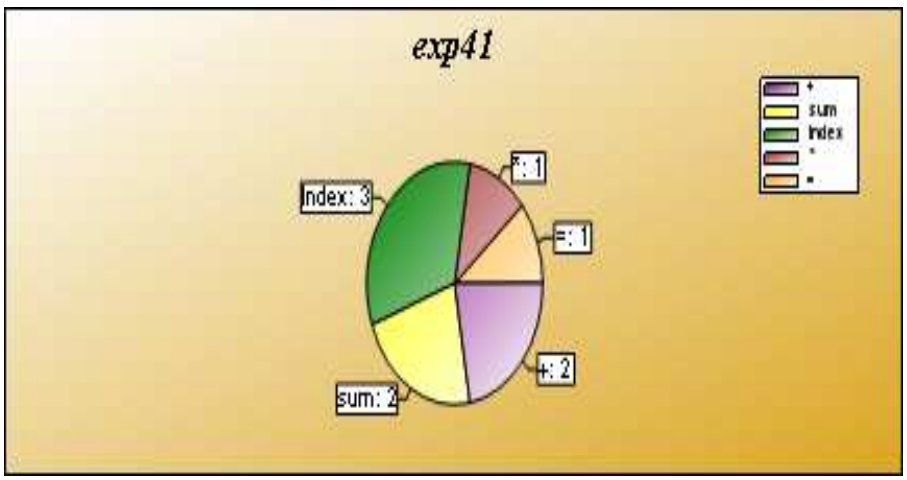


Figure 4.15. References for last expressions of function in Example 2

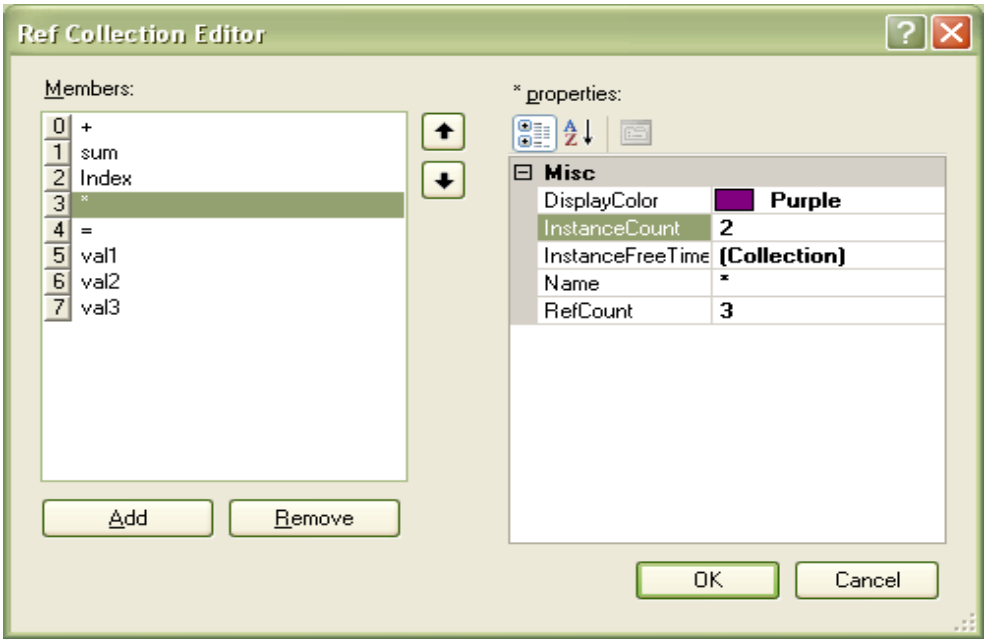


Figure 4.16. Reference collection editor in Example 2

In this example, we show three different decisions. Therefore, we have three scheduling analysis. If there is only one instance for each operator used in If body's expressions, the resource scheduling graph shown in Figure 4.17 is produced. When the number of "+" resource is two, the resource scheduling graph shown in Figure 4.18 is produced. When the number of both "+" and "*" resources are three, the resource scheduling graph shown in Figure 4.19 is produced.

It must be noted that incrementing the number of the resources decrease the execution delay of the BB. We have execution delays of 28, 20, and 12 cycles for our three decisions, respectively. This shows that HLS designer must decide which metric is important for his design. In this example, we show that when the number of the hardware blokc units are increased execution delay decreases. In other words, although we have a hardware cost here we get performance increase.

Finally, to use RH(+) IDE efficiently, HLS designer must use analysis tools together efficiently. Their results give new ideas about the design and trigger new changes in decisions. These changes may conclude a different target architecture requirement. As a result, target architecture properties can be changed during HLS design.

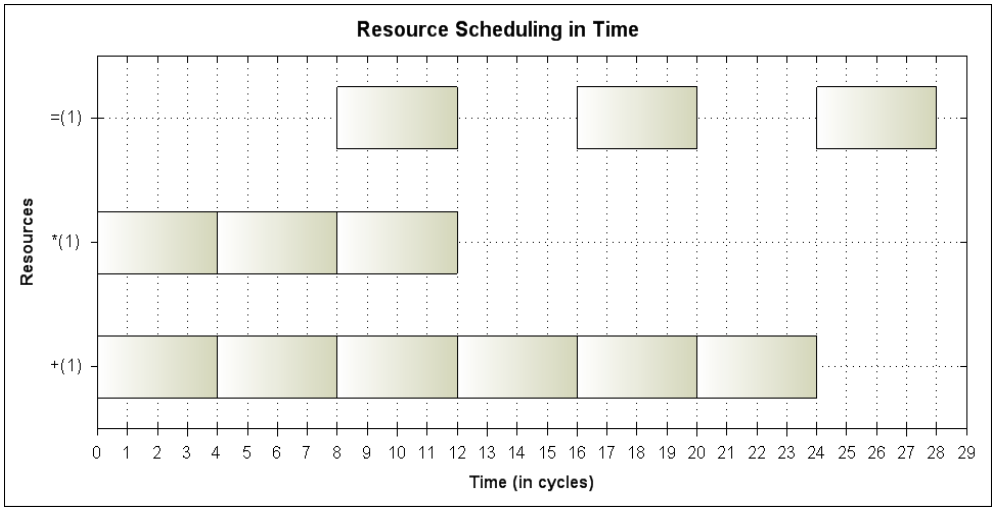


Figure 4.17. Scheduling for If: one instance for resources in Example 2

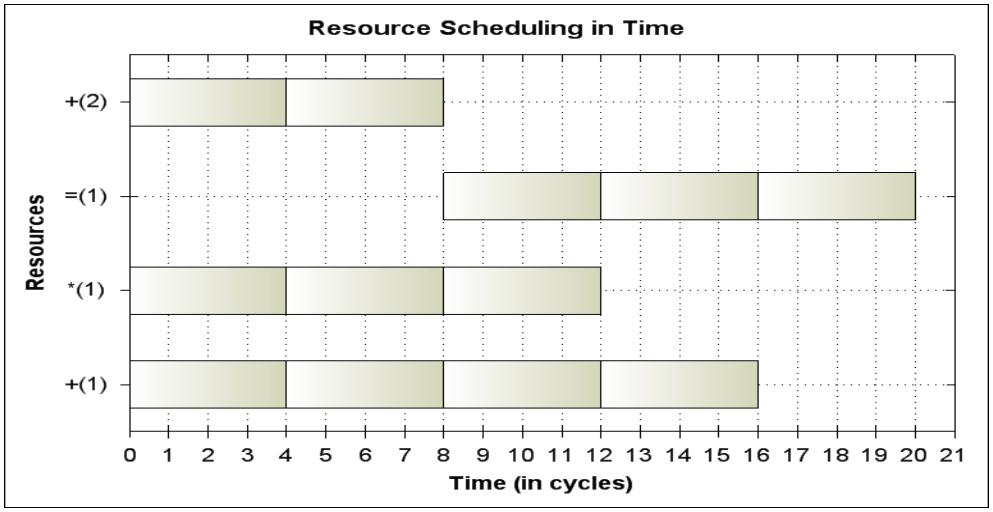


Figure 4.18. Scheduling for If: two "+" in Example 2

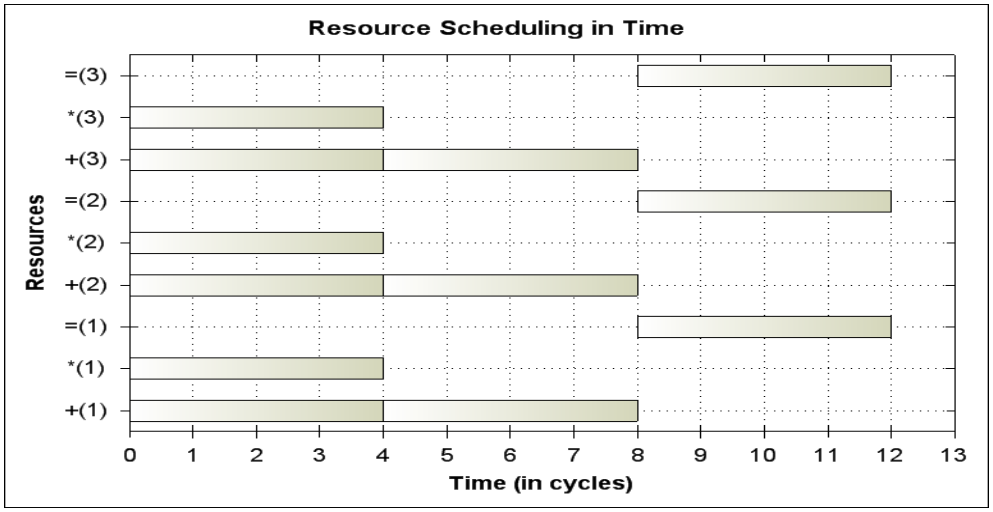


Figure 4.19. Scheduling for If: three "+" and "*" in Example 2

4.3.3. Example 3

In this example, code shown in Figure 4.20 will be considered. Here there is an expression box containing a series of statements. In previous examples, we show resource references in detail and we do not need showing them again here. We present a work by using resource scheduling analysis tool. We have three different decisions about the mentioned expression block. As a result, we have three resource scheduling graphs for this example.

```
Expressions;exp17;exp17;
.=(val1,.+(.+(sum,Index),.*(sum,Index)));
.=(val2,.+(.+(sum,Index),.*(sum,Index)));
.=(val3,.+(.+(sum,Index),.*(sum,Index)));
ExpressionsEnd
```

Figure 4.20. Program for Example 3

We have three statements line in this example. In each line, we have the same operation but operation results are written to the different variables: val1, val2, val3. It must be noted that input variables, which are sum and Index, are only used as input. In other words, they are not modified in any statement line.

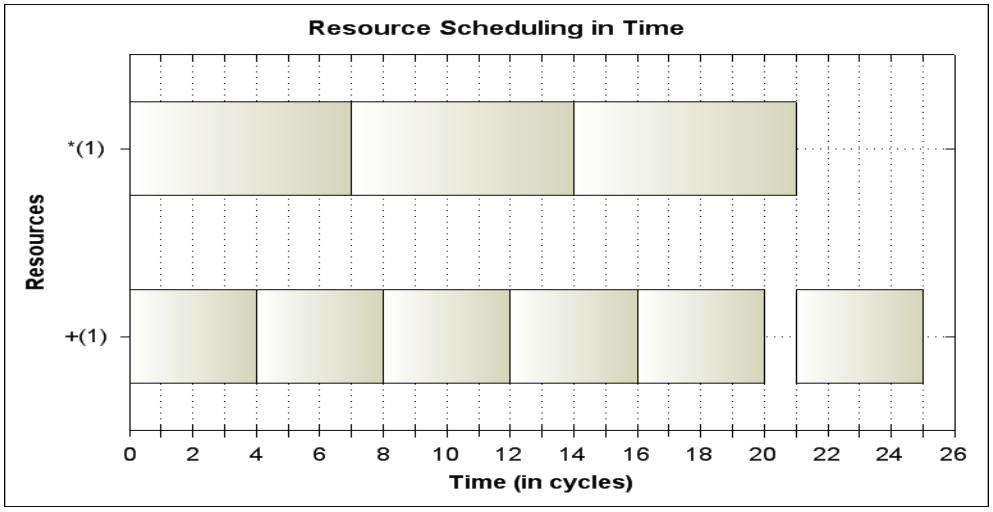


Figure 4.21. Scheduling with one "+" and one "*" in Example 3

We have two operators in this example: "+" and "*". Their execution time values are 4 cycles and 7 cycles, respectively. We show three decision results as resource

scheduling graphs. Firstly, in the case of there are one "+" and one "*" resource, Figure 4.21 is produced. Secondly, if we add a second "*" instance to this block, Figure 4.22 is produced. Lastly, if "+" and "*" resources have three instances on reconfigurable hardware when this block is running, Figure 4.23 is produced. We want to note that incrementing the number of the resources decrease the execution delay of the BB. We have execution delays of 25, 18, and 11 cycles for our three decisions, respectively.

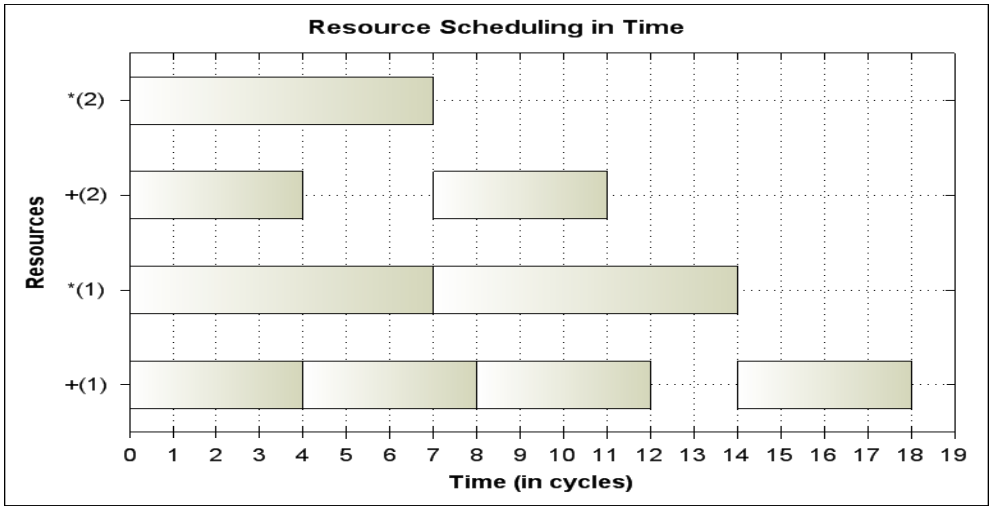


Figure 4.22. Scheduling with two "+" and two "*" in Example 3

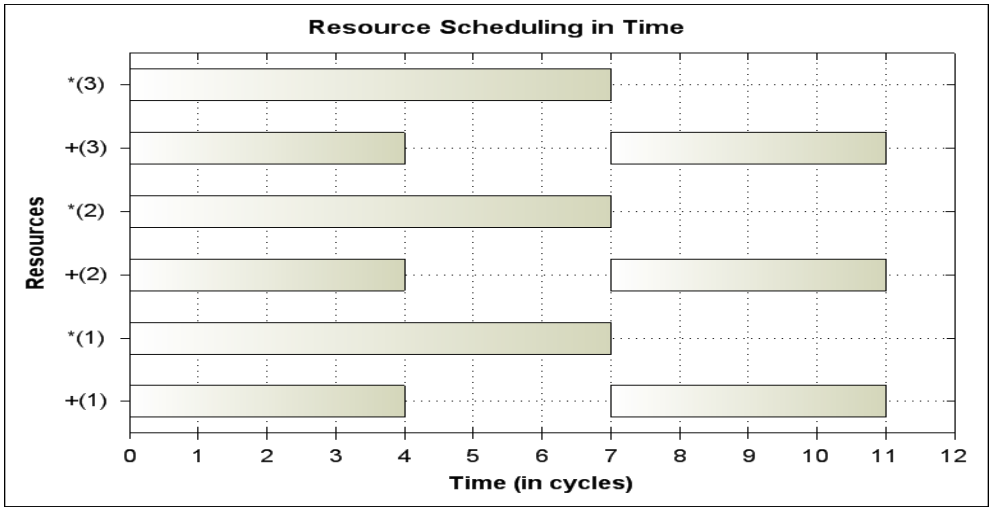


Figure 4.23. Scheduling with three "+" and three "*" in Example 3

5. RH(+) IDE: TRAFFIC CONTROLLER

In this chapter, a complete embedded application will be presented for a traffic controller. The block diagram of the whole design is shown in Figure 5.1.

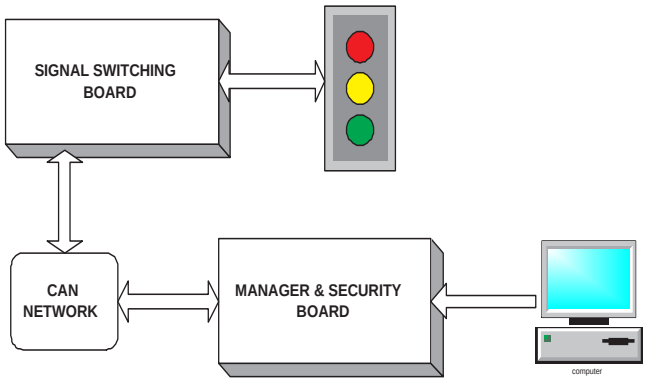


Figure 5.1. Application block diagram

There are two boards: Manager and Security and Signal Switching. Former decides new signal for traffic and send this signal to the latter. The latter switches current signal to new signal. Then, it reads current signal and send it back to the former board. Travelling data is signal.

In the Traffic Conroller example, Manager and Security Board will be considered. This board's possible hardware collection is shown in Figure 5.2. UART, I2C, CAN, eeprom components have functions to provide abstraction. Also, there are two processors: Manager and Security.

5.1. Component Abstraction

The abstraction is provided via the interface functions. The behaviour of these functions is not necessary to know. For that, we have only protoype of these functions. For the abstraction, the data about the operands of the function is sufficient. From RHPlus design, we can call them with their names and appropriate function operands. Also, for the analysis, we need to know the execution delays of them and resources they use.

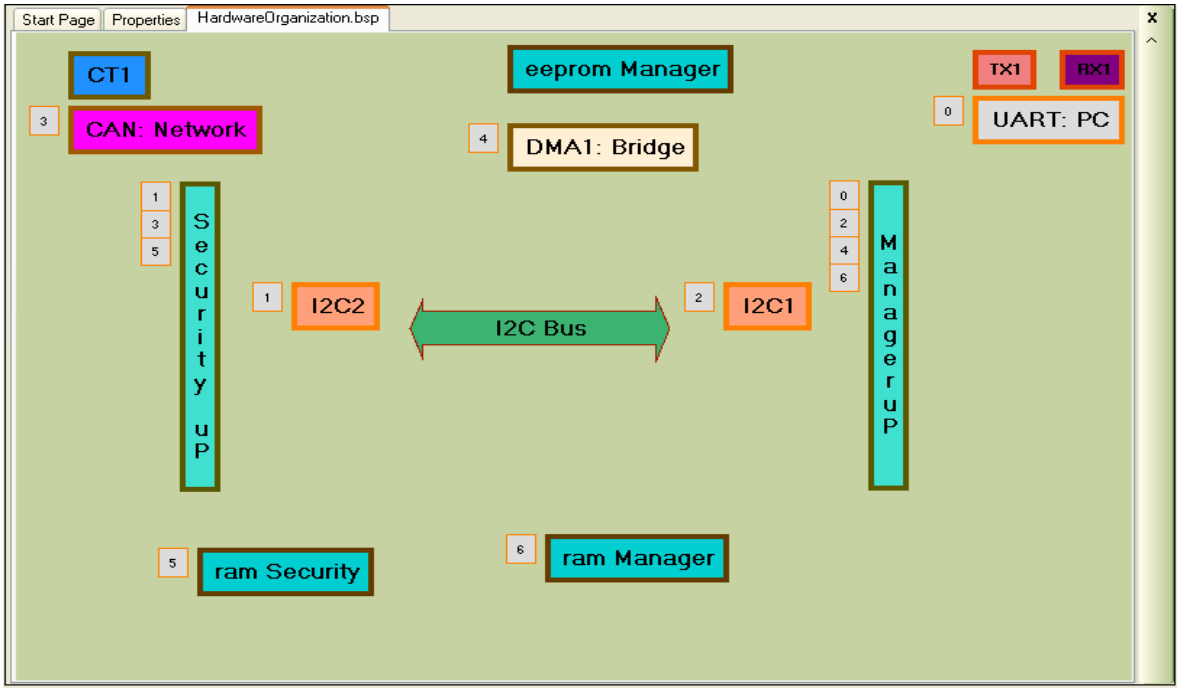


Figure 5.2. Hardware organization

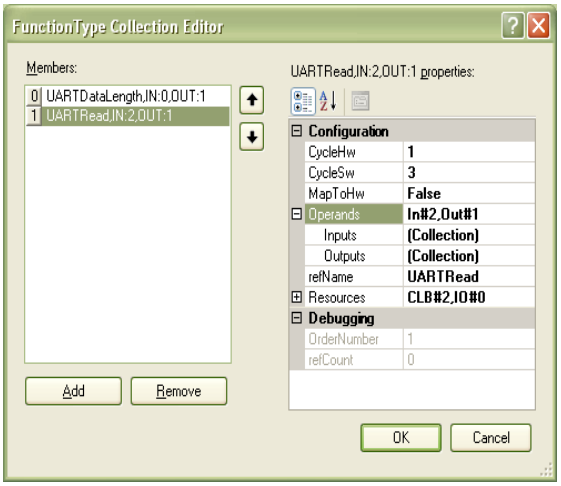
Components' abstraction functions are listed below. Location of abstraction functions for UART, I2C1, and EEPROM is "ram Manager". Location of abstraction functions for I2C2, and CAN is "ram Security". CycleHw is used for them.

- UART: In Figure 5.3, (a) shows functions for abstraction of UART. UARTDataLength has one output. Output is the length of data received from UART interface. Output type is General. CycleHw is 1. UARTRead is used to read UART received data into a buffer. First input is buffer. Its type is Array. It is modified in function. Second input is the length of data that will be read. Its type is General. CycleHw is 8.
- EEPROM: In Figure 5.3, (b) shows functions for abstraction of EEPROM. For eepromWrite, first parameter is the buffer that will be written, Array. Second is buffer length, General. Third is type of data that will be written, General. CycleHw is 16. For eepromRead, first input is the buffer that data is read into, Array. It will be modified in function. Second is buffer length, General. Third is type of data that will be read, General. CycleHw is 12. EepromRead2 has one input. It is type of data that will be read, General. It has one output. It is data

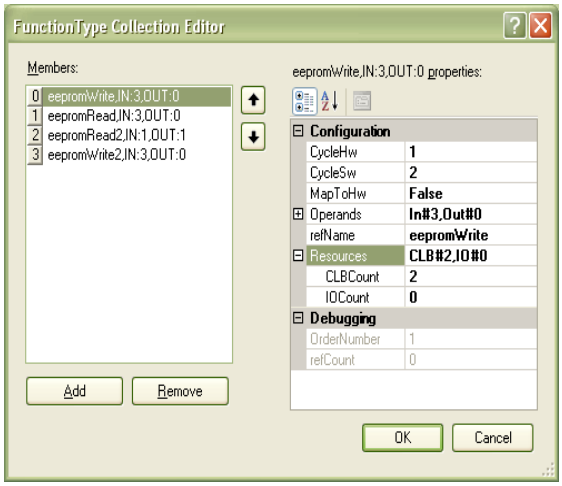
read, General. CycleHw is 6. For eepromWrite2, first parameter is the data that will be written, General. Second is data length, General. This length is always 1. Third is type of data that will be written, General. CycleHw is 8.

- I2C1: In Figure 5.3, (c) shows abstraction functions. For I2CWrite, first input is the data that will be written, General. Second is buffer length, General. This length is always 1. Third is type of data that will be written, General. CycleHw is 6. For I2CRead, first input is the buffer that data is read into, General. It will be modified in function. Second is buffer length, General. This length is always 1. It has one output. It is type of data read, General. CycleHw is 3.
- I2C2: In Figure 5.3, (d) shows functions for abstraction of I2C2. I2CDataLengthS returns the length of data received, General. CycleHw is 1. For I2CReadS, first input is the buffer that data is read into, General. It will be modified in function. Second is buffer length, General. This length is always 1. It has one output. It is type of data read, General. CycleHw is 3. For I2CWriteS, first input is the buffer that will be written, General. Second is buffer length, General. This length is always 1. Third is type of data that will be written, General. CycleHw is 6.
- CAN: In Figure 5.3, (e) shows functions for abstraction of CAN. CANDataLength returns the length of data received from CAN bus, General. CycleHw is 1. For CANRead, first input is the buffer that data is read into, General. It will be modified in function. Second is buffer length, General. This length is always 1. CycleHw is 2. For CANWrite, first input is the buffer that will be written, General. Second is buffer length, General. This length is always 1. Third is type of data that will be written, General. CycleHw is 1.

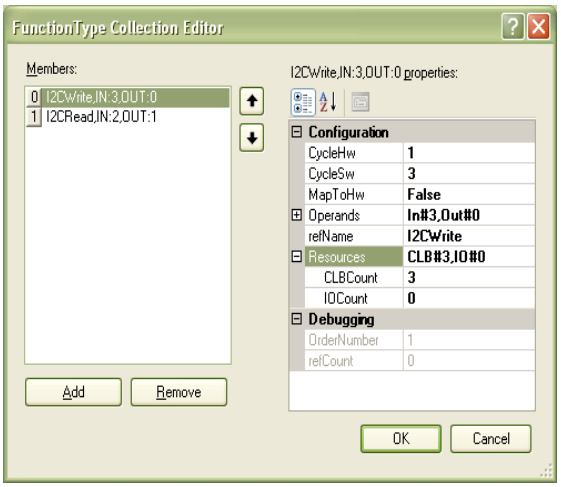
In the interface functions, there are the ones specific to Manager and Security processors. It can be seen from the hardware abstraction file that some hardware block units are assigned to the only one of the processors. These are the results of the hardware design. For example I2C1 and I2C2 are the same type of hardware block buy the former is used by the Manager processor and the second is used by the Security processor. The interface functions for them are different. Thus we can have different type of functionality for them. The names of these functions are also different because the name must be unique for the all entries.



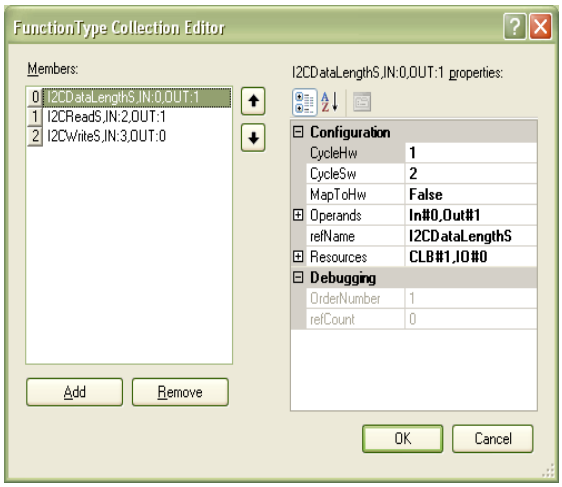
(a) UART



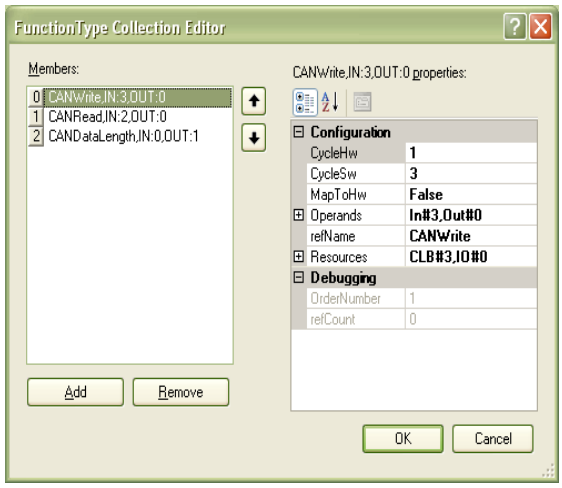
(b) EEPROM



(c) I2C1



(d) I2C2



(e) CAN

Figure 5.3. Abstraction for hardware organization

5.2. Operators

In Figure 5.4, operators that will be used in this example are shown. Functionalities of operators are not known. They will be meaningful at lower levels. In figure, there is "secondDelay" operator. Its parameter is delay value in terms of seconds. Its cycle value is unique so all delay values will have same cycle in Resource Scheduling graph. This cycle value can be adjusted to its real value at lower levels. Other operators are ">", "++", "==", "[]", "FOREVER". "FOREVER" has no operand so its name is not shown in figure. Cycle value of all operators is 1 in this example.

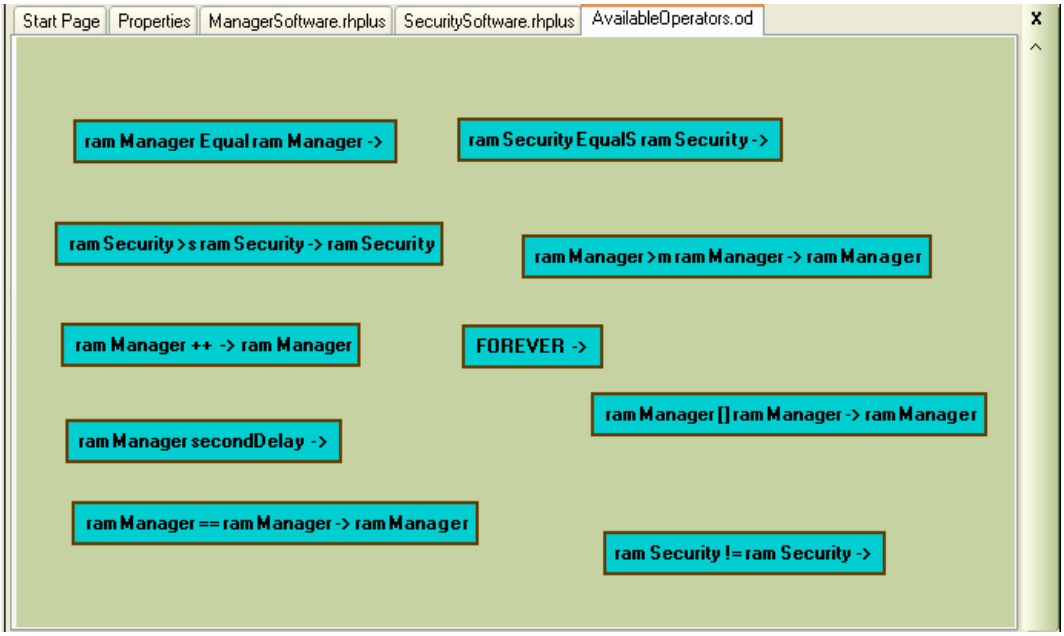


Figure 5.4. Available operators

We want to note that in our OD design, we have the operators used by both Manager and Security processors. We have a global scope. In this example, we do not see the advantage of this global scope. However, it may be possible that the hardware block units used by Security processor can also be attached to the Manager processor. In this case, we prevent the double work-effort by using global scope. In other case, HLS designer must have two OD files and add the same operators to both of these files. After creating the OD file, we can access to all operators from RHPlus files. For that, we use the names of the operators with compatible operators. The operands of the operators must be compatible with the definition of the operator.

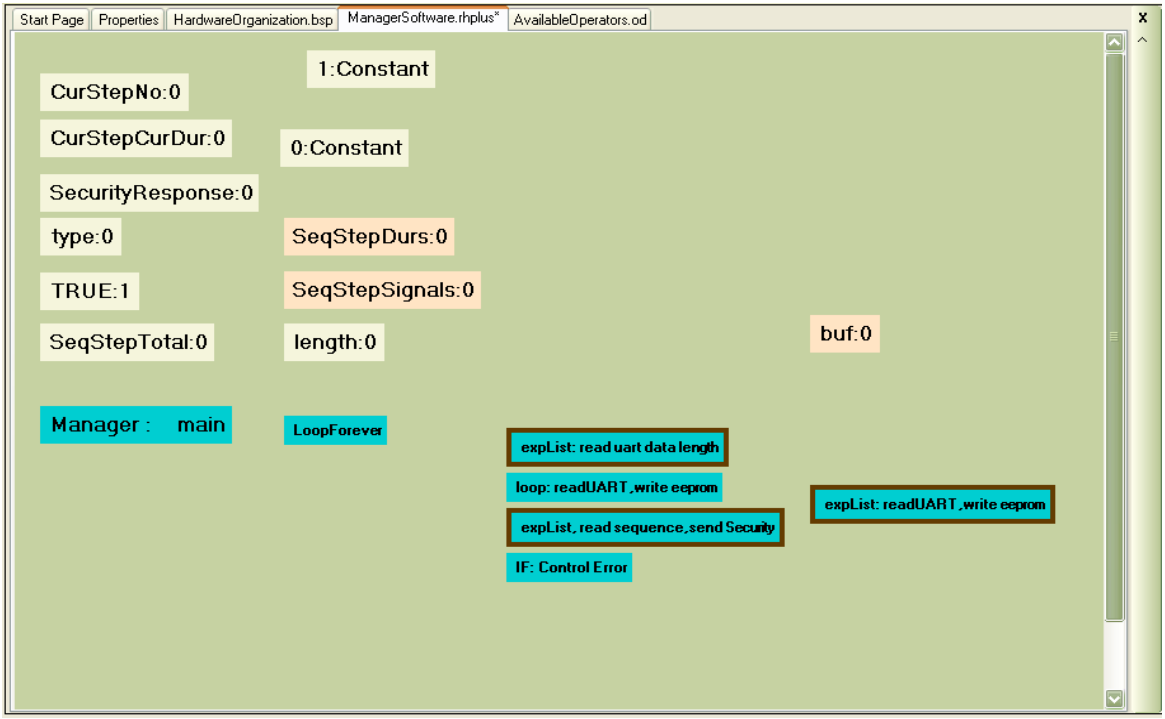


Figure 5.5. Manager program

5.3. Manager Processor

Manager communicates with a PC via UART interface. UART has RX and TX capabilities. Manager use I2C1 to connect I2C Bus. It writes data on bus. This data is read by Security. Manager accesses EEPROM vi DMA1 bridge. Manager uses ram Manager to store its local data. Manager program flow is described as below:

- If demand exists, Manager can get signaling sequence from PC application. Each program data is written to EEPROM for further use of them.
- Manager reads current signal sequence and durations of each step from eeprom.
- Manager sends current signal to the Security.
- Manager delays itself for step duration of current signal.
- Then Manager waits for the Security response. There may an error or no error.
- If there is an error content in response of Security, this is logged to EEPROM.

In Figure 5.5, (a) and (b) shows visual design and LRH(+) software for Manager. Program flow described above can be seen in this code. This code is main function of Manager. It has a forever loop. In traffic signalization, a signal sequence is a signal program that will be run periodically.

When CurStepNo is the last step, it will return to the first step. Then, signals for first step will be shown. These signals are stored in SeqStepSignals array. Step durations are stored in *SeqStepDurs* array. Current step no is CurStepNo. type is type of data processed during program execution. In this example, SeqStepTotal is 16, and CurStepNo is a 4-bit General data.

Possible paths are shown in Figure 5.6. We see five possible paths. THe worst case path is the critical path for the Manager processor software. It can be see that when all conditional blocks are visited, worst case occurs. The execution delay of this path is 70 cycles. Resource Scheduling Graph for longest basic block is shown in Figure 5.7. Also, we append the BB code to the graph. The execution delay for the BB is 38 cycles.

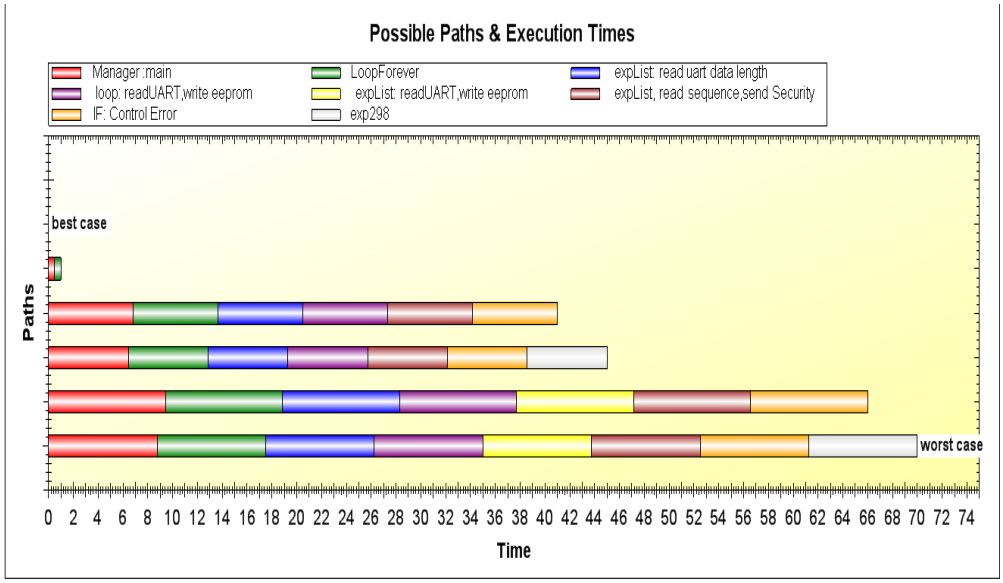


Figure 5.6. Manager possible paths

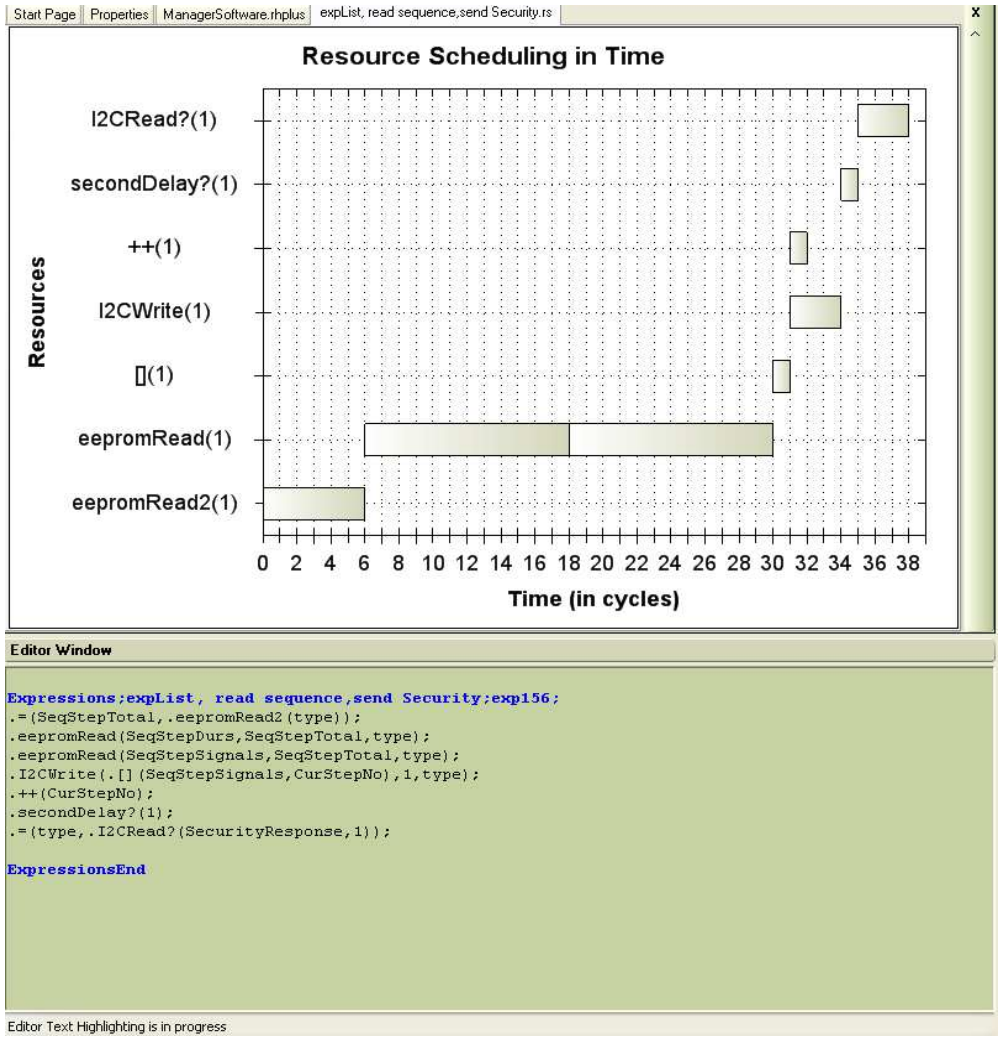


Figure 5.7. Manager resource scheduling

Resource References Graph for the longest basic block is shown in Figure 5.8. The resources are the operators, the functions, and the variables. The most frequently used variable is type, five times. We refer eepromRead for two times having no data dependent to each other. Therefore, it can be selected that there are two instances for this function. In this example, we have only one instance.

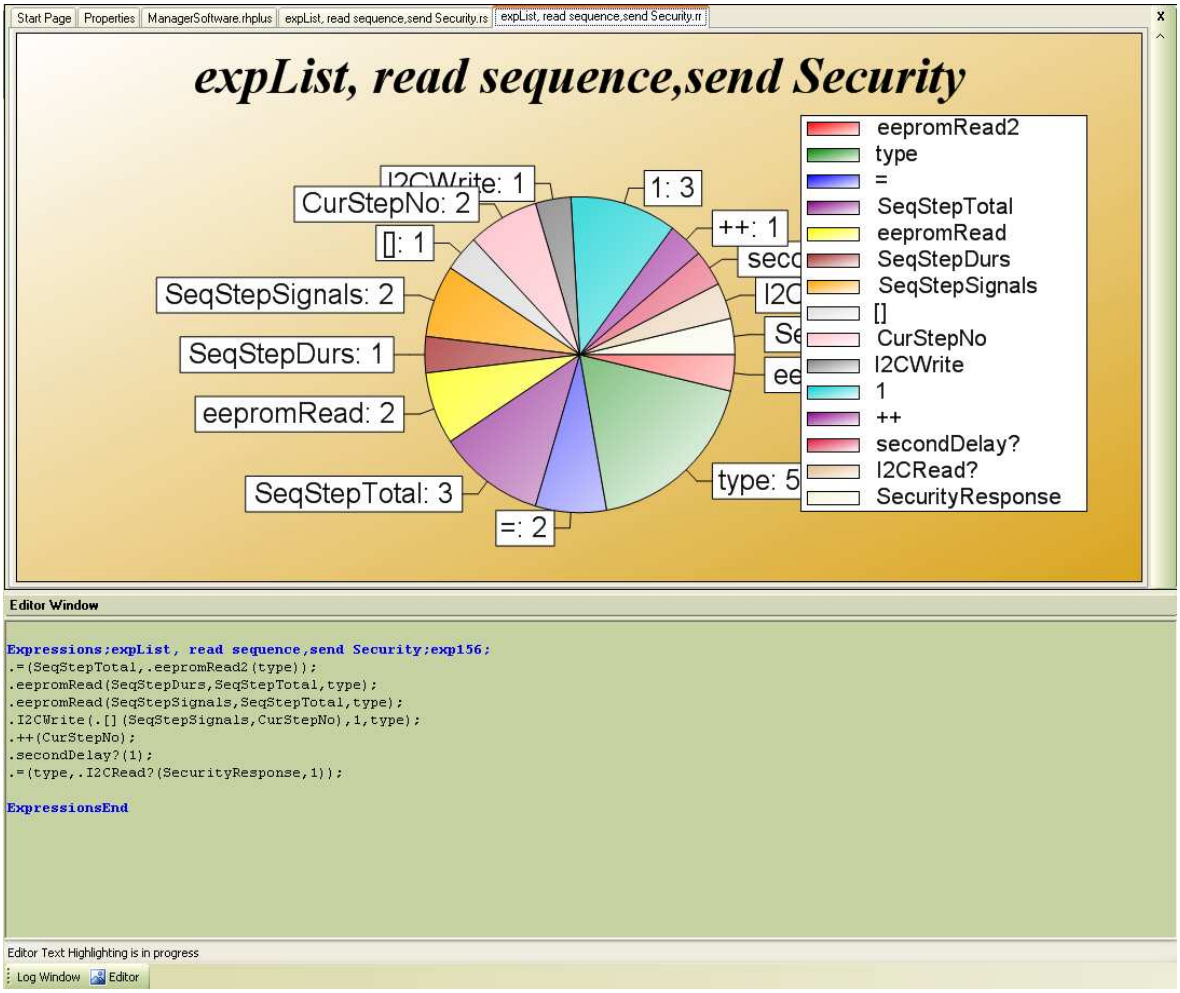
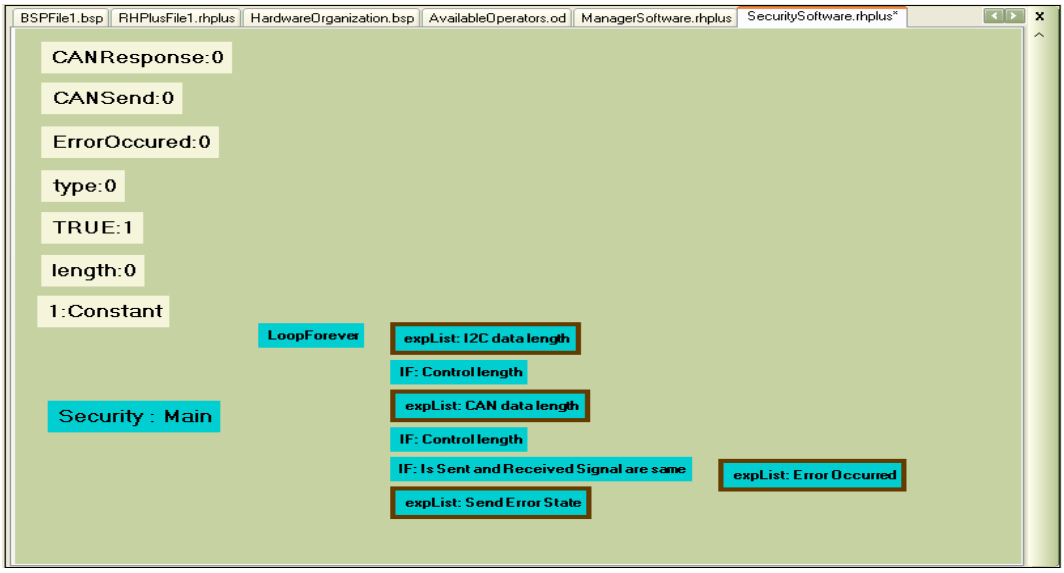


Figure 5.8. Manager resource references for the longest BB

5.4. Security Processor

Security processor communicates with Manager processor by using I2C2. Security processor is connected to CAN network. It broadcasts data, which is read by Signal Switching Board, to CAN network. CAN hardware block is connected to CAN network by using CAN Transmitter. Security uses ram Security to store its local data. This location is used while defining variables, and operands for operators and functions.



Editor Window

```
Function;Security :          Main;F1942;
No input
No output

 Loop;LoopForever;L703;
 .FOREVER();

  Expressions;expList: I2C data length;exp733;
  .=(length,.I2CDataLengthS());
  ExpressionsEnd
  If;IF: Control length;IF129;
  .>s(length,0);

   Expressions;expList: Read;exp802;
   .=(type,.I2CReadS(CANSend,1));
   .CANWrite(CANSend,1,type);
   ExpressionsEnd

  IfEnd
  Expressions;expList: CAN data length;exp741;
  .=(length,.CANDataLength());
  ExpressionsEnd
  If;IF: Control length;IF137;
  .>s(length,0);

   Expressions;expList: Read;exp820;
   .CANRead(CANResponse,1);

   ExpressionsEnd

  IfEnd
  If;IF: Is Sent and Received Signal are same;IF145;
  .!=(CANResponse, CANSend);

   Expressions;expList: Error Occurred;exp838;
   .=(ErrorOccured, TRUE);
   ExpressionsEnd

  IfEnd
  Expressions;expList: Send Error State;exp754;
  .I2CWriteS(ErrorOccured,1,type);

  ExpressionsEnd

 LoopEnd

FunctionEnd
```

Editor Text Highlighting is in progress

Figure 5.9. Security program

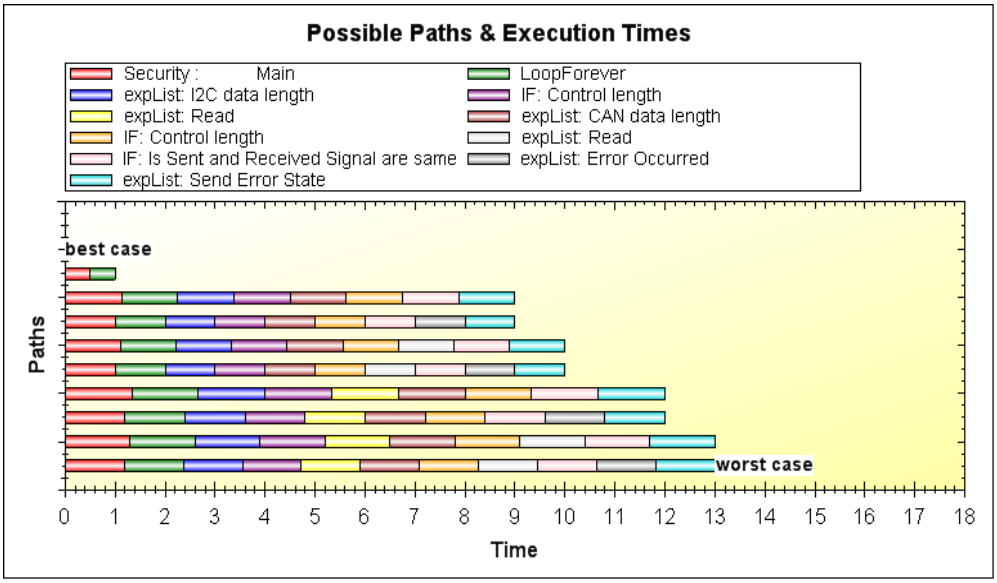


Figure 5.10. Security possible paths

Security program flow is described as below:

- Security controls if new signal data is received from Manager.
- If so, read new signal. Then, inform Signal Switching board of this new signal.
- Control if Signal Switching Board has sent response. Here, response is the measured signal value at signal outputs.
- Compare Signal Switching Board response with signal sent from Security.
- If sent and received signals are not equal, there is an error. Update error information.
- Send error information to Manager.

In Figure 5.9, (a) and (b) shows visual design and LRH(+) software for Security, respectively. Program flow described above can be seen in this code. This code is main function of Security. It has a forever loop. When new signals are received from Manager, it is sent to Signal Switching Module via CAN bus.

Possible path is shown in Figure 5.10. There are nine paths and the worst case is the critical path. Its execution delay is 13 cycles. When all conditional blocks are visited, worst case occurs. Resource References Graph for the longest basic block is shown in Figure 5.11. The variable, CANSend, is referred for two times. Resource

Scheduling Graph for longest basic block is shown in Figure 5.12. The execution delay of the BB is 5 cycles.

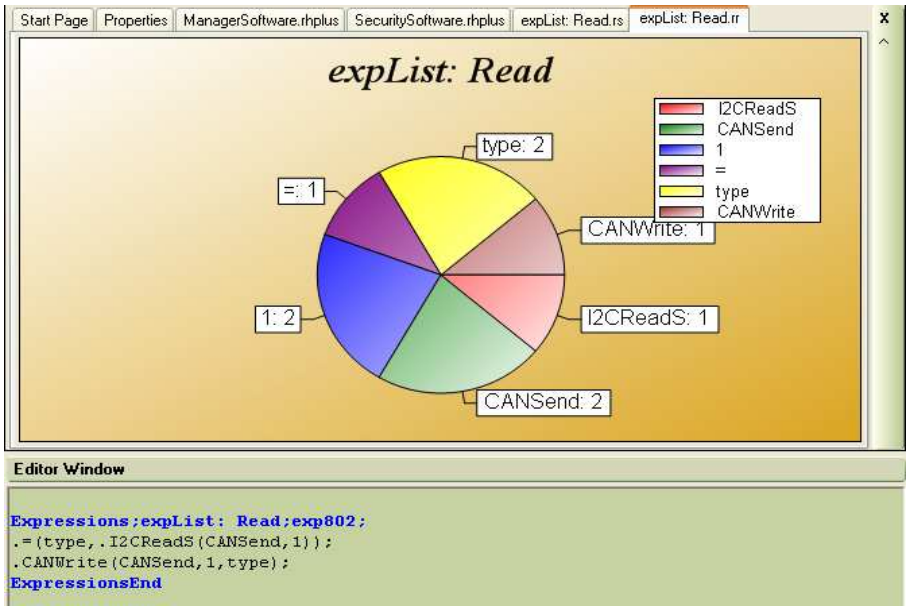


Figure 5.11. Security resource references for the longest BB

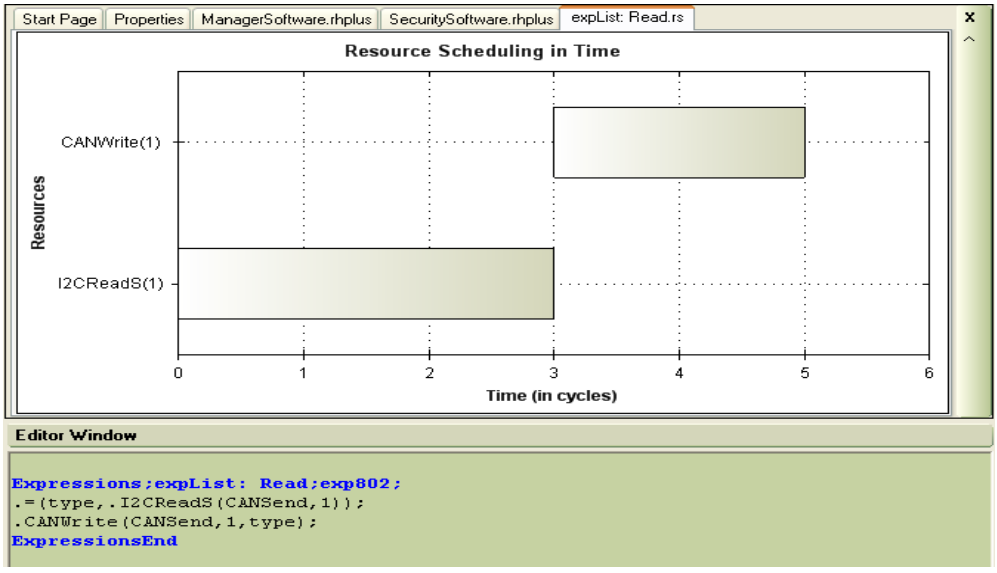


Figure 5.12. Security resource scheduling

6. RH(+) IDE: CLASS DIAGRAMS

In this chapter, we present class organization in RH(+) IDE design. We present 10 diagrams. First of all, we consider our Base Module, which is used to derive other components. FunctionType is the second diagram we explain. MultipleExpressions class is designed to store LRH(+) code. The base of MultipleExpressions class is SubExpression class. They are also described. We detail our analysis tool classes: Resource References, Resource Scheduling, and Possible Paths and Execution Times. To understand how output file is generated, we explain OutputFile class. Lastly, hardware library class is presented.

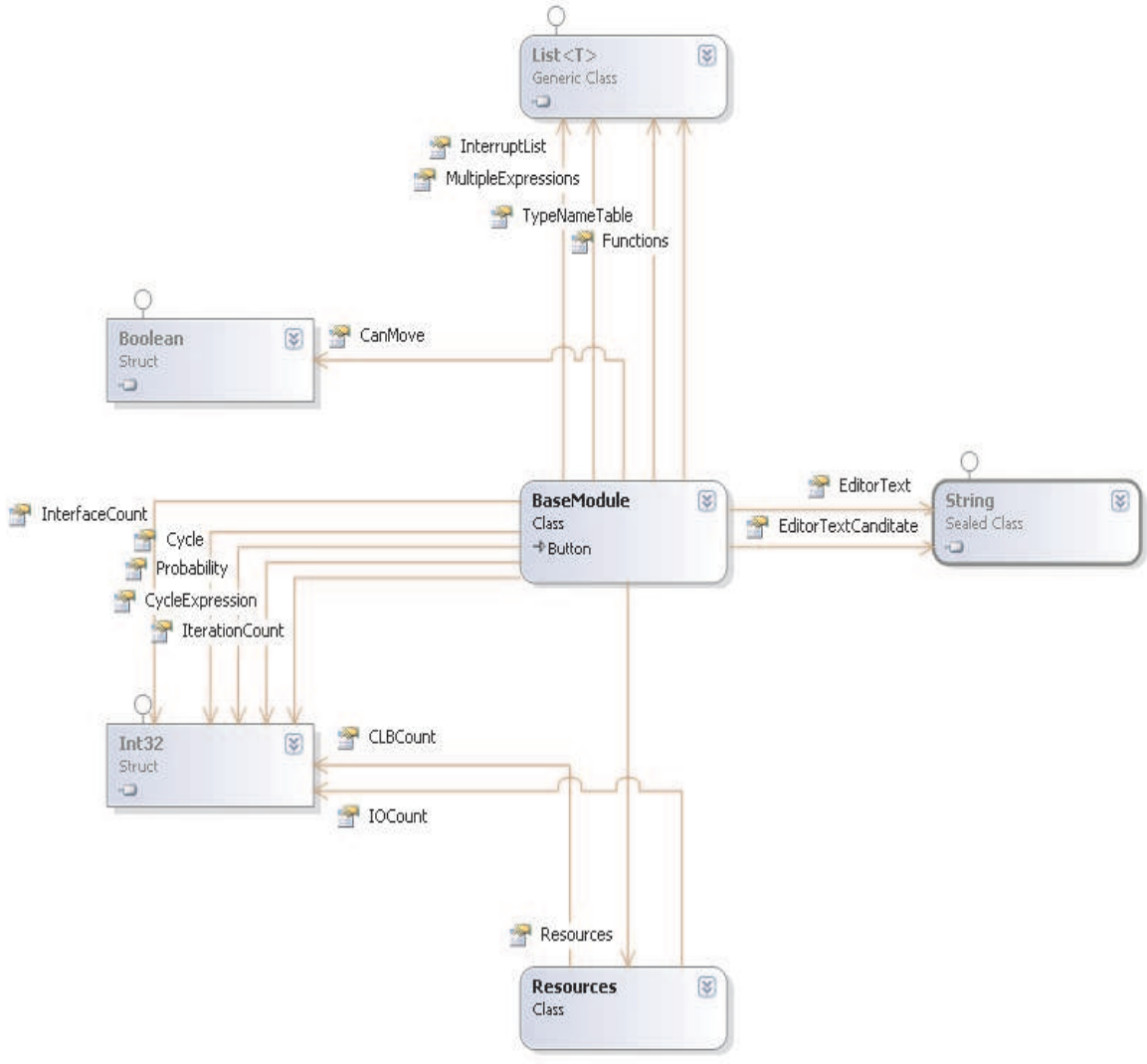


Figure 6.1. Base module

In Figure 6.1, we see properties of Base Module. We provide visual design support. CanMove property shows if a component can move. For example, when RX component is integrated with UART component it cannot move anymore. In other words, RX can only move with UART after they are connected. Also, an elseIf component can be connected to an if component. In this case, elseIf is stepper for if component. After they are integrated, elseIf cannot move. If a component cannot move, CanMove property is false.

Functions are used by BSP components. These are interface functions to provide hardware abstraction. TypeNameTable is implemented to store component relations. For example, while developing **.rhplus* file, a component may follow another component or a component can be assigned as the first component for its scope. These assignments are an entry for TypeNameTable. MultipleExpressions class is used to store LRH(+) code. It can have many single expressions. InterruptList class is used by BSP components. Block hardware units can have interrupt sources. These are identified via class.

EditorText is LRH(+) code for the module derived from Base Module. This code can be a condition if module is a conditional one like loop. EditorText can also be a list of statements if module is an expression list block. EditorText is the last validated LRH(+) code. This means that there is no lexical or syntax error in this code part. However, there is also EditorTextCanditate, which is the last edited text for the module expression. EditorTextCanditate may not have a correct format. HLS designer can travel between code parts. When he revisits a module expression, EditorTextCanditate is presented to him. If EditorTextCanditate has a correct format, it is copied to EditorText for the following analyses.

IterationCount is used by loop component. It is the possible iteration count for the loop. This is used while calculating worst case path. Probability is also used by conditional components: loop, if, and elseIf. It is the probability of the condition is true. Cycle is used by many components. It is the execution times for interface functions BSP components. Also, it is execution delay for the operators. These values

are decided by the user. Besides, Cycle is by RHplus components. However, there is no user-entry in this case. Cycle for RHplus elements is updated by RH(+) IDE. HLS designer can watch these values via Properties Window. CycleExpression is used by components having expressions. There are two types of expressions: condition and a list of statements. These are BB for RH(+) IDE. The execution delays of these BB are assigned to CycleExpression. InterfaceCount is used by some BSP components. For example, memory locations are added to DMA as interfaces. The total number of the interfaces for a components is assigned to InterfaceCount. All modules have Resources property. These are the number of CLB (CLBCount) and IO pins (IOCount). These are LLS details.

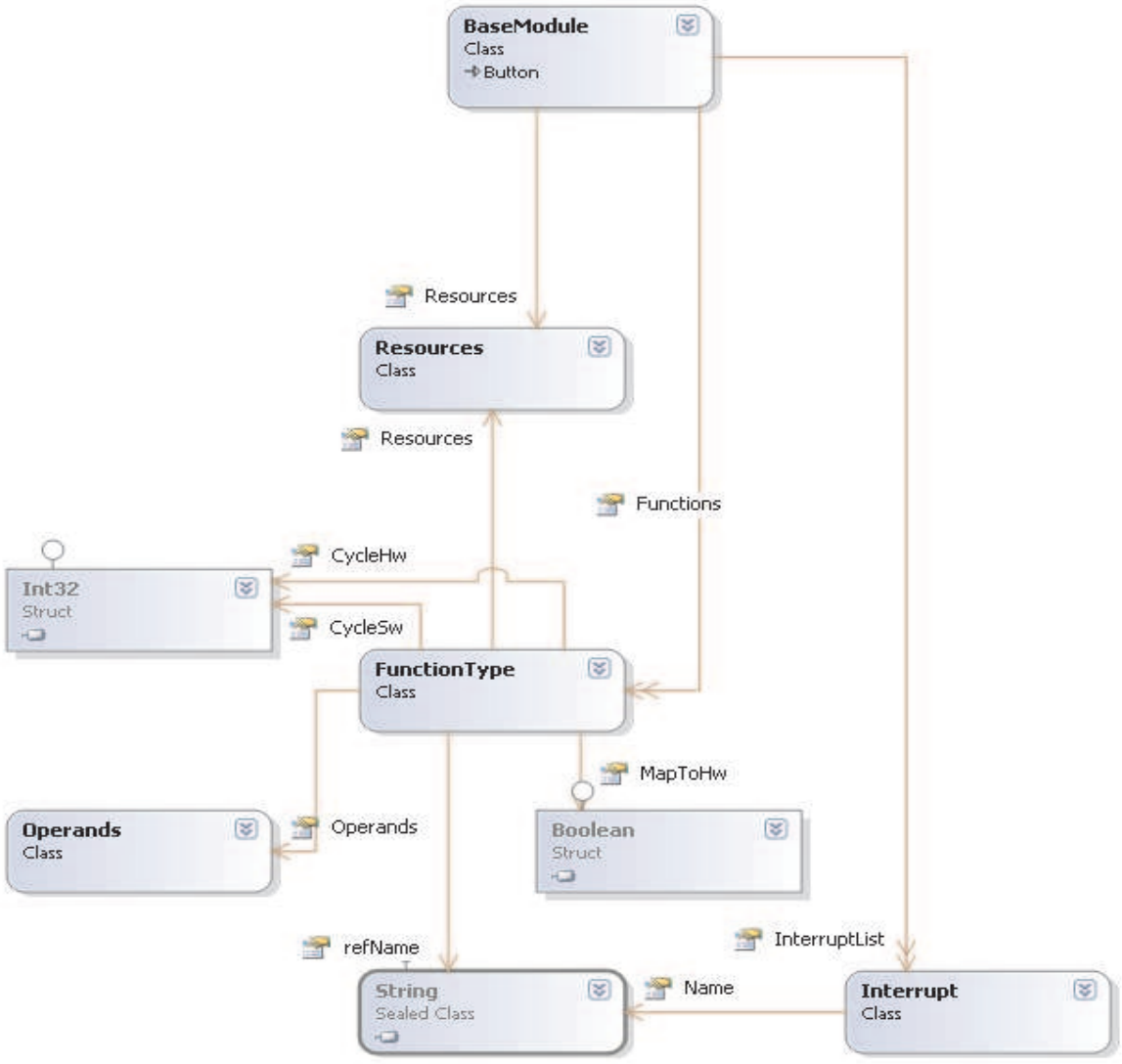


Figure 6.2. Function type

In Figure 6.2, BSP component may have interface functions. Also, there are functions in **.rhplus* files. refNames is the name of the function. Operands are inputs and outputs of the function. A function can be mapped to hardware at run-time if MapToHw is true. When it is mapped to hardware, it is execution time is CycleHw. If it is executed in software, CycleSw is the execution delay of it. Furthermore, we see that BSP components have a list of Interrupt that is InterruptList. These are the possible interrupts BSP element can cause.

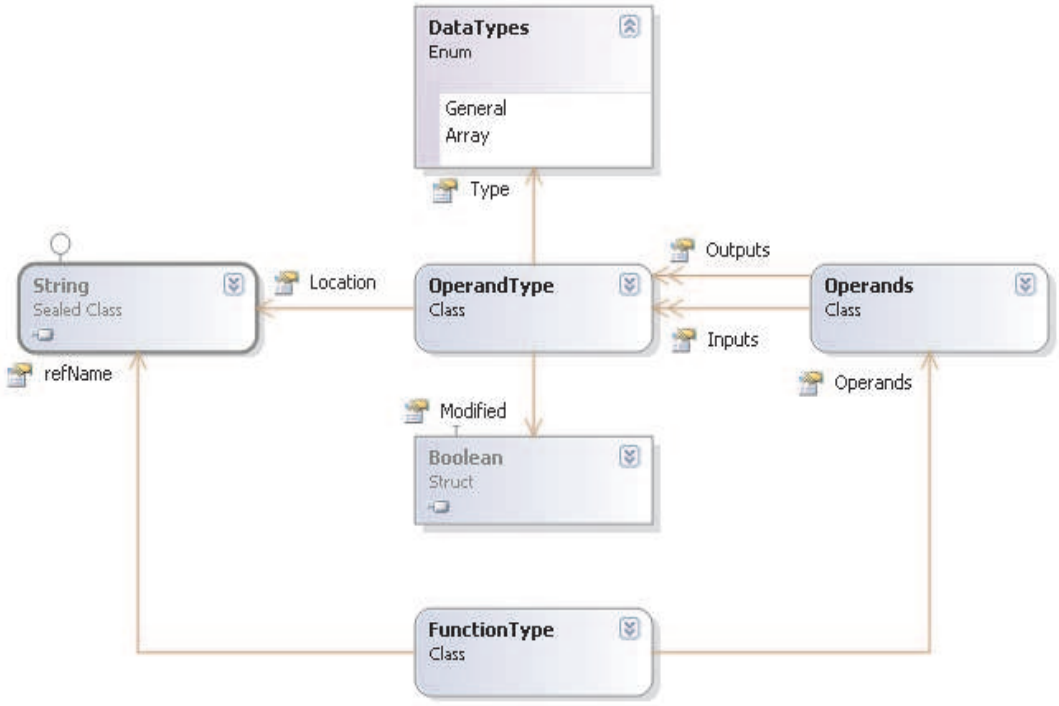


Figure 6.3. Function operands

In Figure 6.3, we see how operands are represented in software. Functions can have operands. Operands may be inputs or outputs. They are type of OperandType. An operand location is given with Location. Operand type can be General or Array. It is decided by Type property. Sometimes, we want to know if an operand is modified in a function. This is a requirement for interface functions because we have no data about behaviour of them. Therefore, to produce correct analysis results, we need this Modified property. If Modified property is true, at the time of applying the related interface function, we understand that we must wait at the end of interface function operation to use this operand in another operation.

In Figure 6.4, we present MultipleExpressions which is a list of SingleExpression. SingleExpression may a condition or a line of statement ending with ';' character. SingleExpression is a list of SubExpression. SingleExpression has a start time to start its execution. The time at which its execution ends are also required. These are stored in TimeStart and TimeEnd, respectively.

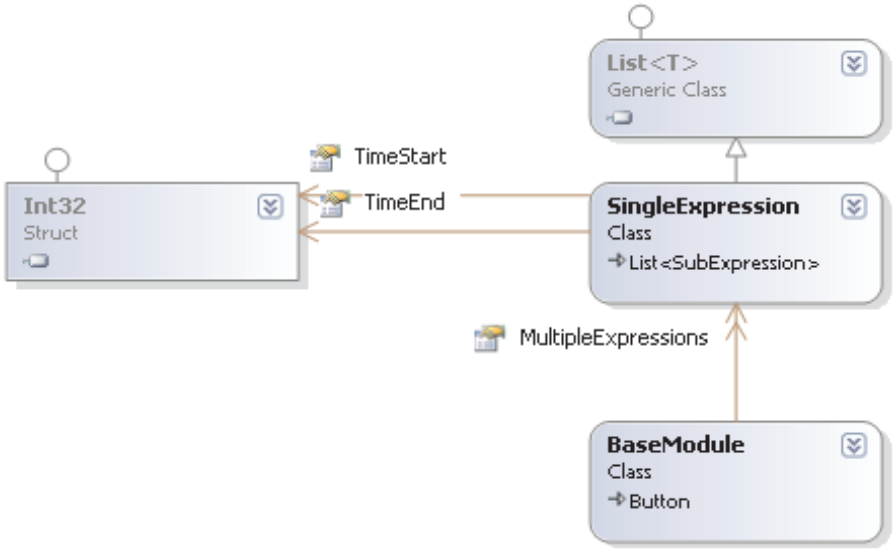


Figure 6.4. Multiple expressions

In Figure 6.5, SubExpression class is presented. SubExpression is an operation operated on an operator or function and their operands. A list of SubExpression constructs SingleExpression. SubExpression has an operator name. This can be name of an operator defined in **.od* file or name of **.rhplus* functions or BSP interface functions. OpName property represents this name. OperandNames property has input and output names. These are the variables HLS designer defines in **.rhplus* file. Types of these operands are stored in OperandTypes property.

For the analysis of the design, we need resources used by these sub-expressions. Therefore, we store it in Resources property. The operator used in SubExpression may have a number of instances. The upper bound for this number is given by HLS designer by using Properties Window. The first free instance is used by the current sub-expression. The index for this instance is stored in InstanceIndex. Cycle is the execution delay sub-expression takes. TimeStart shows the time at which sub-expression

starts to execute. The end time of execution is shown by TimeEnd. These values are used while scheduling the resources to the sub-expressions.

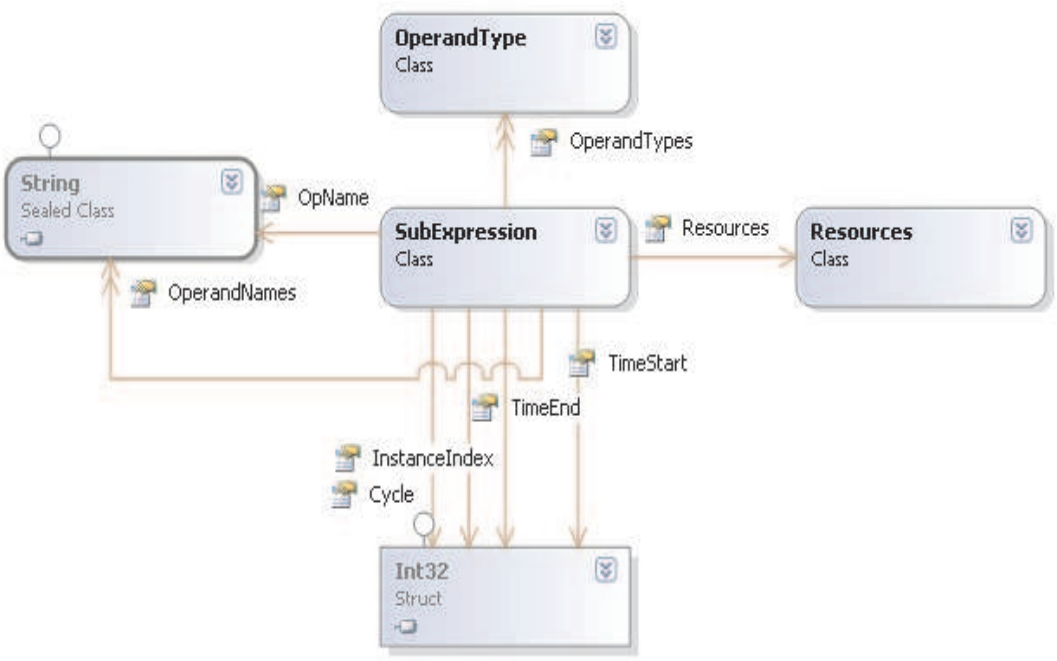


Figure 6.5. Sub-expressions

In Figure 6.6, we see class organization required by Resource References analysis tool. At the top of the classes, we use RefGroup class which is a list of RefList class, which is also a list of Ref class. Operator, functions, and resources can be referenced. Name property has the name of the reference. Sometimes, we want to implement the behaviour of an operator. In this case, we call these operators as complex operators. The behaviour is given by using existing basic operators which are in **.od* file. For analysis tools, we need the number of resources used by this reference. It is stored in Resources property.

HLS designer can change the number of the instances for the current reference. This number is stored in InstanceCount. While scheduling instances to sub-expressions, there may be instances in use. The time when an instance will be free is stored in InstanceFreeTimes integer list. The number of the references of a variable is updated by RH(+) IDE. RefCount property has this value.

While scheduling resources, we have two states for the references: modified and input state. In a sub-expression, a reference can be modified. If this is the case, it cannot be input for another operation until the end of the operation in which it is used. Also, a reference can only be an input in a sub-expression. In this case, it may be input for other operations at the same time. However, it cannot be an operand for another operation which modifes it. We use ModifiedList and InputList properties to store the time intervals for this reference usage. These are list of Interval class. An interval is defined with a start time and end time, which are TimeStart and TimeEnd respectively. For ModifiedList property, entries in this list shows in which time intervals, reference is modified. InputList property has interval entries showing when reference is used as input.

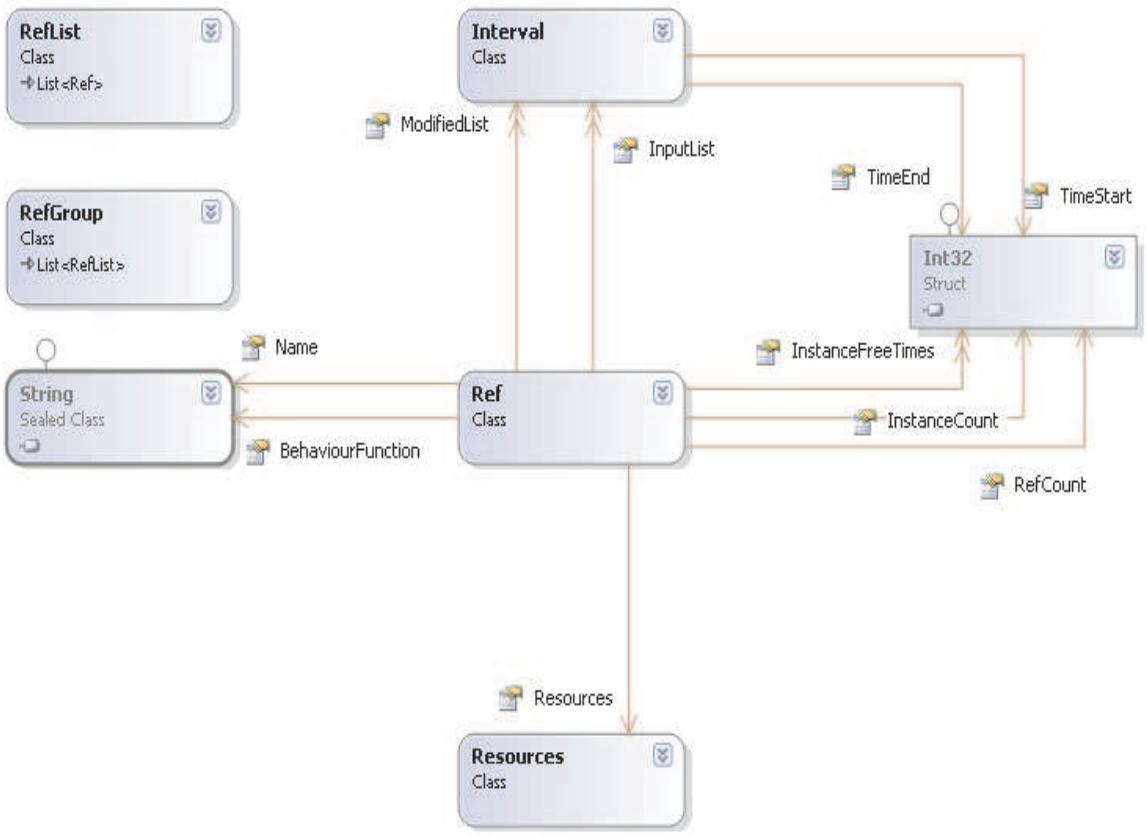


Figure 6.6. Resource references analysis

In Figure 6.7, the class organization used by Resource Scheduler analysis tool is shown. ResourceScheduleResults is a list of ResourceScheduleResult. MaximumEnd property shows the execution time of the worst case path which is also critical path

for the owner block of the analysis. ResourceScheduleResult has a property named ResourceName. This shows which resource is being scheduled. This can be the name of an operator or a function. TimeStart is start time and TimeEnd is end time, for the resources in scheduling graph.

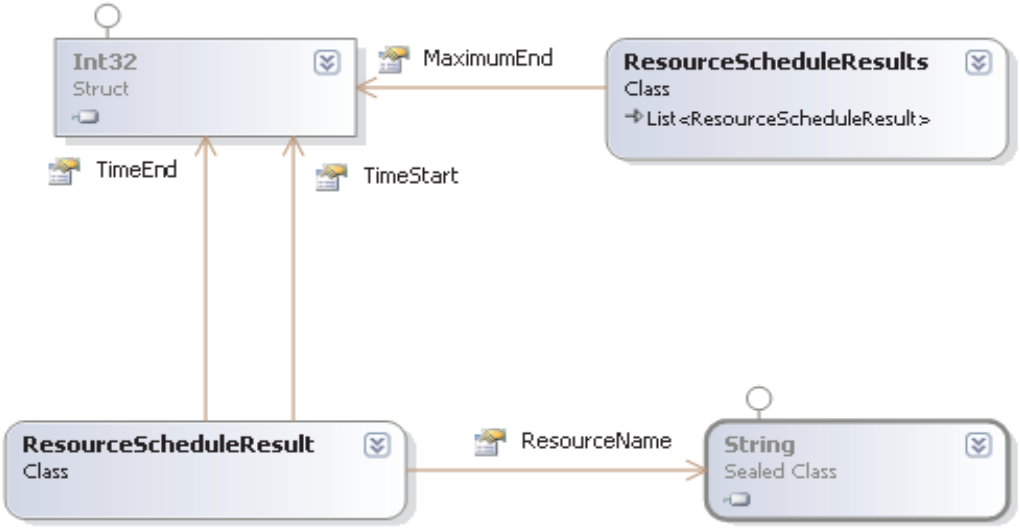


Figure 6.7. Resource scheduling analysis

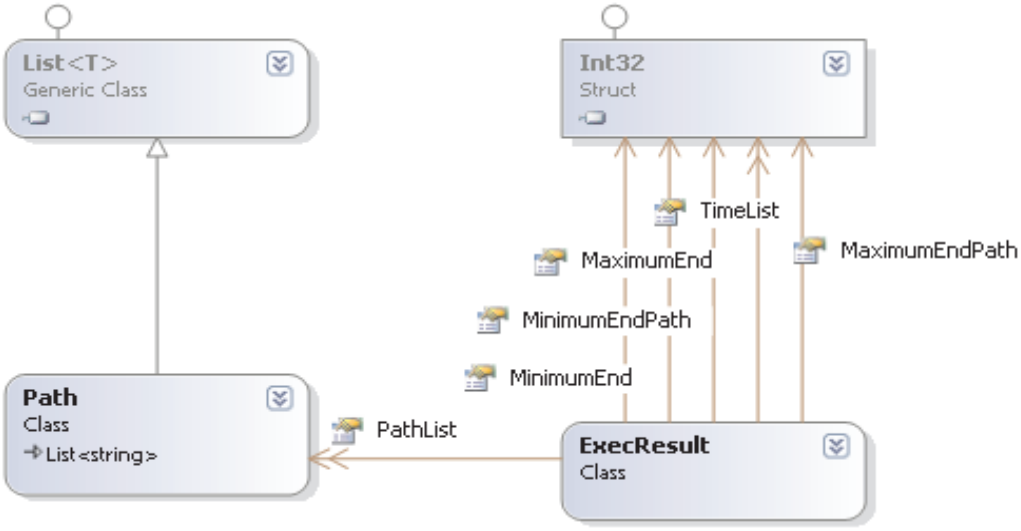


Figure 6.8. Possible execution paths and times analysis

In Figure 6.8, the organization for the class used by Possible Execution Path Finder analysis tool is presented. Possible paths are stored in PathList which is a list of Path, which is a list of string. Path entries are block names which are path nodes.

TimeList has the execution delays for each path node. It is a list of integer. These time values are accumulated to find the execution delay of the path. There may be a number of paths. MinimumEndPath is the path index having the minimum execution delay. This delay is stored in MinimumEnd. MaximumEnd is the execution delay of the path having index of MaximumEndPath.

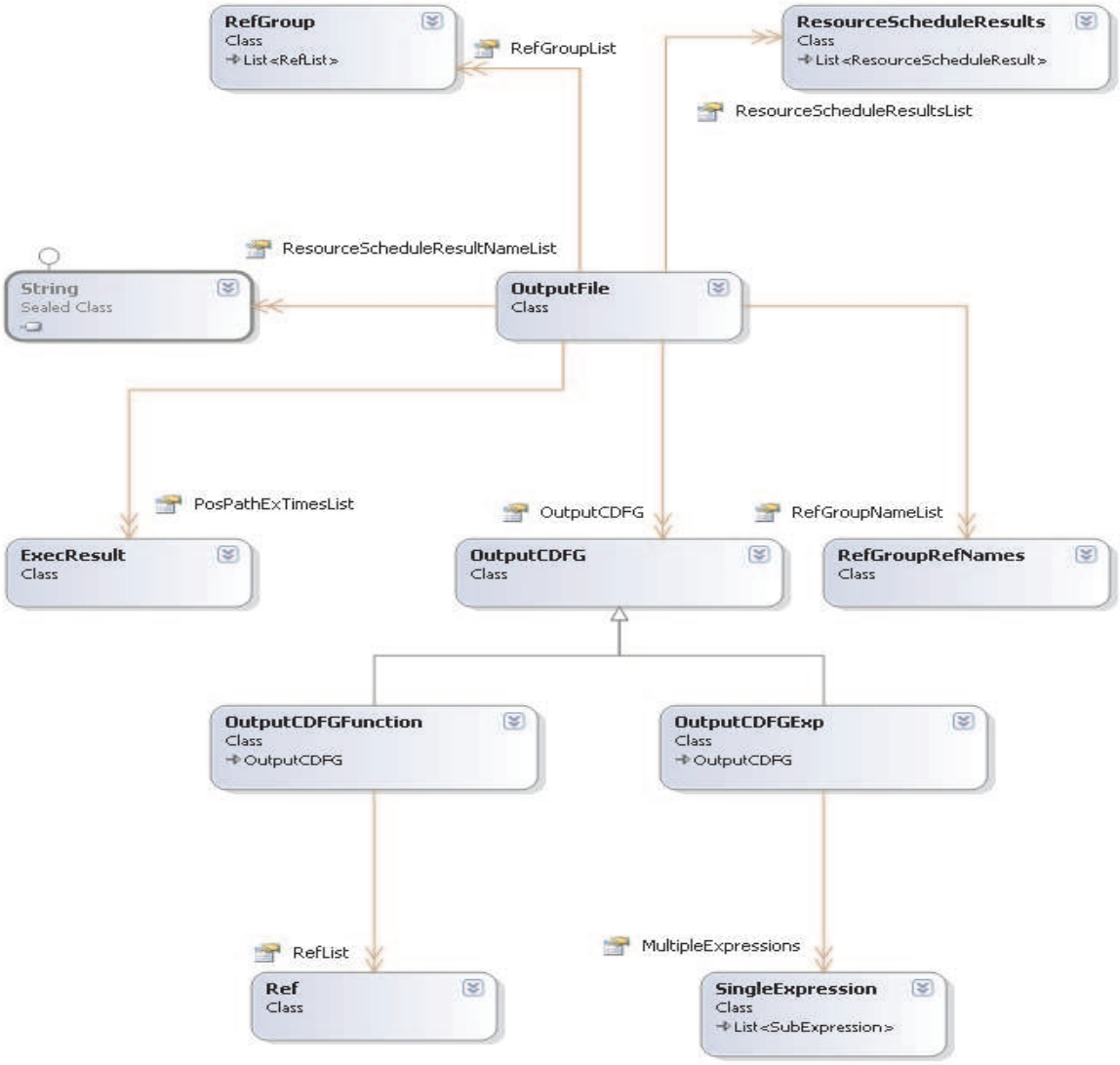


Figure 6.9. Output file

In Figure 6.9, we present the class organization required to produce output file of RH(+) IDE. We keep outputs of the resource reference analysis in a list of RefGroup which is RefGroupList. We apply resource references analysis tool onto all functions. The name of these functions are kept in RefGroupNameList. The outputs

of the resource scheduling analysis are stored in ResourceScheduleResultsList which is a list of ResourceScheduleResults. We apply scheduling analysis onto each BB. The names of the corresponding BB are stored in ResourceScheduleResultNameList. PossiblePathExTimesList is a list of ExecResult. It has outputs of possible path analysis for the functions. Besides, we add CDFG data into output file. This is IR for LRH(+) software. OutputCDFG is the base class for different types of CDFG nodes. OutputCDFGFunction has a different content. RefList for each block under function are listed. OutputCDFGExp is used by the components having expressions. We add MultipleExpressions used by them into output file. Components not using OutputCDFGFunction and OutputCDFGExp use OutputCDFG base class to provide data to output file.

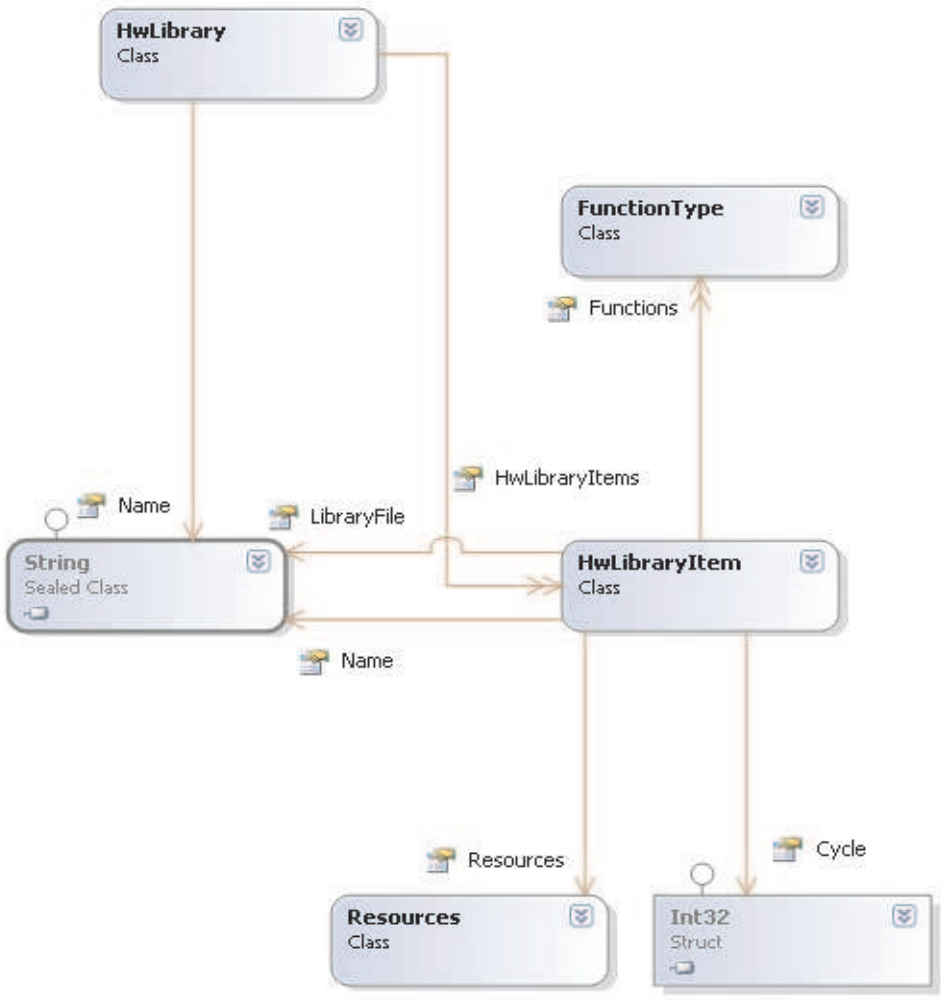


Figure 6.10. Hardware library

In Figure 6.10, the class organization providing hardware library creation ability is presented. A name for the library can be assigned by using Name property. HwLibrary is a list of HwLibraryItem. Items can have LLS details: Cycle and Resources. A name for the item can be determined with Name property. Interface functions can be determined by using Functions property. Instead defining LLS details by using provided properties, designer can give a filepath refering a file which has these LLS details. This path is stored in LibraryFile.

7. CONCLUSIONS AND FUTURE WORK

In this thesis, we have two proposals. The first one is a model, RH(+), for embedded system development on run-time reconfigurable hardware. The second one is a framework, FRH(+), following the requirements of the model. Lastly, we implement RH(+) IDE, which meets a subset requirements of FRH(+) proposes. These are the main concepts FRH(+) has. The most important property is that HLS designer must interact with last intermediate representation of program code. RH(+) IDE provides this property thanks to the subtools. HLS designer can analyze the program with presented subtools, Resource Scheduler, Resorce References, and Possible Paths and Execution Times. After analyzing the program, HLS designer can change his previous settings, which are the resource constraints.

Furthermore, the lower target architecture dependent steps are black box for FRH(+). Bridge between the lower steps and FRH(+) is the Output File of RH(+) IDE. It includes results of analysis subtools. Program flow is given in terms of CDFG in this file. In addition, possible paths and their execution times are included for each function. These execution times have the best case and the worst case. The worst case is the critical path of the function. The number of the references for all variables, operators, and functions are added to Output File. In LLS design, by using this file further optimizations can be applied.

Using the structure FRH(+) proposes, low level abstraction is successfully provided. This abstraction is managed by HLS designer. Abstraction is applied for a virtual board, BSP. This board is defined by HLS designer. By using these capabilities, HLS designer does not need his target hardware to start development of the application. This decreases development life-cycle. Also, it enables preparing prototypes in software. BSP has the possible component list. These components are candidates for hardware mapping. If they are referred from the application they are mapped to the hardware. This strategy increases efficient use of run-time reconfigurable hardware. For example, let us consider an embedded application which has run-time reconfigurable

hardware, and a third party modem. Also, let us assume that application hardware needs a driver hardware to access this modem. Then, we assume, in this application, the modem is used only once in a day. In this use, embedded device sends event logs to a central department computer. After sending these event logs, modem will not be used again until 24 hours pass. In this scenario, modem driver must be configured on reconfigurable hardware only when it is required. In other words, when application code receives the expression like `SendModemData(data, length);`, modem driver is mapped to the hardware. At the end of active scope of the modem use, reconfigurable hardware used by the modem driver can be used by another module. By using FRH(+), the number of the resources the modem driver module requires is known. Thus, the possibility if modem can be mapped to the hardware can be decided when it is referred.

Moreover, reconfigurable hardware is the new owner of the embedded system design applications. Therefore, new approaches arise about it, however, many of them uses the existing traditional approaches as a base. They are insufficient to meet the RH(+) requirements. The origin of the traditional models depends on the static hardware. They do not consider reconfigrable hardware. To adapt them for reconfigurable hardware modifications on them are applied. These modifications take them away from their original aims. The most appropriate example for that is UML. There are modification examples on UML. They modify UML to enable it to use as a programming language. There are attempts to conver UML to byte-codes [32, 33]. Aim of UML, modeling systems, diverges by using it in these solutions.

Instead, in this work, a new approach to provide solution for the application that will be run on run-time reconfigurable hardware has been proposed. Its feautures are described, and the problems why traditional ones do not meet the requirements for it are expressed. In this work, a second solution, FRH(+), is proposed over the first solution, RH(+) Model. Also, RH(+) IDE, which has been implemented, partially meets the FRH(+) requirements.

Run-time reconfiguration is the most important property, which target architecture must have, to be able to apply our work. We select design-time decisions for

run-time reconfiguration. By using design-time decisions instead of run-time decisions, we get an error-free design because we know all possible states of the target architecture at design-time. These states can be assigned by HLS designer as constraints on LRH(+) software.

As a future work, virtual machine having capability running the output of FRH(+) should be implemented. This virtual machine will run on board specified in board support package. Also, debugging ability must be added to the system. In addition real-time operating systems are useful to enable multitasking for an embedded application. Since LRH(+) has many common programming language structures, any operating system can be ported to it. We state that we consider design-time reconfiguration decisions in our work. However, run-time reconfiguration possibility must be presented to HLS designer. He must select the most appropriate decision method for his design. Also, specifying a hybrid solution combining design-time and run-time decisions must be possible to design more complex systems.

APPENDIX A: AVERAGE TAKER EXAMPLE

We present an example calculating average of ten numbers. First of all, follow the following steps to start example. After applying these steps, file organization on "Solution Explorer" must be as in Figure A.1.

- Open our RH(+) IDE.
- Go to "File/New/Project".
- Enter project filename as AverageTaker. Click "Save".
- Look at the "Solution Explorer" window. You must see a node whose text is AverageTaker. On the left of this text, project icon must be seen.
- Right click to the AverageTaker node.
- Go to "Add/New Item".
- Select "BSP" from "RH(+) Files".
- Enter *AverageTaker.bsp* as filename.
- Click "Enter".
- Look at the "Solution Explorer" window. You must see a node whose text is *AverageTaker.bsp* under node whose text is AverageTaker. On the left of this text, **.bsp* file icon must be seen.
- Right click to the AverageTaker node.
- Go to "Add/New Item".
- Select "Operator Definition" from "RH(+) Files".
- Enter *AverageTaker.od* as filename.
- Click "Enter".
- Look at the "Solution Explorer" window. You must see a node whose text is *AverageTaker.od* under node whose text is AverageTaker. On the left of this text, **.od* file icon must be seen.
- Right click to the AverageTaker node.
- Go to "Add/New Item".
- Select "RH(+)" from "RH(+) Files".
- Enter *AverageTaker.rhplus* as filename.

- Click "Enter".
- Look at the "Solution Explorer" window. You must see a node whose text is *AverageTaker.rhplus* under node whose text is AverageTaker. On the left of this text, **.rhplus* file icon must be seen.

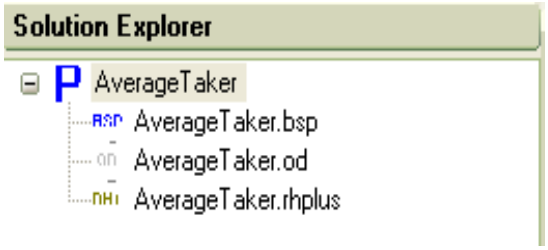


Figure A.1. Solution explorer

Secondly, we must design our board support package. In this file, we define possible components on target architecture. Follow the following steps to create *AverageTaker.bsp* file.

- Double click *AverageTaker.bsp*. You see a tabpage for this file on the center.
- Go to "Toolbox".
- Expand "Processor" tab.
- Mouse down on "Core".
- Drag it towards to the *AverageTaker.bsp* file.
- Drop it onto the *AverageTaker.bsp* file. You must see a rectangle with text of Processor1.
- Go to "Toolbox".
- Expand "Location" tab.
- Mouse down on "Core".
- Drag it towards to the *AverageTaker.bsp* file.
- Drop it onto the *AverageTaker.bsp* file. You must see a rectangle with text of MEM1.

By using the above steps, we create the components on **.bsp* file. Processor1 is the processor AverageTaker software will run on. MEM1 is a memory location that will

be used by the processor. We relate Processor1 and MEM1 with the following steps. After applying these steps, *AverageTaker.bsp* file must be as in Figure A.2.

- Mouse down on MEM1 with rigth mouse button.
- Drag it towards Processor1.
- Drop it onto Processor1. You must see a small square on the left-top corner of both Processor1 and MEM1. This squares are called as interface. The text in them are same. The text is 0.

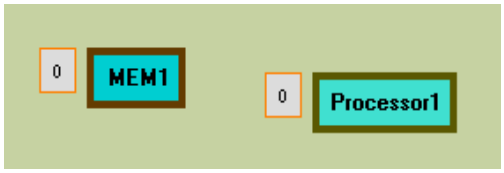


Figure A.2. BSP file

Use the following steps, we assign *AverageTaker.bsp* as the board support package for the project:

- Go to the "Solution Explorer" window.
- Right click to the *AverageTaker.bsp*.
- Click to "Set As Project's BSP File".

Thirdly, we need four operators to implement this example. These are for array indexing, smaller than comparision, summation, and division. Follow the following steps to create *AverageTaker.od* file. With these steps, we define "+" operator. Repeat these steps to add operators: "[]", "<", and "/". All operators have two inputs, one output. Cycles are seven, four, and five, respectively. CLBCount are three, 15, and 24, respectively. Select Array type for the first input of "[]". After applying these steps, *AverageTaker.od* file must be as in Figure A.3.

- Double click *AverageTaker.od*. You see a tabpage for this file on the center.
- Go to "Toolbox".

- Expand "Operator Definition" tab.
- Mouse down on "Operator".
- Drag it towards to the *AverageTaker.od* file.
- Drop it onto the *AverageTaker.od* file. You must see a rectangle with text of "->".
- Click this rectangle.
- Go to "Properties" window.
- Write "+" as OpName.
- Expand Operands.
- Select Inputs.
- Click to the little square on the right side of the text of Collection.
- Click Add button. This is the first operand for the operator.
- Click Add button. This is the second operand for the operator.
- Select Outputs.
- Click to the little square on the right side of the text of Collection.
- Click "Add" button. This is the output operand for the operator.
- Select Cycle. Write 2 for value of it.
- Expand Resources.
- Select CLBCount. Write 2 for value of it.

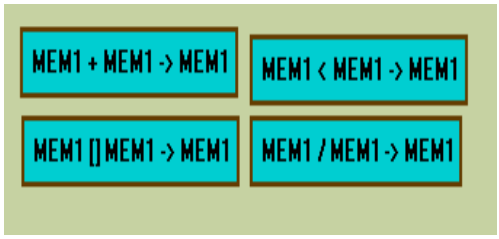


Figure A.3. OD file

Use the following steps, we assign *AverageTaker.od* as the operator definition file for the project:

- Go to the "Solution Explorer" window.
- Right click to the *AverageTaker.od*.

- Click to "Set As Project's OD File".

Use the following steps to control this assignment:

- Go to the "Toolbox" window.
- Look at the end of the "Toolbox" window. You must see "Operators" tab.
- Expand "Operators" tab. You must see all operators we defined, shown in Figure A.4.

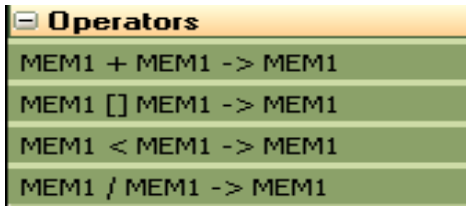


Figure A.4. Toolbox operators

Moreover, we write software code with LRH(+). To create General data whose refNames are index, 10, 1, and result, use the following steps:

- Double click *AverageTaker.rhplus*. You see a tabpage for this file on the center.
- Go to "Toolbox".
- Expand "General/Array Data" tab.
- Mouse down on "General".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of G1.
- Click this rectangle.
- Go to "Properties" window.
- Expand General.
- Select refName. Write sum for value of it.
- Select Location.
- Click the down button on the right side.
- Select MEM1.

To add an array to the file, use the following steps:

- Go to "Toolbox".
- Expand "General/Array Data" tab.
- Mouse down on "Array".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of A1.
- Click this rectangle.
- Go to "Properties" window.
- Expand Array.
- Select refName. Write Values for value of it.
- Select Location.
- Click the down button on the right side.
- Select MEM1.

To add a function to the file, use the following steps:

- Go to "Toolbox".
- Expand "Global Blocks" tab.
- Mouse down on "Function".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of G1.
- Click this rectangle.
- Go to "Properties" window.
- Select refName. Write AverageTaker for value of it.

We add function body by using the following steps. At the end of function body writing, when you click `Average:F1`, you must see the code shown in Table A.1 in "Editor" window. We show visual design in Figure A.5.

- Click "AverageTaker:F1" function on the file while pressing SHIFT key.
- Control your current scope by looking at the left-bottom text, "AverageTaker.rhplus: AverageTaker:F1".
- Go to "Toolbox".
- Expand "Global Blocks" tab.
- Mouse down on "Function".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of *F1*.
- Click this rectangle.
- Go to "Properties" window.
- Select *refName*. Write `AverageTaker` for value of it.
- Go to "Toolbox".
- Expand "Conditional Blocks" tab.
- Mouse down on "Loop".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of *L1*.
- Press `L` key.
- Control your last pressed key by looking at the left-bottom text, `L is being pressed now`.
- Click `L1` rectangle. You must see a line ending with an arrow from `AverageTaker:F1` to `L1`. `L1` is the first block in function's body. You can control this by looking at the left-bottom text, `SUC:L1 is the first block`.
- Click `L1` rectangle.
- Go to "Editor" window.
- Write `.<(index,10);` between blue text lines.
- Click `L1` function on the file while pressing `Shift` key.
- Control your current scope by looking at the left-bottom text, `AverageTaker.rhplus: AverageTaker:F1:L1`.
- Go to "Toolbox".
- Expand "Expressions" tab.

- Mouse down on "Statement List".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of *exp1*.
- Press `L` key.
- Control your last pressed key by looking at the left-bottom text, `L is being pressed now`.
- Click `L1` rectangle. You must see a line ending with an arrow from `L1` to `exp1`. `exp1` is the first block in loop's body. You can control this by looking at the left-bottom text, `SUC:exp1 is the first block`.
- Click `exp1` rectangle.
- Go to "Editor" window.
- Write code shown in Table A.2 between blue text lines.
- Go to `Function` scope by clicking '^' button on the rigth side of the file, under the `x` button.
- Go to "Toolbox".
- Expand "Expressions" tab.
- Mouse down on "Statement List".
- Drag it towards to the *AverageTaker.rhplus* file.
- Drop it onto the *AverageTaker.rhplus* file. You must see a rectangle with text of `exp2`.
- Press `F` key.
- Control your last pressed key by looking at the left-bottom text, `F is being pressed now`.
- Mouse down on `exp2` with rigth mouse button.
- Drag it towards `L1`.
- Drop it onto `L1`. You must see that `exp2` must be moved from its current location on the file to the under of the `L1`.
- Click `exp2` rectangle.
- Go to "Editor" window.
- Write `.=(result,./(sum,10));` between blue text lines.

Table A.1. LRH(+) code for average example

```
Function;AverageTaker:F1;F1;
No input
No output
  Loop;L1;L1;
  .<(index,10);
    Expressions;exp1;exp1;
      .=(sum,.+(sum,.[](Values,index)));
      .=(index,.+(index,1));
    ExpressionsEnd
  LoopEnd
  Expressions;exp2;exp2;
    .=(result,./(sum,10));
  ExpressionsEnd
FunctionEnd
```

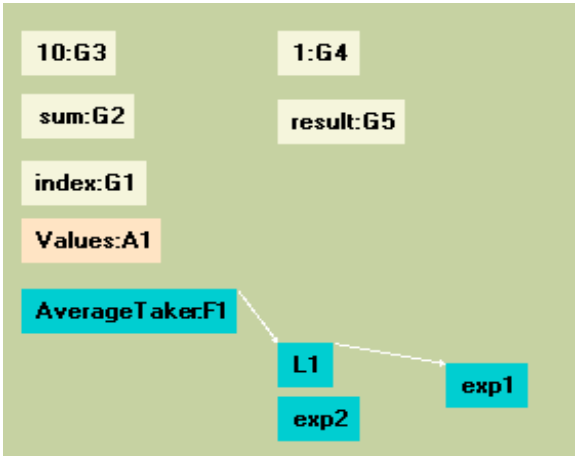


Figure A.5. Visual design

Table A.2. LRH(+) small code part for average example

```
.=(sum,.+(sum,.[](Values,index)));
.=(index,.+(index,1));
```

A.1. Analysis Tools

Use the following steps to activate resource references analysis tool:

- Click `AverageTaker:F1`.
- Go to "Build/Resource References (DFG Style)" menu.
- Click it. You must see a new tabpage on the center with name *AverageTaker:f1.rr*, shown in Figure A.6. You can follow the number of references for each BB, `L1`, `exp1`, and `exp2`.

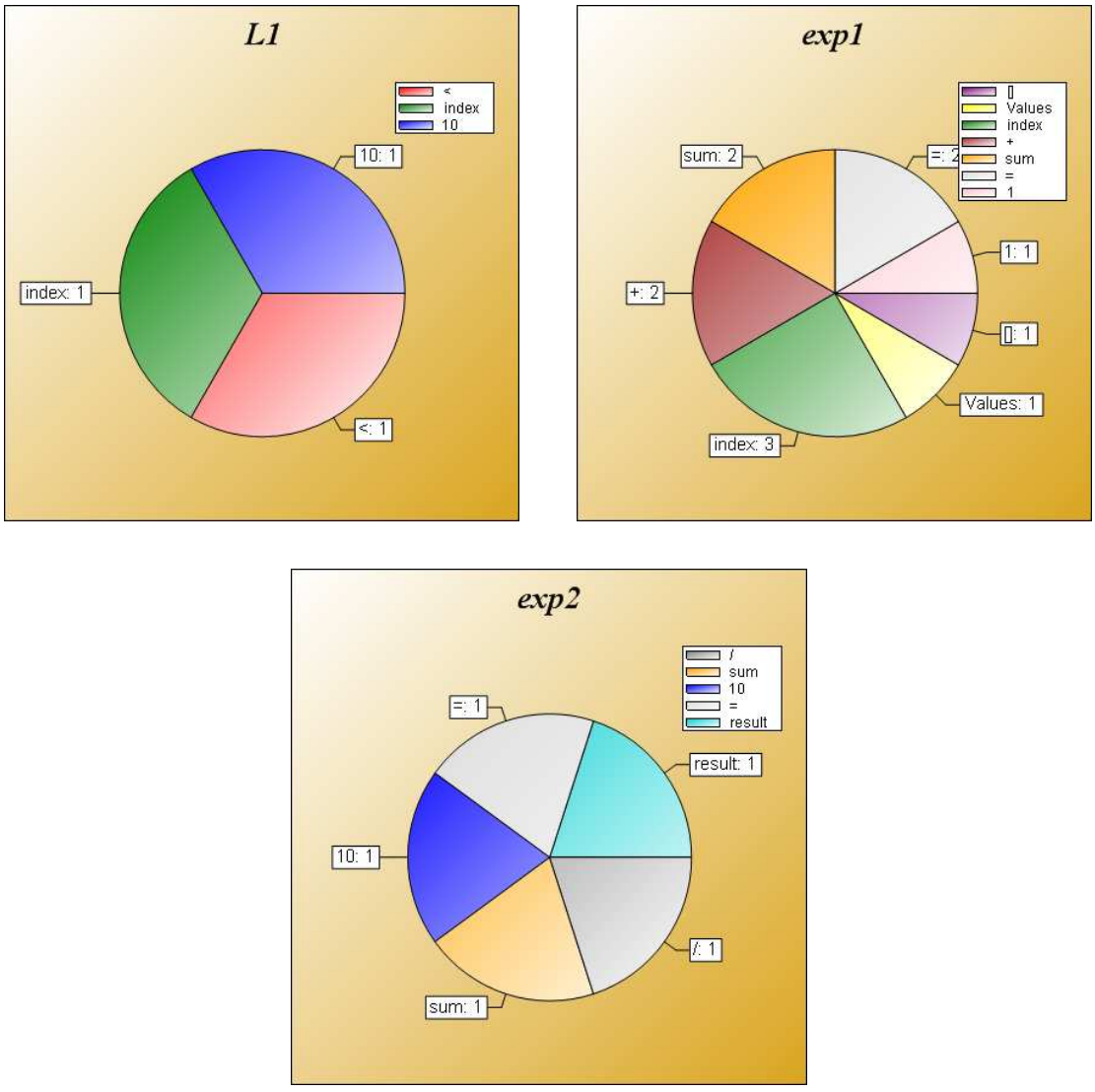


Figure A.6. Resource references

Use the following steps to activate possible execution times analysis tool:

- Click `AverageTaker:F1`.
- Go to "Build/Possible Execution Times and Paths" menu.
- Click it. You must see a new tabpage on the center with name *AverageTaker:f1.pp*, shown in Figure A.7. Worst case path is *AverageTaker:F1 -> L1 -> exp1 -> exp2*. It takes 20 cylces.

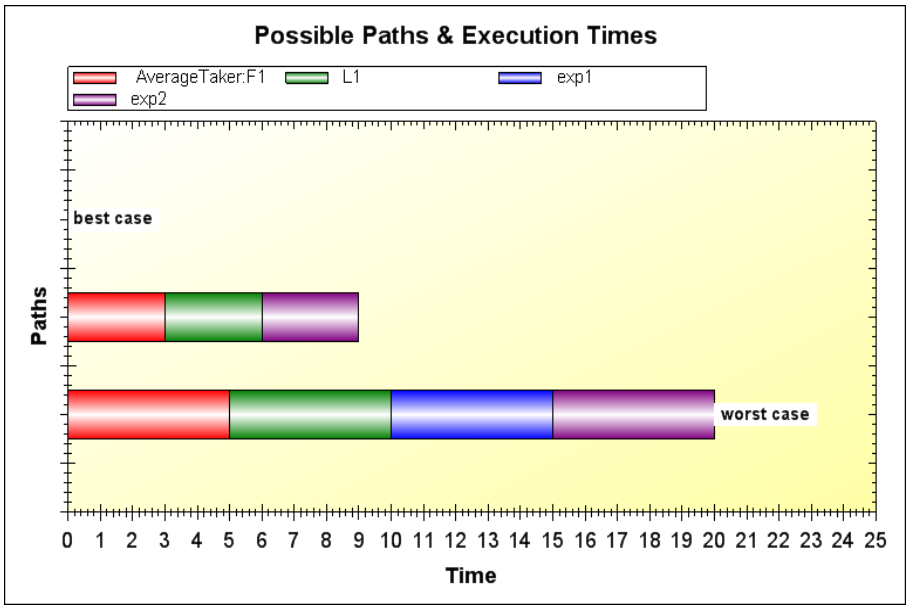


Figure A.7. Possible paths and execution times

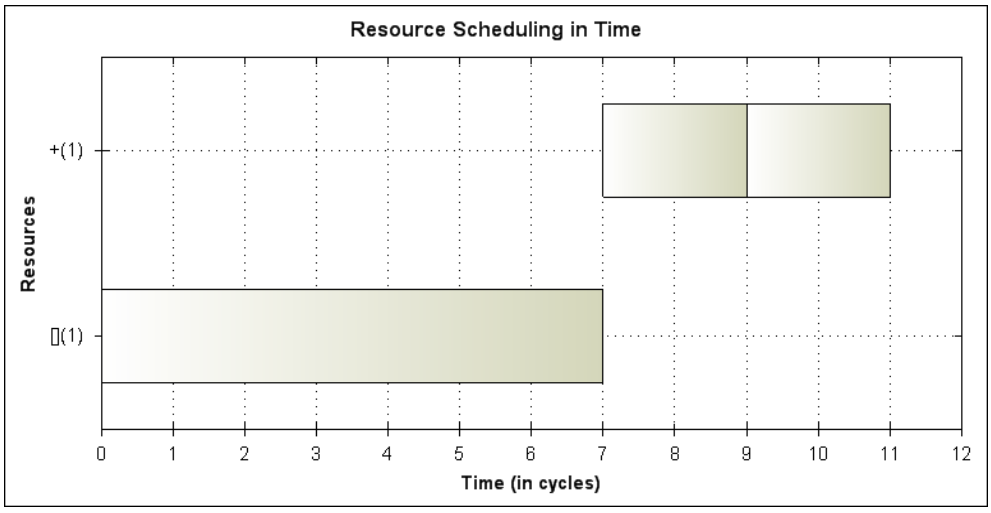


Figure A.8. Resource scheduling for exp1

Use the following steps to activate resource scheduler analysis tool:

- Click `exp1`.
- Go to "Build/Resource Scheduler" menu.
- Click it. You must see a new tabpage on the center with name *exp1.rs*, shown in Figure A.8. It takes 11 cycles.

Lastly, to get output file of FRH(+), use the following steps:

- Click `AverageTaker:F1`.
- Go to "Build/Build Output File (CDFG Style)" menu.
- Click it. You must see a new file with name of *AverageTaker.rhplus.cdfg* in the project directory. This file is input for LLS design. You can open it with any text editor. It is an XML file.

APPENDIX B: EXAMPLE OUTPUT FILE

The example output file is shown in Table B.1. Possible execution times list is under PosPathExTimesList tag. TimeList is the execution delay of the path shown in "PathList -> ArrayOfString" tag. In the example 13 is delay of the path, F1942, L703, exp733, IF129, exp802, exp741, IF137, exp820, IF145, exp838, exp754. RefGroupList has the references for the BBs. Under the tag, "RefGroupList -> ArrayOfArrayOfRef", an example reference for "RefGroupNameList -> RefGroupRefNames -> string" is shown. CDFG starts with ButtonModifiedCDFGList tag. ButtonModifiedCDFG is a node in CDFG. Name tag shows the name of node. Expressions contained in this BB are given with MultipleExpressions tag. ResourceScheduleResultsList includes scheduling of resources for each BB. ResourceScheduleResultNameList lists the owner of the previous scheduling results. ResourceName is the resource used between TimeStart and TimeEnd.

Table B.1. Example output file

```
<?xml version="1.0" encoding="utf-8"?>
<IRFile xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<PosPathExTimesList>
<ExecResult>
<TimeList>
<int>13</int>
</TimeList>
<MaximumEnd>13/MaximumEnd>
<MaximumEndPath>1</MaximumEndPath>
<MinimumEnd>0</MinimumEnd>
<MinimumEndPath>10</MinimumEndPath>
<PathList>
<ArrayOfString>
<string>F1942</string>
```

Table B.1. (continued)


```
<string>L703</string>
<string>exp733</string>
<string>IF129</string>
<string>exp802</string>
<string>exp741</string>
<string>IF137</string>
<string>exp820</string>
<string>IF145</string>
<string>exp838</string>
<string>exp754</string>
</ArrayOfString>
</PathList>
</ExecResult>
</PosPathExTimesList>
<RefGroupList>
<ArrayOfArrayOfRef>
<ArrayOfRef>
<Ref>
<BehaviourFunction />
<InstanceCount>1</InstanceCount>
<InstanceFreeTimes>
<int>3</int>
</InstanceFreeTimes>
<RefCount>1</RefCount>
<Resources>
<CLBCount>3</CLBCount>
<IOCount>0</IOCount>
</Resources>
<Name>I2CWriteS</Name>
<DisplayColor />
```

Table B.1. (continued)


```
<ModifiedList />
<InputList />
</Ref>
</ArrayOfRef>
</ArrayOfArrayOfRef>
</RefGroupList>
<RefGroupNameList>
<RefGroupRefNames>
<GroupName>F1942</GroupName>
<RefListNames>
<string>Security: Main</string>
<string>LoopForever</string>
<string>expList: I2C data length</string>
<string>IF: Control length</string>
<string>expList: Read</string>
<string>0:Constant</string>
<string>expList: CAN data length</string>
<string>IF: Control length</string>
<string>expList: Read</string>
<string>IF: Is Sent and Received Signal are same</string>
<string>expList: Error Occurred</string>
<string>expList: Send Error State</string>
</RefListNames>
</RefGroupRefNames>
</RefGroupNameList>
<ButtonModifiedCDFGList>
<ButtonModifiedCDFG xsi:type="ButtonModifiedCDFGExp">
<TypeNameTable />
<Name>exp820</Name>
<MultipleExpressions>
```

Table B.1. (continued)

```
<ArrayOfSubExpression>
<SubExpression>
<Number>2272</Number>
<Name>SE2264/Name>
<OperandTypes />
<OpName>CANRead</OpName>
<Resources>
<CLBCount>3</CLBCount>
<IOCount>0</IOCount>
</Resources>
<Cycle>2</Cycle>
<OperandNames>
<string>CANRead</string>
<string>CANResponse</string>
<string>1</string>
</OperandNames>
<OperandModifieds>
<boolean>true</boolean>
<boolean>false</boolean>
</OperandModifieds>
<InstanceIndex>1</InstanceIndex>
<TimeStart>0</TimeStart>
<TimeEnd>2</TimeEnd>
</SubExpression>
</ArrayOfSubExpression>
</MultipleExpressions>
</ButtonModifiedCDFG>
</ButtonModifiedCDFGList>
<ResourceScheduleResultsList>
<ArrayOfResourceScheduleResult>
```

Table B.1. (continued)

<ResourceScheduleResult>

<ResourceName>I2CReadS(1)</ResourceName>

<TimeStart>0</TimeStart>

<TimeEnd>3</TimeEnd>

</ResourceScheduleResult>

<ResourceScheduleResult>

<ResourceName>CANWrite(1)</ResourceName>

<TimeStart>3</TimeStart>

<TimeEnd>5</TimeEnd>

</ResourceScheduleResult>

</ArrayOfResourceScheduleResult>

</ResourceScheduleResultsList>

<ResourceScheduleResultNameList>

<string>L703</string>

<string>exp733</string>

<string>IF129</string>

<string>exp741</string>

<string>IF137</string>

<string>IF145</string>

<string>exp754</string>

<string>exp802</string>

<string>exp820</string>

<string>exp838</string>

</ResourceScheduleResultNameList>

</IRFile>

APPENDIX C: RH(+) IDE SOFTWARE DETAILS

C.1. Development Environment

We have used Microsoft Visual Studio 2005 for our RH(+) IDE development. Programming language is C#. Framework version is 2.0. Therefore, to be able to run RH(+) IDE, .NET framework 2.0 must be installed in the computer. In this chapter, we list the files included in our software design. This data will be useful to understand the overall structure of the software.

C.2. Software Design

In our software solution, we have three projects shown in Figure C.1. The name of the solution is *RHPlusIDE*. The first project is *RHPlusIDE*, which is our IDE, shown in Figure C.2. *ZedGraph* is an external project included to our project, shown in Figure C.3. It is used to show our analysis tool results. *RHPlusIDESetup* is constructed to produce a distribution for our IDE, shown in Figure C.4.

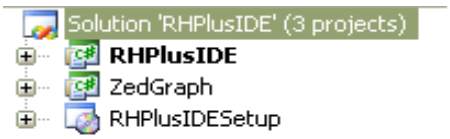


Figure C.1. Solution with three projects

C.3. RHPlusIDE Project

Under *Basic* directory, we have some basic data used by many classes. *Access-Rigths* class is used by memory locations. It includes the abilities that are read, write, rewrite for the block hardware units. They may be true or false. *Converters* include a number of classes. They are required to enable HLS designer modifying the component properties via *Properties Window*. *FrequencyUnit* class includes frequency value and its unit.

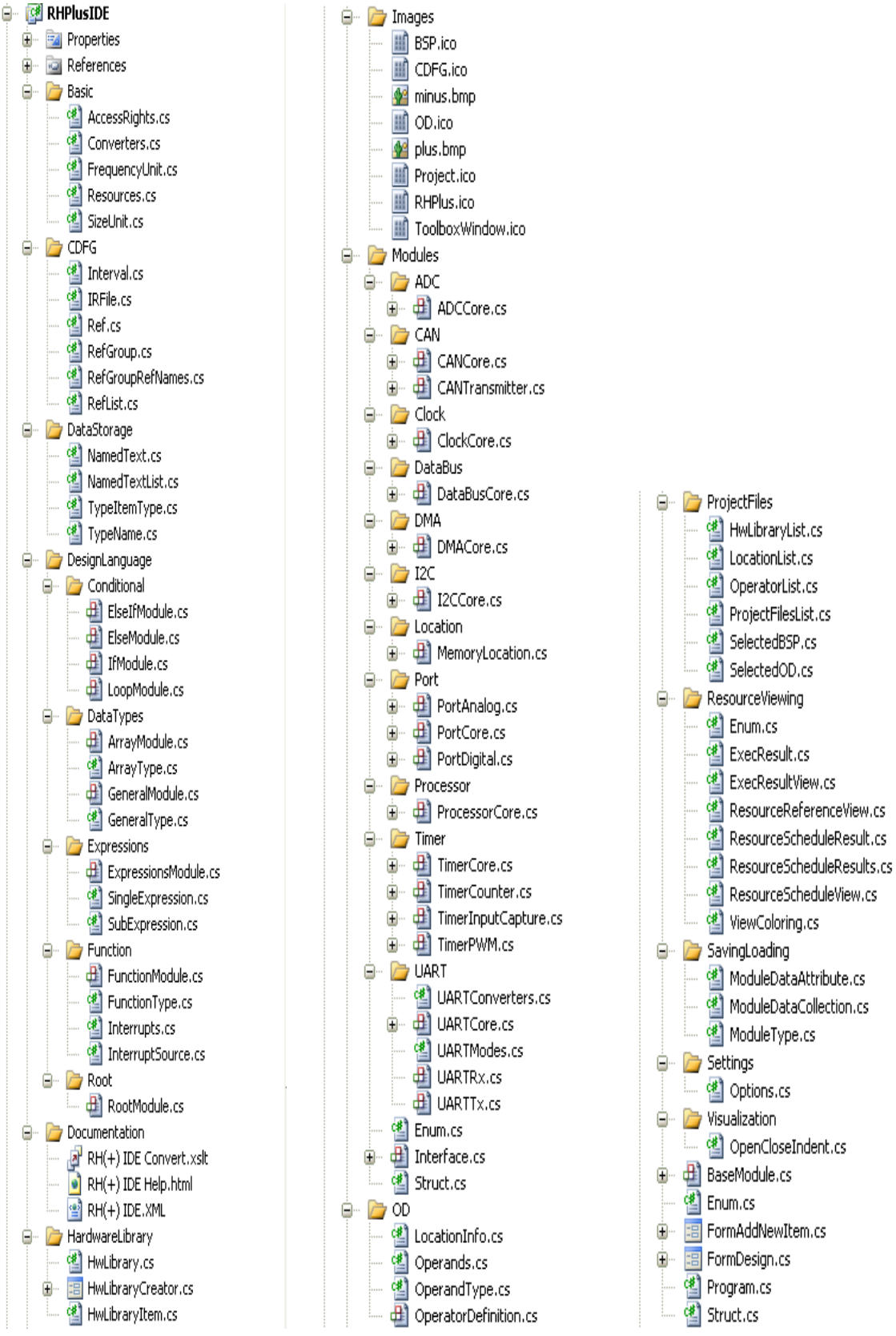


Figure C.2. RH(+) IDE project

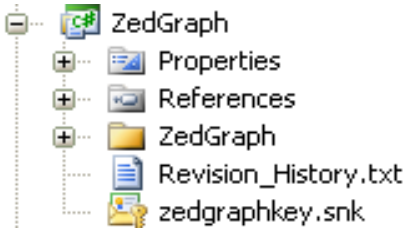


Figure C.3. ZedGraph project

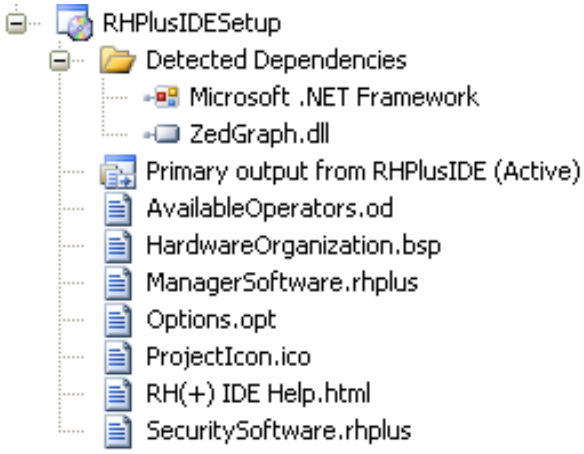


Figure C.4. RHPlusIDESetup project

Moreover, under *Basic* directory, we see *Resources* that are the basic detail for LLS design. It has the number of CLBs and I/O pins. *SizeUnit* class shows a length. It may be the length of a memory. It has length value and its unit. For example, length is 32, and unit is KiloByte.

Under *CDFG* directory, we have a list of classes used while producing output file of the framework. *Interval* class has data, interval start and end, to show a time interval. *IRFile* class has the data collection that will be included in output file. *Ref* class is used to represent a reference in a *BB*. RefGroup is a list of *RefList*. *RefGroupRefNames* has the names of each *RefList*. RefList is a list of *Ref*.

Under *DataStorage* directory, we have some strategy to store run-time data in our project files. NamedText class has two strings: name and text. *NamedTextList* is a list of *NamedText*. *TypeItemType* is used to represent the design relation between different modules. *TypeName* is used to represent the run-time relation between different modules.

Under *DesignLanguage* directory, we introduce LRH(+). *Conditional* directory has the control flow structures. There are four classes: *ElseIfModule*, *ElseModule*, *IfModule*, and *LoopModule*. *DataTypes* directory has the *GeneralModule* and *ArrayModule* classes. We have also *GeneralType*, and *ArrayType* classes to be able to refer *GeneralModule* and *ArrayModule* classes. Under *Expressions* directory, we have classes to represent *BB* expressions. *ExpressionsModule* is used to add a list of statements to our LRH(+) software. *SingleExpression* class shows one line of statement or a condition (a list of operations). *SubExpression* represents only one operation. Under *Function* directory, we have classses about functions. *FunctionModule* is used to instantiate a function. *FunctionType* class is the unique data a function has. A function may be attached as an interrupt. *Interrupt* class represents an interrupt. An ISR may have different interrupt resources. They are shown with *InterruptSource* class. Under *Root* directory, we have *RootModule* class. It is used symbolically in LRH(+) design. It is the root node in design tree and invisible to the HLS designer.

Under *Documentation* directory, we provide documents about the project. A help file for the IDE is created. This is *RH(+) IDE Help.html*. It is accessed from the IDE by pressing *F1* or from *Help menu*. *RH(+) IDE.XML* is the software documentation. It is created by Microsoft Visual Studio. This xml file is converted to a documentation file via *RH(+) IDE Convert.xslt*.

Under *HardwareLibrary* directory, there are classes to create a hardware library. *HwLibrary* represents a hardware library. *HwLibraryItem* is an entry in it. *HwLibraryCreator* is used to manage hardware libraries. Under *Images* directory, the images used in the IDE menu lists are included.

Under Modules directory, there are classes used in **.bsp* file design. The classes are: *ADCCore*, *CANCore*, *CANTransmitter*, *ClockCore*, *DataBusCore*, *DMACore*, *I2CCore*, *MemoryLocation*, *PortAnalog*, *PortCore*, *PortDigital*, *ProcessorCore*, *TimerCore*, *TimerCounter*, *TimerInputCapture*, *TimerPWM*, *UARTCore*, *UARTRx*, *UARTTx*, and *Interface*. *Enum* file has enumerations special to these modules. *Struct* file also has structures special to these modules.

Under *OD* directory, operator definition related classes are considered. *LocationInfo* class has location of an operand. *Operand* class represents an operand. *OperandType* shows location and data type together. *OperatorDefinition* is the main class to represent an operation.

Under *ProjectFiles* directory, there are classes specific to the current active project in IDE. *HwLibraryList* has the hardware libraries included for the active project. *LocationList* has the location of the selected BSP file for the project. *OperatorList* has the operators of the selected OD file for the project. *ProjectFilesList* has the contents of the **.prj* files. *SelectedBSP* is the BSP file used in the project. *SelectedOD* is the OD file used in the project.

Under *ResourceViewing* directory, there are classes used by analysis tools. *Enum* file has enumerations special to these analysis tools. *ExecResult* is used to store possible path analysis. *ExecResultView* constructs the visual graph to show the results shown by *ExecResult*. *ResourceReferenceView* constructs the visual graph to show the references for each *BB*. *ResourceScheduleResult* is the schedule for a resource. *ResourceScheduleResults* is a list of schedules for a list of resources. *ResourceScheduleView* constructs the visual graph to show the results shown by *ResourceScheduleResults*. *ViewColoring* class is used to assign different colors for each *BB* in the analysis graphs.

Under *SavingLoading* directory, there are classes used to save and load project files. *ModuleDataAttribute* has an attribute class. This attribute is used by the software designer if he wants to save a field of a class. *ModuleDataCollection* has a list of fields having *ModuleDataAttribute*. *ModuleType* is the intermediate representation of the design files. Design files have converted a list of objects in type of *ModuleType*. Then, these objects are saved to the disk. While loading, disk files are loaded to a list of objects in type of *ModuleType*. Then, they are converted to the design files.

Under *Settings* directory, we have *Options* class for IDE. These options are saved after changes on it. Under *Visualization* directory, there is a class, *OpenCloseIndent*. It is used to provide indentation in *Editor Window* for LRH(+) design file. *Base-*

Module class is the base class for the components are located under *DesignLanguage*, *Modules*, and *OD* directories. *Enum* file has enumerations common to all directories. *FormAddNewItem* is used when HLS designer wants to add a new file into the project. It lists the available file types to HLS designer. *FormDesign* is the main file of the project, the main screen presented to HLS designer. *Program* is the main entry point for Microsoft Visual Studio C# projects. Structures common to all directories are located in *Struct*.

REFERENCES

1. Sangiovanni-Vincentelli, A. and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems", *IEEE Des. Test*, Vol. 18, No. 6, pp. 23–33, 2001.

2. Local Interconnect Network, http://www.lin-subbus.org, 2006.

3. Controller Area Network, http://www.can-cia.org/can/, 2006.

4. Compton, K. and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Comput. Surv.*, Vol. 34, No. 2, pp. 171–210, 2002.

5. NEC Electronics, http://www.necel.com, 2006.

6. SystemC Community, http://www.systemc.org, 2006.

7. Ku, D. and G. DeMicheli, *HardwareC – A Language for Hardware Design (Version 2.0)*, Technical Report, Stanford, CA, USA, 1990.

8. Cristina, D., C. Peixoto and D. C. da Silva Junior, "A Framework for Architectural Description of Embedded System", *International Workshop on Software and Compilers for Embedded Systems, SCOPES*, Vol. 3199, pp. 2–16, Amsterdam, September 2004.

9. Schliebusch, O., A. Hoffmann, A. Nohl, G. Braun and H. Meyr, "Architecture Implementation Using the Machine Description Language LISA", *Proceedings of the 2002 Conference on Asia South Pacific Design Automation/VLSI Design (ASP-DAC'02)*, pp. 239–244, IEEE Computer Society, Washington, DC, USA, 2002.

10. Damasevicius, R., "A Subset-Based Comparison of Main Design Languages", *Information Technology and Control, Kaunas, Technologija*, Vol. 1, No. 30, pp. 49–56, 2004.

11. Posadas, H., F. Herrera, V. Fernandez, P. Sanchez and E.Villar, "Single Source Design Environment for Embedded Systems Based on SystemC", *Design Automation for Embedded Systems*, Vol. 9, No. 4, pp. 293–312, December 2004.

12. Cesario, W. O., G. Nicolescu, L. Gauthier, D. Lyonnard and A. A. Jerraya, "Colif: A Design Representation for Application-Specific Multiprocessor SOCs", *IEEE Des. Test*, Vol. 18, No. 5, pp. 8–20, 2001.

13. Cesario, W. O., D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, L. Gauthier and M. Diaz-Nava, "Multiprocessor SoC Platforms: A Component-Based Design Approach", *IEEE Des. Test*, Vol. 19, No. 6, pp. 52–63, 2002.

14. Sgroi, M., L. Lavagno and A. Sangiovanni-Vincentelli, "Formal Models for Embedded System Design", *IEEE Des. Test*, Vol. 17, No. 2, pp. 14–27, 2000.

15. Object Management Group, "UML Resource Page", http://www.uml.org, 2006.

16. Oliver, I., "Applying UML and MDA to Real Systems Design", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*, pp. 70–71, IEEE Computer Society, Washington, DC, USA, 2005.

17. Raistrick, C., P. Francis and J. Wright, *Model Driven Architecture with Executable UML(TM)*, Cambridge University Press, New York, NY, USA, 2004.

18. Mellor, S. J., "Executable and Translatable UML", http://www.embedded.com/story/OEG20030115S0043, 2003.

19. Mellor, S. J., "STSC CrossTalk - Executable and Translatable UML", http://www.stsc.hill.af.mil/crossTalk/2004/09/0409Mellor.html, September 2006.

20. Ienne, P. and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*, Morgan Kaufmann Publishers, 2006.

21. Mishra, P. and N. Dutt, *Architecture Description Languages*, Morgan Kaufmann

Publishers, 2006.

22. Fauth, A., J. V. Praet and M. Freericks, "Describing Instruction Set Processors Using nML", *Proceedings of the 1995 European Conference on Design and Test (EDTC'95)*, pp. 503–507, IEEE Computer Society, Washington, DC, USA, 1995.

23. Halambi, A., P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability", *Design, Automation and Test in Europe (DATE'99)*, pp. 485–490, 1999.

24. Halambi, A., A. Shrivastava, N. Dutt and A. Nicolau, "A Customizable Compiler Framework for Embedded Systems", *International Workshop on Software and Compilers for Embedded Systems, SCOPES*, Springer, March 2001.

25. Zivojnovic, V., S. Pees and H. Meyr, "LISA - Machine Description Language and Generic Machine Model for HW/SW Co-design", *IEEE Workshop on VLSI Signal Processing IX*, pp. 127–136, San Francisco, 1996.

26. NXP Semiconductors, "I2C", http://www.nxp.com/products/interface_control/i2c/, 2006.

27. Mano, M. M., *Computer System Architecture*, Prentice-Hall, 2001.

28. W3Schools, "XML Tutorial", http://www.w3schools.com/xml/, 2007.

29. Keller, E. R., *Programming Model for Network Processing on an FPGA*, M.S. Thesis, Graduate School of the University of Massachusetts Amherst, Electrical and Computer Engineering, February 2005.

30. Cardoso, J. M. P., "On Estimations for Compiling Software to FPGA-based Systems", *Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, pp. 225–230, IEEE Computer Society, Washington, DC, USA, 2005.

31. Coyle, F. P. and M. A. Thornton, "From UML to HDL: A Model Driven Architectural Approach to Hardware-Software Co-design", *Information Systems: New Generations Conference, ISNG*, pp. 88–93, 2005.

32. Schattkowsky, T., W. Mueller and A. Rettberg, "A Model-Based Approach for Executable Specifications on Reconfigurable Hardware", *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*, pp. 692–697, IEEE Computer Society, Washington, DC, USA, 2005.

33. Schattkowsky, T. and W. Mueller, "A UML Virtual Machine for Embedded Systems", *International Conference on Information Systems - New Generations (ISNG)*, USA, April 2005.

REFERENCES NOT CITED

1. Cardoso, J. M. P. and H. C. Neto, "Compilation for FPGA-Based Reconfigurable Hardware", *IEEE Des. Test*, Vol. 20, No. 2, pp. 65–75, 2003.

2. Gonzalez, R. E., "Xtensa: A Configurable and Extensible Processor", *IEEE Micro*, Vol. 20, No. 2, pp. 60–70, 2000.

3. Doboli, A. and P. Eles, "Scheduling Under Data and Control Dependencies for Heterogeneous Architectures", *ICCD '98: Proceedings of the International Conference on Computer Design*, p. 602, IEEE Computer Society, Washington, DC, USA, 1998.

4. Gajski, D. D. and L. Ramachandran, "Introduction to High-Level Synthesis", *IEEE Des. Test*, Vol. 11, No. 4, pp. 44–54, 1994.

5. Augé, I., F. Pétrot, F. Donnet and P. Gomez, "Platform-based Design from Parallel C Specifications", *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 24, No. 12, pp. 1811–1826, 2005.

6. Leupers, R., "Compiler Design Issues for Embedded Processors", *IEEE Des. Test*, Vol. 19, No. 4, pp. 51–58, 2002.

7. Schaumont, P. and I. Verbauwhede, "A Component-Based Design Environment for ESL Design", *IEEE Des. Test*, Vol. 23, No. 5, pp. 338–347, 2006.

8. Radojevic, I., Z. Salcic and P. S. Roop, "Modeling Embedded Systems: From SystemC and Esterel to DFCharts", *IEEE Des. Test*, Vol. 23, No. 5, pp. 348–358, 2006.

9. Baldassin, A., P. C. Centoducatte and S. Rigo, "Extending the ArchC Language for Automatic Generation of Assemblers", *SBAC-PAD '05: Proceedings of the 17th*

International Symposium on Computer Architecture on High Performance Computing, pp. 60–68, IEEE Computer Society, Washington, DC, USA, 2005.

10. Leupers, R., "Code Generation for Embedded Processors", *ISSS '00: Proceedings of the 13th International Symposium on System Synthesis*, pp. 173–178, IEEE Computer Society, Washington, DC, USA, 2000.

11. Schattkowsky, T. and W. Muller, "Model-Based Design of Embedded Systems", *International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC*, pp. 121–128, 2004.

12. Lee, T. Y., Y. H. Fan, T. H. Yang, C. C. Tsai, W. T. Lee and Y. S. Hwang, "RCGES: Retargetable Code Generation for Embedded Systems.", *ATVA*, Vol. 3299, pp. 415–425, 2004.

13. Kastner, D., "TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses", *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pp. 18–36, Springer-Verlag, Inc., New York, NY, USA, 2003.

14. Martin, G., L. Lavagno and J. Louis-Guerin, "Embedded UML: A Merger of Real-time UML and Co-design", *CODES '01: Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, pp. 23–28, ACM Press, New York, NY, USA, 2001.

15. Nascimento, F. A. M. D., M. F. da S. Oliveira, M. A. Wehrmeister, C. E. Pereira and F. R. Wagner, "MDA-based Approach for Embedded Software Generation from a UML/MOF Repository", *SBCCI '06: Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*, pp. 143–148, ACM Press, New York, NY, USA, 2006.

16. Ahmed, A. and A. Abbas, "BPDL - Processor Definition Language with Support for Cycle Accurate DSP Architectures", *IEEE International Multi Topic Conference,*

IEEE INMIC, pp. 200–204, Lahore, Pakistan, November 2001.

17. Abbas, A., A. Ahmed, A. Ahmed, W. U. Z. Bajwa, A. Anwar and S. Abbasi, "A Retargetable Tool-suite for the Design of Application Specific Instruction Set Processors Using a Machine Description Language", *IEEE International Symposium on Circuits and Systems, ISCAS*, Vol. 1, pp. 425–428, Scottsdale, AZ, September 2002.

18. Goossens, G., D. Lanneer, W. Geurts and J. V. Praet, "Design of ASIPs in Multi-Processor SoCs Using the Chess/Checkers Retargetable Tool Suite", *International Symposium on System-on-Chip, SoC*, Tampere, November 2006, http://www.retarget.com/doc/goossens-SoC06.pdf.

19. Lee, J. Y., H. D. Yoon, J. H. Yang, I. C. Park and C. M. Kyung, "Code Generation for Embedded Processors with Complex Instructions, ICVC", *International Conference on VLSI and CAD*, pp. 525–528, Seoul, South Korea, November 1999.

20. Galanis, M. D., A. Milidonis, G. Theodoridis, D. Soudris and C. E. Goutis, "A Method for Partitioning Applications in Hybrid Reconfigurable Architectures", *Design Automation for Embedded Systems*, Vol. 10, pp. 27–47, 2006.

21. Zivojnovic, V., S. Pees and H. Meyr, "LISA - Machine Description Language and Generic Machine Model for HW/SW Co-design", *IEEE Workshop on VLSI Signal Processing*, pp. 127–136, San Francisco, October 1996.

22. Hohenauer, M., H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr and G. Braun, "Compiler-in-loop Architecture Exploration for Efficient Application Specific Embedded Processor Design", *Design and Elektronik*, WEKA Verlag, Munich, Germany, February 2004, http://www.iss.rwth-aachen.de/4_publikationen/res_pdf/2004HohenauerDE.pdf.

23. Jesman, R., F. M. Vallina and J. Saniie, *MicroBlaze Tutorial Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software*

Tools, Technical Report, Embedded Computing and Signal Processing Laboratory, Illinois Institute of Technology.

24. Lai, Y. T., H. Y. Lai and C. N. Yeh, "Placement for the Reconfigurable Datapath Architecture", *IEEE International Symposium on Circuits and Systems, ISCAS*, Vol. 2, pp. 1875–1878, May 2005.

25. XILINX, *Platform Specification Format Reference Manual*, October 2005.

26. Cardoso, J. M. P. and H. C. Neto, "Fast Hardware Compilation of Behaviors into an FPGA-Based Dynamic Reconfigurable Computing System", *XII Symposium on Integrated Circuits and Systems Design, SBCCI*, pp. 150–153, IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

27. Sun, F., S. Ravi, A. Raghunathan and N. K. Jha, "Synthesis of Application-Specific Heterogeneous Multiprocessor Architectures Using Extensible Processors", *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)*, pp. 551–556, IEEE Computer Society, Washington, DC, USA, 2005.

28. Doucet, F., S. Shukla and R. Gupta, "BALBOA: A Component-Based Design Environment for Composition and Simulation of System Level Models", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22, pp. 1597–1612, December 2003.

29. Target Compiler Technologies N.V., *CHESS/CHECKERS: A Retargetable Toolsuite for Embedded Processors*, Technical Report, Leuven, Belgium, June 2003.

30. ARC International, *ARChitect Processor Configurator: The Power of Configurable Processing at Your Fingertips*, Technical Report, 2005.

31. Kukkala, P., J. Riihimaki, M. Hannikainen, T. D. Hamalainen and K. Kronlof, "UML 2.0 Profile for Embedded System Design", *DATE '05: Proceedings of the*

Conference on Design, Automation and Test in Europe, pp. 710–715, IEEE Computer Society, Washington, DC, USA, 2005.

32. Fauth, A. and A. Knoll, "Automatic Generation of DSP Program Development Tools", *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 457–460, Minneapolis, 1993.

33. Lanneer, D., J. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen and G. Goossens, "CHESS: Retargetable Code Generation for Embedded DSP Processors", *Code Generation for Embedded Processors*, pp. 85–102, 1995.

34. Lohr, F., A. Fauth and M. Freericks, *Sigh/Sim: An Environment for Retargetable Instruction Set Simulation*, Technical Report, Dept. Computer Science, Tech. Univ., Berlin, Germany, 1993.

35. Hoffmann, A., H. Meyr and R. Leupers, *Architecture Exploration for Embedded Processors with Lisa*, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

36. Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, Inc., New York, NY, USA, 1993.

37. Gibilisco, S., *The Illustrated Dictionary of Electronics*, McGraw-Hill/TAB Electronics, 2001.

38. Target Compiler Technologies N.V., http://www.retarget.com, 2006.

39. Tensilica Inc., http://www.tensilica.com, 2005.

40. CoWare Inc., http://www.coware.com, 2007.

41. ARC International, http://www.arc.com, 2007.

42. CoWare, Inc., "Flexible Platform-Based Design With the CoWare N2C Design System", http://www.coware.com/pdf/pbdWhitepaper.pdf, October 2000.

43. Govindarajan, S., "Scheduling Algorithms for High-Level Synthesis", Term Paper, Dept. of ECECS University of Cincinnati, March 1995.

44. Radcliffe, D., *Hardware Synthesis From a Traditional Programming Language*, M.S. Thesis, The University of Queensland, School of Information Technology and Electrical Engineering, October 2002.