General Parser

(General Purpose Bitstream Parser)

Takaaki Ota, ATC/ETD

8/23/1999

Contents

General Parser

(General Purpose Bitstream Parser)

Takaaki Ota, ATC/ETD
8/23/1999

1 Background

A bitstream parser parses through a digital bitstream according to a given set of syntax rules then deciphers 0s and 1s and presents the contents to the user in a human understandable form. It is a very useful and almost necessary tool in digital communication and signal processing product development. When a target system, which processes digital signal, is not behaving as it is designed to, the engineer who designed the system needs to know exactly why the system behaves the wrong way so that he or she can come up with an effective solution to fix that problem. In this process of analyzing the system behavior the engineer must find out what specific input pattern, often overlooked at the time of the original system designed, is triggering the misbehavior of the system. For this purpose, the engineer needs to study and understand the input stream thoroughly along with the analysis of the system itself.

I have created several bitstream-parsing software tools in the past MPEG based product development projects. Each one was made to fit those specific needs of the project. The first parser was a command line program, which parsed MPEG video elementary stream in a batch process. It was then modified to parse DSS specific extended syntax. Based on that program the second major rewriting resulted as an interactive GUI based parser, this not only parsed MPEG video elementary stream but also MPEG transport stream. That was then modified to parse DSS transport stream as well. Yet, another modification made it parse AC3 audio stream too. Users of the software drove those changes. The GUI based tool was especially successful due to its easy to use interface. This success made the tool very popular among engineers and the popularity increased the demand to add further parsing capabilities. Some had asked me to add an EPG data parsing capability, however I think the foundation of the software had already reached its limit a long time ago. Many added features are now tweaks to the original design and were never considered from the beginning. I do not want simply decline those constructive requests from the users yet the technical difficulty does not allow me to further tweak the old foundation. I needed to come up with some fundamentally different approach to meet all these demands. I have been thinking and seeking an answer to this question for some time. Now I think I have found an ultimate answer. The result I named a ‘General Parser’.

2 Limitation of an Adaptive Parser

The GUI parser was an adaptive parser. Because of its effective dialoging capability with a user, it offers the user many options to select. Unlike a command line program, a GUI parser does not force users to remember cryptic option names. Selecting a specific option is simply to bring up the option dialog box, read the option names, and point-and-click the operations of one's need. It was that simple to change the parser behavior from parsing MPEG transport stream to DSS transport stream to video elementary stream to audio stream within MPEG transport stream. However, what happens if a user cannot find an option for a specific need? This is exactly the problem I have mentioned before. The parser program can only perform things that it was originally designed to do but nothing beyond that.

3 A Programmable Parser

To overcome the limitation of the adaptive parser mentioned above, the parser must become programmable so that users can define what the parser parses and how it does. However, what exactly does it mean to make a programmable parser? The parser software itself is already a program that is designed to perform parsing jobs. How can one make a program programmable? In order to do this we need to separate the two distinctly different concepts, which are mixed together in the conventional parser program.

3.1 Syntax Dependent Actions and Syntax Independent Actions

The term syntax here means bitstream syntax of the input bitstream, meaning a rule that governs how to interpret the input bitstream. Any parser action can be categorized into either a syntax dependent action group or a syntax independent action group. For example, an action to read the next X bits from the current location of the input bitstream is a syntax independent action. This action may be operating system dependent but indifferent from what syntax the input stream follows. However, a decision of whether to read the next Y bits immediately following the previous X bits or to skip it all together until the parser sees a bit pattern Z somewhere down the stream, is syntax dependent. Because this decision is based on the previous bitstream information including bits X. Hard coding the syntax independent actions into the heart of the parser, and separating the syntax dependent actions from the parser makes the parser programmable. Syntax dependent actions, however, are the real work that parser must perform therefore the parser somehow must obtain those separated syntax dependent actions. Thus this syntax dependent action is the program that the parser must obtain, understand and execute.

3.2 Describing the Syntax Dependent Actions

It is easy to say things conceptually, like “separate the syntax dependent actions from the parser and make the parser programmable”. But in order to build a real working system things must be defined precisely to detail; otherwise it is just a beautiful blueprint of a building that no one knows how to build.

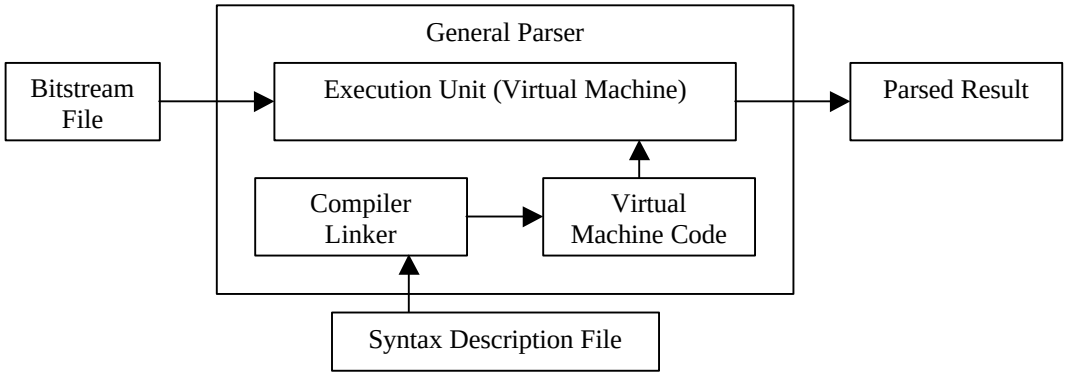
The separated syntax dependent actions must be described in a machine-readable form without containing any ambiguity so that the parser can execute it in a deterministic fashion. One way of realizing this is defining a computer language that describes the syntax dependent actions and the parser interprets this description when it actually parses the input bitstream. An obvious drawback of the interpreting system is the expectation of very slow parsing. Another idea is breaking down the parser dependent actions into very low level primitive operations and describing the entire syntax dependent actions by combining the predefined primitive operations. This is exactly like assembly programming. It may be fast to execute but we can not expect anyone would actually try to use this sophisticated and eccentric tool, which requires learning a strange set of instruction code and writing a unique assembly program. To take advantage of the both approaches and eliminate the shortcomings of each, a language must be defined that is easy to learn and straightforward to describe the syntax dependent actions. Then a set of primitive operations that is fast to execute needs to be defined. Then, a system needs to be developed that translates the syntax dependent actions described in that language into a set of predefined primitive operations. Finally, a system that actually executes those operations needs to be built. In other words I need to 1) define a high level language, 2) define a virtual machine code, 3) write a compiler that translates a source program written in that language into a set of virtual machine code, and 4) implement the virtual machine that executes the virtual machine instructions.

4 Parser Architecture (Operational Model)

4.1 Basic Architecture

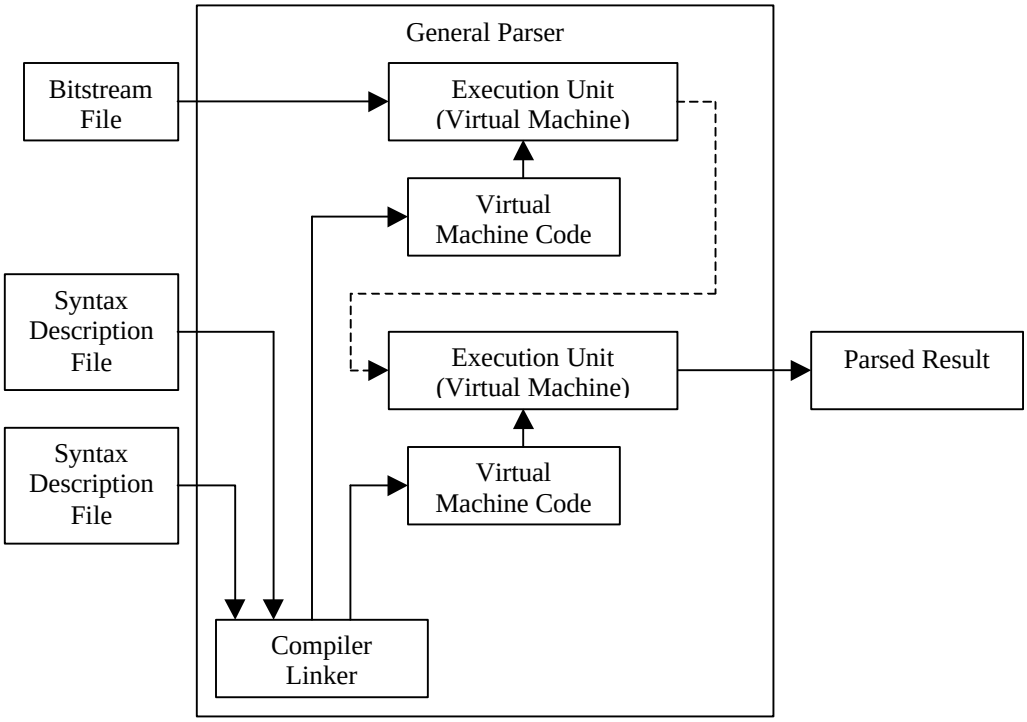
As described in the previous section, the General Parser takes a syntax description file. Then it compiles the syntax description into primitive instructions (virtual machine codes). Finally it executes them as quickly as possible while reading the bitstream file and produces the parsed result.

The following diagram shows the basic relationship between the subject bitstream file, the parser, the syntax description file and the parsed result. The parser is further broken down into the compiler, virtual machine code and the execution unit.



4.2 Extended Architecture

Above architecture can extend to embody more than one execution unit in order to perform multiple parsing operations in a cascaded fashion in a single General Parser. A single compiler can take multiple syntax description files and deliver individually compiled codes to each execution unit. The entire parser system then looks like the following diagram. This particular example contains two execution units however; there is no set limit for the number of execution units. It is governed only by the available computer resource, mainly by the available memory.



The dashed line arrow connecting the first execution unit to the next one is an internal bitstream pipeline. This pipeline transparently interconnects multiple execution units. It means that there is no distinction for the second execution unit whether its input stream is directly coming from a bitstream file or from another execution unit. The pipeline is not a simple sequential stream path. It's implementation is sophisticated enough to permit the second execution unit a random access into the input stream, without requesting the first execution unit finish processing its input entirely. This pipelining or filtering mechanism provides several valuable benefits. First, it allows multiple syntax description files to represent the layered syntax hierarchy, which is commonly employed in the communication standards. Each syntax description file corresponds to each of the layer in the hierarchy. Secondly, it provides users great flexibility in choosing the parsing levels by mixing and matching the different syntax layers. Thirdly, it provides users ease of debugging the syntax description by testing each layer individually instead of testing multi-layered operation at one time.

5 Bitstream Description Language (BDL)

This section discusses about the language for describing the bitstream syntax. The best way of defining a language that is easy to learn and understand is to borrow the features from an established popular language. We know a good example of this in the ISO MPEG standard specification, ISO/IEC 11172-2 and 13818-2. The standard borrows the syntactical notations from the C programming language. The table on the next page is an excerpt from ISO/IEC 13818-2, showing the bitstream syntax of a video sequence header. It hardly requires any additional explanation for you to precisely comprehend how a sequence header is organized from multiple fields of bits, as long as you have sufficient C language knowledge. The benefit of this method is now well recognized by many other standards, which have already adopted it.

I am going to take the same approach defining a language to describe the input bitstream syntax. This language, which is syntactically a subset of C, is now named the ‘Bitstream Description Language’ and will be referred to as BDL for short. BDL is very similar in the way ISO MPEG standard uses the C language as its base, except that it has a few more syntactical rules added to make it a real computer language, which can be compiled and executed.

The goals of BDL are 1) simple and easy to learn 2) capable enough to describe any unknown bitstream syntax and 3) relatively easy to implement its compiler. As a prototype system has been already built, the goal 1 and 3 are met. Whether the goal 2 is satisfied or not, we need to see the feedback from the real use experience.

Table 1 Sequnce Header in ISO/IEC 13818-2

sequence_header() {	**No. of bits**	**Mnemonic**
sequence_header_code	32	bslbf
Horizontal_size_value	12	uimsbf
vertical_size_value	12	uimsbf
aspect_ratio_information	4	uimsbf
frame_rate_code	4	uimsbf
bit_rate_value	18	uimsbf
marker_bit	1	bslbf
vbv_buffer_size_value	10	uimsbf
Constrained_parameters_flag	1	bslbf
load_intra_quantiser_matrix	1	uimsbf
if (load_intra_quantiser_matrix)		
intra_quantiser_matrix[64]	8*64	uimsbf
load_non_intra_quantiser_matrix	1	uimsbf
if (load_non_intra_quantiser_matrix)		
non_intra_quantiser_matrix[64]	8*64	uimsbf
next_start_code()		
}		

Here is an example BDL program that expresses a syntactically identical sequence header. Note that it is very similar to the ISO MPEG syntax presentation but has some additional information such as presentation format and enumeration item names. In general, writing a BDL program from such a standard is effortless.

Table 2 Sequence Header in BDL

```
void sequnce_header(void)
{
      field “0x%08x”, int sequence_header_code:32;
      field "%d" int horizontal_size_value:12;
      field "%d" int vertical_size_value:12;
      field "<%s> " enum {
            aspect_ratio_forbidden,
            aspect_ratio_undefined,
            aspect_ratio_3x4,
            aspect_ratio_9x16,
            aspect_ratio_1x221
      } aspect_ratio_information:4;
      field "<%s>" enum {
            frame_rate_forbidden,
            frame_rate_23_976,
            frame_rate_24_000,
            frame_rate_25_000,
            frame_rate_29_970,
            frame_rate_30_000,
            frame_rate_50_000,
            frame_rate_59_940,
```

```
            frame_rate_60_000
        } frame_rate_code:4;
        field "%d" int bit_rate_value:18;
        field "%d" int marker_bit:1;
        field "%d" int vbv_buffer_size_value:10;
        field "%d" int constrained_parameters_flag:1;
        field "%d" int load_intra_quantiser_matrix:1;
        if(load_intra_quantiser_matrix)
            field "%2d " int intra_quantiser_matrix[8][8]:8;
        field "%d" int load_non_intra_quantiser_matrix:1;
        if(load_non_intra_quantiser_matrix)
            field "%2d " int non_intra_quantiser_matrix[8][8]:8;
}
```

5.1 BDL Specification

Since BDL is a subset of C it is much easier to explain how it is different from C rather than explaining it without a use of any comparison. The following sections describe the BDL specification in comparison with the C language. Each section is based on a syntactical construct and is organized from the top-level construct to the lower level construct. Also to assure the technical clarity of the language specification and to eliminate any ambiguities, the beginning of the each section shows the syntactical construct in BNF (Backus Naur Form); this is a popular tool for describing computer languages. At the end of each section, some examples are presented to help you understand the actual use of the syntactical construct.

5.1.1 BDL Program

<bdl_program> ::= { <declaration> }

A bdl program is the top-level language symbol. It consists of repetition of declarations. The curly brace surrounding the declaration symbol indicates zero or more repetition of declaration symbols. In plain English, this means that a BDL program consists of a bunch of declarations. This is the same as C except that there is no consideration for external declarations in BDL since it does not support linking separately compiled objects. The entire BDL source program must be compiled at a single time.

Example:

```
int keep_going = 1;

void main(void)
{
    while(keep_going) {
        do_this();
    }
}

void do_this(void)
{
    if(looking_at_bits(0x000001, 24))
        keep_going = 0;
    else {
        field int byte:8;
    }
```

```
}
```

This program does not do anything significant. There are three declarations in this example program. One is the declaration of the global variable **keep_going**. Other two are the function **main** and the function **do_this**.

5.1.2 Declaration

<declaration> ::= <statement> | <function_definition>

A declaration is either a statement, see section 5.1.13 below, or a function definition, see section 5.1.3 below. This is unique to BDL and quite different from C language. C treats the function definitions as external declarations and declarations as pure declarations, which are variable declarations. By contrast, BDL treats pure declarations as a kind of statement for the sake of language simplification. So that global variable declarations are treated as a kind of statement. A side effect from this is that you can write some procedures by using statements outside of function definitions. Here, these procedures are named ‘global procedures’. The global procedures are executed even before the main function starts execution.

Example:

Same as the example in section 5.1.1 above

5.1.3 Function Definition

<function_definition> ::= <type_specifier> <literal_name>
(<parameter_list>) <compound_statement>

A BDL function definition form is greatly simplified compared to that of C function definition form. The main difference is that the function name directly follows the type specifier since BDL does not support pointers and structures. The return value from the function is either void, int or char. A BDL function serves as a unit of execution procedure just like a C function does, but also it is a logical grouping of bitstream syntax just like its use in ISO MPEG standard. When a BDL function executes, it parses a group of bits and optionally returns an integer value.

Example:

```
int add2(int x, int y)
{
    return x + y;
}
```

C still allows the old form, original K&R form, of function definition that does not have a function parameter list in the parentheses. By contrast, BDL does not allow this. A function must be declared in a new ANSI declaration form in which the parameter list must be placed in the pair of parentheses following the function name. Each of the parameter must be typed as well. There is no implicit ‘int’ assumption.

5.1.4 Type Specifier

<type_specifier> ::= void | int | char | <enum_specifier>

Bitstream syntax is generally designed for target hardware device to parse the stream in real time in actual application environment, in which floating-point operation is costly to implement and almost prohibitive. Thus, we can safely assume that the values we deal with do not include floating point numbers. The target hardware on which I am going to implement the parser is a popular 386 and above x86 processor PC. Therefore, numbers in BDL are limited to 32bit-integer type or 8bit-character type. BDL also supports enumeration type, which is simply a synonym to integer however, it does provide additional convenience in programming and a better interface when the parser presents the parsed result by the names instead of cryptic numbers. The void type is for syntactical restriction to detect coding errors. A compiler can make use of it to warn a programmer about inconsistencies in a program that some expressions relying on a return value from a void function.

Example:

```
int x;

enum {
      ONE = 1,
      TWO,
      THREE
} count;

void useless(void) {}
```

5.1.5 Literal Name

<literal_name> ::= [A-Za-z_][0-9A-Za-z_]*

A literal name is the name for functions, variables and enumeration identifiers. It can be any pattern that matches the above general expression except those reserved keywords. The meaning of the above general expression is “a series of characters which starts from an alphabet character or an underscore character, and followed by zero or more of alphabet characters, decimal number characters or underscore characters.” Due to the current simplification of the compiler implementation, the maximum length of the literal name is limited to 1024 characters.

Following is a list of reserved keywords that can not be used as literal names.

void, int, char, enum, vlc, field, input, output, goto, for, while, do, if, else, switch, case, default, continue, break, return

Example:

```
int this_is_a_variable_with_long_name_including_numbers_123;
```

5.1.6 Parameter List

<parameter_list> ::= void | <parameter> { , <parameter> }

A parameter list of a function definition is either a single void or a list of parameters separated by comma character.

Example:

```
int add3(int x, int y, int z)
{
    return x + y + z;
}
```

5.1.7 Parameter

<parameter> ::= <type_specifier> <literal_name>

A parameter is a literal name following a type specifier. The name of the parameter is bound to the name scope of the function definition. This is the same as C.

Example:

See section 5.1.6

5.1.8 Enumeration Specifier

<enum_specifier> ::=
enum '{' <enumerator_list> '}' |
enum vlc '{' <vlc_list> '}'

The enumeration specifier has two kinds of forms. One form is a simple enumeration that is similar to the C enumeration. The other one does not exist in C. It takes a post positioned qualifier 'vlc' following the keyword 'enum'. For the details of vlc, see the section 5.1.12 below. Each has different kind of list in the curly brackets. Unlike C, both forms do not support naming the enum type itself thus no type name is allowed between the keyword 'enum' or 'enum vlc' and the following list. The enum specifier works as a type so that you can declare variables based on this type. Unlike int type, enum can be declared without taking variable names. The variableless enum serves as a list of named constants. The enum declaration is bound to the inner most naming scope so is the C enum declaration.

Example:

```
enum {
    false,
    true
};
```

```
enum {
      ten = 10,
      one_hundred = 100,
      one_hundred_one
} special_numbers0, special_numbers1;

enum vlc {
      code0:"1",
      code1:"01",
      code2:"001",
      code3:"0001"
} simple_vlc;
```

5.1.9 Enumerator List

<enumerator_list> ::= <enumerator> { , <enumerator> }

An enumerator list is a list of enumerators, each one separated by a comma.

5.1.10 Enumerator

<enumerator> ::=
<literal_name> |
<literal_name> = <constant_expression> |
<literal_name> = <constant_expression> '..' <constant_expression> |
<literal_name> = '..' <constant_expression> |
<literal_name> = <constant_expression> '@' <constant_expression> |
<literal_name> = <constant_expression> '..' <constant_expression> '@' <constant_expression> |
<literal_name> = '..' <constant_expression> '@' <constant_expression>

An enumerator is a literal name or a literal name followed by an optional equal symbol and value specifier. It gives a name to a specific integer value. The default value of the enumerator starts from zero and increases by one within the enumerator list. When a constant expression is provided the enum value is the result value of the expression evaluation. If no value is provided, an increment of the previous value is assigned. This is the same as C enumerator definition. While the first two forms above are same as C enumerators, the rest are BDL specific forms. Those special forms are only meaningful in field declarations described in the section 5.1.18. The double dot symbol indicates a rage of integer between two numbers and including them. The value on the left side of the dots is the nominal value of the enumerator, meaning when the enumerator name is used in an expression it is substituted by this value. However, any numbers fall in this range is categorized as this enumerator. For example, the variable 'x' in a 4-bit field statement in the following example will be reported as GROUP0 when the value parsed from the bitstream is less than 8. Otherwise, it is reported as GROUP1. The values of GROUP0 and GROUP1, when they are used in an expression, are 0 and 8 respectively.

```
field enum { GROUP0 = 0..7, GROUP1 = ..15 } x:4;
```

The range can be specified in either ascending order or descending order and the nominal value is always on the left side of the double dot symbol. If it takes an ascending form,

the nominal value is the smallest integer in the range. In the descending order, the nominal value is the largest integer within the range.

The form with '@' symbol takes an optional masking value after the '@' symbol. The bits specified as 1 in the mask value are ignored in the comparison. For example the following two declarations are equivalent.

```
field enum { GROUP0 = 0..7, GROUP1 = ..15 } x:4;
field enum { GROUP0 = 0 @ 7, GROUP1 = 8 @ 7 } x:4;
```

5.1.11 VLC List

<vlc_list> ::= <vlc> { , <vlc> }

A vlc list is a list of vlcs, each separated by a comma. It defines a set of vlc codes.

5.1.12 VLC

<vlc> ::= <literal_name> : "CODE_PATTERN" [= <constant_expression>]

The name vlc stands for variable length code. This form gives a name to a specific vlc code pattern. The CODE_PATTERN is surrounded by a pair of double quotation marks. It is a string consisting of either character 0 or character 1, which specifies a binary code pattern. The value of vlc follows the rule of enumerator, which is assigned by the order in the list, and is not the binary value of the CODE_PATERN.

Example:

```
enum vlc {
      CODE_A : "1",
      CODE_B : "01",
      CODE_C : "001" = 3
      CODE_D : "0001"
} code_table;
```

This defines a variable **code_table** that takes one of four values; **CODE_A, CODE_B, CODE_C** or **CODE_D**. Their values are 0, 1, 3 and 4 respectively.

5.1.13 Statement

<statement> ::=
<compound_statement> |
<expression_statement> |
<selection_statement> |
<iteration_statement> |
<declaration_statement> |
<jump_statement> |
<label_statement>

A statement is one of the above listed statement groups.

5.1.14 Compound Statement

<compound_statement> ::= ‘{‘ { <statement> } ‘}’

A compound statement consists of zero or more statements surrounded by a pair of curly braces. A compound statement groups multiple statements and make them as if one statement syntactically. A compound statement also creates a new name space thus the local names can override the same names already defined outside of the compound statement. This behavior is same as C.

5.1.15 Expression Statement

<expression_statement> ::= ; | <expression> ;

The expression statement consists of an expression followed by a semicolon or just a semicolon.

5.1.16 Selection Statement

<selection_statement> ::=
if (<expression>) <statement> |
if (<expression>) <statement> else <statement> |
switch (<expression>) <statement>

The selection statement selects the flow of operation. Above three forms operates exactly same as defined in C.

5.1.17 Iteration Statement

<iteration_statement> ::=
while (<expression>) <statement> |
do <statement> while (<expression>) |
for ([<expression>] ; [<expression>] ; [<expression>]) statement

The iteration statement provides looping operation. All three forms operates exactly same as defined in C.

5.1.18 Declaration Statement

<declaration_statement> ::=
<type_specifier> { <declarator> } ; |
field [output] [“FORMAT”] <type_specifier>
<field_declarator> [: <constant_expression>] ; |
input <input_info> ; |
input ‘{‘ <input_info_list> ‘}’ |
output <variable_reference> : <constant_expression> ;

There are five forms in declaration statement. The first form is the most typical one, which consists of a type specifier followed by zero or more of declarators then terminated with a semicolon. The second form takes a keyword ‘field’, an optional keyword ‘output’ and an optional FORMAT string in front of a field declarator, see 5.1.21 below. In addition, the second form has an optional trailer consisting of colon and constant expression before the terminating semicolon. The last three forms, starting with keyword ‘input’ or ‘output’, will be discussed at the end of this section.

BDL removes C’s restriction on where the declarations are allowed. C prohibits the declarations anywhere but the beginning of a compound statement block or the beginning of a function body block. BDL allows declaration to be placed at anywhere a statement is allowed. In this aspect BDL resembles more like C++, however the following example, which is OK in C++, is illegal in BDL since ‘for’ does not take statements in the parentheses, see 5.1.17 above.

```
for(int i = 0; i < 10; i++) { ... }  /* this generates syntax error */
```

The first one of the five declaration forms declares standard variables. The declarator, described in a later section, maybe a scalar variable or an array variable with or without initialization expression. If the type specifier is enum, the following declarator may be omitted. In this case the enum identifiers serves as named constants within the name space of this declaration. The following examples all belong to the first form.

```
int x, y, z;
char c = ‘a’, d, n = ‘\n’;
int array[10], matrix[3][4];
enum {
      LOW, MID, HIGH
} level;
enum {
      FALSE = 0,
      TRUE = 1
};
```

While all the above declaration have the similar effect as the variable declaration in C, the form with ‘field’ keyword is unique in BDL. The keyword ‘field’ means that the declaring variable corresponds to a certain bit field of the input bitstream. It is a combination of variable declaration and automatic initialization of the variable with the value taken from the input bitstream. For int, char and enum types in the second form, the trailing colon and constant expression are mandatory because the value of the constant expression specifies the bit width of that field, which is necessary for bit extraction. When a BDL program executes, whenever the parser encounters the field statement it retrieves the number of specified bits from the bitstream and advance the current read location in the stream by that amount. The retrieved value is stored in the declared variable. If a negative bit width is specified, that field is considered as little endian field, otherwise all regular fields are treated as big endian field. If the type is enum vlc the colon and constant expression are prohibited because the width of the variable length code is undetermined until the parser sees the actual bitstream. When parser executes the enum vlc declaration, it tries to match one of the codes defined in that vlc declaration and extracts the matched code from the bitstream. Note that the value stored in the declared

variable is not the extracted vlc code itself but the enum value that is determined by the order in the declaration, starting from 0 and incrementing by 1. This is because a scalar vlc code value has no significance to identify itself, for example codes “001” and “0001” are valid and different vlc codes, but both shares an identical code value 1. While parsing vlc, if none of the code matches the bit pattern at the current location, the program terminates with a vlc decode error. The FORMAT is a string surrounded by a pair of double quotations. It specifies the presentation format of that field. If there is no FORMAT specified the bitstream field is extracted and the variable is initialized with that value but nothing is presented to the user. The definition of the format follows that of the C standard library function ‘printf’. For example, if you wish to present an integer value surrounded by a pair of parenthesis use a format string like “(%d)”. The available format characters for each type are listed in the Table 3 Format Characters and their Order below. You may specify as little as zero format-characters within the entire format string but if you use them they must match the kind and the order listed in the table.

Table 3 Format Characters and their Order

Type	Max No. of values	Format Characters (must follow a ‘%’ character to be effective) in the format string	Note
int	1	**[d | c | x | X]**	an integer value
enum	2	**[s [d | c | x | X]]**	first string is the enum entry identifier name, optional second integer value is the enum value
enum vlc	4	**[s [d | c | x | X [d | c | x | X [d | c | x | X]]]]**	first string is the enum entry identifier name, optional second integer is the enum value, optional third integer value is the vlc code value, optional fourth integer value is the code bit width

Here is an example of some filed declaration.

```
field "%d" int horizontal_size_value:12;
field "<%s> (%d)" enum {
   aspect_ratio_forbidden,
   aspect_ratio_undefined,
   aspect_ratio_3x4,
   aspect_ratio_9x16,
   aspect_ratio_1x221
  } aspect_ratio_information:4;
field "%2d " int intra_quantiser_matrix[8][8]:8;
```

The following two examples compare big endian (normal) field and little endian field.

field int big:12;

first		big:12												last	
x	x	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	x	x

field int little:-12;

first		little:-12												last	
x	x	b7	b6	b5	b4	b3	b2	b1	b0	b11	b10	b9	b8	x	x

The form with ‘input’ keyword serves for interactive parameter passing. When the parser encounters the input declaration it asks user to provide a value and the declared variable is initialized with the user provided value. There are two forms for input declaration, one with single input information and the other with multiple input information in a list. Under GUI operation each input statement generates a single dialog interaction, thus multiple input statements with a single input information for each, and a single input statement with multiple input information as a list makes difference. The former presents many dialog boxes, one for each input statement while the latter presents a single dialog box that contains all input information in the list. Here is an example of input statements. Each input information must provide a initialization value, which is provided as a default value in the user interaction.

```
input “How many packets?” int number_of_packet = 0;
input {
      “coordinate X:” x  = 0;
      “coordinate Y:” y = 0;
}
```

Unlike other declaration forms, the form with ‘output’ keyword does not really declare variables. It is categorized as one of the declaration statement only from its syntactical similarity point of view. What it actually does is send the content value of the variable to the output stream, which goes into an inter-process communication pipeline. For this purpose the colon proceeded constant expression must be provided in order to specify the bit width of the output field. Obviously, the variable in output statement must be declared before hand. Here is an example of output statements.

```
output x:8;
output matrix[8]:8;
```

5.1.19 Input Information

<input_info> ::= “PROMPT” <type_specifier> <init_declarator>

Input information consists of prompting string, type specifier and initialized declarator. The PROMPT is a string presented to the user at the time of inquiry.

5.1.20 Input Information List

<input_info_list> ::=
<input_info> ; |
<input_info_list> <input_info> ;

Input information list is a one or more input information each terminated with a semicolon.

5.1.21 Field Declarator

<field_declarator> ::=
<direct_declarator> |

<direct_declarator> ‘[‘ <constant_expression> : <expression> ‘]’

A field declarator takes a form of direct declarator (see 5.1.24 below) or a direct declarator followed by a special array dimension. It is a kind of direct declarator, which can take an extra expression in the last element of the array size specifier. The role of the constant expression is the same as regular dimension specification that defines the size of the array. The expression following the colon character specifies how many elements of the declared space in this dimension to be filled when the bits are extracted from the bitstream. For example, the following field declaration reserves 3 by 5 character array space then fills the 3 by 4 space with 12 of 8bit data extracted from the bitstream. Unfilled space, array[n][4] in this case, are guaranteed to be cleared to zero.

Example:

```
field array[3][5:4]:8;
```

5.1.22 Declarator

<declarator> ::= <init_declarator> | <direct_declarator>

A declarator is either an initialized declarator or a direct declarator.

5.1.23 Initialized Declarator

<init_declarator> ::= <literal_name> = <assignment_expression>

An initialized declarator consists of a literal name followed by an equal sign and assignment expression.

Example:

```
int a = 3;
int b = c + 4;
```

5.1.24 Direct Declarator

<direct_declarator> ::=
<literal_name> |
<direct_declarator> ‘[‘ <constant_expression> ‘]’

A direct declarator consists of simply a literal name or a literal name followed by any number of array dimension specifications. Since array form is not in the initialized declarator, BDL does not provide array initialization.

Example:

```
int a;
int array[3][4][5];
```

5.1.25 Jump Statement

<jump_statement> ::=
continue ; |
break ; |
return [<expression>] ; |
goto <literal_name> ;

The jump statement is one of the four forms listed above. They all follow the same rule and behave the same as C counterparts.

5.1.26 Label Statement

<label_statement> ::=
case <constant_expression> : <statement> |
default : <statement> |
<literal_name> : <statement>

The label statement is one of the above three forms. They are compatible with C counterparts.

5.1.27 Expression

<expression> ::= <assignment_expression> | <expression> , <assignment_expression>

An expression is either a single assignment expression or multiple assignment expressions concatenated with a comma between them. This definition of expression is identical with that of C. The comma works as a comma operator.

5.1.28 Assignment Expression

<assignment_expression> ::=
<conditional_expression> |
<variable_reference> <assignment_operator> <assignment_expression>

An assignment expression is either a conditional expression or an assignment, which right hand value is also an assignment expression. This definition is identical with the assignment expression definition in C.

5.1.29 Conditional Expression

<conditional_expression> ::=
<binary_expression> |
<binary_expression> ? <expression> : <conditional_expression>

A conditional expression can be a binary expression or a combination of binary expression, expression and conditional expression concatenated with ‘?:’ ternary operator. This definition is same as C.

5.1.30 Variable Reference

<variable_reference> ::=
<literal_name> |
<variable_reference> '[' <expression> ']'

A variable reference can take two forms. One form is a simple literal name. This is to refer to a simple variable. The other is an indexed array element form, which provides a reference to an array element. The level of indirection must match the declared one. This is different from C. Since BDL does not support pointer types, unmatched level of indirection introduces an unknown type.

Example:

```
int a = 1;
int b[3][4];
int c = b[a][a + 1];
a = b[0];
```

The last line is an error because the level of indirection of b[0] does not match the declared one.

5.1.31 Assignment Operator

<assignment_operator> ::=
= | += | -= | *= | /= | %= | |= | ^= | &= | <<= | >>=

Their definitions are identical to the C counterparts. The precedence of the operators including assignment operators are defined the same as C.

5.1.32 Binary Expression

<binary_expression> ::=
<unary_expression> |
<binary_expression> <binary_operator> <binary_expression>

A binary expression is either a unary expression or two binary expressions concatenated with a binary operator.

5.1.33 Binary Operator

<binary_operator> ::= * | / | % | + | - | << | >> | < | <= | > | >= | == | != | & | ^ | '|' | && | '||'

The definition of binary operators including their precedence is the same as C.

5.1.34 Unary Expression

<unary_expression> ::=
<postfix_expression> |
++ <variable_reference> |
-- <variable_reference> |
+ <unary_expression> |

- <unary_expression> |
~ <unary_expression> |
! <unary_expression>

An unary expression is either a postfix expression, a pre-increment/pre-decrement operator followed by a variable reference or a unary operator followed by a unary expression. This is the same as C except sizeof operation and address operations ‘&’ and ‘*’.

5.1.35 Postfix Expression

<postfix_expression> ::=
<primary_expression> |
<function_call> |
<variable_reference> |
<variable_reference> ++ |
<variable_reference> --

A postfix expression is a primary expression, a function call, a variable reference or a variable reference followed by a post-increment/post-decrement operator. This definition is the same as C except BDL does not have pointer postfix operators ‘.’ and ‘->’.

5.1.36 Primary Expression

<primary_expression> ::=
<constant> |
<string_constant> |
(<expression>)

A primary expression is a constant, string or an expression surrounded by a pair of parentheses, same as C.

5.1.37 Constant

<constant> ::=
0b[01][01]* |
0o[0-7][0-7]* |
0[0-7][0-7]* |
0x[0-9A-Fa-f][0-9A-Fa-f]* |
[0-9][0-9]* |
\'(\\.|[^\’])*\'

The constant is an integer value expressed in the above general expression forms. The first one is a binary constant. The second and the third one are octal constants. The fourth one is a hexadecimal constant. The fifth one is a decimal constant. The last form is a character constant. A character string surrounded by a pair of single quote characters is treated as a character constant, which value is the ASCII code of the first character. All standard character escape sequences defined in C, \n, \t, \v, \b, \r, \f, \a, \\, \?, \’, \”, \ooo and \xhh are available.

5.1.38 String Constant

<string_constant> ::= \"(\\.|[^\\\"])*\"

A string constant is a string of characters surrounded by a pair of double quote characters. Its value is the memory address where the first character is stored. Since BDL does not have a pointer type, the value is treated as an integer. All standard character escape sequences defined in C, \n, \t, \v, \b, \r, \f, \a, \\, \?, \', \", \ooo and \xhh are available. A string can contain negligible new lines by placing '\' character at the end of line. The new line followed by the '\' is ignored. In addition, multiple string constants separated by white spaces are concatenated as a single string constant, as is same in C.

5.1.39 Control Line

#include "FILE_NAME" |
#include <FILE_NAME>

There is only one kind of control line in BDL. The "#include" inserts the contents of the specified file into this location. It works exactly same as C pre-processor's "#include". There is no distinction between two forms. In both cases, the include-file is expected to be in the current directory of the parser.

6 Virtual Machine (VM)

The virtual machine (VM) is the engine that parses the bitstream according to the given syntax rule, however the syntax rule VM sees is no more in BDL form. The original BDL program is now compiled into a set of primitive instructions. Execution of the primitive instructions provides the VM a great runtime performance compared to interpreting BDL.

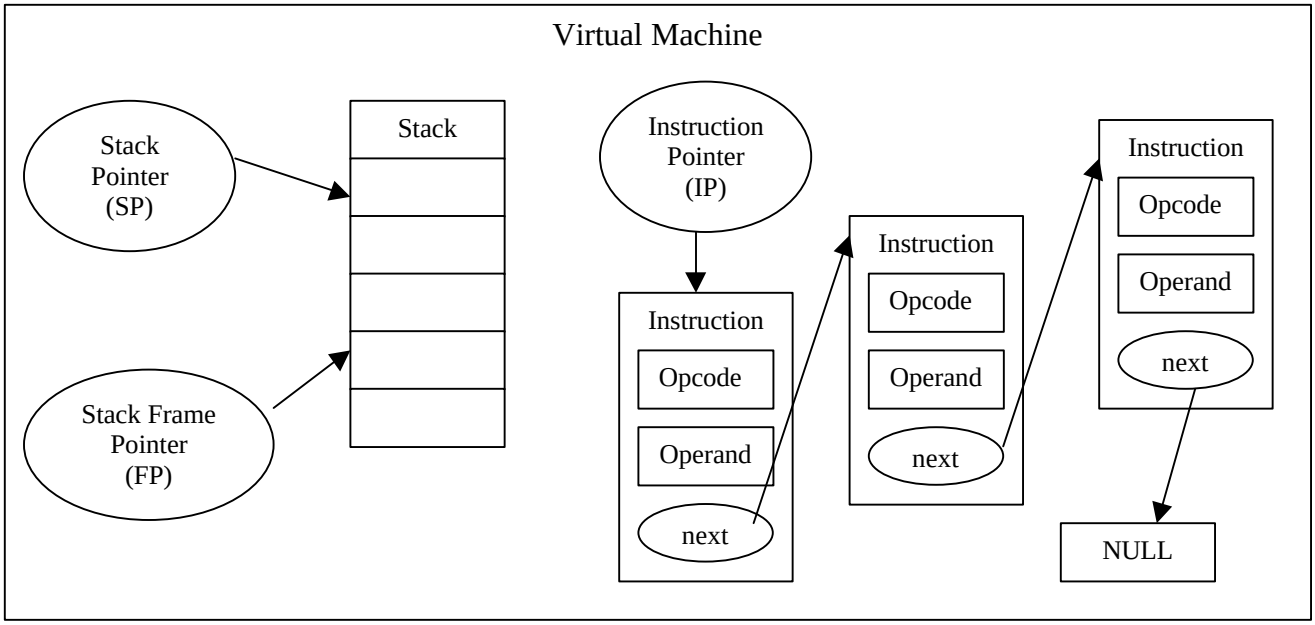
6.1 VM Architecture

The VM design is based on two major considerations. One, it must be fast in execution. Two, it must be designed to help simplifying both engine itself and the compiler implementation. For the fast execution it has small number of instructions and has only a few addressing modes. To help simplify the compiler design the engine is based on a register-less stack machine. It has no general-purpose register file at all. It has only three special purpose pointers, an instruction pointer (IP), a stack pointer (SP) and a stack frame pointer (FP). All the temporary values in computation are kept in the stack storage space.

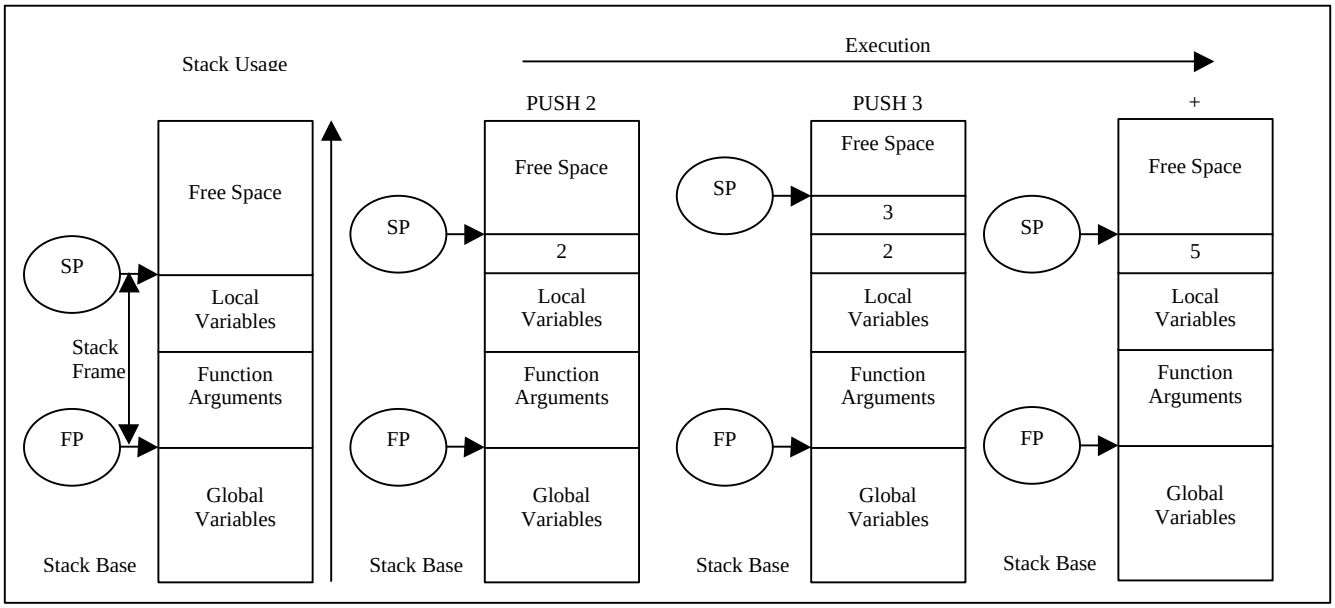
There is no intention of building this VM in hardware so there are a few decisions made that may not be suitable for hardware implementation. For example, the compiled instructions are not laid out in a linear memory space as all the real processors do. They are rather scattered around in memory but as a whole, they form a single linked list. This eliminates the need for compiler to keep many temporary data before generating the final codes and removes the cumbersome address resolving computation.

A few limited operations that require the fastest performance are implemented as a set of native functions. These operations include bit pattern search and bit retrieval services. A BDL program can call these functions just as if other functions defined in the BDL program. For this purpose, the VM provides a thunking mechanism between BDL function call and native function call.

The following diagram shows the VM internals.



VM keeps all data in stack storage. There is no heap thus even global variables are allocated in the stack space. VM keeps track of stack usage with two pointers, stack pointer (SP) and stack frame pointer (FP). The left most picture in the next diagram illustrates the stack usage right after it enters a function. All the global variables are allocated at the bottom of the stack. The beginning of the stack frame is pointed by FP and the function arguments are allocated at the bottom of the stack frame. The local variables are allocated after the function arguments. Finally, the SP points at the beginning of the free stack space after the local variables. As the function executes the stack space is occupied and released dynamically. The diagram shows how an expression "2+3" evaluates on the stack. When function call nests, a new stack frame is created from the current SP location as new stack frame sits on top of the previous one.



6.2 Virtual Machine Instruction Set (Opcode)

Each instruction handler is implemented as a small C function. An 8-bit opcode is assigned to each instruction. For the fastest execution, VM uses table-lookup to map an opcode to a handler function. It has two groups of table for this purpose. Here is how a single execution cycle is performed. 1) Get the opcode from the instruction that is pointed by the IP. 2) Select a table for that opcode. 3) Look up the function from the table with the opcode as an index to the table. 4) Execute that function.

Opcode is grouped into two groups, non-load/store group and load/store group. The next table shows the bit allocation for each group of opcode.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
non-load/store group	0	0:Relative 1:Absolute	Identifier					
load-store group	1	0:Relative 1:Absolute	0:Direct 1:Indirect	0:Int 1:Char	Identifier			

The following table shows the currently implemented VM instruction set.

opcode	mnemonic	operation
0x00	HALT	halts VM execution
0x01	NOP	no operation
0x02	PUSH	*stack_ptr++ = operand;
0x03	POP	stack_ptr--;
0x04	DUP	*stack_ptr = *(stack_ptr – 1); stack_ptr++;
0x05	RESERVE	stack_ptr += operand
0x06	PUSHFRM	creates new stack frame
0x07	POPFRM	restore previous stack frame
0x08	PUSHSCP	reserved
0x09	POPSCP	reserved
0x0a	LOGICAL_OR	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) || *stack_ptr;
0x0b	LOGICAL_AND	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) && *stack_ptr;
0x0c	OR	--stack_ptr; *(stack_ptr - 1) |= *stack_ptr;
0x0d	XOR	--stack_ptr; *(stack_ptr - 1) ^= *stack_ptr;
0x0e	AND	--stack_ptr; *(stack_ptr - 1) &= *stack_ptr;
0x0f	EQUAL	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) == *stack_ptr;
0x10	NOT_EQUAL	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) != *stack_ptr;
0x11	LT	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) < *stack_ptr;
0x12	LE	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) <= *stack_ptr;
0x13	GT	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) > *stack_ptr;
0x14	GE	--stack_ptr; *(stack_ptr - 1) = *(stack_ptr - 1) >= *stack_ptr;
0x15	SL	--stack_ptr; *(stack_ptr - 1) <<= *stack_ptr;
0x16	SR	--stack_ptr; *(stack_ptr - 1) >>= *stack_ptr;
0x17	ADD	--stack_ptr; *(stack_ptr - 1) += *stack_ptr;
0x18	SUB	--stack_ptr; *(stack_ptr - 1) -= *stack_ptr;
0x19	MUL	--stack_ptr; *(stack_ptr - 1) *= *stack_ptr;
0x1a	DIV	--stack_ptr; *(stack_ptr - 1) /= *stack_ptr;
0x1b	MOD	--stack_ptr; *(stack_ptr - 1) %= *stack_ptr;
0x1c	2COMP	*(stack_ptr - 1) = -*(stack_ptr - 1);
0x1d	1COMP	*(stack_ptr - 1) = ~*(stack_ptr - 1);
0x1e	NOT	*(stack_ptr - 1) = !*(stack_ptr - 1);

0x1f	JMP	instruction_ptr = operand;
0x20	JZ	if(*--stack_ptr == 0) instruction_ptr = operand;
0x21	JNZ	if(*--stack_ptr != 0) instruction_ptr = operand;
0x22	JSR	call subroutine
0x23	RET	return subroutine
0x24	SUBR	call native function
0x25	FIELD	read bit field
0x26	INPUT	read input
0x27	OUTPUT	write output
0x28	VM2NATV	convert VM address to native
0x80	LD	*stack_ptr++ = stack[operand];
0x81	ST	stack[operand] = *(stack_ptr - 1);
0x82	ADD_ST	*(stack_ptr – 1) = stack[operand] += *(stack_ptr – 1);
0x83	SUB_ST	*(stack_ptr – 1) = stack[operand] -= *(stack_ptr – 1);
0x84	MUL_ST	*(stack_ptr – 1) = stack[operand] *= *(stack_ptr – 1);
0x85	DIV_ST	*(stack_ptr – 1) = stack[operand] /= *(stack_ptr – 1);
0x86	MOD_ST	*(stack_ptr – 1) = stack[operand] %= *(stack_ptr – 1);
0x87	OR_ST	*(stack_ptr – 1) = stack[operand] |= *(stack_ptr – 1);
0x88	XOR_ST	*(stack_ptr – 1) = stack[operand] ^= *(stack_ptr – 1);
0x89	AND_ST	*(stack_ptr – 1) = stack[operand] &= *(stack_ptr – 1);
0x8a	SL_ST	*(stack_ptr – 1) = stack[operand] <<= *(stack_ptr – 1);
0x8b	SR_ST	*(stack_ptr – 1) = stack[operand] >>= *(stack_ptr – 1);
0x8c	PRE_INC	*stack_ptr++ = ++stack[operand];
0x8d	PRE_DEC	*stack_ptr++ = --stack[operand];
0x8e	POST_INC	*stack_ptr++ = stack[operand]++;
0x8f	POST_DEC	*stack_ptr++ = stack[operand]--;

7 Built-in Functions

There are some operations that:

(1) execute at extreme frequency in the course of regular parsing process and are computationally expensive to perform in BDL.
(2) involves low-level system interaction.
(3) map directly to native library functions that are not trivial to implement.

These operations are provided in the form of built-in functions, which are compiled in native instruction set. By doing so, the type (1) operation gain a great runtime performance. There is simply no way to write type (2) functions in BDL alone. Type (3) functions are too complicated to ask users to implement. The built-in functions can be regarded as pre-defined library functions. That means BDL programs can freely make calls to built-in functions in the same way they call BDL functions, without a need of declaring anything for built-in functions. Following is a table of built-in functions.

function name	type	description
search_word	(1)	Advances the input stream read pointer until it encounters a specified word pattern
look_at_bits	(1)	Peek at a specified series of bits at a specified location
looking_at_bits	(1)	Test if a specified pattern of bits exists at a specified location
tell_bytes	(1),(2)	Get the current byte location within the input stream
tell_bits	(1),(2)	Get the current bit location within the input stream
exit	(2)	Finish the program execution
printf	(2),(3)	Send a formatted character string to the console
sprintf	(2),(3)	Store a formatted character string into a specified memory location

The rest of the subsections in this section describe each of the built-in functions in depth.

7.1 search_word

DEFINITION

```
int search_word( int word_pattern,
                 int word_mask = 0);
```

DESCRIPTION

Search for word_pattern from the current bit location in the input stream toward the end of the stream. Word_pattern is a 32bit value. First it discards the unaligned bits then the search operation starts from the nearest byte aligned location in byte by byte manner. When it finishes the search the current bit location is set to the beginning of the matched pattern which is always byte aligned.

ARGUMENTS

int word_pattern — A 32bit pattern to look for

int word_mask = 0	An optional mask bits. The bits set to one in this mask do not contribute to the matching criterion.

RETURN VALUE

The matched word. It may not necessarily be same as the word_pattern value depending on the word_mask. Weh word_mask is 0 the return value is identical to the word_pattern.

NOTE

When it reaches the end of the input stream before finding the specified pattern, the program will exit with a bitstream-exhausted exception.

7.2 look_at_bits

DEFINITION

```
int look_at_bits( int bit_length,
                  int bit_offset = 0);
```

DESCRIPTION

Retrieves specified number of bits from the specified location. It does not affect the current bit location in the stream.

ARGUMENTS

int bit_length	The number of bits to retrieve. Must be 32 or less.
int bit_offset = 0	An optional bit location offset specifying at where the retrieval should take place. This is a relative offset from the current location thus 0 means the bit retrieval is performed from the current location. Positive number specifies bit offset toward the end of the stream. Negative number specifies bit offset toward the beginning of the stream.

RETURN VALUE

Retrieved value

NOTE

When it reaches the end of stream before acquiring the specified bits the program will exit with a bitstream-exhausted exception.

7.3 looking_at_bits

DEFINITION

```
int looking_at_bits(int bit_pattern,
                    int bit_length,
                    int bit_mask = 0,
                    int bit_offset = 0);
```

DESCRIPTION

Test if a pattern of bits are seen at the specified bit location. It does not affect the current bit location in the stream.

ARGUMENTS

int bit_pattern	A maximum 32bit long bit pattern to test against the stream.
int bit_length	The length of bit_pattern. Must be 32 or less. If the length is less than 32, both bit_pattern and bit_mask are treated as LSB justified.
int bit_mask = 0	An optional mask bits. The bits set to one in this mask do not contribute to the maching criterion.
int bit_offset = 0	An optional bit location offset specifying at where the matching operation should take place. This is a relative offset from the current location thus 0 means the matching operation is performed from the current location. Positive number specifies bit offset toward the end of the stream. Negative number specifies bit offset toward the beginning of the stream.

RETURN VALUE

1 if the pattern matches. Otherwise 0.

NOTE

When it reaches the end of stream before finding the specified pattern the program will exit with a bitstream-exhausted exception.

7.4 tell_bytes

DEFINITION

```
int tell_bytes(void);
```

DESCRIPTION

Gets the current read position within the input stream.

ARGUMENTS

none

RETURN VALUE

The number of bytes at the current stream read position. The very first byte in the stream is counted as zero. The definition of the current position is the byte location, which is not yet read but to be read at the next read.

NOTE

Since the return value is a 32bit signed integer, the return value is not correct if the position exceeds 2.15G byte location.

7.5 tell_bits

DEFINITION

```
int tell_bits(void);
```

DESCRIPTION

Gets the current read position within the input stream.

ARGUMENTS

none

RETURN VALUE

The number of bits at the current stream read position. The very first bit int the stream is counted as zero. The definition of the current position is the bit location, which is not yeat read but to be read at the next read.

NOTE

Since the return value is a 32bit signed integer, the rerun value is not correct if the position exceeds 2.15G bit (268M byte) location.

7.6 exit

DEFINITION

```
void exit(int status = 0);
```

DESCRIPTION

Terminates the program execution and starts clean-up procedure.

ARGUMENTS

int status = 0	An optional return status.

RETURN VALUE

none

NOTE

This function will not return to the caller.

7.7 printf

DEFINITION

int printf(const char *format, ...);

DESCRIPTION

Performs exactly same as 'printf' function in a C standard library. The output is displayed in the console are.

ARGUMENTS

const char *format	A constant string that specifies the printing format.

Any number of optional arguments can follow. It is the programmer's responsibility to match the arguments with the format specification.

RETURN VALUE

The number of format units.

NOTE

Format specifiers other than integer, character and string are meaningless since the system does not support floating point numbers.

7.8 sprintf

DEFINITION

int sprintf(char *str, const char *format, ...);

DESCRIPTION

Performs exactly same as 'sprintf' function in a C standard library.

ARGUMENTS

char *str	An address from where the formatted string will be stored.
const char *format	A constant string that specifies the printing format.

Any number of optional arguments can follow. It is the programmer's responsibility to match the arguments with the format specification.

RETURN VALUE

The number of format units.

NOTE

Format specifiers other than integer, character and string are meaningless since the system does not support floating point numbers.

8 Prototype General Parser

A prototype General Parser has been built based on the design described in this paper. This prototype fully implements all the specifications we have discussed here. However, the purpose of building this prototype was a proof of concept; to see if it really works as designed on the paper. Therefor it is not the best implementation as a final product. Although currently there is no GUI interface, the code is well structured in a modular fashion so that it is prepared to be placed in a different interface environment.

The prototype General Parser, gp.exe, is built as a Win32 command line application. It takes BDL file names and input bitstream file name as command line arguments and produces the parsed result out to the standard output. Here is the available command switches and command line syntax.

Usage: gp [-c] [-d] [-v] [<bdl_source_file>...] [-b <bitstream_file>]

- -c code generation only and no execution
- -d output debug information
- -v get version information
- -b specifies the bitstream file

9 Sample Code and Execution Output

9.1 Sample 1

The next example program does not perform any parsing activity. It is just small enough to provide easy comparison between BDL source program and the compiled instructions.

```
/* sample program – t.c - */

int fact(int n)
{
  if(n > 0)
    return n * fact(n - 1);
  return 1;
}

void main(void)
{
  int i;
  for(i = 0; i < 10; i++) {
    printf("fact(%d) = %d\n", i, fact(i));
  }
}
```

```
D:\ota\project\bison\gp>gp -c -d t.c
compiling [t.c]...
pass1 ...
pass2 ...
linking ...
t.c:1:          00000000        RESERVE 0
                00000001        PUSHFRM
                00000002        RESERVE 4
                00000003        JSR     00000016        ; t.c:13:               ; main()
                00000004        POPFRM
                00000005        POP
                00000006        HALT
t.c:5:          00000007        LD      (+0)            ; ===== fact() =====
                00000008        PUSH    0               ; CONSTANT
                00000009        >
                0000000a        JZ      00000014        ; t.c:7:
t.c:6:          0000000b        LD      (+0)            ; var 'n'
                0000000c        PUSHFRM
                0000000d        LD      (+0)            ; var 'n'
                0000000e        PUSH    1               ; CONSTANT
                0000000f        -
                00000010        JSR     00000007        ; t.c:5:                ; fact()
                00000011        POPFRM
                00000012        *
                00000013        RET
t.c:7:          00000014        PUSH    1               ; CONSTANT
                00000015        RET
t.c:13:         00000016        PUSH    0               ; ===== main() =====
                00000017        ST      (+0)            ; var 'i'
                00000018        POP
                00000019        LD      (+0)            ; var 'i'
                0000001a        PUSH    10              ; CONSTANT
                0000001b        <
                0000001c        JZ      0000002a        ; t.c:16:
t.c:14:         0000001d        PUSHFRM
                0000001e        PUSH    7941312 ;       "fact(%d) = %d
"
                0000001f        LD      (+0)            ; var 'i'
                00000020        PUSHFRM
                00000021        LD      (+0)            ; var 'i'
                00000022        JSR     00000007        ; t.c:5:                ; fact()
                00000023        POPFRM
                00000024        SUBR    040c920         ; native function printf()
                00000025        POPFRM
                00000026        POP
t.c:13:         00000027        ()++    (+0)            ; var 'i'
                00000028        POP
                00000029        JMP     00000019        ; t.c:13:
t.c:16:         0000002a        RET
code generated successfully
```

9.2 Sample 2

This example is really too long to show all. Things after the sequence_header function are omitted. It parses MPEG2 video elementary stream. After this list is the example parsed result.

```
enum {
  FALSE,
  TRUE
};

enum {
  START_CODE                = 0x00000100,
  PICTURE_START_CODE        = 0x00000100,
  USER_DATA_START_CODE      = 0x000001b2,
  SEQUENCE_START_CODE       = 0x000001b3,
  SEQUENCE_ERROR_START_CODE = 0x000001b4,
  EXTENSION_START_CODE      = 0x000001b5,
  SEQUENCE_END_CODE         = 0x000001b7,
  GROUP_START_CODE          = 0x000001b8
};

int number_of_frame_center_offsets = 0;

void main(void)
{
  while(1)
    video_sequence();
}

void video_sequence(void)
{
  search_word(SEQUENCE_START_CODE);
  sequence_header();
  if(search_word(START_CODE, 0xff) == EXTENSION_START_CODE) {
    sequence_extension();
    do {
      extension_and_user_data(0);
      do {
        if(search_word(START_CODE, 0xff), looking_at_bits(GROUP_START_CODE, 32)) {
          group_of_pictures_header();
          extension_and_user_data(1);
        }
        picture_header();
        picture_coding_extension();
        extension_and_user_data(2);
        picture_data();
      } while(search_word(START_CODE, 0xff),
              looking_at_bits(PICTURE_START_CODE, 32) ||
              looking_at_bits(GROUP_START_CODE, 32));
      if(search_word(START_CODE, 0xff), !looking_at_bits(SEQUENCE_END_CODE, 32)) {
        sequence_header();
        sequence_extension();
      }
    } while(search_word(START_CODE, 0xff),
            !looking_at_bits(SEQUENCE_END_CODE, 32));
  } else {
    /* ISO/IEC 11172-2 */
  }
}

void sequence_header(void)
{
  field "0x%08x" int sequence_header_code:32;
  field "%d" int horizontal_size_value:12;
  field "%d" int vertical_size_value:12;
  field "<%s> (%d)" enum {
    aspect_ratio_forbidden,
```

```
    aspect_ratio_undefined,
    aspect_ratio_3x4,
    aspect_ratio_9x16,
    aspect_ratio_1x221
  } aspect_ratio_information:4;
  field "<%s> (%d)" enum {
    frame_rate_forbidden,
    frame_rate_23_976,
    frame_rate_24_000,
    frame_rate_25_000,
    frame_rate_29_970,
    frame_rate_30_000,
    frame_rate_50_000,
    frame_rate_59_940,
    frame_rate_60_000
  } frame_rate_code:4;
  field "%d" int bit_rate_value:18;
  field "%d" int marker_bit:1;
  field "%d" int vbv_buffer_size_value:10;
  field "%d" int constrained_parameters_flag:1;
  field "%d" int load_intra_quantiser_matrix:1;
  if(load_intra_quantiser_matrix)
    field "%2d " int intra_quantiser_matrix[8][8]:8;
  field "%d" int load_non_intra_quantiser_matrix:1;
  if(load_non_intra_quantiser_matrix)
    field "%2d " int non_intra_quantiser_matrix[8][8]:8;
}
```

Here is the beginning part of the execution result.

```
D:\ota\project\bison\gp>gp test.bdl -b sony2.mpg
compiling [test.bdl]...
pass1 ...
pass2 ...
linking ...
code generated successfully
execution starts at ...       Fri Aug 20 14:17:43 1999
[main]
  [video_sequence]
    [sequence_header]
      sequence_header_code:32                          : 0x000001b3
      horizontal_size_value:12                         : 704
      vertical_size_value:12                           : 480
      aspect_ratio_information:4                       : <aspect_ratio_3x4> (2)
      frame_rate_code:4                                : <frame_rate_30_000> (5)
      bit_rate_value:18                                : 37500
      marker_bit:1                                     : 1
      vbv_buffer_size_value:10                         : 112
      constrained_parameters_flag:1                    : 0
      load_intra_quantiser_matrix:1                    : 0
      load_non_intra_quantiser_matrix:1                : 0
    [sequence_extension]
      extension_start_code:32                          : 0x000001b5
      extension_start_code_identifier:4                : 1
      profile_and_level_escape:1                       : 0
      profile_identification:3                         : <Main_Profile> (4)
      level_identification:4                           : <Main_Level> (8)
      progressive_sequence:1                           : 0
      chroma_format:2                                  : <_4_2_0> (1)
      horizontal_size_extension:2                      : 0
      vertical_size_extension:2                        : 0
      bit_rate_extension:12                            : 0
      marker_bit:1                                     : 1
      vbv_buffer_size_extension:8                      : 0
      low_delay:1                                      : 0
      frame_rate_extension_n:2                         : 0
      frame_rate_extension_d:5                         : 0
    [extension_and_user_data]
```

```
    [extension_data]
      extension_start_code:32                          : 0x000001b5
      [sequence_display_extension]
        extension_start_code_identifier:4                : 2
        video_format:3                                   : 0
        colour_description:1                             : 0
        display_horizontal_size:14                       : 120
        marker_bit:1                                     : 1
        display_vertical_size:14                         : 120
    [user_data]
      user_data_start_code:32                          : 0x000001b2
      user_data:8                                      : 0x01
      user_data:8                                      : 0x02
      user_data:8                                      : 0x03
      user_data:8                                      : 0x04
      user_data:8                                      : 0x05
      user_data:8                                      : 0x06
      user_data:8                                      : 0x07
      user_data:8                                      : 0x08
      user_data:8                                      : 0x09
      user_data:8                                      : 0x0a
  [group_of_pictures_header]
    group_start_code:32                              : 0x000001b8
    time_code:25                                     : 0x00001000
    closed_gop:1                                     : 1
    broken_link:1                                    : 0
  [extension_and_user_data]
    [user_data]
      user_data_start_code:32                          : 0x000001b2
      user_data:8                                      : 0x0b
      user_data:8                                      : 0x0c
      user_data:8                                      : 0x0d
      user_data:8                                      : 0x0e
      user_data:8                                      : 0x0f
      user_data:8                                      : 0x10
      user_data:8                                      : 0x11
      user_data:8                                      : 0x12
      user_data:8                                      : 0x13
      user_data:8                                      : 0x14
  [picture_header]
    picture_start_code:32                            : 0x00000100
    temporal_reference:10                            : 2
    picture_coding_type:3                            : <I> (1)
    vbv_delay:16                                     : 0x1c3c
    extra_bit_picture:1                              : 0
  [picture_coding_extension]
    extension_start_code:32                          : 0x000001b5
    extension_start_code_identifier:4                : 8
    f_code[2][2]:4                                   :  4  4
                                                     : 15 15
                                                     :
    intra_dc_precision:2                             : 0
    picture_structure:2                              : <Bottom_Field> (2)
    top_field_first:1                                : 0
    frame_pred_frame_dct:1                           : 0
    concealment_motion_vectors:1                     : 1
    q_scale_type:1                                   : 1
    intra_vlc_format:1                               : 0
    alternate_scan:1                                 : 1
    repeat_first_field:1                             : 0
    chroma_420_type:1                                : 0
    progressive_frame:1                              : 0
    composite_display_flag:1                         : 0
  [extension_and_user_data]
    [extension_data]
      extension_start_code:32                          : 0x000001b5
      [picture_display_extension]
        extension_start_code_identifier:4                : 7
    [user_data]
      user_data_start_code:32                          : 0x000001b2
      user_data:8                                      : 0x15
```

```
      user_data:8                                    : 0x16
      user_data:8                                    : 0x17
      user_data:8                                    : 0x18
      user_data:8                                    : 0x19
      user_data:8                                    : 0x1a
      user_data:8                                    : 0x1b
      user_data:8                                    : 0x1c
      user_data:8                                    : 0x1d
      user_data:8                                    : 0x1e
  [picture_data]
  [picture_header]
    picture_start_code:32                          : 0x00000100
    temporal_reference:10                          : 2
    picture_coding_type:3                          : <P> (2)
    vbv_delay:16                                   : 0x1dac
    full_pel_forward_vector:1                      : 0
    forward_f_code:3                               : 7
    extra_bit_picture:1                            : 0
  [picture_coding_extension]
    extension_start_code:32                        : 0x000001b5
    extension_start_code_identifier:4              : 8
    f_code[2][2]:4                                 :  2  2
                                                   : 15 15
                                                   :
    intra_dc_precision:2                           : 1
    picture_structure:2                            : <Top_Field> (1)
    top_field_first:1                              : 0
    frame_pred_frame_dct:1                         : 0
    concealment_motion_vectors:1                   : 0
    q_scale_type:1                                 : 1
    intra_vlc_format:1                             : 1
    alternate_scan:1                               : 1
    repeat_first_field:1                           : 0
    chroma_420_type:1                              : 0
    progressive_frame:1                            : 0
    composite_display_flag:1                       : 0
  [extension_and_user_data]
    [extension_data]
      extension_start_code:32                        : 0x000001b5
      [quant_matrix_extension]
        extension_start_code_identifier:4              : 3
        load_intra_quantiser_matrix:1                  : 1
        intra_quantiser_matrix[8][8]:8                 :  8 17 17 18 18 18 19 19
                                                       : 19 19 20 20 20 20 20 21
                                                       : 21 21 21 21 21 22 22 22
                                                       : 22 22 22 22 23 23 23 23
                                                       : 23 23 23 23 24 24 24 25
                                                       : 24 24 24 25 26 26 26 26
                                                       : 25 27 27 27 27 27 28 28
                                                       : 28 28 30 30 30 31 31 33
                                                       :
        load_non_intra_quantiser_matrix:1              : 1
        non_intra_quantiser_matrix[8][8]:8             : 16 17 17 18 18 18 19 19
                                                       : 19 19 20 20 20 20 20 21
                                                       : 21 21 21 21 21 22 22 22
                                                       : 22 22 22 22 23 23 23 23
                                                       : 23 23 23 23 24 24 24 25
                                                       : 24 24 24 25 26 26 26 26
                                                       : 25 27 27 27 27 27 28 28
                                                       : 28 28 30 30 30 31 31 33
                                                       :
        load_chroma_intra_quantiser_matrix:1           : 0
        load_chroma_non_intra_quantiser_matrix:1       : 0
      extension_start_code:32                        : 0x000001b5
      [picture_display_extension]
        extension_start_code_identifier:4              : 7
    [user_data]
      user_data_start_code:32                        : 0x000001b2
      user_data:8                                    : 0x15
      user_data:8                                    : 0x16
      user_data:8                                    : 0x17
```

```
      user_data:8                                   : 0x18
      user_data:8                                   : 0x19
      user_data:8                                   : 0x1a
      user_data:8                                   : 0x1b
      user_data:8                                   : 0x1c
      user_data:8                                   : 0x1d
      user_data:8                                   : 0x1e
  [picture_data]
  [picture_header]
    picture_start_code:32                         : 0x00000100
    temporal_reference:10                         : 0
    picture_coding_type:3                         : <B> (3)
    vbv_delay:16                                  : 0x1858
    full_pel_forward_vector:1                     : 0
    forward_f_code:3                              : 7
    full_pel_backward_vector:1                    : 0
    backward_f_code:3                             : 7
    extra_bit_picture:1                           : 0
  [picture_coding_extension]
    extension_start_code:32                       : 0x000001b5
    extension_start_code_identifier:4             : 8
    f_code[2][2]:4                                :  2  2
                                                  :  3  3
                                                  :
    intra_dc_precision:2                          : 0
    picture_structure:2                           : <Top_Field> (1)
    top_field_first:1                             : 0
    frame_pred_frame_dct:1                        : 0
    concealment_motion_vectors:1                  : 1
    q_scale_type:1                                : 0
    intra_vlc_format:1                            : 0
    alternate_scan:1                              : 1
    repeat_first_field:1                          : 0
    chroma_420_type:1                             : 0
    progressive_frame:1                           : 0
    composite_display_flag:1                      : 0
  [extension_and_user_data]
    [extension_data]
      extension_start_code:32                       : 0x000001b5
      [picture_display_extension]
        extension_start_code_identifier:4             : 7
    [user_data]
      user_data_start_code:32                       : 0x000001b2
      user_data:8                                   : 0x15
      user_data:8                                   : 0x16
      user_data:8                                   : 0x17
      user_data:8                                   : 0x18
      user_data:8                                   : 0x19
      user_data:8                                   : 0x1a
      user_data:8                                   : 0x1b
      user_data:8                                   : 0x1c
      user_data:8                                   : 0x1d
      user_data:8                                   : 0x1e
  [picture_data]
```

10 Conclusion

The General Parser - a programmable bitstream parser - concept has been thoroughly studied. The detail specifications are examined and defined. A prototype system has been put together and the concept is now proven to be a realistic solution.

11 Acknowledgement

The work environment I am in has free atmosphere that allows individual to pursue one's own technical interest, which somehow relates to day to day work but may not necessarily be tied to it directly. This kind of free atmosphere is important to foster creativity, which is an essential element to the R&D activity. I find the management I work for does understand this and I do appreciate that. I also want to express gratitude to some of my colleagues. Those who showed technical interest in this work became beta testers of the prototype system and provided many constructive comments; without them, the system would not be as rich as it is defined today.