Arpl Language Reference

Table of Contents

Overview

Language Representation
ARPL has more ways to represent constructs than some other languages. There are functionally redundant keywords (for instance ‘until’) that allow expressing meaning in a more variable manner. Hopefully, this is a human readble improvement over some other languages.

ARPL does not require expressions to be surrounded with ‘(‘ and ‘)’ in all cases if it can be determined by the compiler what is part of the expression. It is recommended to use the brackets however. For instance the ‘if’ statement may be written as:

```
if a < 10 then dosomething();
```

ARPL has the following:

- run-time type identification (via typenum())
- exception handling (via try/throw/catch) (experimental)
- function prolog / epilog control
- multiple case constants eg. case ‘1’,’2’,’3’:
- assembler code (asm)
- pascal calling conventions (pascal)
- no calling conventions (naked)
- inline code, auto-inlining
- additional loop constructs (until, loop, forever)
- true/false are defined as 1 and 0 respectively
- thread storage class
- structure alignment control
- firstcall blocks
- block naming
- branch prediction hints
- bit slicing
- nested functions

Compiler Options

Option	Description
-fno-exceptions	This option tells the compiler not to generate code for processing exceptions. It results in smaller code, however the try/catch mechanism will no longer work.
-finline[n]	Set the compiler's threshold to inline functions. The default is five instructions.
-fpoll[n]	This option tells the compiler to generate code to poll for interrupts periodically.
-o[pxrcl]	This option disables optimizations done by the compiler causing really poor code to be generated. p – this disables the peephole optimization step x – this disables optimization of expressions (constants) r – this disables the allocation of register variables and common subexpression elimination. Also turns off constant optimizations. c – this disables optimizations done during code generation l – this disables loop invariant optimization -o by itself disables all optimizations done by the compiler
-os	Optimize for size. This will disable optimizations that increase code size such as loop inversions.
-w	This option disables wchar_t as a keyword. This keyword is sometimes #defined rather than being built into some compilers.
-S	generate assembly code with source code in comments.
-a<n>	Set number of bits used for addressing to <n>

Riscv Register Usage

Register usage is as outline in the Riscv documentation, with the exception of x27 which the compiler uses for the global pointer to read-only data.

Regno	ABI Name	Description
x0	0	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address / link register
x6,x7	t1,t2	Temporaries
x8	fp /s0	Frame pointer
x9	s1	Saved register
x10,x11	a0,a1	Function arguments / return value
x12-x17	a2-a7	Function arguments
x18-x26	s2-s10	Saved registers
x27	gp1,s11	Ro data global pointer, saved register
x28-x31	t3-t6	Temporaries

Compiler Primitive Types

Types supported by the compiler are the same as C types with the following exceptions / additions:
byte – declares a byte, 8 bits, sized variable, may be applied to a float
char – a char variable is always 16-bits in size
half – declares a half-precision, 16 bits, floating point value
integer – is a synonym for int
single – declares a single precision, 32 bits floating-point value (short float may also be used)
quad – declares a quad precision, 128-bit floating-point value (long double may also be used)
decimal – declares a decimal floating-point variable of 128-bits. Approximately 34 decimal digits.
vector – declares a vector variable, 512 bit / 64 bytes bucket
vector_mask declares a vector masking variable

A **short int** is 32-bits, a **long int** is 128-bit, and an **int** is 64-bits.

The following additions have been made:

typenum(<type>)

allow run-time type identification. It returns a hash code for the type specified. It works the same way the sizeof() operator works, but it returns a code for the type, rather than the types size.

ARPL supports a simple try/throw/catch mechanism. A catch statement without a variable declaration catches all exceptions.

try begin <statement> end
catch(var decl) begin
end
catch(var decl)
begin
end
catch begin
end

Types:

arpl specific keywords

__attribute__

__attribute__ defines attributes associated with functions. Currently the only defined attribute is __no_temps which indicates to the compiler that the function does not use any temporary registers. This allows the compiler to omit code to save and restore temporaries around function calls. This is used primarily for functions defined in assembly language.

```
Example:
	extern signed byte KeybdGetStatus() __attribute__(__no_temps);
	extern byte KeybdGetScancode() __attribute__(__no_temps);
```

__bmap

__bmap causes the compiler to emit code using the byte mapping, bmap, instruction. This intrinsic function may be used to perform permutations on a value.

```
#include <string.h>

void *(memset)(void *s, integer c, size_t n)
begin   /* store c throughout unsigned char s[n] */
	const unsigned byte uc = c;
	unsigned byte *su = (unsigned byte *)s;
	unsigned long *pl;
	long m;

	// Source all bytes of m from byte zero
	m = __bmap(c,0);
	if ((s & 0xf) == 0 and n >= 16) begin
		pl = (unsigned long *)s;
		for (; n >= 16; ++pl, n -= 16)
			*pl = m;
		su = (unsigned byte *)pl;
	end
	for (; n > 0; ++su, --n)
		*su = uc;
	return (s);
end
```

__check

__check causes the compiler to output a bounds checking instruction. The bounds expression must be of the format shown in the example.

```
Example:
	__check (hMbx; 0; 1024);
```

The first expression must be greater than or equal to the second expression and less than the third expression or a processor check interrupt will be invoked. Note any valid expressions may be used.

__leaf

In some cases, the compiler can not tell if a function is a leaf function, for instance in prototypes of external functions. The __leaf keyword indicates to the compiler that function is a leaf function and uses the link register to perform a return.

__mulf

__mulf causes the compiler to output a fast, single cycle multiply instruction. The fast multiply instruction is limited to 24 x 16 bits.

```
Example:
       ndx = __mulf (row, 56);
```

__sync()

The __sync() intrinsic causes the compiler to generate a fence or sync operation. The __sync() intrinsic accepts a single constant integer value which is passed to the assembler for building the instruction.

Example:

```
__sync(0xFFFF)
```

addrof

This is a synonym for the '&' character in expressions. It evaluates to the address of the specified value.

align()

The align keyword is used to specify structure alignment in memory. For example, the following structure will be aligned on 64-byte boundaries even though the structure itself is smaller in size.

```
record my_rec align(64) begin
   byte name[40];
end
```

Place the align keyword just before the opening begin of a record declaration.

Note that specifying the record alignment overrides the compiler's capability to automatically determine alignment. Care must be taken to specify a alignment that is at least the size of the record.

Taking the size of a structure with an alignment specified returns the alignment.

and

logically 'and' values.
Example:

```
if (a and b) then begin
end
```

'and' resolves to a Boolean value of true (1) or false (0).

asm [__leafs]

The asm keyboard allows assembler code to be placed in a arpl function. The compiler does not process the block of assembler code, It simply copies it verbatim to the output. Global variables may be referenced by name by following the compiler convention of adding an '_' to the name. Stack arguments have to be specifically addressed referenced to the fp register. Register arguments can use the register directly.

```
pascal void SetRunningTCB(hTCB ht)
begin
   asm begin
      lw     tr,32[fp]     ; this references the ht variable
      asli   tr,tr,#10
      add    tr,tr,#_tcbs          ; this is a global variable reference
   end
end
```

The __leafs keyword indicates that the assembler code contains leafs (calls to other functions). Using the __leafs keyword causes the compiler to emit code to save and restore the subroutine linkage register.

```
// ------------------------------------------------------------------------
----
// Set an IRQ vector
// ------------------------------------------------------------------------
----

void set_vector(unsigned int vecno, unsigned int rout)
begin
     if (vecno > 255) return;
     if (rout == 0) return;
        asm __leafs begin
                     lw            r2,32[fp]
```

```
                        lw              r1,40[fp]
                        call            set_vector
        end
end
```

begin

This keyword “begins” a compound statement.

case

Case statement may have more than one case constant specified by separating the constants with commas.

ARPL:

```
switch (option) begin
case 1,2,3,4:
        printf(“option 1-4);
case 5:
        printf(“option 5”);
end
```

Standard C:

```
switch (option) {
case 1:
case 2:
case 3:
case 4:
        printf(“option 1-4);
case 5:
        printf(“option 5”);
}
```

The compiler will make use of a jump table if there are enough cases and the density of the cases is greater than 33%. The table must be at least 33% populated. Otherwise, the compiler reverts to using a series of branches in a binary tree pattern. If there are just a few cases (<4) then a linear series of branch testing is used.

If the case options are powers of two, the compiler may generate more efficient code using BBS and BBC instructions. BBS means branch-on-bit-set.

catch (<type>)

The catch statement “catches” a specific type of object used for exception handling. A catch handler corresponds to object type used in the throw statement. If the thrown object is not of a type caught by a local catch handler, then a search for the correct catch handler will continue at a more outer level.
The type may be specified as an ellipsis which will then match any type, but the object thrown is not available to the catch handler. The type may also be omitted which will also match any type.
If none of the catches match the error object then catch statements at more outer levels are checked.
When exception processing is enabled the compiler always generates a default catch for each function. The default catch merely returns up the try/catch chain. The default catch will perform the same cleanup as would be activated by a return statement.

```
try begin
  < some code>
end
catch (int erc) begin
   < process error code>
end
catch begin
   < catch anything else>
end
```

class

ARPL features a simple class keyword which may be used to implement classes. A class is very similar to a record except that class methods may be declared to be part of the class. Classes in ARPL can have only single inheritance. Methods may be overloaded with different parameters.

coroutine

arpl allows coroutines to be implemented by using a spec for the coroutine function. A coroutine may suspend processing by using the **yield** keyword allowing another routine to executed. Coroutines are managed with the use of some state stored in the data area. The coroutine keeps track of where it gave up control via the **yield** keyword. Processing will return to the point after the **yield** keyword when the coroutine is transferred to. Coroutines may exit using the **return** statement.

```
coroutine(test_stack) void test_rout(integer a, integer b) begin
end
```

coroutine accepts a parameter which specifies the stack to use for the coroutine.

decimal

decimal is used to declare a variable of the decimal type. Decimal floating-point arithmetic is performed instead of binary floating-point. The decimal type has approximately 34 digits of decimal precision.

delete

delete calls the run-time function __delete() to delete an object allocated by the new operator. __delete() takes a pointer to the object which was returned by new() and deallocates the object from the heap. If an object is deleted it is immediately deallocated and is not garbage collected. Delete does not call an object destructor. The object should be destroyed before using delete.

end

Represents the end of a compound statement.

enum

The enum keyword defines a set of enumerated values which gives a list of identifiers specific values for each identifier. This allows the identifiers to be used as numeric constants in the program text. Enumeration values are 16-bit signed integer values. Typical use is as switch cases.

The stride may be specified for the enumeration by following the enum keyword with the parenthesized stride value. The value must be a constant. If not specified the enumeration will increment by one.

In the following example the enumeration will decrement by 1 the value for each enumerated constant. so BADARG is equal to -1. This may be useful in cases where functions return a negative value indicating error or a positive value indicating proper operation.

```
enum(-1) begin
        OKAY = 0,
        BADARG
end
```

The following will increment the enumeration value by 32.

```
enum (0x20) begin
        ErrorClass0 = 0,
        ErrorClass1,
        ErrorClass2
end
```

This may be useful to define constants for bit-fields. The following example shows usage for a bit-field at bit position 7.

```
enum (0x80) begin
        GFX_POINT = 0,
```

```
        GFX_LINE,    // will equal 0x080
        GFX_RECT     // will equal 0x100
end
```

enum may also indicate a powers-of enumeration as in:

enum (*2) begin SelectA, SelectB, SelectC end

In which case SelectA would be equal to one, SelectB equal to two, and SelectC equal to four.
The powers-of enumeration causes the enumeration to multiply by the enumeration value for each step instead of adding. Different power series can be defined by setting one of the enumeration values to an arbitrary value. From that point on the values will multiply. So, “enum (*2) { A, B = 3, C }; will assign the values 1, 6, and 12 to A, B, C.

Note that a floating-point number may be used to specify the increment, however the enumeration is always integer values. The calculated enumeration value is rounded down to an integer value.

Enumeration values are only 16-bit signed integer values.

epilog

The epilog keyword identifies a block of code to be executed as the function epilog code. An epilog block maybe placed anywhere in a function, but the compiler will output it at the function’s return point.

```
naked myfunction()
begin
      // other code
        epilog asm begin
            // do some epilog work here, eg. setup return values
        end
end
```

false

False is a predefined boolean constant with the value of zero.

firstcall

The firstcall keyword defines a statement that is to be executed only once the first time a function is called.

```
firstcall begin
        printf("this prints the first time.");
end
```

The compiler automatically generates a static variable in the data segment that controls the firstcall block. The firstcall statement is equivalent to:

```
static char first=1;
if (first) begin
   first = 0;
   <other statements>
end
```

The if statement may also precede the firstcall for readability.

forever

Forever is a loop construct that allows writing an unconditional loop. A forever loop should have an alternate means of exiting. For instance, a break statement will exit the loop.

```
forever begin
        printf("this prints forever.");
end
```

generic, _Generic

The generic statement acts like a function and returns a value corresponding to the type of the specified variable. The variable is specified followed by a list of types similar to a case statement. If there is a default keyword then if the type of the variable does not match any of the types listed then the default value will be returned.

```
integer x;
x = generic(var, int: 1, char: 2, long 3: default: 4);
```

The generic statement may return values of any type, but it is best to keep all the return values of the same type so that the result may be assigned to a variable of specific type. Any non-comma expression may be used to return a result including calling a function.

half

half is used to declare a half precision binary floating point value. Half precision values occupy 16-bits. Float values are double precision binary 64-bit values by default.

inline

The inline keyword may be applied to a function declaration to cause the compiler to emit the function "inline" with other code. Every time the inline function is called, the code for the function is replicated inline. It is best to use register parameters to inline code.
Inline accepts an optional parameter indicating the number of lines below which the function will inline. This may be used to prevent inlining of functions.

inline(10) void myfunc()

will inline the function myfunc() if it has fewer than 10 lines of code.

if

If statements can accept a branch predictor hint. The hint must be a constant value determined at compile time. The syntax adds an options second expression ';' into the expression clause as shown below.

```
if (a < 10; 1)    // predict taken all the time
…
```

interrupt

A function declared as an interrupt routine will use an interrupt return instruction at the end, and may also emit code to store registers on the stack on entry, and load them from the stack on exit. Interrupt may accept an integer parameter indicating which registers to save and reload. Registers are identified in a bit mask with the lowest bit representing the lowest numbered register.

interrupt(0xfffffffa) my_irq() begin end

will save all the registers on the stack except x0 and the stack pointer for Riscv.

The interrupt keyword may also be preceded by a keyword indicating which operating mode the interrupt is for. Four supported operating modes are **__user**,

__supervisor, **__hypervisor**, and **__machine**. This affects the return instruction used at the end of the interrupt routine.

naked

The naked keyword causes the compiler to omit all the conventional stack operations required to call a function. (Omits function prologue and epilogue code) It's use is primarily to allow inline assembler code to handle function calling conventions instead of allowing the compiler to handle the calling convention. The naked keyword may also be applied to the switch() statement to cause the compiler to omit bounds checking on the switch.
A naked function also omits the default exception handler.

```
naked myfunction()
begin
        asm begin
        end
end
```

new

The new operator generates a call to the run-time library function __new(). __new() will allocate storage for the object on the heap using malloc(). The object is given a 64-byte header and the object state is set to new. Objects whose state is new will not be deleted automatically by the garbage collector. They must be deleted using the delete keyword. new may be preceded by the keyword 'auto' which indicates that automatic garbage collection should be used for the object. In this case the objects state is set to auto new.
new() does not call the object's constructor.

or

The 'or' keyword performs a logical or of values.
Example:

```
        if (a or b) begin
        end
```

pascal

The pascal keyword causes the compiler to use the pascal calling convention rather than the usual C calling convention. For the pascal calling convention, function arguments are popped off the stack by the called routine. This may allow slightly faster and smaller code in some circumstances.

```
pascal char myfunction(int arg1, int arg2)
begin
end
```

Note the default calling convention for the compiler is pascal.

prolog

The prolog keyword identifies a block of code to be executed as the function prolog. A prolog block may be placed anywhere in a function, but the compiler will output it at the function's entry point.

```
naked myfunction()
begin
        prolog asm begin
            // do some prolog work here, eg. setup stack parameters
        end
end
```

Prolog code is not as highly optimized as other code as the prolog is generally placed before register variables are saved. That means it cannot make use of register variables.

quad

quad is used to declare a quad precision binary floating point value. Quad precision values occupy 128-bits. Float values are double precision binary 64-bit values by default.

register

The compiler will not use registers to pass arguments to functions unless specifically instructed to do so. The register keyword is used for this. The compiler automatically uses registers for temporaries and other variables where possible. Using the register keyword on anything other than an argument is likely

to be ignored. Note the compiler allocates storage space on the stack for variables even if they are in registers. This storage space is often unused.

single

single is used to declare a single precision binary floating point value. Single precision values occupy 32-bits. Float values are double precision binary 64-bit values by default.

stop

stop causes a CPU stop instruction to be emitted. An integer constant may be provided as an argument to stop.

switch

The naked keyword may be applied to the switch() statement to cause the compiler to omit bounds checking. Normally the compiler will check the switch variable to ensure that it's within the range of the defined case values. With a naked switch the compiler assumes that the switch value is between the minimum and maximum case value in the switch statement. Naked switchs result in faster code, but results are undefined if the switch is out of range. For a naked switch if the switch value isn't valid then the program will likely crash. So use with caution.

Regular switch:

```
;             switch (btn) begin
              lw     r3,-32[bp]
              ldi    r4,#1
              ldi    r5,#9
              chk    r3,r4,r5,BIOSMain_13
              sub    r3,r3,#1
              shl    r3,r3,#3
              lw     r3,BIOSMain_19[r3]
              jal    r0,0[r3]
```

Naked Switch:

```
;             switch(btn; naked) {
              lw     r3,-32[bp]
              sub    r3,r3,#1
              shl    r3,r3,#3
              lw     r3,BIOSMain_19[r3]
              jal    r0,0[r3]
```

Note that if the minimum case value is zero then the code may omit the subtract of the minimum value making the switch slightly faster.

then

'then' is defined as a keyword. Its only purpose is to make code more readable. It may be used with 'if' statements in which case it is ignored.

throw

Throw acts in a similar fashion to the return statement. Throw returns to the latest catch handler. The latest catch handler does not have to be defined in the current routine or a previous routine. Throwing an exception will walk backwards up the stack to the most recently defined catch handler. Unlike c++ throw does not automatically destroy objects created in the subroutine or method.

thread

The ‘thread’ keyword may be applied in variable declarations to indicate that a variable is thread-local. Thread local variables are treated like static declarations by the compiler, except that the variable’s storage is allocated in the thread-local-storage segment (tls).

thread int varname;

true

true is a predefined Boolean constant with a value of one.

try begin end

Try defines a try – catch block. A block of statements followed by a series of catch statements. Try causes the compiler to output code to point to the catch block of the try statement. This pointer will be used by subsequent throw statements.

When exception handling is enabled every function has a default exception handler that merely returns up to the next higher exception handler.

typenum()

Typenum() works like the sizeof() operator, but it returns a hashcode representing the type, rather than the size of the type. Typenum() can be used to identify types at run-time.

```
record tag begin int i; end;

main()
begin
        integer n;

        n = typenum(record tag);
end
```

The compiler numbers the types it encounters in a program, up to 10,000 types are supported. Pointers to types add 10,000 to the hash number for each level of pointer. Program should not depend on specific hash numbers as the hash numbers of type may vary between different versions of the compiler. The compiler uses a 15-bit hash number which may be encoded into a three character ascii string.

until

Until is a loop construct that allows writing a loop that continues until a condition is true. Until and while are almost the same except that until waits for the inverted condition. It may have better semantics in some circumstances to use until instead of inverting a while condition. while(!x) is better represented as until(x)

```
x = 0;
until (x==10) begin
        printf("this prints 10 times.");
        x = x + 1;
end
```

using

The using keyword is used to activate features of ARPL. In particular a name mangler used for classes can be activated by using the phrase ‘using name mangler;’ Without activating the name mangler all class methods have global scope.

yield

The **yield** keyword is used to suspend the execution of a coroutine and transfer execution to a different routine. The **yield** records its current location so that a transfer back to the routine may continue from the yield point.

name mangler

The name mangler generates unique names for methods so that there is no name clash for overloaded or derived methods. A hash string representing the type, method parameter and return value types is added to the method name.

__cdecl

Specifies that the default calling convention is the C calling convention

pascal

Specifies that the default calling convention is the Paacal calling convention.

real names

Disables name mangling.

&&&

The &&& operator indicates to the compiler that a safe optimization is to generate code that executes both sides of the operator then use an ‘and’ operation to determine the result. This may eliminate branches.

|||

The ||| operator indicates to the compiler that a safe optimization is to generate code that executes both sides of the operator then use an ‘or’ operation to determine the result. This may eliminate branches.

??

?? is the multiplex operator. It selects one of a series of expressions from a colon separated list. If the value on the left hand side of ?? is zero, then the first expression is selected, if the value is one, then the second expression is selected, and so on. The last expression in the list acts as a default. The following example returns the value 4 as there is no expression for the tenth selection.

```
integer main()
begin
        integer a = 10;
        integer b = 1;
        integer c = 2;
        integer d = 3;
        integer e = 4;

        return ( a ?? b : c : d : e );
end
```

The ?? operator is similar to the ? operator working in the reverse order. The control expression for ? evaluates to a boolean value returning true or false (1 or 0). The control expression for ?? evaluates to an enumeration (0,1,2,3…).

a ?? b : c; is the same as a ? c : b;

Character Constants

String constants may be pre-pended with one of ‘B’, ‘W’, ‘T’, or ‘O’ to indicate the size of encoded characters. ‘B’ = byte, ‘W’ = wyde (16-bit), ‘T’ = tetra(32-bit), and ‘O’ = octa (64-bit).

```
Example: The following string is encoded as bytes:
B”? = display help”
```

String constants may also be placed inline with code using the letter ‘I’ as a prefix. Inlined string constants are automatically 16-bit encoded.

Block Naming

The compiler supports named compound statement blocks. To name a compound statement follow the opening begin with a colon then the name.

```
void SomeFunc()
begin
    while (x) begin: x_name
      <other statements>
    end
end
```

An eventual goal for the compiler is to have the break statement be able to identify which block statement to break out of.

Nested Functions

arpl allows nesting of functions. Functions may be defined within other functions, making them local to the function. Locally defined functions may access variables at a more outer function level. However, there are performance implications for variable access of this type. Outer local variables need to be dereferenced multiple times by the compiler, hence access to them is slower.

```
integer
main##__BASEFILE__()
begin
        int i;

 i = 47;
        inline(20) int foo1() begin
                return 43;
        end
        int sub1(int a, int b) begin
                int g, h;
                int sub2(int c, int d) begin
```

```
                c = c + g + i;
                d = d + h;
                return (c*d);
            end
            g = 2; h = 3;
            return (a+b);
        end
        printf("%d", foo());
        printf("%d", foo1()*8);
        sieve();
end
```

Calling Convention

ARPL uses the pascal calling convention by default. However, for functions with variable argument lists the variable portion of the list must be popped off the stack by the caller, as the called routine does not know how many variable arguments there are. The called routine will only pop the fixed portion of the argument list. The 'C' calling convention may be specified using the keyword __cdecl.

Bit Slicing

The compiler allows bits slices of primitive types to be taken. Specific bits of a variable may be selected using the indexing operator[]. In the following example bits 26 to 31 of the variable ab are updated, then bits 6 to 10 is returned with 21 subtracted from the value.

```
integer main(int a, int b, int c)
begin
  integer ab;
  ab[31:26] = a + b + c[15:5];
  return (ab[10:6]-21);
end
```

The indicated bits must specify the high bit followed by a colon then the low bit. Bits are numbered from 0 to 63.

Bit slicing aids the compiler in determining the use of bit field instructions that may be available on the processor. Bit manipulations can also be done with bitwise operators (~, &, |, ^, <<, and >>).

Array Handling Differences from ‘C’

The following is “in the works”. It may or may not work.

Arrays may be passed by value using the standard declaration of an array as a parameter. In ‘C’ arrays are always passed by reference.

In ARPL:

```
SomeFn(integer ary[50]) begin
end
```

Declares a function that accepts an array of 50 integers passed by value. Declaring the function the same way in ‘C’ results in a reference to the array being passed to the function rather than the array values.

In order to pass an array by reference in ARPL the pointer indicator ‘*’ must be used as in the following:

```
SomeFn(int *ary) begin
end
```

It is not recommended to pass large arrays or structures around in a program by value as program performance may be adversely affected. Passing aggregate types by value causes the compiler to output code to copy the values. The alternative, passing references around is significantly faster.

Vector Operations

Vector operations are support with the vector type which indicates a vector is to be used. Vectors are 64-byte wide variables that may take on elements based on the subtype declared.

vector float vec;

Will declare a vector variable consisting of float type elements.

Vector masking variable may be declared as a **vector_mask**. The masking scope is applied with parenthesis.

```
vector_mask mvar;
vector res, a, b;
res = mvar(a + b);
```

Exception handling

When exception handling is enabled, ARPL generates a default exception handler for each function. The action of the default handler is merely to return to the next higher exception handler. If an exception handler is not coded for the function then the default handler will be in effect. During function prolog, the address of any exception handler is stored in the stack frame at 16[$FP]. When a throw() operation occurs the address of the exception handler is loaded from 16[$FP] and jumped to. Throw() loads the exception type into register $a1 and the exception value into $a0. The catch handlers check the exception type in $a1 against the type of exception they handle. If there is no match, the next catch handler tests the value. If none of the catch handlers match the exception type then the default handler which returns up the exception chain is jumped to.

This causes an unhandled exception to unwind the stack just as a return would, then return to the caller's exception handler address rather than the normal return address.

To receive hardware exceptions the operating system must be notified of which exceptions are desired. A 256-bit bit mask is used to track which exceptions the application will respond to. This bitmap is stored in the applications ACB. When a selected exception occurs the OS forces the cause code into $a0, and the "exception" type into $a1 then causes the application to resume execution at the exception handler the next time the application is active.

Because of the simplicity of the exception handling mechanism objects created in the function are not automatically destroyed. That means it's necessary to keep track of which objects got created and destroy them in the catch handler.

Garbage Collection

If a function or method uses the new operator, then a call is made to the run-time library function __AddGarbage() when the function returns. The __AddGarbage() function moves objects off the function's object list onto the garbage collector's list.

Limitations

The ARPL compiler has some built in limitations, most of the limits are far beyond what would be required for ordinary use:

The maximum number of function parameters: 20
The maximum number of symbols in a translation unit: 32,000.
The maximum number of functions in a translation unit: 3,000
The maximum number of different types allowed: 32,000
The maximum number of labels in a translation unit: 65,000.
The maximum number of dimensions that may be specified for an array is 19.
The maximum number of different expressions in a function: 450.

Number of basic blocks in a function: 10,000 max.

Return Block