Creating a Back End for VPCC
Revised 3/1/91

Jack W. Davidson
David B. Whalley

Computer Science Report No. RM-88-01
March 1, 1991

The C Virtual Machine Instruction Set Definition

1. Retargeting

vpcc (Very Portable C Compiler), as its name implies, is very portable. The following steps are required to retarget *vpcc* for a new machine.

1. Update the SZtypes in the file "macdefs" to define the sizes of C types (e.g. `char`, `int`, `float`, `double`, etc.).
2. Update the `argsize` function in "genccode.c" to define the sizes of types when pushed on or popped off the run-time stack.
3. Define either `RIGHTTOLEFT` or `LEFTTORIGHT` in "genccode.c" to select the order of evaluation of arguments to functions.

2. Expanding the Intermediate Code

To produce a compiler the intermediate language must be expanded to represent instructions on the target machine. Each time an intermediate language operator is produced by *vpcc*, a function is invoked that begins with 'c_' and usually has the name of that operator. For instance, when an **`ICON`** operator is produced the `c_icon` function is called. Note that the arguments to the intermediate code operators correspond to the arguments in the functions. For instance, `c_icon` has the arguments `typeid`, `class`, and `n`. **`ICON`** has the same arguments. The values for the ***opcode*** arguments, which represent the intermediate language operators, can be found in the header file, "manifest". An argument called ***typeid*** indicates a C type. The possible basic types are:

Type	Description
`UNDEF`	undefined type such as the type of a void function
`FARG`	function argument
`CHAR`	character
`SHORT`	short integer
`INT`	integer
`LONG`	long integer
`FLOAT`	floating point
`DOUBLE`	double precision floating point
`STRTY`	structure
`UNIONTY`	union
`ENUMTY`	enumeration
`MOETY`	member of an enumeration
`UCHAR`	unsigned character
`USHORT`	unsigned short integer
`UNSIGNED`	unsigned integer
`ULONG`	unsigned long integer

Type values are either a basic type or a composite of a basic type with a modifier. The possible values of a modifier are as follows:

Modifier	Description
`PTR`	pointer to type
`FTN`	function returning type
`ARY`	array of type

The modifiers are read right to left in the type value, starting with the field adjacent to the basic type. For instance

```
ARY + (PTR << 2) + (FTN << 4) + DOUBLE
```

represents the type of an array of pointers to functions returning doubles. When manipulating type values in *vpcc*

one usually uses macros. The available macros to manipulate type values are:

Macro	Description
`MODTYPE(t,b)`	set basic type of `t` to `b`
`BTYPE(t)`	returns basic type of `t`
`ISUNSIGNED(t)`	returns true if `t` is an unsigned type
`UNSIGNABLE(t)`	returns true if `t` is one of the 4 basic types that can become unsigned
`ENUNSIGN(t)`	returns the unsigned type of the signed type `t`
`DEUNSIGN(t)`	returns the signed type of the unsigned type `t`
`ISPTR(t)`	returns true if `t` is the type of a pointer
`ISFTN(t)`	returns true if `t` is the type of a function
`ISARY(t)`	returns true if `t` is the type of an array
`INCREF(t)`	returns a type representing a pointer to `t`
`DECREF(t)`	returns a type after removing the first operator from `t`

The definition of the basic types, modifiers, and type macros are defined in the header file, "manifest". An argument called ***class*** indicates the storage class of the value being referenced. The possible values for ***class*** are:

Class	Description
`SNULL`	Constant value
`AUTO`	Automatic variable
`EXTERN`	Externally defined variable
`STATIC`	Static variable (local or global)
`REGISTER`	Register defined variable (parameter or automatic)
`EXTDEF`	Externally defined and initialized variable
`LABEL`	Label id
`MOS`	Member of a structure
`PARAM`	Parameter variable
`STNAME`	Structure name
`MOU`	Member of a union
`UNAME`	Union name
`TYPEDEF`	Type definition name
`FORTRAN`	Fortran function
`ENAME`	Enumeration name
`MOE`	Member of an enumeration
`UFORTRAN`	External fortran function
`USTATIC`	Undefined static variable

Class values are defined in the header file "mfile1". An argument called ***blkno*** indicates a block number used to uniquely identify a local variable. An argument called ***id*** contains the name of a variable. The following pages give a description of how each of these routines should be declared, the intermediate language operators to which it applies, and a written and notational description of the code the routine must generate. The notational description uses a C-like language with the following conventions:

Notation	Description
`addr`	the address of a memory location
`addr(x)`	returns the address **`x`**
`mem[x]`	a memory reference at location **`x`**
`pass(x)`	pushes **`x`** onto the run-time stack
`pop`	pops the value off the top of the CGS
`push(x)`	pushes **`x`** onto the top of the CGS
`mask(x)`	generate a **`x`**-bit mask of ones
`v`	a value
←	assignment

where CGS means the compiler generated stack.

```
void c_aoper(opcode, typeid)
int opcode;
TWORD typeid;
```

The appropriate operation is determined by the opcode.

Binary operations:

Opcode	Description
PLUS	addition
MINUS	subtraction
MUL	multiplication
AND	bitwise and
OR	bitwise or
ER	bitwise exclusive or
DIV	division
MOD	remainder
LS	left shift
RS	right shift

Code is emitted to pop the top two values on the stack, perform the binary operation, and push the result back onto the stack.

v1 ← pop;v2 ← pop;push(v2 *opcode* v1);

Unary operations:

Opcode	Description
UMINUS	unary minus
COMPL	bitwise negation

Code is emitted to pop the top value on the stack, perform the unary operation, and push the result back onto the stack.

v ← pop;push(*opcode* v);

```
void c_assign(typeid)
TWORD typeid;
```

ASSIGN - store a value in a memory location

This routine emits code that pops an address off the stack, pops a value off the stack, and stores the value at the location in memory specified by the address.

addr ← pop;v ← pop;mem[addr] ← v;

```
void c_bgnstmt(lineno)
int lineno;
```

BGNSTMT - begin a new statement

This routine is invoked before code for a source language statement is emitted. The ***lineno*** indicates the source line number of the statement This routine is useful for outputting debugging information.

```
void c_call(typeid, nargs, lenargs)
TWORD typeid;
int nargs, lenargs;
```

CALL - invoke a function

This routine generates code to call a function. ***typeid*** indicates the type of value that is returned by the

function. ***nargs*** indicates the number of arguments that have been pushed on the run time stack preceeding the call. ***lenargs*** indicates the total length of all the arguments in bytes. The address of the routine to be called is popped off the stack, the call is generated and the return value (if ***typeid*** is not UNDEF indicating a void function call) is pushed on the stack.

```
addr ← pop;push(environ);push(retaddr);pc ← addr;push(retv);
```

```
void c_casel(start, range)
long start, range;
```

CASEL - switch statement branch

This routine is invoked to generate code for a switch statement where the value of the switch expression can be indexed efficiently into the next (***range+1***) WORD statements. ***start*** indicates the lowest value of a case label for the switch and ***range*** indicates the number of ascending successive values for case labels within the switch. The value of the switch expression is popped off the stack and is used to determine the destination of the branch.

```
v ← pop;pc ← (v - start) < range ? mem[pc+(v - start)] : pc;
```

```
void c_cmpeq(typeid, n, l)
TWORD typeid;
int n, l;
```

CMPEQ - compare and branch if equal

This routine is invoked to generate code for a switch statement where the value of the switch expression cannot be indexed efficiently. If the constant ***n*** is equal to the value on top of the stack then a branch to the label ***l*** will occur.

```
v ← pop;pc ← v == n ? l : pc;
```

```
void c_cmpgt(typeid, n, l)
TWORD typeid;
int n, l;
```

CMPGT - compare and branch if greater than

This routine is invoked to generate code for a switch statement where the value of the switch expression cannot be indexed efficiently. If the value on top of the stack is greater than the constant ***n*** then a branch to the label ***l*** will occur.

```
v ← pop;pc ← v > n ? l : pc;
```

```
void c_dc(typeid, class, val, flag)
TWORD typeid;
int class, val, flag;
```

DC - define a constant

This routine is called to generate code to have a memory location initialized with an integer value. When ***class*** is SNULL and ***val*** is UNDEF then ***val*** indicates the amount of space to reserve for the variable. When ***class*** is SNULL and ***val*** is not UNDEF then the ***val*** argument represents the initial value. If ***class*** is LABEL, then ***val*** is the label number. If ***class*** is FIELD, then the field size in bits is found in the lower six bits of the class. The initializer for the field is found in ***val***. The ***flag*** is used to indicate the end of an initialization. A one indicates the initialization is complete. For example, to initialize a structure that contains bit fields may require the output of several initializers. It is necessary to indicate that there are no more, so the assembled bit pattern can be output to the assembly or RTL file. See also

c_sdc.

```
void c_decl(typeid, id, class, size, blkno)
TWORD typeid;
int class, size, blkno;
char *id;
```

DECL - define a variable

This routine is invoked to generate code for a variable declaration. A variable can be a global, a parameter, or a local. The type of declaration can be distinguished by the value of the ***blkno***.

type	**blkno**
global	0
parameter	1
local	>= 2

This routine can be called to generate code associated with a global declaration. ***class*** can be EXTERN, EXTDEF, or STATIC. If ***class*** is EXTDEF or if it is STATIC and ***id*** does not begin with an '*', then an initial value for the variable should follow (c_dc, c_fdc, c_sdc). Whether or not the variable is to be initialized may affect the form of the declaration. If the variable is not initialized then space must be allocated by the code generated for declaring the variable. If the variable is initialized then the space allocated may be implicit from the code generated from the initial value that is to follow.

This routine can be called to generate code for the declaration of an argument. ***class*** can either be PARAM or REGISTER. ***size*** indicates the size of the parameter. Typically a symbol will be defined representing an offset from the argument pointer to allow later access to the argument. A counter should usually be updated to represent the total size of the arguments encountered so far.

This routine can be invoked to generate code for a local declaration. The possible values for ***class*** include STATIC, AUTO, and REGISTER. If ***class*** is STATIC and the first character of ***id*** is '*', then the variable is not initialized. Otherwise if ***class*** is STATIC then the variable is initialized. This means that the initialization value will follow the local declaration. This may affect how the declaration of the static variable is made. For instance, if there is no initial value then ***size*** would be used to reserve space for the static variable. If the class is AUTO or REGISTER (and REGISTER is treated as an AUTO), then ***size*** is used to determine the offset of the local variable. Typically a counter will be updated to reflect the total size of the local variables so far. ***blkno*** can also be used to uniquely identify the local variable. If ***blkno*** > 2 then it is possible that more than one local variable in different blocks in the function will have the same name. Thus ***blkno*** can be appended to the name of the local variable. The name of the local variable in this routine is typically defined as a symbol representing the offset from the frame pointer. Note that ***blkno*** is not needed for static variables since each local static identifier is unique. On some machines alignment of local variables on offsets larger than byte boundaries may be desired.

```
void c_deref(typeid)
TWORD typeid;
```

DEREF - indirection

This routine is called to generate code to reference a value at a memory location. The address on top of the stack is popped off and the value at the memory location of the address is pushed onto the stack.

```
addr ← pop;push(mem[addr]);
```

```
void c_dup(typeid)
TWORD typeid;
```

DUP - duplicate the top of the stack

This routine duplicates the value on top of the stack.

```
v ← pop;push(v);push(v);
```

```
void c_efunc(n)
int n;
```

EFUNC - end of a function

This routine generates code for the end of a function. It also can be used to define symbols such as the size of the locals for the routine, the register mask, etc.. ***n*** represents the number of bytes for the function's locals. Usually this value is not used to define the size of the locals since alignment of local variables may cause additional bytes to be used.

```
void c_end()
```

This routine is not associated with any intermediate language operator. It is called when the end of the source file is encountered. If any code needs to be generated at the end of the file, this routine is useful for that purpose. Often this routine will take no action.

```
void c_endinit(typeid)
TWORD typeid;
```

INIT - global initialization sequence boundary

This routine is invoked at the end of a global variable initialization sequence. This sequence is bracketed by the **INIT** operators. Since this sequence is used to initialize a global variable, only constant expressions are allowed. Typically the code expander will maintain its own initialization stack to construct this value. In this routine, the actual generation of the code to produce the initial value may be produced.

```
void c_epdef()
```

EPDEF - end of prologue code for a procedure

This routine is invoked after code has been generated for the declarations of a function. This may provide a place where certain actions can be performed such as saving registers. Usually these types of activities can be done in the `c_func` routine. Consequently, often this routine takes no action.

```
void c_fcon(typeid, class, n)
TWORD typeid;
int class;
double n;
```

ICON - floating point constant

This routine is invoked to generate code to generate a floating point constant. ***typeid*** indicates whether the constant is to be a float or a double. ***n*** is the value of the constant. This value is pushed on the stack. Often ***class*** is not used since this routine only pushes a value.

```
push(n);
```

```
void c_fdc(typeid, class, value)
TWORD typeid;
int class;
double value;
```

DC - define constant

This routine is called to generate code for a memory location to be initialized with an floating point value. ***typeid*** can either be float or double. ***value*** is the floating point constant that is to be loaded into memory. ***class*** may be used to specify the type of the memory address, but often this argument is not used since this routine is called immediately following the declaration of the variable (ex: `c_gbl`). Note: besides outputting the three arguments passed to this routine, it also outputs a zero word. Some **DC** operations (like when outputting a string) require an ***offset***. See `c_sdc`.

```
void c_fld(typeid, n)
TWORD typeid;
long n;
```

FLD - extract field of bits

This routine will generate code to cause an address to be popped off the stack, the extraction of a set of bits from the memory location pointed to by the address, and these bits to be pushed back onto the stack. ***typeid*** will indicate an integral type. ***n*** indicates how the bits are to be extracted. The six least significant bits of the value in ***n*** represent the number of bits to extract. The remaining bits are used to indicate the offset in bits from the start of the memory location that the field structure occupies. This bit offset indicates the least significant bit of the desired bit field.

```
addr ← pop;v ← mem[addr];offset ← n >> 6;numbits ← n & 077;
v ← v >> offset;v ← v & mask(numbits);push(v);
```

```
void c_func(typeid, id, class)
TWORD typeid;
int class;
char *id;
```

FUNC - beginning of a function

This routine is called to generate code for the start of a function. There are several actions that could occur in this routine. Code should be generated declaring the function so that the function may be called. ***class*** indicates whether the function can be called from an external file. If not then ***class*** would be `STATIC` or `USTATIC`. Registers that the function uses should be saved if a callee save model is being used. Also counters representing the space used by the arguments and local variables for the function need to be initialized.

```
void c_goto(n)
int n;
```

GOTO - unconditional branch

This routine is called to generate code for a goto statement and is sometimes used in the generation of other control constructs. Code is generated that causes a unconditional branch to the label indicated by ***n***. Often the top of the stack needs to be popped if the stack is not empty. This is in support of constructs such as the "? :" operation. (A register holds the value of the true part of the "? :" operation. The false part will assign a value to the same register.)

```
pc ← n;
```

```
void c_icon(typeid, class, n)
TWORD typeid;
int class, n;
```

ICON - integer constant

This routine is invoked to generate code for an integer constant. ***class*** can either be SNULL or LABEL. If SNULL then ***n*** is the value of the constant. This value is pushed on the stack. If it is LABEL then ***n*** identifies a label that has the address of the value. Typically an "L" is used as a prefix to the label identifier, indicating it is a local static. The address of the label is pushed onto the stack.

```
push(n); or push(addr(Ln));
```

```
void c_ifld(typeid, n)
TWORD typeid;
long n;
```

IFLD - insert a field of bits

This routine will generate code to cause the insertion of a set of bits into memory. The address of the memory location is popped off the stack. A value containing the field of bits is then popped off the stack. The bits are inserted into the memory location. ***typeid*** will represent an integral type. ***n*** indicates how the bits are to be inserted. The six least significant bits of the value in ***n*** represent the number of bits to insert. The remaining bits are used to indicate the offset in bits from the start of the memory location that the field structure occupies. This bit offset indicates the least significant bit of the desired bit field.

```
addr ← pop;bitfield ← pop;offset ← n >> 6;numbits  -n & 077;
dstv ← mem[addr];dstv ← dstv & (mask(numbits) << offset);
bitfield ← bitfield << offset; dstv ← dstv | bitfield;
mem[addr] ← v;
```

```
void c_initaoper(opcode, typeid)
int opcode;
TWORD typeid;
```

PLUS - addition
MINUS - subtraction

This routine is invoked to generate an arithmetic operation involving initializing a global variable. The only values allowed at this time for ***opcode*** are PLUS and MINUS. Such operations occur when a global variable is initialized with the offset of the address of another global variable. This routine will only be called at some point following the call of c_startinit. The routines, c_startinit and c_endinit, bracket the global initialization operation. This routine will usually take a different action than its counterpart c_aoper. This is because the initial value must be calculated at compile time and thus registers cannot be used. Typically, a stack is maintained and an assembly-time expression is constructed.

```
void c_initicon(typeid, class, n)
TWORD typeid;
int class, n;
```

ICON - integer constant

This routine is invoked to generate an integer constant used in an arithmetic operation involving the initialization of a global variable. The integer constant is pushed on an internal initialization stack. See c_startinit, c_endinit, and c_initaoper for more details on initializing global variables.

```
void c_initname(typeid, id, class, blkno)
TWORD typeid;
int class, blkno;
char *id;
```

NAME - identifier reference

This routine is invoked to generate the address of a variable name referenced in an arithmetic operation involving the initialization of a global variable. The address of the name is pushed on an internal initialization stack. See `c_startinit`, `c_endinit`, and `c_initaoper` for more details on initializing global variables.

```
void c_initsconv(oldtype, newtype)
TWORD oldtype, newtype;
```

SCONV - convert

This routine is invoked to generate a conversion in an arithmetic operation involving the initialization of a global variable. Typically no action would occur unless conversions were allowed in assembly time expressions. Usually the conversion would be from an integer to a ptr type. See `c_startinit`, `c_endinit`, and `c_initaoper` for more details on initializing global variables.

```
void c_jmp(opcode, typeid, label)
int opcode, label;
TWORD typeid;
```

JMPEQ - jump if equal
JMPGE - jump if greater than or equal
JMPGT - jump if greater than
JMPLE - jump if less than or equal
JMPLT - jump if less than
JMPNE - jump if not equal

This routine is invoked to generate code for a conditional jump. The ***opcode*** represents the type of conditional jump (the intermediate opcodes listed above). The ***label*** represents the destination of the conditional jump if the comparison is true. Two values are popped off the stack and code is generated to compare the two values and perform the conditional jump.

v1 ← pop;v2 ← pop;pc ← (v2 *opcode* v1) ? *label* : pc;

```
void c_llabel(n)
int n;
```

LLABEL - generate local label

This routine is called to generate code for a label. These labels are generally targets of branches and thus are referred to as local labels. ***n*** identifies the label. A label may typically be prefixed with a 'L'. This depends on the target assembly language.

```
void c_name(typeid, id, class, blkno)
TWORD typeid;
int class, blkno;
char *id;
```

NAME - reference an identifier

This routine is called to generate code for the address of the name of an identifier that is referenced. The address of the name should be pushed on the stack. The form of the name will depend upon the value of ***class***. ***blkno*** may be used to help uniquely identify variables having an `AUTO` class (see

c_lcl).

push(addr(*id*));

```
void c_pconv(oldtype, newtype)
TWORD oldtype, newtype;
```

PCONV - pointer conversion

This routine is called to generate code to convert a pointer of one type to a pointer of a new type. The value on the top of the stack is converted and remains on top of the stack. On many machines no action is required to do this conversion.

```
void c_pusha(typeid)
TWORD typeid;
```

PUSHA - push an address

This routine is called to generate code that pops an address off the code generator stack and pushes it onto the run-time stack.

addr ← pop;pass(addr);

```
void c_pushv(typeid)
TWORD typeid;
```

PUSHV - push a value

This routine is called to generate code to push a value. The value on top of the stack is popped off and pushed onto the run-time stack. The value may need to be converted to a different type before it is pushed on the run-time stack. For instance most implementations of C require that a float be converted to a double before it is passed. The size of the values being passed need not be calculated since this is done by *vpcc* in the `argsize` routine in the file "genccode.c" and the total will be passed to `c_call`.

v ← pop;pass(v);

```
void c_reasg0(typeid)
TWORD typeid;
```

REASG0 - reassign special register

This routine reassigns a special register created using **USE0** to a standard pseudo register (a register on the compiler-generated stack). Often no action is required when a register is popped off the stack in `c_goto`.

```
void c_rel(opcode, typeid)
int opcode;
TWORD typeid;
```

EQ - set if equal
NE - set if not equal
LE - set if less than or equal
LT - set if less than
GE - set if greater than or equal
GT - set if greater than

This routine emits code to pop the top two values off the stack, compare the values according to ***opcode*** (which can have the intermediate code values listed above), and push a 1 on the stack if the comparison

is true or a 0 if it is not.

```
v1 ← pop;v2 ← pop;push(v2 opcode v1 ? 1 : 0);
```

```
void c_return(typeid)
TWORD typeid;
```

RETURN - return to caller

This routine is called to generate code for a return statement. If ***typeid*** is not UNDEF denoting a void function, the value on top of the stack is popped off and placed in a location that is known to the caller. Code is also generated to cause a return to the caller. The exact set of activities required depends upon the calling sequence. Usually these activities are shared between the c_call, c_func, and c_return routines.

```
v ← pop;addr ← pop;pop(environ);push(v);pc ← addr;
```

```
void c_reuse0(typeid)
TWORD typeid;
```

REUSE0 - place value in special register assigned using **USE0**

This routine pops the top of the stack and assigns that value to a special register assigned in c_use0. As in c_reasg0 often no action is required if c_goto can pop the top of the stack. See c_goto for more information.

```
void c_rotate(typeid)
TWORD typeid;
```

ROTATE - rotate 3rd element on stack to the top of the stack

This routine rotates the item that is two elements below the top of the stack up to the top of the stack. This is sometimes used in augmented assignments, post increments and decrements, etc.

```
v1 ← pop;v2 ← pop;v3 ← pop;push(v2);push(v1);push(v3);
```

```
void c_sconv(oldtype, newtype)
TWORd oldtype, newtype;
```

SCONV - convert

This routine generates code to convert the value on top of the stack from the old type to the new type.

```
void c_sdc(typeid, class, id, 0)
TWORD typeid;
int class;
char *id;
```

DC - define constant

This routine generates code to place the address of a string in memory. ***class*** indicates how to reference the address of the string. The string will either be some global class or a static (ex:LS19). ***id*** contains the name of that string. This routine is called immediately following the generation of code for the declaration of a variable. Note: the zero argument is to maintain compatibility with the other **DC** calls.

```
void c_setc()
```

SETC - setup for call

This routine is invoked just before code is generated that pushes arguments onto the run-time stack for a

call. Any actions that need to be taken before the arguments are pushed on the stack can be generated by this routine.

```
void c_slabel(n)
int n;
```

SLABEL - string label

This routine generates code to declare a string label. ***n*** identifies the label. Typically "LS" may preceed the value of n as the name of the string label to indicate that it is a local static.

```
void c_sseg(n)
int n;
```

SSEG - set segment

This routine issues code to change the segment. The segment types for ***n*** are defined in the file mfile1. They are as follows:

Code	Name	Description
0	PROG	program
1	DATA	standard data
2	ADATA	array data
3	STRNG	nonglobal string definition
4	ISTRNG	global string definition
5	STAB	symbol table

Often machines will not distinguish between any of these segments except the program segment.

```
void c_starg(typeid, size)
TWORD typeid;
int size;
```

STARG - structure argument

This routine generates code to pop the address of a structure argument off the stack and push the value of the structure argument onto the run-time stack. ***size*** indicates how many bytes are to be passed.

```
addr ← pop;strncpy(v,addr,size);pass(v);
```

```
void c_startinit(typeid)
TWORD typeid;
```

INIT - global initialization sequence boundary

This routine is called to indicate the start of a global variable initialization sequence. This sequence will continue until the `c_endinit` routine is invoked. Since the constant expression is typically maintained on an internal stack, this routine usually will initialize that stack.

```
void c_stasg(typeid, size)
TWORD typeid;
int size;
```

STASG - structure assignment

This routine generates code for a structure assignment. It first pops the destination address off the stack. It then uses the source address on the top of the stack to copy ***size*** bytes from the source location to the destination location in memory.

```
daddr ← pop;saddr ← pop;strncpy(daddr,saddr,size);
```

```
void c_swap(typeid)
TWORD typeid;
```

SWAP - swap top two elements on stack

This routine swaps the top two values on the stack. This is used by *vpcc* to sometimes accomplish augmented assignments, post increments and decrements, etc.

```
v1 ← pop;v2 ← pop;push(v1);push(v2);
```

```
void c_switchs(typeid, n, st, l)
TWORD typeid;
int n, st, l;
```

SWITCHS - start of switch environment

This routine generates code for the start of the switch statement. The switch consists of ***n*** cases, the first case being ***st***, and the range of values between cases is ***l***. Often no action is needed in this routine. This will of course depend on the target machine.

```
void c_switchv(typeid)
TWORD typeid;
```

SWITCHV - switch value

This routine will generate code to pop the top value off the stack and assign it to a special location to be used as a switch value. Depending upon how the switch can be implemented in the target machine, this may require converting the value to a different type.

```
void c_use0(typeid)
TWORD typeid;
```

USE0 - place value in special register

This routine will generate code to pop off the value on the top of the stack and place it into a special register. As with `c_reuse0` and `c_reasg0`, often no action is required. See `c_goto`.

```
void c_word(n)
int n;
```

WORD - address of case label for a switch statement

This routine will generate code to cause a definition to hold the address of the label ***n***. Usually the definition will be the size of a word. This label address is used in conjunction with the `c_casel` routine.