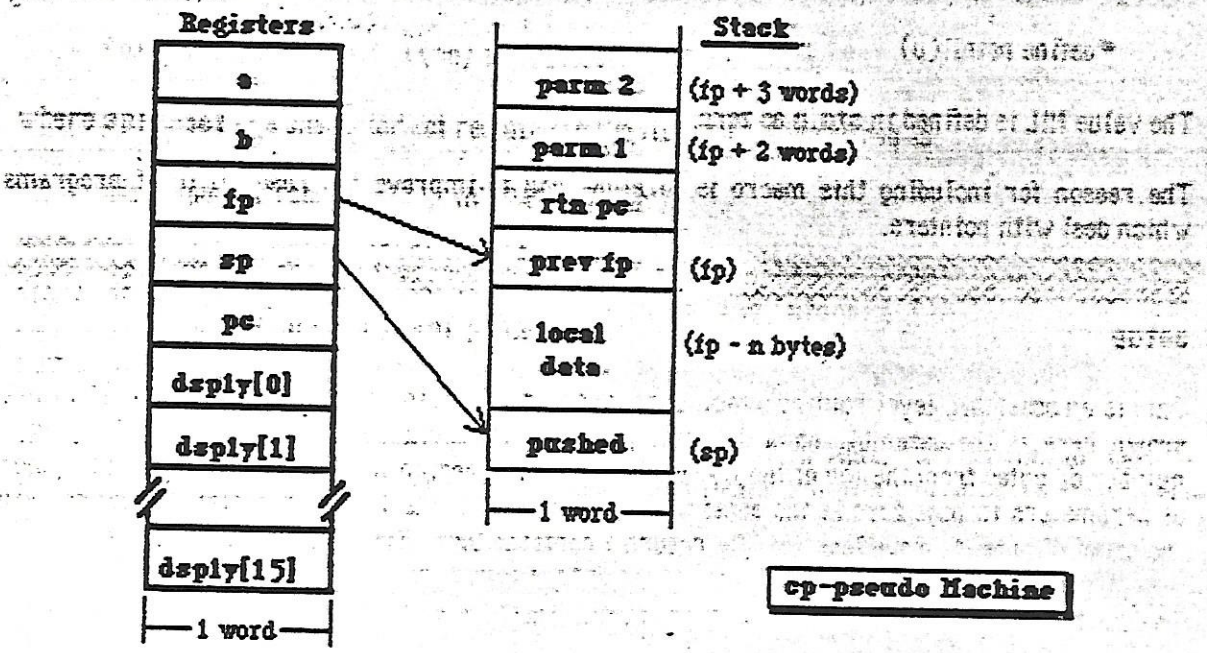
Description of the C-Compiler Pseudo Machine and cp-Codes

The C-Compiler produces an intermediate code (here called cp-Code) which may be converted to native code for some machine, or interpreted. It is based on a pseudo machine with a stack, and two accumulators. Each register is presumed to be a word wide, where a word is the size of machine pointers. The registers, and machine model are:



Register		
a	accumulator a	
b	accumulator b	
fp	frame pointer	(points to stack location of previous fp)
sp	stack pointer	(points to last word pushed on stack)
pc	program counter	(points to next instruction to execute)
dsply	display regs	(an array of frame pointers for each of 16 static lex levels)

Accumulator **a** is the primary register, capable of holding data and addresses. Arithmetic always acts upon the data in this register, and leaves the result there. Indirect loads and calls always act on the address in register **a**.

Accumulator **b** is the secondary register used to hold the second operand for a binary operation, or the destination address for indirect store operations.

The frame pointer is used as the base address register for function parameters (positive offsets) and local variables (negative offsets). This register always points into the stack somewhere. The previous value of the **fp** (prior to the entry link instruction) is stored at the address in **fp**.

The stack may only contain word sized values. Thus, bytes are always coerced to word before being pushed. A push may act on the **pc**, **fp**, accum **a**, immediate data, or the contents of some address. A pop from the stack may store a value into the **pc**, **fp**, or accum **b**.

The display regs need some further explanation... When C functions are nested, the nesting level is referred to as the static lex level of the function. Most normal C functions have a static lex level of zero. When a function containing nested functions is entered, it must save its frame pointer in the display at the appropriate static lex level so that nested functions can gain access

to its local variables.

For purposes of description, the cp-Code instructions have been categorized into several groups. These are:

Arithmetic

add		a = a + b
and		a = a & b
com		a = ~a
div		a = a / b (signed)
divu		a = a / b (unsigned)
dvi	n	a = a / n (unsigned)
inc	n	a = a + n
mod		a = a % b (signed)
modu		a = a % b (unsigned)
mpi	n	a = a * n
mul		a = a * b
neg		a = -a
not		a = !a
or		a = a | b
shl		a = a << b (b is unsigned shift count)
shr		a = a >> b (signed shift; b is unsigned shift count)
shru		a = a >> b (unsigned shift; b is unsigned shift count)
sub		a = a - b
xor		a = a ^ b

Call & Return

isp	n	sp = sp + n
jsr	addr	++pc, push pc, pc = addr
jsi		++pc, push pc, pc = a
lnk	n	push fp, fp = sp, sp = sp - n
lnkv	lev,n	push dsply[lev], push fp, fp = sp, sp = sp - n
rtn		fp -> sp, pop fp, pop pc
rtnv	lev	sp = fp, pop fp, pop dsply[lev], pop pc

Loads

ldc	n	a = n	
ldcb	n	b = n	
lea	addr	a = addr	
ldb	addr	a = (addr).b	(unsigned byte)
ldw	addr	a = (addr).w	
lla	n	a = fp + n	
llb	n	a = (fp + n).b	(unsigned byte)
llw	n	a = (fp + n).w	
lib	n	a = (a + n).b	(unsigned byte)
liw	n	a = (a + n).w	
lvb	lev,n	a = (dsply[lev] + n).b	(unsigned byte)
lvw	lev,n	a = (dsply[lev] + n).w	
lbit	n1, n2	isolate bitField in a, n1 = bit offset, n2 = width	(note 1)
tab		b = a	

Pushes

pea	addr	push addr	
pla	n	push fp + n	
pdc	n	push n	
pdb	addr	push (addr).b	(unsigned byte)
pdw	addr	push (addr).w	
plb	n	push (fp + n).b	(unsigned byte)
plw	n	push (fp + n).w	
pib	n	push (a + n).b	(unsigned byte)
piw	n	push (a + n).w	
pvb	lev,n	push (dsply[lev] + n).b	(unsigned byte)
pvw	lev,n	push (dsply[lev] + n).w	
psh		push a	
pop		pop stack into b	

Stores

stb	addr	(addr).b = a	
stw	addr	(addr).w = a	
slb	n	(fp + n).b = a	
slw	n	(fp + n).w = a	
sib	n	(b + n).b = a	
siw	n	(b + n).w = a	
svb	lev,n	(dsply[lev] + n).b = a	
svw	lev,n	(dsply[lev] + n).w = a	
sbit	n1,n2	prep for bitField store, n1 = bit offset, n2 = width	(note 2)
mov	n	(b) -> (a) for n bytes, a unchanged	
incl		add element a to bitset at (b), a = b	
excl		remove element a from bitset at (b), a = b	

Tests

The following tests always yield a value of 1 if the test is true, or 0 if the test is false.

in	a = b in bitset at (a)	
eq	a = (a == b)	
ne	a = (a != b)	
gt	a = (a > b)	
ge	a = (a >= b)	
le	a = (a <= b)	
lt	a = (a < b)	
eqz	a = (a == 0)	
nez	a = (a != 0)	
ltz	a = (a < 0)	
lez	a = (a <= 0)	
gez	a = (a >= 0)	
gtz	a = (a > 0)	
ugt	a = (a > b)	(unsigned)
uge	a = (a >= b)	(unsigned)
ule	a = (a <= b)	(unsigned)
ult	a = (a < b)	(unsigned)

Jumps

jeq	**addr**	if a == b then pc = addr, else ++pc
jne	**addr**	if a != b then pc = addr, else ++pc
jlt	**addr**	if a < b then pc = addr, else ++pc
jle	**addr**	if a <= b then pc = addr, else ++pc
jge	**addr**	if a >= b then pc = addr, else ++pc
jgt	**addr**	if a > b then pc = addr, else ++pc
jeqz	**addr**	if a == 0 then pc = addr, else ++pc
jnez	**addr**	if a != 0 then pc = addr, else ++pc
jltz	**addr**	if a < 0 then pc = addr, else ++pc
jlez	**addr**	if a <= 0 then pc = addr, else ++pc
jgez	**addr**	if a >= 0 then pc = addr, else ++pc
jgtz	**addr**	if a > 0 then pc = addr, else ++pc
jult	**addr**	if a < b (unsigned) then pc = addr, else ++pc
jule	**addr**	if a <= b (unsigned) then pc = addr, else ++pc
juge	**addr**	if a >= b (unsigned) then pc = addr, else ++pc
jugt	**addr**	if a > b (unsigned) then pc = addr, else ++pc
jmp	**addr**	pc = addr
swt	**addr**	perform a switch block (note 3)

Compiler Codes

The C compiler may produce some cp-Codes which were not covered above... These are effectively synonyms for one of the above codes, and are produced to aid native code generators. These codes are:

ldu	**addr**	same as **ldb** since bytes are always unsigned	
llu	**n**	same as **llb**	(ditto)
liu	**n**	same as **lib**	(ditto)
lvu	**lev,n**	same as **lvb**	(ditto)
pdu	**addr**	same as **pdb**	(ditto)
plu	**n**	same as **plb**	(ditto)
piu	**n**	same as **pib**	(ditto)
pvu	**lev,n**	same as **pvb**	(ditto)
taa		same as **tab**, used to indicate preparation of b for indirect addressing	
poa		same as **pop**, used to indicate preparation of b for indirect addressing	
jsr	**n,addr**	same as **jsr addr**, followed by **isp n** (n ignored if zero)	
jsi	**n**	same as **jsi**, followed by **isp n** (n ignored if zero)	

Additionally, when the cp-Code Assembler sees a **lnk** or **lnkv** instruction, it generates a 6502 **brk** instruction just ahead of the instruction to signal the interpreter that the following is cp-Code, instead of native code. The compiler guarantees that a **lnk** or **lnkv** instruction is generated only once for each function -- at its entry point.

Note 1

The bitfield instruction **lbit** operates on data already in accumulator a. The immediate data provide the offset of the least significant bit in the word, and the width in bits of the field. Bit numbering proceeds right to left in bytes and words, with the least significant bit being bit #0.

Note 2

The bitfield instruction **sbit** is always used immediately ahead of a store operator. Its presence indicates to the interpreter that a bitfield store is to be performed, with the offset and width provided. This is necessary because a bitfield store is really a mask and shift of the data in accumulator a, then logically or'ed with the byte or word of the store's target address (with its corresponding bitfield bits zero'd first), finally followed by a byte or word data store. (Whew!) Note that the operation must leave the contents of accumulator a unchanged.

Note 3

The **swt** instruction performs a switch block operation with the table at addr. The contents of the table is a list of 2-word entries. The first word in each entry contains the address of a label, the second word contains the test value. The last (default) entry contains a NIL in the first word, and the address of the default label in the second word.

The 6502 Version of the C Engine

The top end of page zero of the 6502 memory contains a number of reserved locations used to simulate a 16-bit C machine. This section discusses the function of these C registers. You should be familiar with the operation of the cp-Code machine in general before reading this section. The next page shows a diagram of the C-Engine Registers and their locations in page zero of the 6502 memory map. These registers and their locations are predefined for you by including the file **regs65.ah** at the front of your Assembly program.

When you write Assembly language modules to interface with C programs or other modules from the standard library you must respect a few rules about register use, parameter passing, and returning of results...

C always passes parameters on the C-Stack -- you should too! The C-Stack is located somewhere in high memory and the sp register always points at its base, or the location of the last pushed item. Stack items are always one word (2 bytes) wide, with the low byte of the word occupying the lowest stack address.

Pushing a parameter on the stack involves first decrementing the stack pointer register, sp, by 2 and then storing the bytes using the "[sp],y" addressing mode. If you only want to pass a byte parameter, you should pass the byte as the low byte of the stack parameter word, and zero the high byte.

Parameters expected by your functions will be located at addresses pointed to by the stack pointer register, sp, and above. You can access them with the "[sp],y" addressing mode. You can also store into the y register the number of bytes of parameters less one, that you expect, and call the library routine **setup** with a 6502 **jsr** instruction. This function copies the parameters in order from the C-Stack to page zero locations beginning with address 00. Be careful not to ask for too many parameters, lest you overrun other reserved page zero locations. (See the **Page Zero Memory Map**). On return from **setup**, the y register will contain 0xff.

Results from C routines are always returned in the primary register, r1. This is where you should place any results of your routines. Again, the result is always a word wide, low byte first. If you only want to return a byte result, you should zero the high byte of r1 before returning. (C++ 6502 considers bytes as unsigned quantities).

Parameters pushed on the stack must always be removed by the routine that put them there. Simply increment the stack pointer register, sp, by the number of bytes pushed -- this will always be a multiple of 2 bytes!

The other registers are divided into two camps as far as you are concerned. You must respect the contents of the fp, sp, pc, and dsply registers. That is, if you alter those page zero locations you should restore them to their original state before returning.

The other registers, r1, r2, r3, r4, jp, and smask are free for you to use as you like, with results being returned in r1. These registers are used during the execution of a cp-Code instruction, which for all intents and purposes happens indivisibly. That is, a cp-Code instruction always executes to completion before your routine could be activated.

If you are writing interrupt handlers, however, a single cp-Code instruction could be interrupted. In such a case **none** of the registers should be altered by your handler!

Your Assembly routines should always return with a 6502 rts instruction.

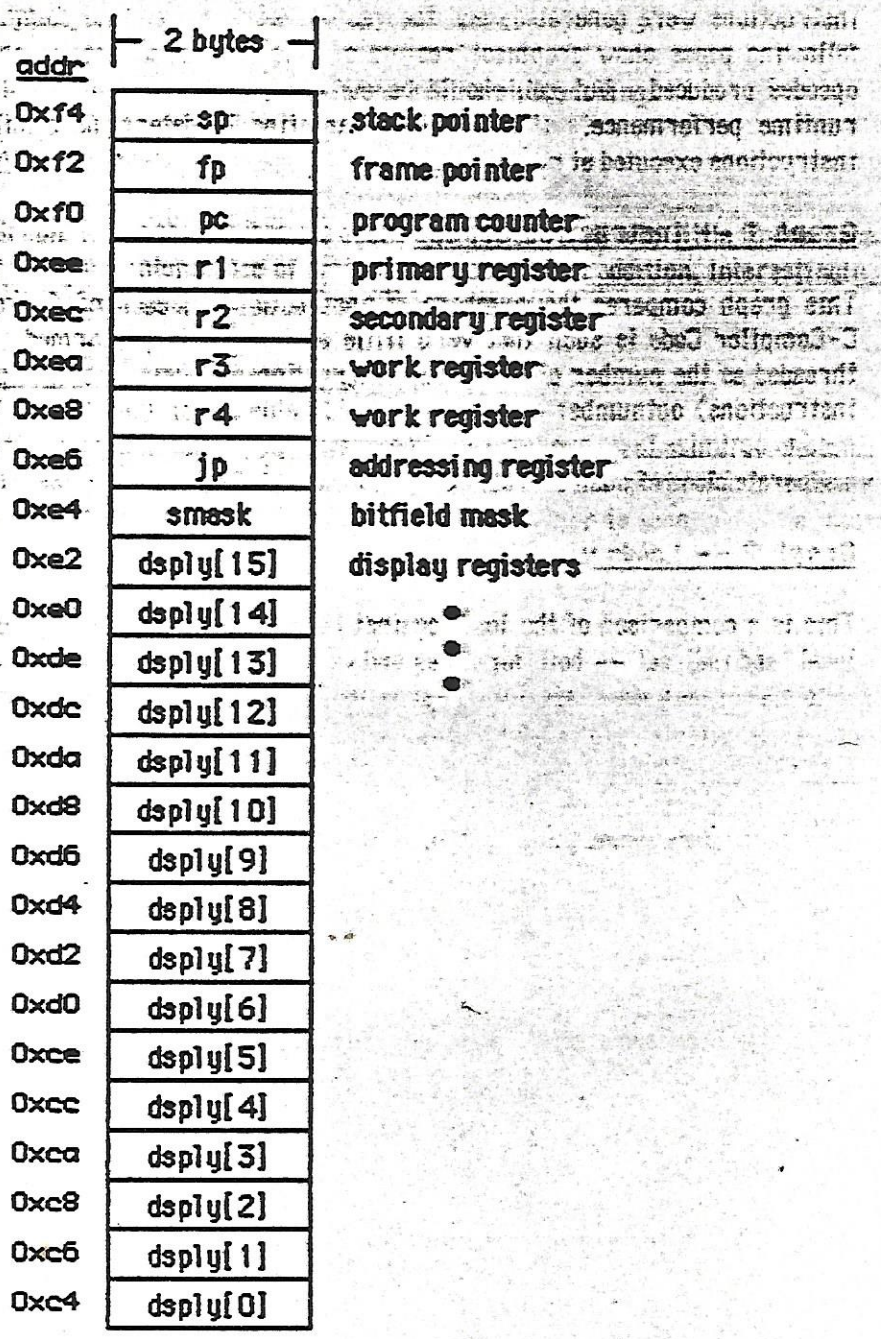
The state of the C-Engine is represented uniquely (interrupts not withstanding) by the set of dsply registers, the pc, fp, and sp registers -- and -- the 6502 machine address stored at the top of the machine stack in page 1, and the value of the machine stack pointer.

The functions **setjmp** and **longjmp** use this information. **setjmp** makes a copy of the current machine state. **longjmp** restores the machine state from this copy to effect a direct return to a known state.

This is precisely what happens on an **exit** call from a C program. The **exit** function performs a **longjmp** to a state preserved by the operating system before it launched your program.

The header file **setjmp.h** contains a declaration of a **JMPVECT** structure which can be used to hold a copy of the machine state.

These functions may also find use in your programs for emergency and error handling situations. The descriptions of the individual functions show examples of their use.



The C Engine Registers in Page Zero