C-SKY V1.0 Applications Binary Interface Standards Manual

Release 0.4

csky

Nov 26, 2018

C-SKY MicroSystems Co.,LTD

Address: Floor 15,Building A,Tiantang Software Park,No.3,Xidoumen Road, Xihu District,Hangzhou

Postcode: 310012

Website: www.c-sky.com

Contents

CHAPTER 1

About this document

1.1 Abstract

This manual defines the CSKY V1.0 CPU Applications Binary Interface (ABI). The ABI is a set of interface standards that writers of compilers and assemblers must use when creating tools for the CSKY V1.0 CPU architecture.These standards cover run-time aspects as well as object formats to be used by compatible tool chains. A standard definition ensures that all CSKY V1.0 CPU tools are compatible and can interoperate.

Although compiler support routines are provided, this manual does not describe how to write CSKY V1.0 CPU development tools, does not define the services provided by an operating system, and does not define a set of libraries. Those tasks must be performed by suppliers of tools, libraries, and operating systems.

1.2 Purpose

The standards defined in this manual ensure that all chains of development tools for C-SKY V1.0 CPU will be fully compatible. Fully compatible tools can interoperate, and thus make it possible to select an optimum tool for each link in the chain, rather than selecting an entire chain on the basis of overall performance. The Technology Center of Hangzhou C-SKY Microsystems Co., Ltd will provide a test suite to verify compliance with published standards.

The standards in this manual also ensure that compatible libraries of binary components can be created and maintained. Such libraries make it possible for developers to synthesize applications from binary components, and can make libraries of common services stored in on-chip ROM available to applications executing from off-chip ROM. With established standards, developers can build up libraries over time with the assurance of continued compatibility.

Two overriding goals are reflected in this manual:

- Use of interfaces that allow future optimizations for performance and energy.

For example, whenever possible, registers are used to pass arguments, even though always using the stack might be easier. Small programs whose working sets fit into the registers are thus not forced to make unnecessary memory references to the stack just to satisfy the linkage convention.

- Use of interfaces that are compatible with legacy "C" code written for the C-SKY V1.0 whenever possible.

For example, whenever possible, CK510/CK610 rules are used to build an argument list. This not only fits the CK510/CK610 programmer s expectations, but easily supports old code that doesn t use the ANSI standard macros for handling variable argument lists.

1.3 References

Table 1.1: The references

GC++ABI	http://www.codesourcery.com/cxx-abi/abi.html	Generic C++ ABI
GDWARF	http://dwarf.freestandards.org/Dwarf3Std.php	DWARF 3.0, the generic debug format
GABI	http://www.sco.com/developers/gabi/ ...	Generic ELF, 17th December 2003 draft
GLSB	http://www.linuxbase.org/spec/refspecs/ ...	gLSB v1.2 Linux Standard Base
Open BSD	http://www.openbsd.org/	Open BSD standard

1.4 Current status and anticipated changes

1. This manual has been released publicly. This manual is meant to be expandable.
2. Anticipated changes to this document include typographical corrections and clarifications.
3. Future standard The C++ ABI for C-SKY(Exception handle, etc.) may be added into the manual.
4. PE object file format is anticipated to be added to this manual.
5. The Linux system interface for compiled application programs (The ABI for C-SKY Linux) is anticipated to be added to this manual.
6. TLS for Linux ABI, Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process allocate locations in which to store thread-specific data.

1.5 Overview

Standards in this manual are intended to preclude creation of incompatible development tools for the CSKY V1.0, by ensuring binary compatibility between:

Object modules generated by different tool chains

Object modules and the CSKY V1.0 processor

Object modules and source level debugging tools

Current definitions include the following types of standards.

1.5.1 Low-Level Run-Time Binary Interface Standards

- Processor specific binary interface — the instruction set, representation of fundamental data types, and exception handling
- Function calling conventions — how arguments are passed and results are returned, how registers are assigned, and how the calling stack is organized

1.5.2 Object File Binary Interface Standards

- Header convention
- Section layout
- Symbol table format
- Relocation information format
- Debugging information format

1.5.3 Source-Level Standards

- C language — preprocessor predefines, in-line assembly, and name mapping
- Assembler — syntax and directives

1.5.4 Library Standards

- Compiler assist libraries — floating point, and long-long integer

1.6 Change history

Table 1.2: Record of Change

Revision	Date	Change by	Description
V0.1	2006-10-09	Chunqiang Li	1. First public release 2. Procedure call standard for C-SKY 3. The syntax used to insert assembly language statements into C language programs 4. C89 run-time library 5. ELF for C-SKY 6. DWARF 1.1 for C-SKY
V0.2	2010-06-28	Chunqiang Li	1. Run-time ABI for C-SKY 2. Shared Libaray interface 3. DWARF 2.0 for C-SKY 4. The format of the manual
V0.3	2010-08-02	Chunqiang Li	1. Add CK600 MMU & SPM Registers Description 2. Add Floatpoint & HI/LO Registers Description 3. Fix the bugs in **Function Address** Description for PIC 4. Fix some bugs
V0.4	2018-04-10	Luxia Feng	1. Add software name in Register Assignments

CHAPTER 2

Low-level Binary Interfaces

2.1 Processor Architecture

C-SKY processor is the 32-bit high-performance and low-power embedded processor designed for embedded system and SoC development. Its architecture and micro-achitecture which have extensible instruction set, configurable hardware, re-synthesis, friendly integration and so on is independently designed. In addition, it is excellent in power management. It implements several strategies to reduce power consumption including statically designed and dynamic power supply management,low voltage supply,entering low power mode and closing internal function modules.

So far C-SKY V1.0 possesses IP has two subserials: CK500 including CK510,Ck520,CK510(ESM) and CK600 including CK610, ck620 and ck610(ESM-F). CK510 is the first generation of C-SKY IP, and CK610 is the second generation of C-SKY IP which has more efficient performance than CK510. CK520/CK620 inserts OMFLIP, MAC, MTLO, MTHI, MFHI and MFLO instructions based on CK510/CK610 instruction set.

The "E" means DSP enhancing, the "S" means SPM, the "M" means MMU, and the "-F" means Float Point.

The complete C-SKY V1.0 (CK500&CK600) architecture is described in the CK500 & CK600 Reference Manual.

2.1.1 Registers

The C-SKY V1.0 ABI defines how to use the 16 general-purpose 32-bit registers of the C-SKY V1.0 processor. These registers'names range from r0 to r15.

C-SKY V1.0 Co-processor 0 has up to 31 control registers. These registers'names range from cr0 to cr30. The control registers are shown in Table 2.1 C-SKY Control Registers. These control registers can access with **mtcr/mfcr** instructions.

Table 2.1: C-SKY Control Registers

Register Use Convention		
Reg	**Name**	**Function**
cr0	psr, cr0	Processor Status Register
cr1	vbr, cr1	Vector Base Register
cr2	epsr, cr2	Shadow Exception PSR
cr3	fpsr, cr3	Shadow Fast Interrupt PSR
cr4	epc, cr4	Shadow Exception Program Counter
cr5	fpc, cr5	Shadow Fast Interrupt PC
cr6	ss0, cr6	Supervisor Scratch Register
cr7	ss1, cr7	Supervisor Scratch Register
cr8	ss2, cr8	Supervisor Scratch Register
cr9	ss3, cr9	Supervisor Scratch Register
cr10	ss4, cr10	Supervisor Scratch Register
cr11	gcr, cr11	Global Control Register
cr12	gsr, cr12	Global Status Register
cr13	cpidr	Product ID Register
cr14	cr14	Reserved
cr15	cr15	Coprocessor control register (CK610 only)
cr16	cr16	Reserved
cr17	cfr	Cache Flush Register
cr18	ccr	Cache Config Register
cr19	capr	Cachable and Access Popedom Register (MGU processor only)
cr20	pacr	Protected Area Config Register (MGU processor only)
cr21	prsr	Protected Area Select Register (MGU processor only)
cr22	cr22	MMU Index register (CK510M only, Reserved for others)
cr23	cr23	MMU random register (CK510M only, Reserved for others)
cr24	cr24	MMU EntryLo0 register (CK510M only, Reserved for others)
cr25	cr25	MMU EntryLo1 register (CK510M only, Reserved for others)
cr26	cr26	MMU EntryHi/Bad VPN register (CK510M only, Reserved for others)
cr27	cr27	MMU Context register (CK510M only, Reserved for others)
cr28	cr28	MMU Page mask register (CK510M only, Reserved for others)
cr29	cr29	MMU wired register (CK510M only, Reserved for others)
cr30	cr30	MMU control instruction register (CK510M only, Reserved for others)
cr31	cr31	Not existed

The ABI does not mandate the semantics of the C-SKY Hardware Accelerator Interface (HAI),because these semantics vary between C-SKY implementations based on particular chips.

CK600 subserial of C-SKY V1.0 provides instruction encodings which are up to other 15 co-processors (except for co-processor 0) to move, load, and store values. Co-processor 15 includes some control registers for SPM/MMU, etc. Please refer to CK600 manuals.

2.1.2 Fundamental Data Types

The C-SKY processor works with the following fundamental data types:

unsigned byte of eight bits

unsigned halfword of 16 bits

unsigned word of 32 bits

signed byte of eight bits

signed halfword of 16 bits

signed word of 32 bits

As the above list indicates, the data sizes are 8-bit bytes, 16-bit halfwords and 32-bit words. The mapping between these data types and the C language fundamental data types is shown in Table 2.2 Mapping of C Fundamental Data Types to the C-SKY.

Table 2.2: Mapping of C Fundamental Data Types to the C-SKY

Fundamental Data Types			
ANSI C	**Size**	**Align**	**C-SKY**
char	1	1	unsigned byte
unsigned char	1	1	unsigned byte
signed char	1	1	signed byte
short	2	2	signed halfword
unsigned short	2	2	unsigned halfword
signed short	2	2	signed halfword
long	4	4	signed word
unsigned long	4	4	unsigned word
signed long	4	4	signed word
int	4	4	signed word
unsigned int	4	4	unsigned word
signed int	4	4	signed word
enum	4	4	signed word
data point	4	4	unsigned word
function ptr	4	4	unsigned word
long long	8	8	signed word:unsigned word
unsigned long long	8	8	unsigned word[2]
float	4	4	unsigned word
double	8	8	unsigned word[2]
long double	8	8	unsigned word[2]

Memory access for unsigned byte-sized data is directly supported by the ld.b (load byte) and st.b (store byte) instructions. Signed byte-sized access requires the sextb (sign extension) instruction and the ld.b successively. Access to unsigned halfword-sized data is directly supported by the ld.h (load halfword) and st.h (store halfword) instructions. Signed halfword access requires the sexth (sign extension) instruction and the ld.h successively. Memory access to word-sized data is supported by through ld.w (load word) and st.w (store word) instructions. The ld.w suffices for both signed and unsigned word access because the operation sets all 32 bits of the loaded register.

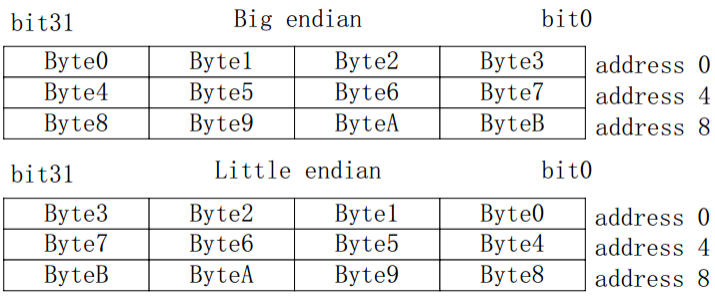

Figure 2.1: Data Organization in Memory

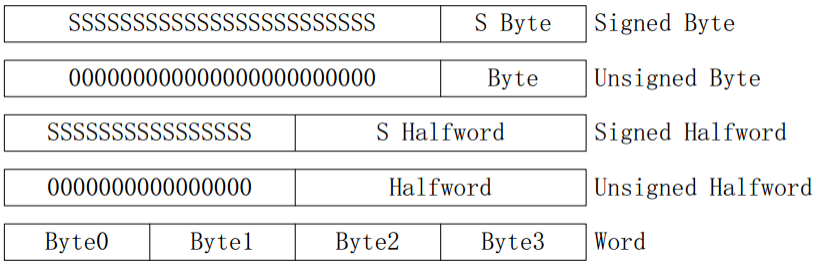

Figure 2.2: Data Organization in Registers

C-SKY supports the second standard's complement data formats. The operand size for each instruction is either explicitly encoded in the instruction (load/store instructions) or implicitly defined by the instruction operation (index operations, byte extraction). Typically, instructions operate on all 32 bits of the source operand(s) and generate a 32-bit result.

C-SKY memory may be viewed from either a Big Endian or Little Endian byte ordering perspective depending on the processor configuration (see Figure 2.1 Data Organization in Memory). In Big Endian mode (the default operating mode), the most significant byte (byte 0) of word 0 is located at address 0. For Little Endian mode, the most significant byte of word 0 is located at address 3. Fundamental data in memory is always naturally aligned, i.e., a *long* data is 4-byte aligned, a *short* data is 2-byte aligned.

Within registers, bits are numbered within a word starting with bit 31 as the most significant bit (see Figure 2.2 Data Organization in Registers). By convention, byte 0 of a register is the most significant byte regardless of Endian mode. This is only an issue when executing the xtrb[0-3] instructions.

The C-SKY processor currently does not support the long long int data type with 64-bit operations.However, compliant compilers must emulate the data type. The long long int data type, both signed and unsigned, is eight bytes in length and 8-byte aligned.

C-SKY processor currently does not support floating point data. However, compliant compilers must support its use. The floating point format to be used is the IEEE standard for *float* and *double* data types. Support for the *long double* data type is optional but compilers must conform to the IEEE standard format when provided.

Alignments are specifically chosen to avoid the possibility of access faults in the middle of an instruction (with the exception of load/store multiple).

2.1.3 Compound Data Types

Arrays, structures, unions, and bit fields have different alignment characteristics.

Arrays have the same alignment as their individual elements.

Unions and structures have the most restrictive alignment of their members. A structure containing a char data type,a short data type, and an int data type must have 4-byte alignment to match the alignment of the int data field. In addition, the size of a union or structure must be an integral multiple of its alignment. Padding must be applied to the end of a union or structure to make its size have a multiple of the alignment. Members must be aligned within a union or structure according to their type; padding must be introduced between members as necessary to meet this alignment requirement.

Bit fields neither exceed 32 bits nor cross a word (32 bit) boundary. Bit fields of signed short and unsigned short type are further restricted to 16 bits in size and cannot cross 16-bit boundaries. Bit fields of signed char and unsigned char types are further restricted to eight bits in size and cannot cross 8-bit boundaries. Zero-width bit fields pad to the next 8, 16, or 32 bit boundary for char, short,and int types respectively. Outside of these restrictions, bit fields are packed together with no padding each other.

Bit fields are assigned in big-endian order, i.e., the first bit field occupies the most significant bits while subsequent fields occupy lesser bits. Unsigned bit fields range from 0 to $2^w - 1$ where "w" is the size in bits. Signed bit fields range from -2^w to $2^w - 1$ Plain int bit fields are unsigned.

Bit fields impose alignment restrictions on their enclosing structure or union. The fundamental type of the bit field (e.g., char, short, int) imposes an alignment on the entire structure.

In the following example, the structure has alignment whose size is more than 4-byte and will have size of four bytes because the fundamental type of the bit fields is int, which requires 4-byte alignment. The second structure, named after less, requires only 1-byte alignment because that is the requirement of the fundamental type (char) used in that structure. The alignments are driven by the underlying type, not the width of the fields. These alignments are to be considered along with any other structure members. Struct named after careful requires 4-byte alignment; its bit fields only require 1-byte alignment, but the "fluffy" requires 4-byte alignment.

```
struct more {
  int first : 3;
  unsigned int second : 8;
};

struct less {
  unsigned char third : 3;
  unsigned char fourth : 8;
};

struct careful {
  unsigned char third : 3;
  unsigned char fourth : 8;
  int fluffy;
};
```

Fields within structures and unions begin on the next possible suitably aligned boundary for their data type. For non-bit fields, this is a suitable byte alignment. Bit fields begin at the next available bit offset with the following exception: the first bit field after a non-bit field member will be allocated on the next available byte boundary.

In the following example, the offset of the "c" is one byte. The structure itself has 4-byte alignment and is four bytes in size because of the alignment restrictions introduced by using the "int" underlying data type for the bit field.

```
struct s {
  int bf : 5;
  char c;
};
```

This behavior is consistent with the other UNIX System V Release 4 ABIs.

2.2 Function Calling Conventions

2.2.1 Register Assignments

Table 2.3 Register Assignments shows the required register mapping for function calls. Some registers, such as the stack pointer, have specific purposes, while others are used for local variables, or communicating function call arguments and returning values.

Certain registers are bound to their purpose because specific instructions use them. For instance, subroutine call instructions write the returning address into r15. The instructions shall be used to save and restore registers on entry and exit from a function use r0 as a base register, making it most appropriate for the stack pointer register.

Refer to "**Argument Passing**" for an explanation of argument words and how they are allocated. Refer to "**Return Values**" for an explanation of the returning buffer address.

Table 2.3: Register Assignments

Register Use Convention			
Name	**Software Name**	**Usage**	**Cross-Call Status**
r0	r0	Stack Pointer	Preserved
r1	r1	Scratch	Destroyed
r2	a0	Argument Word 1/Return Buffer Address	Destroyed/Preserved
r3-r7	a1-a5	Argument Word 2-6	Destroyed
r8-r13	l0-l5	Local	Preserved
r14	l10/gb	Local / GOT Base Address for PIC	Preserved
r15	lr	Link/Scratch	(Return Address)
r16-r19	l6-l9	Local	Preserved
r20-r25	t0-t5	Temporary registers used for expression evaluation	Destroyed
r31	tls	TLS register	Preserved
pc	pc	Prgogram counter, cannot be accessed directly by instructions.	
hi	hi	Multiply special register. Holds the most significant 32 bits of multiply	Destroyed
lo	lo	Multiply special register. Holds the least significant 32 bits of multiply	Destroyed

Float Point Registers

The CK600 of C-SKY V1.0 provides instruction encodings to move, load, and store values for up to 16 co-processors.

Co-processor 1 adds 32 32-bit floating-point general registers. Each even/odd pair of the 32 floating-point general registers can be used as either a 32-bit single-precision floating-point register or as a 64-bit double-precision floating-point register. For single-precision values, the even-numbered floating-point register holds the value. For double-precision values, the even-numbered floating-point register holds the least significant 32 bits of the value and the odd-numbered floating-point register holds the most significant 32 bits of the value. This is always true, regardless of the byte ordering conventions in use (big endian or little endian).

Floating-point data representation is specified in IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Table 2.4 Float Point Registers describes the conventions for using the floating-point registers.

Table 2.4: Float Point Registers

Register Use Convention		
Name	**Usage**	**Cross-Call Status**
fr0-fr31	Stack Pointer	Destroyed

Cross-Call Lifetimes

The 16 general-purpose registers are split between those preserved and those destroyed across function calls. This balances the need for callers to keep values in registers across calls against the need for simple

leaf subroutines to perform operations without allocating stack space and saving registers. The preserved registers are called non-volatile registers. The registers that are destroyed are called volatile registers.

Registers ranged from r8 to r14 are preserved because the load and store multiple instructions deal with registers ranged from N to 15. It is easy to save these registers in a single instruction.

The called subroutine can use any of the argument and scratch registers without concern for restoring their values. Preserved registers must be saved before being used and restored before returning to the caller. While the called function is not specifically required to save and restore r15, on entry r15 usually contains the return address, so the value must be preserved in order for execution to resume at the address of the instruction that follows the subroutine call.

Preserving r15 with the LDM and STM instructions is simple. Many implementations that save other non-volatile registers will also save r15. This is particularly useful when the called subroutine itself makes further subroutine calls.

The caller must preserve any essential data stored in argument registers and scratch registers. Data in these registers does not survive function calls. In particular, r1 is designated as a scratch register upon entry to a subroutine, and used to calculate stack frame adjustments for subroutines with large stack frames.

There is no register dedicated as a frame pointer. For non-alloca() functions, the frame pointer can always be expressed as an offset from the stack pointer. For alloca() functions and functions with very large frames, a frame pointer can be synthesized into one of the non-volatile registers.

Eliminating the dedicated frame pointer makes another register available for general use, with a corresponding improvement in generated code. This affects stack tracing for debugging. See 2.3 Runtime Debugging Support for additional information.

2.2.2 Stack Frame Layout

The stack pointer points to the bottom (low address) of the stack frame. Space at lower addresses than the stack pointer is considered invalid and may actually be unaddressable. The stack pointer value must always be a multiple of eight.

Figure 2.3 Stack Frame Layouts; First() calls Second() calls Third() shows typical stack frames for three functions, indicating the relative positions of local variables,parameters, and return addresses. The outbound argument overflow must be located at the bottom (low address) of the frame. Any incoming argument spill generated for varargs and stdarg processing must be at the top (high address) of the frame. Space allocated by **Alloca()** must reside between the outbound argument overflow and local variables areas.

The caller must stack argument variables that do not fit in the argument registers in the outbound argument overflow area. If all outbound arguments fit in registers, this area is not required. A caller may allocate argument overflow space sufficient for the worst-case call, use portions of it as necessary, and not change the stack pointer between calls.

The caller must reserve stack space for return variables that do not fit in the first two argument registers (e.g., structure returns). This return buffer area is typically located with the local variables. This space is typically allocated only in functions that make calls returning structures, and is not required.

The caller may stack the return address (r15) and the content of other local registers in the register save area upon entry to the called subroutine. If a called routine does not modify local variables (including r15), this area is not required.

Local variables that do not fit into the local registers are allocated space in the Local Variable area of the stack.If there are no such variables,this area is not required.

Beyond these requirements, a routine is free to manage its stack frame in any way desired.

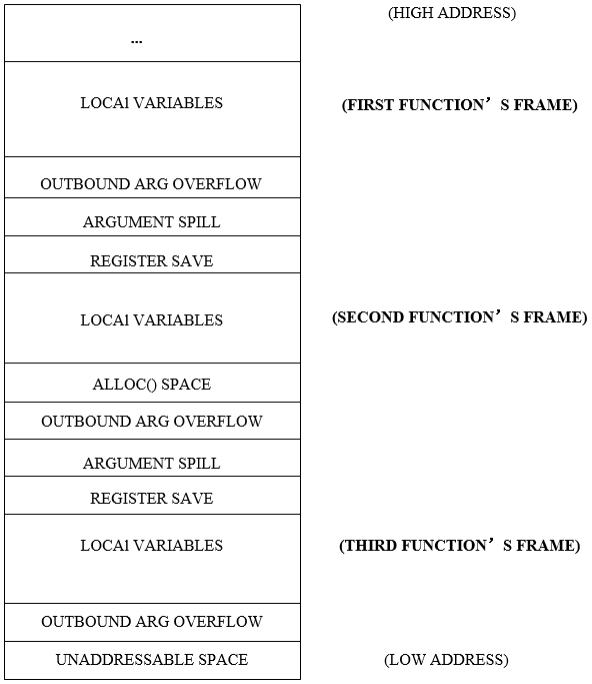

Figure 2.3: Stack Frame Layouts; First() calls Second() calls Third()

2.2.2.1 Extending the Stack

Stack maintenance is the responsibility of system software. In some environments, it may be valuable for compilers to probe the stack as they extend it in order to allow memory protection hardware to provide "guard pages".

2.2.3 Argument Passing

The C-SKY CPU uses six registers (r2–r7) to pass the first six words of arguments from the caller to the called routine. If additional argument space is required, the caller is responsible for allocating this space on the stack. This space (if needed by a particular caller) is typically allocated upon entry to a subroutine, reused for each of the calls made from that subroutine that have more arguments than fit into the six registers used for subroutine calls, and deallocated only at the caller‘s exit point. All argument overflow allocation and deallocation is the responsibility of the caller.

At entry to a subroutine, the first word of any argument overflow can be found at the address contained in the stack pointer. Subsequent overflow words are located at successively larger addresses.

2.2.3.1 Scalar Arguments

Arguments are passed using registers r2 through r7, with no more than one argument assigned per register. Argument values that are smaller than a 32-bit register occupy a full register.

In addition, small argument values are right justified and possibly extended within the register. Small signed arguments (e.g., shorts) are sign extended; small unsigned arguments (e.g., unsigned shorts) are zero extended, while other small values (e.g.,structures of less than four bytes) are not extended, leaving the upper bits of the register undefined. The caller is responsible for sign and zero extensions. Small arguments that

are passed via the argument overflow mechanism are placed in the overflow word with the same orientation they would have if passed in a register; a char is passed in th low-order byte of an overflow word. Such small overflow arguments need not be sign extended within the argument word as they would be if passed in a register.

Arguments larger than a register must be assigned to multiple argument registers as long as there are argument registers available. Arguments that would be aligned on eight-byte boundaries in memory (double, long double, long long, or structures or unions containing a double, long double or long long) must begin in an even numbered register.Once all the argument registers are used, or if there are not enough registers left to hold a large argument, the argument and any subsequent arguments must be placed in the overflow area described above.

Large arguments must not be split when there are too few argument registers to hold the entire argument.

The caller is responsible for allocating argument overflow space and for deallocating any space needed for argument overflow. The only argument space that may be allocated or deallocated by the called routine is space used to place the register arguments in memory. This may be necessary for stdargs or structure parameters.

Alignment is forced for atomic data types; fundamental data types are not split.

2.2.3.2 Structure Arguments

Structures passed as arguments can be partially or wholly passed through the argument registers. A structure argument may overflow onto the stack only when all argument registers are full. In these cases, the caller must adjust the stack pointer to allocate the overflow area.

Structure arguments that are smaller than 32 bits have their value right justified within the argument register. The unused upper bits within the register are undefined.

Structure arguments larger than 32 bits are packed into consecutive registers. Structures that are not integral multiples of 32 bits in size have their final bits left justified within the appropriate register. This allows those bits to be stored with a 32-bit operation and be adjacent to the preceding portion of the structure.

2.2.4 Variable Arguments

The stdarg C macros provide a mechanism to handle variable length argument lists. The caller might not know whether the called function handles variable arguments, so the called routine is responsible for handling the nuances of variable argument lists.

2.2.4.1 Spilling Register Arguments

Variable argument lists are most easily handled by spilling one or more of the register arguments so that they are adjacent to any overflow arguments that are on the stack at function entry. The typical sequence should extend the stack several words, spill the argument registers after the last named argument into this space, and then proceed with the normal prologues to allocate a stack frame and save any non-volatile registers.

The stdarg macros can use the address of the first stored argument register for the va_start macro. The va_arg macro advances this pointer by an amount appropriate to the size of the type specified.

2.2.4.2 Legacy Code Compatibility

The C-SKY CPU linkage convention provides a way for variable argument lists to be handled in a way that is compatible with legacy C code written for processors where the entire argument list is passed in memory.

The legacy behavior uses several more instructions, stack slots, and memory references than required by strict interpretation of the ANSI C standards. Tool generators must provide this legacy behavior as an option. It is not required as a default behavior.

To provide compatibility, the called function must spill all the argument registers, rather than just those beyond the registers that hold the named arguments. This is more pessimistic than required for the stdarg definitions, but provides the most compatibility.

Spilling is triggered for functions that take the address of any of their arguments.This allows non-standard varargs code (C code that works on processors with all arguments passed in memory) to run on the C-SKY CPU.

The spilled arguments are a snapshot of their values at the time the function is entered.This requirement does not force the compiler to generate code that keeps the "live" value of the parameters in memory. For example, the following would not be required to print out the value "4".

```
func(int a, int b, int c, ...)
{
  int *ip;
  use(c);
  ip = &b;
  ip++;
  *ip = 4;
  printf ("C now has value %d\n", c);
}
```

The compiler is free to keep the value of c live in a register. The only requirement is to save a snapshot of the parameter passing registers (e.g., ranging frome r2 to r7) during the function prologue.

2.2.5 Return Values

2.2.5.1 Scalar Values

Subroutines return values in the argument registers. Return values smaller than 32 bits occupy a full register. These must be right justified and zero or sign extended to 32 bits before return (refer to "**Scalar Arguments**"). Return values of 32 bits or fewer are returned in register r2.

Return values between 33 and 64 bits are returned in the register pair r2/r3. The portion of the data that would reside at a lower address if stored in memory is in r2. For example, r2 would contain the most significant 32 bits of the long long data type.

Return values larger than eight bytes are treated as structure return values and are returned through memory. The return value is placed in a caller-supplied buffer. The buffer address is passed from the caller to the called routine as a hidden first argument in register r2.

2.2.5.2 Structure Values

Structures can be returned in one of two ways.

Small structures (eight bytes or fewer) are returned in the register pair r2/r3. If the structure consists of four or fewer bytes, the value is returned in r2, right justified. This matches the way it would be justified when passed as an argument. If the structure consists of five to eight bytes, the first four bytes are returned in r2 and the trailing portion of the structure is returned left justified in r3.

This alignment is chosen to generate good code for code sequences such as that wom takes a structure argument of the same type returned by bat. The only work required is to perhaps change registers if the call to wom has the structure in some place other than r2/r3.


```
wom(..., bat(), ...)
```

Structures larger than eight bytes are placed in a buffer provided by the caller. The caller must provide for a buffer of sufficient size; the buffer is typically allocated on the stack to provide re-entrancy and to avoid any race conditions where a static buffer may be overwritten. The address of the buffer is passed to the called function as a hidden first argument and arrives in register r2. The normal arguments start in register r3 instead of in r2, within the fundamental data type constraints.

The caller must provide this buffer for large structures even when the caller does not use the return value (e.g., the function was called to achieve a side-effect). The called routine can thus assume that the buffer pointer is valid and need not check the pointer value passed in r2.

When r2 is used to pass a buffer address, the called routine must preserve the value passed through r2. The caller can thus assume that r2 is preserved when the buffer address of a large structure is passed in r2. This is similar to the way in which strcat and memcpy return their respective destination addresses.

Often, the temporary buffer that is used for such structure returns is immediately used as a source for a memcpy to a final destination. For example, the sequence will often be compiled with sfunc returning into a temporary buffer, which is immediately copied into s. Although the caller must know the address of the temporary buffer in order to provide it to the called routine, the address need not be recalculated. In turn, the called routine can use the address to copy the results into the temporary buffer using memcpy, which returns the destination address (e.g., r2 has the desired value), or passes it to in-line code which uses r2 as a base register.

```
struct s {...} s, sfunc();
s = sfunc();
```

2.3 Runtime Debugging Support

The most difficult aspect of C-SKY CPU debugging is stack tracing. Tracing is complicated because the linkage convention does not mandate a frame pointer register and does not provide any back-chain construct. This section describes rules for generating function prologues that can be easily decoded by a debugger to determine the size of a stack frame, the location of the return address, and the location of any saved non-volatile registers.

2.3.1 Function Prologues

Function prologues acquire stack space needed by the function to store local variables. This includes space the function uses to save non-volatile registers. Prologue instruction sequences can take a number of forms. A set of working assumptions about function prologues follows.

The function prologue is the only place in the function that acquires stack space, other than later calls to alloca().

The function prologue uses only the following classes of instructions.

- subi r0,imm (Note that this might appear multiple times in a prologue)
- stm rn-r15,(r0)
- st.w rx,(r0,disp)
- instructions that set and modify r1.
- These are presumed to establish values for a relatively large frame. This sequence includes one of the

following instructions:

```
lr.w r1, imm
movi r1, imm
bgeni r1, imm
bmaski r1, imm
```

followed by zero or more of:

```
addi r1, imm
subi r1, imm
rsubi r1, imm
not r1
rotli r1, imm
bseti r1, imm
bclri r1, imm
ixh r1,r1
ixw r1,r1
```

followed by:

```
sub r0, r1
```

Whether lrw or the other sequence is used, the r1 value is subtracted from r0 to increase stack space. While this sequence is allowed to occur multiple times, code generators should generate a single literal of the appropriate value (e.g., summing two constants) rather than perform two subtractions.

- mov rn,r0

This is optional support for traceback through alloca()-using functions, and also marks the final instruction in the prologue.

The function prologue is organized roughly as:

- If stdarg, acquire space to store volatile registers; store volatile registers.
- Acquire space to store non-volatile registers.
- Store non-volatile registers that may be modified in this function.
- Acquire any additional stack space required. This space acquisition might be folded in with earlier ones if the total space allocated is no more than 32 bytes.
- If needed in this function, copy the stack pointer into one of the non-volatile registers to act as a frame pointer.
- Larger frames should allocate the register save space and then allocate the remainder of the required stack space rather than perform a single large stack acquisition. If the stack is acquired in a single allocation before the non-volatile registers are saved, then another base register is needed to reach the location for the stored registers. The prologue recognition code in the debugger does not recognize using alternate base registers to store the non-volatile registers as being part of the prologue.

This sequence allows the stack pointer to be modified several times.

2.3.2 Stack Tracing

Stack tracing for the CKCORE depends on the ability to determine the entry point for a function, given a PC value in that function. Since there are no unique prologue-only patterns in the instruction stream that can be identified by scanning backwards from the current PC, a symbol table for the executable file must be present. The symbols need not be complete DWARF information.

Placing a specific byte pattern just before the prologue is not sufficient to identify the beginning of a function because the pattern can also appear within the body of the function as part of a literal table. In code-size sensitive environments, the extra space consumed by such a byte pattern is undesirable.

The stack tracing code iteratively performs the following:

1. Get the current PC
2. Find the beginning of the containing function. Stop if this can't be determined.
3. Decode the prologue starting at the function's entry.
4. Determine the "top of frame" from the framesize information described in the prologue. This is either an adjustment to the stack pointer or a "pseudo-frame pointer" if the prologue ends with a frame pointer generating instruction.
5. Recover stored non-volatile registers based on the offsets described in the prologue.Repeat for the next frame.

CHAPTER 3

High-level Language Issues

3.1 C Preprocessor Predefines

All C language compilers must predefine the symbol ___CKCORE___, ___ckcore___, with the value "1" to indicate that the compiler targets the C-SKY processor. In the future, this value may be changed to correspond to different versions of the chip.

3.2 C In-Line Assembly Syntax

3.2.1 Overview

We can instruct the compiler to insert the code of a function into the code of its callers, to the point where actually the call is to be made. Such functions are inline functions. Sounds similar to a Macro, Indeed there are similarities.

This method of inlining reduces the function-call overhead. And if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function‘s code needs to be included. The effect on code size is less predictable, it depends on the particular case. To declare an inline function, we‘ve to use the keyword inline in its declaration.

Inline assembly is important primarily because of its ability to operate and make its output visible on C variables. Because of this capability, "asm" works as an interface between the assembly instructions and the "C" program that contains it.

3.2.2 Basic In line

format of basic inline assembly is very much straight forward. Its basic form is:

asm("assembly code");

Example:

```
/* moves the contents of r1 to r0 */
asm("mov r0, r1");
/*move 0x2 to the r2 */
__asm__("movi r2, 0x2");
```

You might have noticed that here I‘ve used asm and ___asm___. Both are valid. We can use ___asm___ if the keyword asm conflicts with something in our program. If we have more than one instructions, we write one per line in double quotes, and also suffix a ‘n‘ and ‘t‘ to the instruction. This is because compiler sends each instruction as a string to assembler and by using the newline/tab we send correctly formatted lines to the assembler.

Example:

```
__asm__ ("mov r8, r0\n\t"
        "mov r1, r9\n\t"
        "stw r1, (r8,4)\n\t");
```

If in our code we touch (ie, change the contents) some registers and return from asm without fixing those changes, something bad is going to happen. This is because compiler have no idea about the changes in the register contents and this leads us to trouble, especially when compiler makes some optimizations. It will suppose that some register contains the value of some variable that we might have changed without informing compiler, and it continues like nothing happened. What we can do is either use those instructions having no side effects or fix things when we quit or wait for something to crash. This is where we want some extended functionality. Extended asm provides us with that functionality.

3.2.3 Extended asm

In basic inline assembly, we had only instructions. In extended assembly, we can also specify the operands. It allows us to specify the input registers, output registers and a list of clobbered registers. It is not mandatory to specify the registers to use, we can leave that head ache to compiler and that probably fit into compiler‘s optimization scheme better. Anyway the basic format is:

```
asm ( assembler template
    : output operands /* optional */
    : input operands /* optional */
    : list of clobbered registers /* optional */
    );
```

The assembler template consists of assembly instructions. Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands would go.

Example:

```
asm ("cmpei %0, 0\n\t"
    "bt 1\n\t"
    "stw %0, (%1, 0)"
    "1:\n\t"
    : /* no output registers */
    : "r" (count), "r"(dest)
```

(continues on next page)

(continued from previous page)

```
    : "memory"
    );
```

The above inline fills if count!=0, store count into the memory which dest point to. It also says to compiler that, the contents of memory is changed. Let us see one more example to make things more clearer.

```
int a=10, b;
asm ("mov r1, %1;
    mov %0, r1"
    :"=r"(b) /* output */
    :"r"(a) /* input */
    :"r1" /* clobbered register */
    );
```

Here what we did is we made the value of "b" equal to that of "a" using assembly instructions. Some points of interest are:

- "b" is the output operand, referred to by %0 and "a" is the input operand, referred to by %1.
- "r" is a constraint on the operands. We'll see constraints in detail later. For the time being, "r" says to COMPILER to use any register for storing the operands. output operand constraint should have a constraint modifier "=". And this modifier says that it is the output operand and is write-only.
- There are two %'s prefixed to the register name. This helps COMPILER to distinguish between the operands and registers. operands have a single % as prefix.
- The clobbered register r1 after the third colon tells COMPILER that the value of r1 is to be modified inside "asm",so COMPILER won't use this register to store any other value.

When the execution of "asm" is complete, "b" will reflect the updated value, as it is specified as an output operand. In other words, the change made to "b" inside "asm" is supposed to be reflected outside the "asm".

3.2.3.1 Assembler Template

The assembler template contains the set of assembly instructions that gets inserted inside the C program. The format is like: either each instruction should be enclosed within double quotes, or the entire group of instructions should be within double quotes. Each instruction should also end with a delimiter. The valid delimiters are newline(n) and semicolon(;). "n" may be followed by a tab(t). Operands corresponding to the C expressions are represented by %0, %1 … etc.

3.2.3.2 Operands

C expressions serve as operands for the assembly instructions inside "asm". Each operand is written as first an operand constraint in double quotes. For output operands, there'll be a constraint modifier also within the quotes and then follows the C expression which stands for the operand. ie,"constraint" (C expression) is the general form. For output operands an additional modifier will be there. Constraints are primarily used to decide the addressing modes for operands. They are also used in specifying the registers to be used.

If using more than one operand, they are separated by comma.

In the assembler template, each operand is referenced by numbers. Numbering is done as follows. If there are a total of n operands (both input and output inclusive), then the first output operand is numbered 0, continuing in increasing order, and the last input operand is numbered n-1. The maximum number of operands is as we saw in the previous section.

Output operand expressions must be values. The input operands are not restricted like this. They may be expressions. The extended asm feature is most often used for machine instructions the compiler itself does not know as existing ;-). If the output expression cannot be directly addressed (for example, it is a bit-field), our constraint must allow a register. In that case, COMPILER will use the register as the output of the asm, and then store that register contents into the output.

As stated above, ordinary output operands must be write-only; COMPILER will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm also supports input-output or read-write operands.

So now we concentrate on some examples. We want to add a number by 5. For that we use the instruction add.

```
asm ("mov %0, %1\n\t"
    "cmplt %0, %0\n\t"
    "addc %0, 5"
    : "=r" (five_times_x)
    : "r" (x)
    );
```

Here our input is in ‘x‘. We didn‘t specify the register to be used. COMPILER will choose some register for input, one for output and does what we desired. If we want the input and output to reside in the same register, we can instruct COMPILER to do so. Here we use those types of read-write operands. By specifying proper constraints, here we do it.

```
asm ("cmplt %0, %0\n\t"
      "addc %0, 5"
    : "=r" (five_times_x)
    : "0" (x)
    );
```

Now the input and output operands are in the same register. But we don‘t know which is that register.

In all the two examples above, we didn‘t put any register to the clobber list. why? In the first two examples, COMPILER decides the registers and it knows what changes happen.

3.2.3.3 Clobber List

Some instructions clobber some hardware registers. We have to list those registers in the clobber-list, ie the field after the third “:” in the asm function. This is to inform compiler that we will use and modify them ourselves. So compiler will not assume that the values it loads into these registers will be valid. We shouldn‘t list the input and output registers in this list. Because, compiler knows that “asm” uses them (because they are specified explicitly as constraints). If the instructions use any other registers, implicitly or explicitly (and the registers are not present either in input or in the output constraint list), then those registers have to be specified in the clobbered list.

If our instruction can alter the condition code register, we have to add “cc” to the list of clobbered registers.

If our instruction modifies memory in an unpredictable fashion, add “memory” to the list of clobbered registers. This will cause compiler to not keep memory values cached in registers across the assembler instruction. We also have to add the volatile keyword if the memory affected is not listed in the inputs or outputs of the asm.

We can read and write the clobbered registers as many times as we like. Consider the example of multiple instructions in a template; it assumes the subroutine _foo accepts arguments in registers r1 and r2.


```
asm ("movl r2, %0 \n\t
      movl r3, %1 \n\t
      jsri _foo"
    : /* no outputs */
    : "g" (from), "g" (to)
    : "r2", "r3"
    );
```

3.2.3.4 Volatile

If you are familiar with kernel sources or some beautiful code like that, you must have seen many functions declared as volatile or ___volatile___ which follows an asm or ___asm___.

If our assembly statement must execute where we put it, (i.e. must not be moved out of a loop as an optimization), put the keyword volatile after asm and before the ()'s. So to keep it from moving, deleting and all, we declare it as

asm volatile (... : ... : ... : ...);

Use ___volatile___ when we have to be very much careful.

If our assembly is just for doing some calculations and doesn't have any side effects, it's better not to use the keyword volatile. Avoiding it helps compiler in optimizing the code and making it more beautiful.

In the section Some Useful Recipes, there are many examples for inline asm functions.There we can see the clobber-list in detail.

3.2.3.5 Constraints

Constraints can say whether an operand may be in a register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values (ie range of values) it may have.... etc.

There are a number of constraints of which only a few are used frequently. We'll have a look at those constraints.

1. Register operand constraint

When operands are specified using this constraint, they get stored in General Purpose Registers(GPR). Take the following example:

```
asm ("mov %0, %1\n"
    :"=r"(myval)
    : "=r"(inval));
```

Here the variable myval is kept in a register, the value in inval is copied onto that register. When the "r" constraint is specified, compiler may keep the variable in any of the available GPRs. To specify the register, you must directly specify the register names by using specific register constraints. They are:

r	r0-r15
a	r1-r14
b	r1
d	r1–r13 (FP CP)
h	hi(ck510e/ck520)
l	lo(ck510e/ck520)
u	r1–r7 (CK610)

For example:

```
__asm__ __volatile__ ("mthi %1"
                      :"=h"(j)
                      :"r"(i));
```

2. Memory operand constraint(m)

When the operands are in the memory, any operations performed on them will occur directly in the memory location, as opposed to register constraints, which first store the value in a register to be modified and then write it back to the memory location. But register constraints are usually used only when they are absolutely necessary for an instruction or they significantly speed up the process. Memory constraints can be used most efficiently in cases where a C variable needs to be updated inside "asm" and you really don't want to use a register to hold its value. For example, the value of input is stored in the memory location **loc**:

```
asm("stw %1, %0"
    :"m"(loc)
    :"r"(input));
```

3. Matching constraints

In some cases, a single variable may serve as both the input and the output operand. Such cases may be specified in "asm" by using matching constraints.

```
asm ("inct %0" :"=a"(var):"0"(var));
```

This constraint can be used:

- In cases where input is read from a variable or the variable is modified and modification is written back to the same variable.
- In cases where separate instances of input and output operands are not necessary.

The most important effect of using matching restraints is that they lead to the efficient use of available registers.

Some other constraints used are:

1. "m" : A memory operand is allowed, with any kind of address that the machine supports in general.
2. "o" : A memory operand is allowed, but only if the address is offsettable. ie, adding a small offset to the address gives a valid address.
3. "V" : A memory operand that is not offsettable. In other words, anything that would fit the "m" constraint but not the "o" constraint.
4. "i" : An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.
5. "n" : An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use "n" rather than "i".
6. "g" : Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.
7. "r" : Register operand constraint, look table given above.
8. "a" : Register operand constraint, look table given above.
9. "b" : Register operand constraint, look table given above.
10. "d" : Register operand constraint, look table given above.

11. “h” : Register operand constraint, look table given above.
12. “l” : Register operand constraint, look table given above.
13. “u” : Register operand constraint, look table given above.
14. “I” : Constant in range 0 to 127(>= 0; <= 127).
15. “J” : Constant in range 1 to 32.
16. “K” : Constant in range 0-31.
17. “L” : Constant in range -32 to -1
18. “M” : exact_log2 (VALUE) >= 0.
19. “N” : (((VALUE)) == -1 || exact_log2 ((VALUE) + 1) >= 0).

While using constraints, for more precise control over the effects of constraints, Compiler provides us with constraint modifiers. Mostly used constraint modifiers are

- “=” : Means that this operand is write-only for this instruction; the previous value is discarded and replaced by output data.
- “&” : Means that this operand is an earlyclobber operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address. An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written.

3.2.4 Examples

- add two numbers

```
int main(void)
{
        int foo = 10, bar = 15;
        __asm__ __volatile__(" cmplt %1, %1\n\t"
                             "addc %1,%2"
                             :"=a"(foo)
                             :"0"(foo), "b"(bar)
                             );
        printf("foo+bar=%d\n", foo);
        return 0;
}
```

The “=” sign shows that it is an output register.

```
__asm__ __volatile__("addu %0,%1\n"
                     : "=m" (my_var)
                     : "ir" (my_int), "m" (my_var)
                     : * no clobber-list */
                     );
```

In the output field, “=m” says that my_var is an output and it is in memory. Similarly, “ir” says that, my_int is an integer and should reside in some register (recall the table we saw above). No registers are in the clobber list.

- Memory access


```
int main (int argc, char **argv)
{
        int i;
        char kk[10];
        char ch;
        __asm__ __volatile__ ("ldw %0, %1"
                              : "=r"(i)
                              : "m"(argc));
        __asm__ __volatile__ ("stw %1, %0"
                              : "=o"(kk)
                              : "r"(i));
        __asm__ __volatile__ ("stw %0, %1"
                              : "=r"(i)
                              : "V"(argc));
        __asm__ __volatile__ ("stw %1, %0"
                              : "=m"(kk[5])
                              : "r"(ch));
        return 0;
}
```

- Linux System calls

In Linux, system calls are implemented using inline assembly. All the system calls are written as macros. For example, a system call with 1 arguments is defined as a macro as shown below.

```
#define _syscall1(type, name, atype, a)                    \
type name(atype a)                                         \
{                                                          \
  register long __name __asm__("r1") = __NR_##name;        \
  register long __res __asm__("r2") = a;                   \
  __asm__ __volatile__ ("trap 0\n\t"                       \
                        : "=r" (__res)                     \
                        : "r" (__name),                    \
                          "0" (__res)                      \
                        : "r1", "r2");                     \
  if ((unsigned long)(__res) >= (unsigned long)(-125))     \
  {                                                        \
    *__errno_location () = -__res;                         \
    __res = -1;                                            \
  }                                                        \
return (type)__res;                                        \
}
```

Whenever a system call with 1 arguments is made, the macro shown above is used to make the call. The syscall number is placed in *r1*, then each parameters in *r2*, And finally "trap 0" is the instruction which makes the system call work. The return value can be collected from *r2*.

Note: "___errno_location()" is a function call, and will return the result in **r2**, and function call for CKCORE will clobber **r1**–**r7**, but 'register long ___res ___asm___("r2")' use "r2" also, so there is a bug in the above example, It must be:

```
{
long __error = __res;
*__errno_location () = -__error;
__res = -1;
}
```

3.3 Name Mapping

Externally visible names in the C language must be mapped through to assembly language without change. For example, the following

```
void testfunc() { return;}
```

generates assembly code similar to the following fragment

```
testfunc:
        jmp r15
```

CHAPTER 4

Object File Formats

C-SKY CPU tools use ELF 1.2 object file formats and DWARF 2.0 debugging information formats, as described in System V Application Binary Interface, from The SantaCruz Operation, Inc. ELF and DWARF provide a suitable basis for representing the information needed for embedded applications. This section describes particular fields in the ELF and DWARF formats that differ from the base standards for those formats.

4.1 ELF Header

e_machine

The e_machine member of the ELF header contains the decimal value 39 (hexadecimal 0x27) which is defined as the name EM_CKCORE.

e_ident

For file identification in e_ident[] must be the values listed in Table 4.1 e_ident Field values.

Table 4.1: e_ident Field values

C-SKY e_ident Fields		
e_ident[EI_CLASS]	ELFCLASS32	For all 32-bit implementations
e_ident[EI_DATA]	ELFDATA2LSB or ELFDATA2MSB	The choice will be governed by the default data order in the execution environment. ELFDATA2LSB: Little Endian ELFDATA2MSB: Big Endian

e_flags

In ABI v0.1, the ELF header e_flags member contains zero, because the C-SKY processor family defines no flags at that time. Now e_flags are shown in Table 4.2 C-SKY-Specified e_flags. Unallocated bits are reserved to future revisions of this specification.

Table 4.2: C-SKY-Specified e_flags

Name	Value	Meaning
EF_CSKY_ABIMASK	0xFF000000	The integer value formed by these four bits identify extensions to the C-SKY ABI v0.1; In ABI v0.1, the ELF header e_flags member contains zero, because the C-SKY processor family defines no flags at that time; Non-zero values indicate the object file or executable contains program text tha uses newer version C-SKY ABI than C-SKY ABI v0.1
EF_CSKY_PIC	0x00000001	This bit is asserted when the file contains position independent code that can be reloacated in memory
EF_CSKY_CPIC	0x00000002	This bit is asserted when the file contains code that follows standard calling sequence rules for calling position independent code. The code in this file is not necessarily position independent. The EF-CSKY-PIC and EF_CSKY_CPIC flags must be mutually exclusive.

4.2 Section Layout

4.2.1 Section Alignment

The object generator (compiler or assembler) provides alignment information for the linker. The default alignment is eight bytes. Object producers must ensure that generated objects specify required alignment. For example, an object file must reflect the fact that four-byte alignment is required in the data section.

4.2.2 Section Attributes

Table 4.3 C-SKY CPU Section Attributes defines section attributes that are available for C-SKY CPU tools. These attributes are additions to the ELF standard flags shown in Table 4.4 ELF Section Attributes.

Table 4.3: C-SKY CPU Section Attributes

CKCORE Section Attribute Flags	
Name	**Value**
SHF_CKCORE_NOREAD	0x80000000

The SHF_CKCORE_NOREAD attribute allows the specification of code that is executable but not readable. Plain ELF assumes that all segments have read attributes, which is why there is no read permission attribute in the ELF attribute list. In embedded applications, “execute-only” sections that allow hiding the implementation are often desirable.

Table 4.4: ELF Section Attributes

ELF Section Attribute Flags	
Name	**Value**
SHF_WRITE	0x00000001
SHF_ALLOC	0x00000002
SHF_EXECINSTR	0x00000004

4.2.3 Special Sections

Various sections hold program and control information. Table 4.4 shows sections used by the system, the indicated types, and attributes. These are additions to ELF standards shown in Table 4.5. The ELF standard reserves section names beginning with a period (“.”), but applications may use those sections if their existing meanings are satisfactory.

C-SKY currently support PIC technique, when compiling PIC, the link editor will create .got and .plt sections, see “**Global Offset Table**” and “**Procedure Linkage Table**”.

Table 4.5: C-SKY CPU Tools Special Sections

C-SKY Section Names for PIC		
Name	**Type**	**Attributes**
.got	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR

Note: It is strongly recommended that read-only constants, such as string literals, be placed into the .rodata section instead of the .text section. The space that these add to .text can have a severe impact on addressability, requiring the use of larger branch instructions and reducing the chances for sharing of values in literal tables.

Table 4.6: ELF Sections

ELF Reserved Section Names		
Name	**Type**	**Attributes**
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_NOBITS	none
.dynamic	SHT_PROGBITS	–
.dynstr	SHT_NOBITS	SHF_ALLOC
.dynsym	SHT_PROGBITS	SHF_ALLOC
.fini	SHT_NOBITS	SHF_ALLOC+SHF_EXECINSTR
.hash	SHT_PROGBITS	SHF_ALLOC
.init	SHT_NOBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	–
.line	SHT_NOBITS	none
.note	SHT_PROGBITS	none
.rel*	SHT_REL	–
.rela*	SHT_RELA	–
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_SYMTAB	none
.strtab	SHT_SYMTAB	–
.symtab	SHT_SYMTAB	–
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

4.3 Symbol Table Format

There are no C-SKY CPU symbol table requirements beyond the base ELF standards.

4.4 Relocation Information Format

4.4.1 Relocation Fields

Relocation entries describe how to alter the instruction and data relocation fields as shown in Table 4.7 Relocation Fields. The choice of the relocation type numbers as encoded in the ELF object file is defined in Table 4.8 Relocation Type Encodings.

Table 4.7: Relocation Fields

word32	This specifies a 32-bit field occupying four bytes. This address is NOT required to be 4-byte aligned.
disp8	This corresponds to the scaled 8-bit displacement addressing mode. The relocation is the low-order 8 bits of the 16 bits addressed in the relocation type. jsri, jmpi, & lrw use this 8-bit displacement addressing mode. But this displacement addressing mode would never be relocated.
disp11	This corresponds to the scaled 11-bit displacement addressing mode. The relocation is the low-order 11 bits of the 16 bits addressed in the relocation type. br, bf, bt & bsr use this 11-bit displacement addressing mode.
pcword32	This specifies a 32-bit field occupying four bytes. This address is NOT required to be 4-byte aligned.

The object file supports the 32-bit relocations for 32-bit data (addressing constants in memory). Both absolute and PC-relative relocations are defined.

Note that the 32 bits where the relocation is to be applied need not be on a 32-bit boundary. The relocation entry points to the address of the 32 bits to be adjusted by the relocation entry. The relocation adds the appropriate value (either the 32-bit value or the 32-bit displacement) to the existing contents of the 32 bits at that address.

A packed data structure can cause a 32-bit relocation to be misaligned in the object file. This might be done with a C compiler extension, or by means of hand-crafted assembly, in order to save data space (but the misaligned data must be accessed piece-wise to avoid alignment exceptions). The linker must be able to deal with this case.

Scaled 11-bit displacement mode is used in br, bf, bt, and bsr instructions. The 11-bit value indicates the number of halfwords from PC+2 to the target address. The relocation entry must point to the 16-bit instruction that contains the displacement.

Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the linker merges one or more relocatable files to form the output. It first determines how to combine and locate the input files; then it updates the symbol values, and finally it performs the relocation.

Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A

This means the addend used to compute the value of the relocatable field.

B

This means the base address at which a shared object has been loaded into memory during execution. Generally a shared object file is built with a 0 base virtual address, but the execution address will be different.

P

This means the place (section offset or address) of the storage unit being relocated (computed using r_offset).

S

This means the value of the symbol whose index resides in the relocation entry, unless the the symbol is STB_LOCAL and is of type STT_SECTION in which case S represents the original sh_addr minus the final sh_addr.

G

This means the offset into the global offset table at which the address of the relocation entry symbol resides during execution. See "PIC Examples" and "Global Offset Table" for more information.

GOT

This means the address of the global offset table. See "**Global Offset Table**".

L

This means the place(section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See "**Procedure Linkage Table**" below for more information.

A relocation entry r_offset value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. Because C-SKY CPU uses only Elf32_Rela relocation entries, the relocated field does not hold the addend, but relocation entry holds it.

4.4.2 Relocation Types

This section describes values and algorithms used for relocations. In particular, it describes values the compiler/assembler must leave in place and how the linker modifies those values.

Table 4.8 Relocation Type Encodings shows semantics of relocation operations. Key S indicates the final value assigned to the symbol referenced in the relocation record. Key A is the addend value specified in the relocation record. Key P indicates the address of the relocation (e.g., the address being modified).

Table 4.8: Relocation Type Encodings

Name	Value	Field	Calculation
R_CKCORE_NONE	0	none	none
R_CKCORE_ADDR32	1	word32	S+A
R_CKCORE_PCRELIMM8BY4	2	dis8	((S+A-P) >>2) & 0xff
R_CKCORE_PCRELIMM11BY2	3	disp11	((S+A–P)>>1)&0x7ff)
R_CKCORE_PCRELIMM4BY2	4	none	Unsupported, deleted
R_CKCORE_PCREL32	5	word32	S+A–P
R_CKCORE_PCRELJSR_IMM11BY2	6	disp11	((S+A–P)>>1)&0x7ff)
R_CKCORE_GNU_VTINHERIT	7	–	??
R_CKCORE_GNU_VTENTRY	8	–	??
R_CKCORE_RELATIVE	9	word32	B+A
R_CKCORE_COPY	10	none	none
R_CKCORE_GLOB_DAT	11	word32	S
R_CKCORE_JUMP_SLOT	12	word32	S
R_CKCORE_GOTOFF	13	word32	S + A – P
R_CKCORE_GOTPC	14	word32	GOT + A - P
R_CKCORE_GOT32	15	word32	G + A
R_CKCORE_PLT32	16	word32	L + A - P

4.4.2.1 Static Relocations in Data Sections

R_CKCORE_ADDR32

In DATA sections, absolute 32-bit relocation adds the relocated symbols value to the existing content of the location specified. Consider the example

```
        data
D1:
        .long 0x10
D2:
        .long SYMBOL+ 1234 <- R_CKCORE_ADDR32 for this word32 field.
```

The object file emitted by the compiler has a relocation entry for SYMBOL that references the address of this word. The existing content of the 32 bits at the specified address are overwritten with the new value.

So in the example, the offset of the relocation is 4, symbol value is SYMBOL in.data section or other section, addend is 1234.

4.4.2.2 Static C-SKY V1 Relocations in Text Sections

R_CKCORE_ADDR32

In TEXT sections, absolute 32-bit relocation adds the relocated symbols value to the existing content of the location specified. Consider the example

```
.text
...
lrw r1, symbol+1234 <- R_CKCORE_ADDR32 for this word32 field.
...
jsri printf <- R_CKCORE_ADDR32 for this word32 field.
```

The object file emitted by the compiler has a relocation entry for symbol that references the address of this word. The existing content of the 32 bits at the specified address are overwritten with the new value.

So for the second relocation entry in the example, the offset is the [**jsri located PC - .text base address**], symbol value is **printf**, addend is **0**.

R_CKCORE_PCRELIMM8BY4

Occur when jmpi/jsri/lrw instructions reference a target that is in a symbol which is identified in a new section. For examble: (jsri has the same case)

```
text
mycode:
        ...
        lrw r1, [myconst] /* r1 = 0x12345678 */
        ...
        .data /* A new section */
myconst:
        .long 0x12345678
```

It is a obsoleted relocation type.

R_CKCORE_PCRELIMM11BY2

Occur when br, bf, bt, and bsr instructions (typically bsr) reference a target that is not in the current object file. They can also occur when the target is in a separate section of the same object file, but these occurrences must be resolved by the compiler/assembler and not appear as relocation entries.

```
        import __exit
        .export tbsr
        .text
tbsr:
        bsr __exit
```

The relocation is calculated as shown in **Table 4-8 Relocation Type Encodings**. The existing contents of the low-order 11 bits of the instruction are overwritten with the newly calculated displacement.

Note: he bsr instruction encoding is the distance from PC+2 to the target. This adjustment ust be made in the compiler/assembler. The emitted relocation record for a bsr to symbol must be to X+(–2); in other words, the symbol must be X and the addend field of the elocation record must contain –2.

R_CKCORE_PCRELIMM4BY2

It is a obsoleted relocation type.

This relocation come from MCORE “loopt” instruction, and C-SKY CPU has no any “loopt”, so this relocation should not appear in any C-SKY CPU binary files.

R_CKCORE_PCREL32

This relocation type computes the difference between a symbol‘s value and the address or section offset to be relocated.

It is a obsoleted relocation type for C-SKY V1.0. May be used by C-SKY V2.0.

R_CKCORE_PCRELJSR_IMM11BY2

Like PCRELIMM11BY2, this relocation indicates that there is a “jsri” at the specified address. There is a separate relocation entry for the literal pool entry that it references (So there are 2 relocation entry for “jsri” when assemble with –jsri2bsr option), but we might be able to change the jsri to a bsr if the target turns out to be close enough [even though we won’t reclaim the literal pool entry, we’ll get some runtime efficiency back]. Note that this is a relocation that we are allowed to safely ignore.

4.4.2.3 Dynamic Relocations

R_CKCORE_RELATIVE

The linker editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_CKCORE_COPY

R_CKCORE_COPY may only appear in executable objects where e_type is set to ET_EXEC. The effect is to cause the dynamic linker to locate the target symbol in a shared library object and then to copy the number of bytes specified by the st_size field to the place. The address of the place is then used to pre-empt all other references to the specified symbol. It is an error if the storage space allocated in the executable is insufficient to hold the full copy of the symbol. If the object being copied contains dynamic relocations then the effect must be as if those relocations were performed before the copy was made.

Note: R_CKCORE_COPY is normally only used in SVr4 type environments where the executable is not position independent and references by the code and read-only data sections cannot be relocated dynamically to refer to an object that is defined in a shared library. The need for copy relocations can be avoided if a compiler generates all code references to such objects indirectly through a dynamically relocatable location, and if all static data references are placed in relocatable regions of the image. In practice, however, this is difficult to achieve without source-code annotation; a better approach is to avoid defining static global data in shared libraries.

R_CKCORE_GLOB_DAT

This relocation type is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to deterimine the correspondence between symbols and global offset table entries.

R_CKCORE_JMP_SLOT

The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a procedure linkage table entry. The dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address, see "Procedure Linkage Table".

R_CKCORE_GOTOFF

This relocation type computes the difference between a symbol's value and the address of the global offset table. It additionally instructs the link editor to build the global offset table.

R_CKCORE_GOT32

When referring to a global DATA or FUNCTION in text section, the compiler and assembler set a R_CKCORE_GOT32 relocation for the linker, then the linker create an entry in GOT and set R_CKCORE_GLOB_DAT for dynamic linkage.

R_CKCORE_PLT32

When calling a FUNCTION in text section, the compiler and assembler create the code such as: calling a FUNCTION@PLT, an set R_CKCORE_PLT32 relocation for the linker. If FUNCTION is a local function, the linker finish the relocation; if FUNCTION is a global function, the linker set R_CKCORE_JMP_SLOT relocation for dynamic linkage.

Table 4.9 Relocation Types for PIC describes the function of relocation types for PIC, and when they are deal with.

Table 4.9: Relocation Types for PIC

Fields	For what	Type In Object File(.o)	Type in .so
In Text Sections	Loading GOT Base Address	R_CKCORE_GOTPC	NULL
	Refer to Local Data or Function	R_CKCORE_GOTOFF	NULL
	Refer to Global Data or Function	R_CKCORE_GOT32	R_CKCORE_GLOB_DAT
	Call Local Function Directly	R_CKCORE_PLT32	NULL
	Call Global Function Directly	R_CKCORE_PLT32	R_CKCORE_JMP_SLOT
In Data Sections	Refer to Local Data or Function	R_CKCORE_ADDR32 w/section	R_CKCORE_RELATIVE
	Refer to Global Data or Function	R_CKCORE_ADDR32 w/sym	R_CKCORE_ADDR32 w/sym

4.5 Program Loading

As the system creates or augments a process image, it logically copies a file segment to a virtual memory segment. When and if the system physically reads the file depends on the program's execution behavior, system load, etc. A process does not require a physical page unless it references a logical page during execution. Processes commonly leave many pages unreferenced; therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose virtual addresses are zero, modulo the file system block size.

Virtual addresses and file offsets for C-SKY CPU segments are congruent modulo 64 KByte (0x10000) or larger powers of 2. Because 64 KBytes is the maximum page size, the files are suitable for paging regardless of physical page size.

Table 4.10 Program Header Segments describes the Executable File Example in Figure 4.1 Executable File example.

Table 4.10: Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0	0x2bf00
p_vaddr	0x400100	0x43bf00
p_paddr	unspecified	unspecified
p_filesz	0x2bf00	0x4e00
p_memsz	0x2bf00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W
p_align	0x10000	0x10000

Because the page size can be larger than the alignment restriction of a segment file offset, up to four file pages can hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page can hold a copy of the beginning of data.
- The first data page can have a copy of the end of text.
- The last data page can contain file information not relevant to the running process.

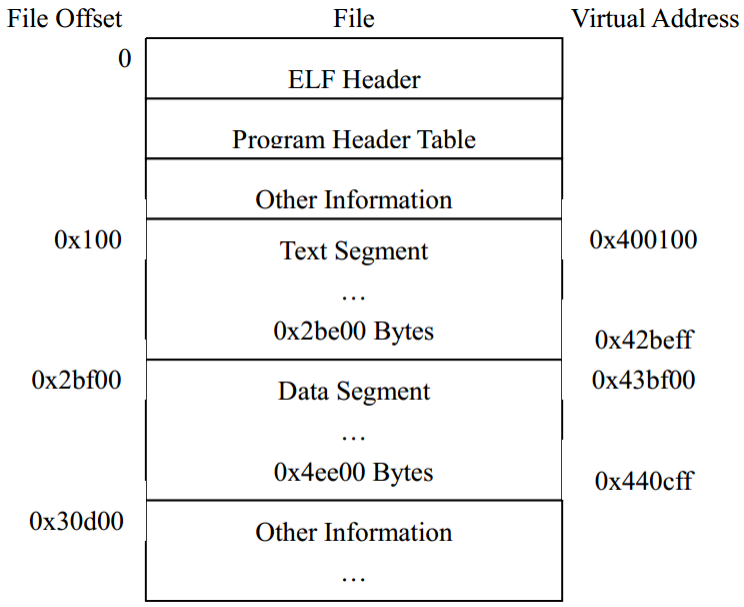

Figure 4.1: Executable File example

Logically, the system enforces the memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example Figure 4.1 Executable File example, the file region holding the end of text and the beginning of data is mapped twice: once at one virtual address for text and once at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data which the system defines to begin with zero values. Thus if the last data page of a file includes information not in the logical memory page, the extraneous data must be set to zero, rather than the unknown contents of the executable file. "Impurities" in the other three pages are not logically part of the process image; whether the system expunges them is unspecified.

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code [see "**PIC Examples**"]. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file, with the system using the p_vaddr values unchanged as virtual addresses. Shared object segments typically contain position-independent code, allowing a segment virtual address to change from one process to another without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the relative positions of the segments. Because position independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

Table 4.11: Shared Object Segment Addresses Example

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0x50000200	0x5002a400	0x50000000
Process 2	0x50010200	0x5003a400	0x50010000
Process 3	0x60020200	0x6004a400	0x60020000
Process 4	0x60030200	0x6005a400	0x60030000

4.6 Dynamic Linking

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without changing the segment images. Thus multiple processes can share a single shared object text segment, even though the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques:

- Control transfer instructions hold addresses relative to the program counter (PC). A PC-relative branch or function call computes its destination address in terms of the current program counter, not relative to any absolute address. If the target location exceeds the allowable offset for PC relative addressing, the program requires an absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the the instructions, the compiler generates code to calculate an absolute address during execution.

Because the processor architecture provides PC relative call, register call and branch instructions, compilers can easily satisfy the first condition.

A global offset table provides information for address calculation. Position-independent object files (executable and shared object files) have a table in their data segment that holds addresses. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual addresses assigned for an individual process. Because data segments are private for each process, the table entries can change - whereas text segments do not change because multiple processes share them.

4.6.1 Dynamic Section

Dynamic section entries give information to the dynamic linker. Some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

DT_PLTGOT

On the C-SKY CPU architecture, this entry's d_ptr member gives the address of the first entry in the global offset table. As mentioned below, the first three global offset table entries are reserved, and two are used to hold procedure linkage table information.

4.6.2 Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries. After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type R_CKCORE_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The first entry (entry 0) in the table is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image. On the C-SKY CPU architecture, the second and third entries the global offset table also are reserved. The second entry (entry 1) is reserved for the ID of this module in the dynamic linker, and the third entry (entry 2) is reserved for a function address in the dynamic linker, which is used in PLT. See Section 4.6.4 Procedure Linkage Table.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For the C-SKY CPU architecture, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table.

```
extern Elf32_Addr _GLOBAL_OFFSET_TABLE_[]
```

The symbol _GLOBAL_OFFSET_TABLE_ must be the base of the .got section, allowing non-negative "subscripts'' into the array of addresses.

4.6.3 Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. The dynamic linker treats such symbol table entries specially. If the dynamic linker is searching for a symbol, and encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

1. If the st_shndx member of the symbol table entry is not SHN_UNDEF, the dynamic linker has found a definition for the symbol and uses its st_value member as the symbol's address.
2. If the st_shndx member is SHN_UNDEF and the symbol is of type STT_FUNC and the st_value member is not zero, the dynamic linker recognizes this entry as special and uses the st_value member as the symbol's address.
3. Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

4.6.4 Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the C-SKY CPU architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text.

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must reside in gb (Refer to Table 2.3 Register Assignments). Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file.

   Consequently, the calling function is responsible for setting the global offset table base register before calling the procedure linkage table entry. So the compiler must create codes to calculate the global offset table base, and set it in gb (GOT base register) at the prologue of the calling function, Just like:

```
Func:
        ...
        bsr L1 /* r15 = L1 = PC+2 now */
L1:
        lrw gb, $GOT-L1
        add gb, r15 /* so gb = $GOT */
```

3. For illustration, assume the program calls name1, then the compiler creates the function calling, such as:

```
Func:
        ...
        /* Calling name1 function created by compiler, r13 can be other registers */
        lrw r13, #offset /* offset = name1@GOT -$GOT */
        add r13, gb
        ld r13, (r13, 0) /* r13 = *(name1@GOT) */
        jsr r13
        ...
```

4. Initially (first time to calling name1), If the dynamic linker is using lazy binding technique, (name1@GOT) in the global offset table holds the address of the instructions in PLT, not the real address of name1. So calling name1 (jsr r13 instruction) transfers control to the label .PLT1.

   If the lazy binding technique is not used in dynamic linker, or the second time to calling name1 when lazy binding, the global offset table holds the real address of name1, the dynamic linking is finished. So if binding directly in the dynamic linker, we need not PLT.

5. For lazy binding, in PLT, each entry includes some instructions, Just like:

```
PLT1: /* for calling name1 */
        subi r0, 24 /* to save arguments in stack for name1 */
```

(continues on next page)

(continued from previous page)

```
        stw r2, (r0, 0)
        stw r3, (r0, 4)
        stw r4, (r0, 8)
        stw r5, (r0, 12)
        stw r6, (r0, 16)
        stw r7, (r0, 20)
        /* load the function address in the dynamic linker */
        ldw r4, (gb, 8)
        /* Prepare the arguments in r2&r3 for the dynamic linker */
        lrw r3, #offset /* the offset of relocation for name1 in .reloc */
        ldw r2, (gb, 4) /* load the ID of this module in the dynamic linker */
        jmp r4 /* transfer the control to the dynamic linker*/
.PLT2:
        ...
```

6. At first, we must save all arguments of name1 on the stack, but does not save link register (r15). So the dynamic linker need not save r2~r7 any more. But must save r8~r15 if they are used in dynamic linker.
7. Secondly, the program load the relocation offset (offset) in .dynamic section to r2. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type R_CKCORE_JMP_SLOT, and its offset will specify the global offset table entry used in step 3. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, name1 in this case.
8. After getting the relocation offset, the program places the value of the second global offset table entry (GOT+ 4)/(gb, 4) into r3, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (GOT + 8)/(gb, 8), which transfers control to the dynamic linker.
9. When the dynamic linker receives control, it looks at the designated relocation entry, finds the symbol's value, stores the "real'' address for name1 in its global offset table entry, and transfers control to the desired destination.
10. Subsequent instructions at step 3 will call directly to name1, without calling the dynamic linker a second time. That is, the jsr instruction at step 3 will transfer to name1, instead of transferring to the .PLT1 instruction.

The LD_BIND_NOW environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type R_CKCORE_JMP_SLOT during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

4.7 PIC Examples

This section discusses example code sequences for basic operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does not prevent a program from conforming to the ABI. Two main object code models are available.

Absolute code

Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program absolute addresses coincide with the process virtual addresses.

Position-independent code

Instructions under this model hold relative addresses, not absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

The following sections describe the differences between absolute code and position-independent code. Code sequences for the models (when different) appear together, allowing easier comparison.

Note: The examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences or to reproduce compiler output or actual assembler syntax.

4.7.1 Function Prologue for PIC

This section describes the function prologue for position-independent code. A function prologue first calculates the address of the global offset table, leaving the value in register gb, This calculation is a constant offset between the text and data segments, known at the time the program is linked.

The offset between the start of a function and the global offset table (known because the global offset table is kept in the data segment) is added to the virtual address of the function to derive the virtual address of the global offset table. This value is maintained in the gb register throughout the function.

After calculating the gb, a function allocates the local stack space, the gb is a called saved register. See the first block of codes in Section 4.6.4 Procedure Linkage Table.

4.7.2 Data Objects

This section describes data objects with static storage duration. The discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack pointer(see Figure 4.2 Absolute Load & Store).

```
extern int src;                 .globl src, dst, ptr
extern int dst;
extern int *ptr;

{                               lrw r7, dst
    …                           lrw r6, ptr
    ptr = &dst;                 stw r7, (r6, 0)

    *ptr = src;                 lrw r7, src
    …                           ldw r4, (r7, 0)
}                               lrw r6, ptr
                                ldw r5, (r6, 0)
                                stw r4, (r5, 0)
```

Figure 4.2: Absolute Load & Store

Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the symbols' offsets into the global offset table. Combining the offset with the global offset table address in gb gives the absolute address of the table entry holding the desired address(see Figure 4.3 Load & Store for PIC) .


```
extern int src;
extern int dst;
extern int *ptr;

{
    …
    ptr = &dst;




    *ptr = src;
    …
}
```

```
.globl src, dst, ptr



lrw r7, dst_got_offset
add r7, gb
ldw r5, (r7, 0)
lrw r6, ptr_got_offset
add r6, gb
ldw r4, (r6, 0)
stw r5, (r4, 0)

lrw r7, src_got_offset
add r7, gb
ldw r4, (r7, 0)
ldw r5, (r4, 0)
lrw r6, ptr_got_offset
add r6, gb
ldw r3, (r6, 0)
ldw r4, (r3, 0)
stw r5, (r4, 0)
```

Figure 4.3: Load & Store for PIC

4.7.3 Function Call

C-SKY CPU Programs use the jump and link instruction, jsri, to make direct function calls. Since the jsri instruction provides 32 bits of address, direct function calls can appoach full address space (0 ~ 4 GByte)(see Figure 4.4 Absolute Direct Function Calling).

```
extern void func ();

{
    …
    func ();
    …
}
```

```
.import    func



    …
    jsri    func
    …
```

Figure 4.4: Absolute Direct Function Calling

Other indirect function calls are done by computing the address of the called function into a register and using the jump and link register, jsr(see Figure 4.5 Absolute Indirect Function Calling).

Calling position independent code functions is always done with the jsr instruction. The global offset table holds the absolute addresses of all position independent functions. (see Figure 4.6 PIC Function Calling)

4.7.4 Branching

C-SKY CPU programs use branch instructions to control execution flow. As defined by the architecture, branch instructions hold a PC-relative value with a 2 KByte range, allowing a jump to locations up to 2 KBytes away in either direction(see Figure 4.7 Branching).


```
extern void (*ptr)();
extern void func();

{
    …
    ptr = func;



    (*ptr) ();
    …
}
```

```
.global    ptr
.import    func


    …
    lrw    r7, func
    lrw    r6, ptr
    stw    r7, (r6, 0)

    lrw    r7, ptr
    ldw    r6, (r7, 0)
    jsr    r6
    …
```

Figure 4.5: Absolute Indirect Function Calling

```
extern void (*ptr)();
extern void func();

{
    …
    ptr = func;



    (*ptr) ();
    …
}
```

```
.global    ptr
.import    func


    …
    lrw    r7, func_got_offset
    add    r7, gb
    ldw    r5, (r7, 0)
    lrw    r6, ptr_got_offset
    add    r6, gb
    ldw    r4, (r6, 0)
    stw    r5, (r4, 0)

    lrw    r6, ptr_got_offset
    add    r6, gb
    ldw    r4, (r6, 0)
    ldw    r5, (r4, 0)
    jsr    r5
    …
```

Figure 4.6: PIC Function Calling

```
Label:
    …
    goto label;
    …
```

```
1:
    …
    br  1b
    …
```

Figure 4.7: Branching

C switch statements provide multiway selection. When case labels of a switch statement satisfy grouping constraints, the compiler implements the selection with an address table. The address table is placed in a .rdata section; this so the linker can properly relocate the entries in the address table. Figure 4.8 Absolute Switch Codes and Figure 4.9 PIC Switch Codes use the following conventions to hide irrelevant details:

- The selection expression resides in register r7;
- Case label constants begin at zero;
- Case labels, default, and the address table use assembly names .Lcasei, .Ldef, and .Ltab, respectively.

Address table entries for absolute code contain virtual addresses; the selection code extracts the value of an entry and jumps to that address. Position-independent table entries hold offsets; the selection code compute the absolute address of a destination.

```
                                    .text
                                    .align     1


…                                   …
switch (j)                          movi    r6, 3
{                                   cmphs   r7, r6
    case 0:                         jbf     .Ldef
        ...                         lrw     r6, .Ltab
    case 2:                         ixw     r6, r7
        ...                         ldw     r7, (r6, 0)
    case3:                          jmp     r7
        ...                         .section   .rodata
    default:                        .align     2
        ...                     .Ltab:
}                                   .long   .Lcase0
                                    .long   .Ldef
                                    .long   .Lcase2
                                    .long   .Lcase3
                                    .text
                                .Lcase0:
                                    …
```

Figure 4.8: Absolute Switch Codes

4.8 Debugging Information Format

C-SKY CPU tools must use DWARF 2.0 debugging information formats, as described in System V Application Binary Interface, from The Santa Cruz Operation, Inc.

Currently, no extensions to the DWARF standard are necessary to provide C-SKY debugging support. However, such extensions may be made in the future.

4.8.1 DWARF Register Numbers

DWARF generally describes the steps a debugger takes to locate variables in a program being debugged in machine-independent terms. However, the way in which the OP_REG and OP_BASEREG atoms are handled is machine-specific — these atoms require that a value (or the pointer to a value) be contained in a machine-specific register.


```
…
switch (j)
{
    case 0:
        ...
    case 2:
        ...
    case3:
        ...
    default:
        ...
}
```

```
    .text
    .align    1

    …
    movi   r6, 3
    cmphs  r7, r6
    jbf    .Ldef
    lrw    r6, .Ltab_got_offset
    add    r6, gb
    ldw    r6, (r6, 0)
    ixw    r6, r7
    ldw    r7, (r6, 0)
    add    r7, gb
    jmp    r7
    .section   .rodata
    .align    2
.Ltab:
    .long  .Lcase0_got_offset
    .long  .Ldef_got_offset
    .long  .Lcase2_got_offset
    .long  .Lcase3_got_offset
    .text
.Lcase0:
    …
```

Figure 4.9: PIC Switch Codes

Table 4.12 DWARF Register Atom Mapping for C-SKY CPU shows the mapping between the values used in those atoms and the CKCORE register set. The entries ranging from to r15 specify the currently active set of general purpose registers; this is usually the primary register set. The entries ranging from r0 tp r15 specify the alternate register file. The control registers are encoded from 32 through 63.

Table 4.12: DWARF Register Atom Mapping for C-SKY CPU

Atom	Register	Atom	Register	Atom	Register	Atom	Register
0	r0	1	r1	2	r2	3	r3
4	r4	5	r5	6	r6	7	r7
8	r8	9	r9	10	r10	11	r11
12	r12	13	r13	14	r14	15	r15
16	r0’	17	r1’	18	r2’	19	r3’
20	r4’	21	r5’	22	r6’	23	r7’
24	r8’	25	r9’	26	r10’	27	r11’
28	r12’	29	r13’	30	r14’	31	r15’
32	cr0	33	cr1	34	cr2	35	cr3
36	cr4	37	cr5	38	cr6	39	cr7
40	cr8	41	cr9	42	cr10	43	cr11
44	cr12	45	cr13	46	cr14	47	cr15
48	cr16	49	cr17	50	cr18	51	cr19
52	cr20	53	cr21	54	cr22	55	cr23
56	cr24	57	cr25	58	cr26	59	cr27
60	cr28	61	cr29	62	cr30	63	cr31
64	cr32						

CHAPTER 5

Run-time Libraries

The content of most libraries are platform and OS dependent. For this reason, they are beyond the scope of this document and are not addressed here. Some library functions are required to provide support for operations that are not supported directly by the C-SKY CPU hardware. These library routines are specified in this section.

5.1 Compiler Assist Libraries

The C-SKY CPU does not currently provide hardware support for floating point data types, nor for long long data types. Compilers should provide the functionality for some of these operations through the use of support library routines. The C-SKY CPU Technology Center requires a single shared support library for all tool sets to eliminate redundant code.

The functions to be provided through support routines include:

Floating point math routines

Long long routines

Compilers that generate in-line code to provide these functions must make no references to the library functions.

Compilers that provide these functions by generating subroutine calls to the support libraries must use the standard interfaces.

In particular, it must be possible to link objects produced with different tool sets into single executables. This requires that:

- Compiler support library names not clash between tool sets
- Compiler support routines follow consistent linkage rules
- Linkers from different tool sets must either use the same support library names and interfaces, or must provide a mechanism to indicate where support libraries can be found.
- Routines in the support libraries must satisfy the following constraints.

- The only external state information used is floating point rounding mode.
- No global state can be modified.
- Identical results must be returned when a routine is re-invoked with the same input arguments.
- Multiple calls with the same input arguments can be collapsed into a single call with a cached result.

These properties permit a compiler to make assumptions about variable lifetimes across library subroutine calls — values in memory won‘t change; previously de-referenced pointers need not be de-referenced again.

5.2 Floating Point Routines

These routines comply with ABI linkage conventions concerning registers that must be preserved across function calls. The routines have no side effects. They do not modify memory except as noted, thus allowing compilers to optimize de-referenced pointer values across calls. The routines always return the same value for the same inputs, allowing compilers to optimize subsequent calls away.

The data formats are as specified in IEEE 754. The routines are not required to compute results as specified in IEEE 754. Implementations of these routines must document the degree to which operations conform to the IEEE standard. Not all users of floating point require IEEE 754 precision and exception handling, and may not want to incur the overhead that complete conformance requires.

5.2.1 Arithmetic functions

Table 5.1: Floating point arithmetic functions

Functions	Description
double ___adddf3(double a, double b)	Return a + b computed to double precision.
double ___subdf3(double a, double b)	Return a – b computed to double precision
double ___muldf3(double a, double b)	Return a * b computed to double precision
double ___divdf3(double a, double b)	Return a / b computed to double precision
double ___negdf2(double a)	Return –a computed to double precision
float ___addsf3(float a, float b)	Return a + b computed to single precision
float ___subsf3(float a, float b)	Return a – b computed to single precision
float ___mulsf3(float a, float b)	Return a * b computed to single precision
float ___divsf3(float a, float b)	Return a / b computed to single precision
float ___negsf2(float a)	Return –a computed to single precision

5.2.2 Conversion functions

Table 5.2: Floating point conversion functions

Functions	Description
double ___extendsfdf2 (float a)	Extend a to the wider mode of their return type.
float ___truncdfsf2 (double a)	Truncate a to the narrower mode of their return type, rounding toward zero.
int ___fixsfsi (float a)	convert a to a signed integer, rounding toward zero.
int ___fixdfsi (double a)	
long long ___fixsfdi (float a)	convert a to a signed long long, rounding toward zero.
long long ___fixdfdi (double a)	
unsigned int ___fixunssfsi (float a)	convert a to an unsigned integer, rounding toward zero. Negative values all become zero.
unsigned int ___fixunsdfsi (double a)	
unsigned long long ___fixunssfdi (float a)	unsigned long long ___fixunsdfdi (double a) toward zero. Negative values all become zero.
unsigned long long ___fixunsdfdi (double a)	
float ___floatsisf (int i)	convert i, a signed integer, to floating point.
double ___floatsidf (int i)	
float ___floatdisf (long i)	convert i, a signed long, to floating point.
double ___floatdidf (long i)	
float ___floatunsisf (unsigned int i)	convert i, an unsigned integer, to floating point.
double ___floatunsidf (unsigned int i)	
float ___floatundisf (unsigned long i)	convert i, an unsigned long, to floating point.
double ___floatundidf (unsigned long i)	

5.2.3 Comparison functions

Table 5.3: Floating point comparison functions

Functions	Description
int ___unordsf2 (float a, float b)	These functions return a nonzero value if either argument is NaN, otherwise 0. There is also a complete group of higher level functions which correspond directly to comparison operators. They implement the ISO C semantics for floating-point comparisons, taking NaN into account. Pay careful attention to the return values defined for each set. Under the hood, all of these routines are implemented as if (___unordXf2 (a, b)) return E; return ___cmpXf2 (a, b); where E is a constant chosen to give the proper behavior for NaN. Thus, the meaning of the return value is different for each set. Do not rely on this implementation; only the semantics documented below are guaranteed.
int ___unorddf2 (double a, double b)	
int ___eqsf2 (float a, float b)	These functions return zero if neither argument is NaN, and a and b are equal.
int ___eqdf2 (double a, double b)	
int ___nesf2 (float a, float b)	These functions return a nonzero value if either argument is NaN, or if a and b are unequal.
int ___nedf2 (double a, double b)	

Continued on next page

Table 5.3 – continued from previous page

Functions	Description
int ___gesf2 (float a, float b) int ___gedf2 (double a, double b)	These functions return a value greater than or equal to zero if neither argument is NaN, and a is greater than or equal to b.
int ___ltsf2 (float a, float b) int ___ltdf2 (double a, double b)	These functions return a value less than zero if neither argument is NaN, and a is strictly less than b.
int ___lesf2 (float a, float b) int ___ledf2 (double a, double b)	These functions return a value less than or equal to zero if neither argument is NaN, and a is less than or equal to b.
int ___gtsf2 (float a, float b) int ___gtdf2 (double a, double b)	These functions return a value greater than zero if neither argument is NaN, and a is strictly greater than b.
int ___cmpsf2 (float a, float b) int ___cmpdf2 (double a, double b)	These functions calculate a <=> b. That is, if a is less than b, they return −1; if a is greater than b, they return 1; and if a and b are equal they return 0. If either argument is NaN they return 1, but you should not rely on this; if NAN is a possibility, use one of the higher-level comparison functions.

5.3 Long Long Integer Routines

These routines comply with ABI linkage conventions concerning registers that must be preserved across function calls. The routines have no side effects. They do not modify memory except as noted, and thus allow compilers to optimize de-referenced pointer values across calls. The routines always return the same value for the same inputs, allowing compilers to optimize subsequent calls away.

5.3.1 Arithmetic functions

Table 5.4: long long arithmetic functions

Functions	Description
long long ___ashldi3 (long long a, int b)	This function return the result of shifting a left by b bits.
long long ___ashrdi3 (long long a, int b)	This function return the result of arithmetically shifting a right by b bits.
long long ___lshrdi3 (long long a, int b)	This function return the result of logically shifting a right by b bits
long ___divsi3 (long a, long b)	These functions return the quotient of the signed division of a and b.
long long ___divdi3 (long long a, long long b)	
long ___modsi3 (long a, long b)	These functions return the remainder of the signed division of a and b.
long long ___moddi3 (long long a, long long b)	
long long ___muldi3 (long long a, long long b)	This function return the product of a and b.
long long ___negdi2 (long long a)	This function return the negation of a.
unsigned long ___udivsi3 (unsigned long a, unsigned long b)	These functions return the quotient of the unsigned division of a and b.
unsigned long long ___udivdi3 (unsigned long long a, unsigned long long b)	
unsigned long long ___udivmoddi4 (unsigned long long a, unsigned long long b, unsigned long long *c)	This function calculate both the quotient and remainder of the unsigned division of a and b. The return value is the quotient, and the remainder is placed in variable pointed to by c.
unsigned long ___umodsi3 (unsigned long a, unsigned long b)	These functions return the remainder of the unsigned division of a and b.
unsigned long long ___umoddi3 (unsigned long long a, unsigned long long b)	

5.3.2 Comparison function

Table 5.5: long long comparison functions

Functions	Description
int ___cmpdi2 (long long a, long long b)	These function perform a signed comparison of a and b. If a is less than b, they return 0; if a is greater than b, they return 2; and if a and b are equal they return 1.
int ___ucmpdi2 (unsigned long long a, unsigned long long b)	These function perform an unsigned comparison of a and b. If a is less than b, they return 0; if a is greater than b, they return 2; and if a and b are equal they return 1.

5.3.3 Trapping Arithmetic Functions

The following functions implement trapping arithmetic. These functions call the libc function abort upon signed arithmetic overflow.

Table 5.6: long long trapping arithmetic functions

Functions	Description
int ___absvsi2 (int a)	This function return the absolute value of a.
long ___absvdi2 (long a)	
int ___addvsi3 (int a, int b)	This function return the sum of a and b; that is a + b.
long ___addvdi3 (long a, long b)	
int ___mulvsi3 (int a, int b)	This function return the product of a and b; that that is a * b.
long ___mulvdi3 (long a, long b)	
int ___negvsi2 (int a)	These functions return the negation of a; that is -a.
long ___negvdi2 (long a)	
int ___subvsi3 (int a, int b)	These functions return the difference between b and a; that is a - b.
long ___subvdi3 (long a, long b)	

5.3.4 Bit Operations

Table 5.7: long long comparison functions

Functions	Description
int ___ffsdi2 (long long a)	These functions return the index of the least significant 1-bit in a, or the value zero if a is zero. The least significant bit is index one.

CHAPTER 6

Assembler Syntax and Directives

6.1 Section

The output of the assembler consists of, in part, sections whose content is determined by the assembler input. Sections containing code are aligned to 2-byte boundaries. Sections containing data are aligned so that the alignment requirements of the data contained in the section is preserved.

6.2 Input Line Lengths

The assembler may limit input lines, but such a limit must be at least 2100 characters in length. This gives the ability to construct an expression containing a symbol of maximum supported length (2048 bytes) and a data-allocation pseudo-instruction. For example:

```
.long longsymbol
```

The assembler is allowed to support longer lines. If the assembler imposes a limit on the length of an input line, the assembler must issue a diagnostic if that limit is exceeded.

6.3 Syntax

An assembler source file contains a list of one or more assembler statements. Each statement is terminated with a newline character or a ";" character. The ";" character does not terminate the statement if it appears within a string literal or inside a comment. Empty statements (i.e. blank lines) are ignored.

Each statement consists of zero or more labels, at most one mnemonic, with the remainder of the statement being arguments specific to the mnemonic.

Labels are symbols that are followed by a ":". Temporary labels are allowed and are indicated by a non-zero digit (1–9) instead of a symbol. Duplicate temporary labels are allowed and references to them are resolved

by searching for the nearest source line with the label. References to temporary labels must have a "b" or "f" suffix appended to the digit to indicate which direction to search.

Labels that begin with "." (period) are considered local labels. The assembler does not include these symbols in the symbol table of the generated object file.

Mnemonics fall into three categories: instructions, pseudo-instructions, and directives. Instruction mnemonics map one-to-one into an C-SKY CPU opcode. Pseudo-instructions map into sequences of C-SKY CPU opcodes. Directives always start with a "." and are used to control the assembly and allocate data areas. All mnemonics are case sensitive and must be specified in lower case.

White space in assembler source files is ignored except as a separator between mnemonics and when embedded within string literals or character constants. Multiple white space characters are functionally equivalent to a single white space character except within literals and character constants.

Comments in assembler source are indicated by the following:

- "//" sequence indicates a comment reaching to the end of the line.
- "#" character, when not part of a valid preprocessing directive, indicates a comment reaching to the end of the line.

Comments are terminated only by the end of the line. The ";" character does not terminate a comment. A multi-line comment, e.g. "/* */", is not supported since most assemblers are inherently line oriented.

Comments can never begin or end within a string literal or character constant.

6.3.1 Preprocessing

The assembler is not required to provide macro preprocessing. This functionality can be provided by existing preprocessors that conform to the ANSI standard. If the assembler does provide preprocessing, then it must conform to the "C" language preprocessing standard and the following paragraph does not apply.

An assembler command line option will enable the following behavior. Any line with a "#" character in the first column is assumed to be line and file information from the preprocessor. The assembler must use this information in error messages. This allows a programmer to relate an error back to the line and file of the original source file before preprocessing. The file and line information from the preprocessor is in the form:

```
# number "filename"
```

Any other preprocessor lines that do not match this form are ignored by treating them as comments.

6.3.2 Symbols

Symbols must begin with a character in the set: a–z, A–Z, . (period), or _ (underscore). The remaining characters in a symbol may be in that set plus the digits 0–9. Symbols are case sensitive and all characters in the symbol are significant. Symbols may be limited in length but that limit must be at least 2048 characters. If there is a limit on symbol length, symbols that exceed the limit must cause an error message to be emitted.

Silent truncation of long symbols is undesirable. This is intended to avoid silent errors where two long symbols differ only at some point after the tools have stopped keeping track of significant characters. The "$" character is not allowed in a symbol name because it is not a universally supported character on non-U.S. keyboards.

The special symbols created by temporary labels can only be referenced within a single source file. These references must consist of a single digit followed by a "b" or "f" to indicate the direction of the nearest matching label.

The "." symbol will always indicate the current location within the current section at the start of the current statement. Thus:

```
movi r3,15
br .
br .
```

results in three instructions, two of which branch to themselves.

The "." symbol is used instead of "*" because it avoids conflicts with "*" as a multiply operator.

6.3.3 Constants

The same constants and lexical expression of constants that are available in C are allowed in the assembly. This includes hex, octal, decimal, float, double, character, and strings. Both character and string constants have characters, " and" respectively, to delimit them. Multiple characters within character constant are each treated like a base 256 number. e.g. "1234" equals 0x31323334.

The syntax of constants is chosen to be familiar to C programmers. The use of special characters in the syntax for constants must be avoided as they are used in expressions. In addition, the "$" character is not a universally supported character on non-U.S. keyboards.

6.3.4 Expressions

Addition, subtraction, multiplication, division, modulus, logical anding, inclusive oring, exclusive oring, negating, complementing, and shifting operations are supported by the assembler for the generation of constants or relocatable expressions in the argument portion of a statement. These operations have the semantics and precedence of their equivalent C language operations. Parenthesis can be used to force particular bindings of operations. All operations are done as if on 32-bit unsigned values. The syntax of expressions is chosen to be familiar to C programmers.

Expressions can involve more than one relocatable value as long as the assembler can resolve the expression to remove all or all but one of the relocatable values. For example, the difference between two labels in the same section reduces to an assemble time constant.

Relocatable expressions must evaluate down to a possibly-zero offset from a relocatable address. The linker is not required to provide the ability to store the value "5 times the value of this relocatable symbol".

6.3.5 Operators and Precedence

Table 6.1 Assembly Expression Operators shows the operators available to the assembly programmer. The table is arranged in order of precedence; the higher precedence operators appear earlier in the table. These are the same operators used in the C language.

Table 6.1: Assembly Expression Operators

Assembly Expression Operators		Precedence
-	unary negation	1
~	unary logical complement	
*	multiplication	2
/	division	
%	modulus	
+	addition	3
-	subtraction	
<<	left shift	4
>>	right shift	
&	logical and	5
^	logical exclusive or	6
|	logical inclusive or	7

Operations may be grouped with parentheses to force a particular precedence.

6.3.6 Instruction Mnemonics

The instruction opcode mnemonics are listed in the C-SKY CPU Reference Manual .

6.3.7 Instruction Arguments

Register arguments within the argument portion of a statement are indicated by the character, “r” or “R” followed by the register number (0 through 15). Register 0 (r0) can also be specified as “sp”.

Instructions that use the PC relative indirect addressing (lrw, jsri, jmpi) take two argument syntaxes. The first syntax is of the form:

```
lrw r0, 0x12345678
lrw r1, 0x4321
lrw r2, 0x4321
lrw r3, 0x4321
```

The assembler collects these argument values into a literal table, possibly allowing several instructions to reuse the same slot, and emit them at an appropriate point in the output. Such a point may be after the nearest unconditional branch. In some situations, such a location might not arise before the span of the lrw/jsri/jmpi instruction is exhausted. In such cases, the assembler must spill the literal table before the span is exhausted and provide a branch around the literal table.

The assembler provides a mechanism that allows the user to force a dump of the currently outstanding literals by using the .literals pseudo-instruction. Any literals that have not yet been emitted are emitted when this directive is encountered. When the assembler input is exhausted, the assembler emits any literals that have not yet been emitted, as if a .literals pseudo-instruction was appended to the assembly source.

Note: The assembler is allowed, but not required, to attempt to optimize code size by doing “optimal” literal placement. This interacts with the expansion of jbt and jbf pseudo-operations. Also, if literals must be output after an instruction that is not an unconditional transfer of control, the assembler must insure that a branch around the literal table is also generated.

The second form uses a [label] notation for the literal. In this case, the supplied argument is the label of the address containing the value to be loaded. This gives the assembler programmer complete control over the placement and sharing of literals.


```
lrw r0,[lit0]
lrw r1,[lit1]
lrw r2,[lit1]
lrw r3,[lit1]
...
.align 4
Lit0: .long 0x12345678
Lit1: .long 0x4321
```

Note: The user is responsible for insuring that the specified label is 4-byte aligned when using the [label] literal syntax.

The C-SKY CPU instruction set does not directly support position independent code so it is up to the assembler programmer or compiler to synthesize PC-relative branches and subroutine calls. To help support this, a 32-bit PC relative argument type is allowed and is indicated by an expression that is evaluated as a delta from ".". Any symbols in the expression must be within the same section as the instruction so the assembler can resolve it to a constant offset. This can be done in the following manner (assuming r1 and r15 are available):

```
bsr .+2
lrw r1,symbol-.
add r1,r15
jsr r1
...
symbol: subi r0,12
```

6.4 Assembler Directives

Assembler directives are used to control the assembly of the source code as well as reserving and/or initializing areas for data. All assembler directive mnemonics begin with a ".".

Only the .align, .comm, and .lcomm directives align the location counter to a known boundary. All other mnemonics, including .long, do not imply alignment. It is up to the assembler programmer or compiler to explicitly align these locations to avoid runtime misalignment faults. For operations that specify alignment values (e.g., .align, .comm, and .lcomm), the value specified is log2 of the alignment. For example, the value "3" specifies 8-byte alignment.

All data values emitted by assembler directives will be in big-endian order.

This alignment behavior is needed to support packed data structures. Packed data structures explicitly allow misaligned fundamental types to save data space at the expense of additional code to pack and unpack the structures. Note that the ABI does not specify how a user expresses such misaligned references at the C source level.

The directive syntax in this manual uses "[" and "]" to indicate an optional field. The "{" and "}" syntax indicates zero or more repetitions of a field.

6.4.1 .align abs-exp [, abs-exp]

Aligns the location counter to the boundary indicated by the first constant expression. The integral alignment argument is log2 of the alignment, e.g. the value "3" specifies 8-byte alignment. Negative alignment values

are treated as zero, indicating 1-byte alignment.

The second, optional expression is the value to be filled into the bytes between the old location and new location. If unspecified, the bytes will be filled with zeros.

Note: The maximum alignment allowed is not constrained by the assembler. But in order for the assembler to be able to resolve expressions between symbols in the section,the linker must guarantee that the resulting section will be aligned to the largest alignment required within the section. This can be true for every loadable section from every source file, so large alignments should be used conservatively to avoid large gaps in the final load image.

6.4.2 .ascii "string" {, "string"}

Reserves and initializes space for one or more strings given. Each assembled string will not be null-terminated and will fill consecutive addresses. No alignment is implied.

6.4.3 .asciz "string" {, "string"}

Same as .ascii except the strings will be null terminated.

6.4.4 .byte exp {, exp}

Assembles consecutive bytes with the one or more values given by the expression(s). No alignment is implied.

Values larger than eight bits are truncated to fit into eight bits. This also generates a warning diagnostic.

6.4.5 .comm symbol, length [, align]

Declares an area of length bytes in the .bss section that will be shared by different files. If another file declares a longer length, then the length will be the maximum of all the declared lengths.

The alignment, if specified, is log2 of the alignment. The value "3" specifies 8-byte alignment. The units are the same as in the .align directive. If no alignment is specified, the assembler will naturally align the symbol according to the largest natural type that can be contained in an entity of that size. Entities of eight bytes and larger are 8-byte aligned, entities of four bytes are 4-byte aligned, entities of two and three bytes are 2-byte aligned, single-byte entities are 1-byte aligned.

6.4.6 .data

Equivalent to:

```
.section .data,"RW"
```

6.4.7 .double float {, float}

Assembles floating point values into IEEE 64-bit floating point numbers. The numbers will be consecutive and no alignment is implied.

6.4.8 .equ symbol, expression

Sets the value of the symbol to the expression. If the expression value cannot be resolved to an absolute or relocatable value after all assembler passes are complete, the assembly will be aborted with an error.

6.4.9 .export symbol {, symbol}

Causes the symbol to appear in the emitted symbol table in the resulting object file. The symbol may be defined within the file or it may be defined within an external file.

6.4.10 .fill count [, size [, value]]

Emits count copies of the value given. Only the least significant size bytes of value are replicated. The size must be a value ranging from one through eight; the default size is one byte. The default value is zero.

All three arguments are integral absolute expressions.

6.4.11 .float float {, float}

Assembles floating point values into IEEE 32-bit floating point numbers. The numbers will be consecutive and no alignment is implied.

6.4.12 .ident "string"

Places the string in the .comment section of the object file reserved for identification purposes. This is used for version tracking and source-to-binary audit trails.

6.4.13 .import symbol {, symbol}

Indicates that the symbols are defined externally from this file. All undefined symbols that are not declared as imported will cause a warning message to be issued by the assembler. Symbols that have been declared external but are not referenced should not appear in the symbol table of the emitted object file.

6.4.14 .literals

Causes the assembler's accumulated literal table for the jmpi, jsri, and lrw instructions for the current section to be emitted. Can be used by the assembler programmer to flush literal tables at the exact point desired.

6.4.15 .lcomm symbol, length [, alignment]

Reserve length bytes for a named local common area in the .bss section. The allocations of symbols in the .bsssection will be in the same order as the .lcomm statements in the source file.

Note: Preserving the allocation order allows the compiler to use fixed offsets from a bss pointer to access several related variables.

The optional alignment value is log2 of the desired alignment; a value of "3" specifies eight byte alignment. If no alignment is specified, the assembler will naturally align the symbol according to the largest natural

type that can be contained in an entity of that size. Entities of eight bytes and larger are 8-byte aligned, entities of four bytes are 4-byte aligned, entities of two and three bytes are 2-byte aligned, single-byte entities are 1-byte aligned.

6.4.16 .long exp {, exp}

Emits four byte values consecutively.

6.4.17 .section name [, "attributes"]

Assemble subsequent statements onto the end of the named section.

Section names obey the same syntax as symbol names.

The attributes supported are the access permissions (read, write, and execute) and the allocation bits (yes or no). Permissions and allocation are indicated by any combination of the letters RWXANrwxan with no separators between them. The attributes are specified as a quoted string. The attribute characters are explained in Table 6.2 CKCORE Section Attribute Encodings.

Table 6.2: CKCORE Section Attribute Encodings

Section Attribute Encodings	
R or r	Section is to be readable.
W or w	Section is to be writable.
X or x	Section contains executable code.
A or a	Section is to be allocated space in the loaded image.
N or n	Section is NOT to be allocate space in the loaded image.

A missing attribute list indicates that the section should have all permissions (RWX) and address space will be allocated in the load map. An empty attribute list (e.g., an empty quoted string) specifies an allocated but inaccessible section.

A missing attribute list generates the default permissions.

Multiple specifications of a section take the attributes from the first specification of the section.

```
.sectionsectionname,"RX"
.sectionsectionname,"RW"
```

The RW attribute is ignored and the section sectionname will have read and execute permissions.

6.4.18 .short exp {, exp}

Emits two byte values consecutively.

6.4.19 .text

Equivalent to:

```
.section.text,"RX"
```

6.4.20 .weak symbol [, symbol]

Specify a weak external symbol definition. If symbol is not otherwise defined at link time, it has the value zero. Multiple symbols can be specified on the same line.

6.5 Pseudo-Instructions

The assembler also supports several pseudo-instructions which are expanded into one or more machine instructions.

Some pseudo-instructions are used to delay selection of instructions until relative addresses are resolved. For example, a smaller relative branch instruction could be emitted instead of a larger absolute jump instruction if the decision is delayed until the branch distance is known.

Other pseudo-instructions are for the assembler programmers convenience. For example, the "clear the condition bit" (clrc) instruction is another mnemonic for a compare of r0 being not equal to r0. Also, the mnemonics for the load/store instructions (ldb, ldh, ldw, stb, sth, stw) have alternate forms (ld.b, ld.h, ld.w, st.b, st.h, st.w).

6.5.1 clrc

Clear the condition code bit (C) in the status register. Emits the opcode equivalent to:

```
cmpne r0,r0
```

6.5.2 cmplei rd, n

Perform a signed comparison of the value in rd with the constant n. N is allowed to have the values 0 through 31. Emits the opcode equivalent to:

```
cmplti rd, n+1
```

6.5.3 cmpls rd, rs

Compare if the unsigned value in rd is lower or the same as the unsigned value in rs. Emits the opcode equivalent to:

```
cmphs rs, rd
```

6.5.4 cmpgt rd, rs

Compare if the signed value in rd is greater than the signed value in rs. Emits the opcode equivalent to:

```
cmplt rs, rd
```

6.5.5 jbsr label

Call the subroutine identified by label. Use the relative branch to subroutine instruction if the subroutine is within range, otherwise use an absolute jump to subroutine. Emits one of the following sequences:

```
        bsr label
Or:
        jsri label
```

6.5.6 jbr label

Continue execution at the instruction identified by label. Use the relative branch instruction if the label is within range, otherwise use an absolute jump to the label. Emits the equivalent of one of the following sequences based on the distance to the target label.

```
        br label
Or:
        jmpi label
```

6.5.7 jbf label

Continue execution at the instruction identified by label only if the condition code bit is false. Use the relative conditional branch instruction if the label is within range, otherwise use a conditional branch around an absolute jump to the label. Emits the equivalent of one of the following sequences based on the distance to the target label.

```
        bf label
Or:
        bt 1f
        jmpi label
  1: ...
```

The temporary label "1" is here for illustration purposes; it is not emitted. The expansion of jbf will not cause a problem for the following fragment.

```
  bt 1
  ...
  jbf label
1: ...
```

6.5.8 jbt label

Continue execution at the instruction identified by label only if the condition code bit is true. Use the relative conditional branch instruction if the label is within range, otherwise use a conditional branch around an absolute jump to the label. Emits the equivalent of one of the following sequences based on the distance to the target label.

```
        bt label
Or:
        bf 1f
        jmpi label
  1: ...
```

The temporary label "1" is here for illustration purposes; it is not emitted. The expansion of jbt will not cause a problem for the following fragment.

```
      bt 1
      ...
      jbtlabel
1: ...
```

6.5.9 neg rd

Negates the value in rd. Emits the opcode equivalent to:

```
rsubi rd,0
```

6.5.10 rotlc rd, 1

Rotates the value in rd left by one bit. The carry bit is rotated into least significant bit while the most significant bit that was rotated out is saved in the carry bit. Emits the opcode equivalent to:

```
addc rd,rd
```

6.5.11 rotri rd, imm

Rotates the value in rd right by the number of bits specified in imm. Emits the opcode equivalent to:

```
rotli rd,32-imm
```

An immediate value of 0 is not allowed.

6.5.12 rts

Return from subroutine. Emits the opcode equivalent to:

```
jmp r15
```

6.5.13 setc

Set the condition code bit (C) in the status register. Emits the opcode equivalent to:

```
cmphs r0,r0
```

6.5.14 tstle rd

Test for a negative or zero value in the specified register. Emits the opcode equivalent to:

```
cmplti rd,1
```

6.5.15 tstlt rd

Test for a negative value in the specified register. Emits the opcode equivalent to:

```
btsti rd,31
```

6.5.16 tstne rd

Test for a non-zero value in the specified register. Emits the opcode equivalent to:

```
cmpnei rd,0
```