.

AssemblerManual

(AssemblyLanguageProgrammer’sGuide)

COFFEE™ RISCCORE

Version0.7

VERSION HISTORY

Version	Date	Changes
0.1	25.08.2004	First draft
0.2	11.09.2004	Additions and corrections (Juha)
0.3	13.10.2004	Corrections to fit `crasm pre 2`
0.4	01.11.2004	Corrections
0.5	21.12.2004	Corrections to fit `crasm pre2.11`
0.6	31.01.2005	Corrections to fit `crasm 1.0`
0.7	18.02.2005	Corrections

TABLE OF CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

OVERVIEW

This manual is a user guide to the COFFEE ™ assembler `crasm`.
Here is a brief summary of how to invoke `crasm`. It is written with Perl.

```
perl crasm [input_file] [-Include path|-I path] [[-binary | -b] | [-hex | -h]] [-help|-h] [-list|-l] [-obj output_file|-o output_file] [-symbols|-s] [-version|-v] [-warnoff|-w] [--version|--v] [-Z]
```

`input_file`

Input file name. Extension isn't matter.

`-binary`

Create separate binary output files. Name of the text segment file is name of the source file plus '_ts.bin'. Name of the data (and bss) segment file is name of the source file plus '_ds.bin'.

`-hex`

Create separate hexadecimal output files. Name of the text segment file is name of the source file plus '_ts.bin'. Name of the data (and bss) segment file is name of the source file plus '_ds.bin'.

`-Include path`

Path (one) for include files. It is allowed to repeat option as many times as needed. Search for include files is done in following sequence: directory where is source file and then paths in order they are defined.

`-help`

Print available options.

`-list`

Turn on listings. Name of the list file is name of the source file with extension 'lst'.

`-obj filename`

Define name of object-file. Default name is name of the source file. Output file always have extension 'out'.

`-symbols`

Do not add local symbols in symbol table.

`-warnoff`

Suppress warning messages.

`-version`

Print version of assembler.

`--version`

Print version of assembler and exit.

`-Z`

Generate an object file even if error was found.

After the program name `crasm`, the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is insignificant.
All options should start with hyphen ('-'), except input file name. An option is a '-' followed by one letter or full name; the case of the letter is important. All options are optional. All options should be separated by at least one space.
Some options expect exactly one file name (or path) to follow them.

1. ARCHITECTURE-BASED CONSIDERATIONS

This chapter describes programming considerations that are determined by the COFFEE™ RISC core architecture. It addresses the following topics:

- Registers (Section 1.1)
- Bit and Byte Ordering (Section 1.2)
- Addressing (Section 1.3)
- Exceptions (Section 1.4)
- Interrupts (Section 1.5)

1.1 Registers

This section discusses the registers that are available and describes how memory organization affects them. See Section 6.1 for information on register use and linkage.

1.1.1 Main Processor Registers

COFFEE™ RISC core has two different register sets for data shown in Table 1-1. The first set (SET 1) is intended to be used by application programs. The second set of registers (SET 2) is for privileged software which could be an operating system or similar. SET 2 is protected from application program. Privileged software can access both sets. There are 32 registers in both sets including general purpose registers (GPRs) and special purpose registers (SPRs).

In addition COFFEE™ has eight condition registers (CRs) which are used with conditional branches or when executing instructions conditionally. These are visible to application software as well as to privileged software.

COFFEE™ has also one memory mapped register bank, CCB (core control block). CCB is for controlling the processor operation and as such should be configured by boot code. CCB also contains few status registers. Note that, CCB can be extended with an external configuration block.

The usage of general purpose registers is not restricted by hardware in any way. See Section 6.1 or compiler documentation for more information about register usage.

SET 1			SET 2		
R0	GPR	32-bit	PR0	GPR	32-bit
R1	GPR	32-bit	PR1	GPR	32-bit
…					
R28	GPR	32-bit	PR28	GPR	32-bit
R29	GPR	32-bit	PR29	PSR	32-bit
R30	GPR	32-bit	PR30	SPSR	32-bit
R31	GPR/LR	32-bit	PR31	GPR/LR	32-bit

Table 1-1. Register sets

1.1.1.1 SET 1 GPRs

SET 1 has 32 identical general purpose registers R0...R31 with one exception: R31 is used as a link register (LR) with some instructions. The programmer is free to use R31 for any other purpose as long as its special behaviour is taken into account. All general purpose registers (and the link register) are 32-bit wide.

1.1.1.2 SET 2 GPRs

SET 2 has 30 identical general purpose registers PR0...PR28 and PR31 with one exception: PR31 is used as a link register by some instructions. The programmer is free to use PR31 for any other purpose as long as its special behaviour is taken into account. All general purpose registers (and the link register) are 32-bit wide.

1.1.1.3 SET 2 SPRs

There is two special purpose registers in SET 2: PSR and SPSR. PSR is 8-bit wide. When reading data from PSR the "non existent" bits are read as zeros. Writing to a read only register (PSR) is ignored.

PSR (register index 29)

Processor Status Register is a read only register shown in Figure 1-1 and contains the flags explained below. Bits 7 down to 5 are reserved for future extensions.

RESERVED	IE	IL	RSWR	RSRD	UM
7...5	4	3	2	1	0

Figure 1-1. Main processor status register

`IE = 1`: Interrupts enabled, `IE = 0`: Interrupts disabled.
`IL = 1`: Instruction word length is 32 bits, `IL = 0`: Instruction word length is 16 bits.
`RSWR` bit selects which register set to use as target:
`RSWR = 1`: SET2, super users set; `RSWR = 0`: SET1, users set.
`RSRD` bit selects which register set to use as source:
`RSRD` = 1: SET2, super users set; `RSRD` = 0: SET1, users set.
`UM` indicates which user mode the processor is in:
`UM` = 0: super user mode, `UM` = 1: user mode.
`RESERVED`: Read as zeros.

SPSR (register index 30)

SPRS is used to save PSR flags when changing user mode by executing `scall` – instruction. It can also be used to set mode flags for the user: IE and IL flags are copied from SPSR to PSR when `retu` instruction is executed. Note that bits 31 down to 5 are writable but only bits 7 down to 0 are saved in case of `scall`.

1.1.1.4 *CRs*

There are eight 3-bit wide condition registers C0...C7 (visible both to application software and privileged software). Condition registers are used with conditional branches or when executing instructions conditionally. Each register contains three flags: Z (Zero), N (Negative) and C (Carry). When executing compare instructions or some arithmetic instructions these three flags are calculated and saved to the selected CR (arithmetic instructions always save flags to C0). When conditionally branching or executing, flags from the selected CR are compared to match a certain condition given by the programmer. See Chapter 3 for more information about instructions and Section 2.9 for more information about conditional execution.

1.1.1.5 CCB registers

In the following, usage and organization of control and status registers is explained. Few things worth noting are discussed first. The core configuration block (CCB) is a memory mapped register bank, which contains various registers for controlling the functionality of the core. It also contains status registers, which cannot be written but are only used by software for monitoring events. CCB registers are organized as a continuous block, that is, memory addresses of the registers reserve a continuous area from the address space. CCB reserves 256 consecutive memory addresses ("byte" addresses) starting from the address defined in the first CCB register (the base address of the block). CCB can be remapped anywhere in the address space by writing a new value to CCB_BASE register. It is also possible to extend the range of configuration registers by writing a suitable address to CCB_END register: Memory accesses in address range [CCB_BASE] + 256 to [CCB_END] are redirected to an external block instead of memory.

Conventions and notes:
Unused bits in registers shorter than 32 bits will appear as zeros but they should be masked by software for future compatibility. Bit indexes range from 0 to 31, 0 corresponding to LSB. When referring to bit positions we simply refer to bit indexes: A bit in position X means a bit with index X. Offsets from 29H to FFH are reserved for future extensions.

symbol: **CCB_BASE[31..0]**
offset: **0H**
reset value: **0001000H**
description: The contents of this register defines the base address of the CCB block. 256 consecutive memory locations starting from [CCB_BASE] are reserved for CCB registers. All memory accesses in range [CCB_BASE] to [CCB_BASE] + 255 map to CCB registers.
notes: The base address has to be aligned to 256B boundary, that is, bits 7 down to 0 has to be zeros. You need to have at least one instruction between the one remapping the CCB (*st* instruction) and one accessing CCB at new location.

symbol: **REGSPC_END[31..0]**
offset: **1H**
reset value: **000100FFH**
description: The contents of this register defines the last address of register address space. All memory accesses in range [CCB_BASE] + 256 to [CCB_END] map to an external register block.

notes:	Addresses [CCB_BASE] to [CCB_BASE] + 255 map to CCB independent of the value in REGSPC_END. The external register block may be any device connected to data memory bus of COFFEE core.
symbol:	**COP0_INT_VEC[31..0]**
offset:	**2H**
reset value:	**00000001H**
description:	The contents of this register defines the entry address of an interrupt service routine for coprocessor 0 interrupts/exceptions.
notes:	See Section 1.5 for more information about interrupts.
symbol:	**COP1_INT_VEC[31..0]**
offset:	**3H**
reset value:	**00000001H**
description:	The contents of this register defines the entry address of an interrupt service routine for coprocessor 1 interrupts/exceptions.
notes:	See Section 1.5 for more information about interrupts.
symbol:	**COP2_INT_VEC[31..0]**
offset:	**4H**
reset value:	**00000001H**
description:	The contents of this register defines the entry address of an interrupt service routine for coprocessor 2 interrupts/exceptions.
notes:	See Section 1.5 for more information about interrupts.
symbol:	**COP3_INT_VEC[31..0]**
offset:	**5H**
reset value:	**00000001H**
description:	The contents of this register defines the entry address of an interrupt service routine for coprocessor 3 interrupts/exceptions.
notes:	See Section 1.5 for more information about interrupts.
symbol:	**EXT_INT0_VEC[31..0]**
offset:	**6H**
reset value:	**00000001H**
description:	The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 0.
notes:	The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol:	**EXT_INT1_VEC[31..0]**
offset:	**7H**
reset value:	**00000001H**
description:	The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 1.
notes:	The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol:	**EXT_INT2_VEC[31..0]**
offset:	**8H**
reset value:	**00000001H**
description:	The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 2.
notes:	The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol:	**EXT_INT3_VEC[31..0]**
offset:	**9H**
reset value:	**00000001H**
description:	The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 3.
notes:	The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol:	**EXT_INT4_VEC[31..0]**
offset:	**AH**
reset value:	**00000001H**
description:	The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 4.
notes:	The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol:	**EXT_INT5_VEC[31..0]**
offset:	**BH**
reset value:	**00000001H**

description: The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 5.
notes: The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol: **EXT_INT6_VEC[31..0]**
offset: **CH**
reset value: **00000001H**
description: The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 6.
notes: The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol: **EXT_INT7_VEC[31..0]**
offset: **DH**
reset value: **00000001H**
description: The contents of this register defines the base address of an interrupt service routine for external/timer interrupt number 7.
notes: The entry address of the interrupt service routine can be the base address directly or a combination of the base address and an offset provided externally. See Section 1.5 for more information about interrupts.

symbol: **INT_MODE_IL[11..0]**
offset: **EH**
reset value: **FFFH**
description: The contents of this register defines whether the interrupt service routines should be executed in 16 bit mode or in 32 bit mode. A high bit ('1') causes the core to switch to 32 bit mode when entering the interrupt service routine in question, a low bit ('0') indicates execution of the service routine in 16 bit mode. Bit positions are associated to interrupt sources as follows:
bit 0 – coprocessor 0, **bit 1 –** coprocessor 1, **bit 2 –** coprocessor 2,
bit 3 – coprocessor 3, **bit 4 –** interrupt 0, **bit 5 –** interrupt 1,
bit 6 – interrupt 2, **bit 7 –** interrupt 3, **bit 8 –** interrupt 4,
bit 9 – interrupt 5, **bit 10 –** interrupt 6, **bit 11 –** interrupt 7
notes: See Section 1.5 for more information about interrupts and Chapter 4 for more information about coprocessor.

symbol: **INT_MODE_UM[11..0]**
offset: **FH**
reset value: **FFFH**
description: The contents of this register defines whether the interrupt service routines should be executed in user mode or in super-user mode. A high bit ('1') causes the core to switch to user mode when entering the interrupt service routine in question; a low bit ('0') indicates execution of the service routine in super-user mode. Bit positions are associated to interrupt sources as follows:
bit 0 – coprocessor 0, **bit 1 –** coprocessor 1, **bit 2 –** coprocessor 2,
bit 3 – coprocessor 3, **bit 4 –** interrupt 0, **bit 5 –** interrupt 1,
bit 6 – interrupt 2, **bit 7 –** interrupt 3, **bit 8 –** interrupt 4,
bit 9 – interrupt 5, **bit 10 –** interrupt 6, **bit 11 –** interrupt 7
notes: See Section 1.5 for more information about interrupts and Chapter 4 for more information about coprocessor.

symbol: **INT_MASK[11..0]**
offset: **10H**
reset value: **000H**
description: Bits in this register can be used to block interrupts from individual sources. A low bit ('0') causes interrupt requests from the corresponding source to be blocked. A high bit ('1') allow requests to pass through. Bit positions are associated to interrupt sources as follows:
bit 0 – coprocessor 0, **bit 1 –** coprocessor 1, **bit 2 –** coprocessor 2,
bit 3 – coprocessor 3, **bit 4 –** interrupt 0, **bit 5 –** interrupt 1,
bit 6 – interrupt 2, **bit 7 –** interrupt 3, **bit 8 –** interrupt 4,
bit 9 – interrupt 5, **bit 10 –** interrupt 6, **bit 11 –** interrupt 7
notes: This mask register does not prevent interrupt requests from entering the INT_PEND register.

symbol: **INT_SERV[11..0]**
offset: **11H**
reset value: **000H**
description: This is a read-only status register having a flag for each interrupt source. A high flag ('1') means that an interrupt request from the corresponding source has been accepted. In practice this means that the interrupt service routine is being executed or it was executed until another request with higher priority interrupted the service routine. In this case there is multiple flags high in the INT_SERV register. Executing `reti` instruction at the end of an interrupt

service routine will cause the corresponding flag to go low. Bit positions are associated to interrupt sources as follows:
bit 0 – coprocessor 0, **bit 1 –** coprocessor 1, **bit 2 –** coprocessor 2,
bit 3 – coprocessor 3, **bit 4 –** interrupt 0, **bit 5 –** interrupt 1,
bit 6 – interrupt 2, **bit 7 –** interrupt 3, **bit 8 –** interrupt 4,
bit 9 – interrupt 5, **bit 10 –** interrupt 6, **bit 11 –** interrupt 7

notes: See Section 1.5 for more information about interrupts.

symbol: **INT_PEND[11..0]**
offset: **12H**
reset value: **000H**
description: This is a read-only status register having a flag for each interrupt source. A high flag ('1') means that an interrupt request from the corresponding source has been detected and is waiting to get accepted. A flag is lowered once the request is accepted and the service routine started. Bit positions are associated to interrupt sources as follows:
bit 0 – coprocessor 0, **bit 1 –** coprocessor 1, **bit 2 –** coprocessor 2,
bit 3 – coprocessor 3, **bit 4 –** interrupt 0, **bit 5 –** interrupt 1,
bit 6 – interrupt 2, **bit 7 –** interrupt 3, **bit 8 –** interrupt 4,
bit 9 – interrupt 5, **bit 10 –** interrupt 6, **bit 11 –** interrupt 7

notes: See Section 1.5 for more information about interrupts or additional interrupt document on how to clear the INT_PEND register by software.

symbol: **EXT_INT_PRI[31..0]**
fields: **PRI7[31..28], PRI6[27..24], PRI5[23..20], PRI4[19..16], PRI3[15..12], PRI2[11..8], PRI1[7..4], PRI0[3..0]**
offset: **13H**
reset value: **00000000H**
description: This register is used to set priorities for external interrupt sources. Each interrupt source is associated with a four bit unsigned value in range from 0 to 15, 0 meaning highest priority. Bitfield **PRI***X* is associated with external interrupt number *X*. X ranges from 0 to 7.

notes: Internal timers of COFFEE can be configured to generate interrupts in which case the timer in question is associated to one of the external interrupts => priority of a timer interrupt shall also be set using EXT_INT_PRI register. See Section 1.5 for more information about interrupts.

symbol: **COP_INT_PRI[15..0]**
offset: **14H**

fields:	**PRI3[15..12], PRI2[11..8], PRI1[7..4], PRI0[3..0]**
reset value:	**0000H**
description:	This register is used to set priorities for coprocessor interrupts/exceptions. Each coprocessor is associated with a four bit unsigned value in range from 0 to 15, 0 meaning highest priority. Bitfield **PRI***X* is associated with coprocessor number *X*. X ranges from 0 to 3.
notes:	See Section 1.5 for more information about interrupts.

symbol:	**EXCEPTION_CS[7..0]**
offset:	**15H**
reset value:	**00H**
description:	This is a read-only register which is used to report the cause of an exception to an exception handler.
notes:	See Section 1.5 for more information about interrupts.

symbol:	**EXCEPTION_PC[31..0]**
offset:	**16H**
reset value:	**00000000H**
description:	This is a read-only register which is used to report the memory address of the instruction which caused an exception. Can be used by exception handler.
notes:	See Section 1.5 for more information about interrupts.

symbol:	**EXCEPTION_PSR[7..0]**
offset:	**17H**
reset value:	**00H**
description:	Contains a copy of processor status flags (PSR) which were valid when the instruction causing an exception was decoded. Can be used by exception handler.
notes:	See Section 1.4 for more information about exceptions.

symbol:	**DMEM_BOUND_LO[31..0]**
offset:	**18H**
reset value:	**00000000H**
description:	This register is used to set the lower limit of a continuous address space for data memory protection. Accesses inside the area defined together with DMEM_BOUND_HI register are either allowed in user mode or blocked while in user mode (allowing accesses outside the area only) depending on memory protection flags in MEM_CONF register. In super user mode the whole address space is accessible.
notes:	The CCB block itself can be protected from user level code by mapping it to protected address space. See Chapter 6 for more details about programming considerations.

symbol:	**DMEM_BOUND_HI[31..0]**
offset:	**19H**
reset value:	**FFFFFFFFH**
description:	This register is used to set the upper limit of a continuous address space for data memory protection. Accesses inside the area defined together with DMEM_BOUND_LO register are either allowed in user mode or blocked while in user mode (allowing accesses outside the area only) depending on memory protection flags in MEM_CONF register. In super user mode the whole address space is accessible.
notes:	The CCB block itself can be protected from user level code by mapping it to protected address space. See Chapter 6 for more details about programming considerations.

symbol:	**IMEM_BOUND_LO[31..0]**
offset:	**1AH**
reset value:	**00000000H**
description:	This register is used to set the lower limit of a continuous address space for instruction memory protection. Fetching instructions from addresses inside the area defined together with IMEM_BOUND_HI register are either allowed in user mode or blocked while in user mode (allowing accesses outside the area only) depending on memory protection flags in MEM_CONF register. In super user mode the whole address space is accessible.
notes:	See Chapter 6 for more details about programming considerations.

symbol:	**IMEM_BOUND_HI[31..0]**
offset:	**1BH**
reset value:	**FFFFFFFFH**
description:	This register is used to set the upper limit of a continuous address space for instruction memory protection. Fetching instructions from addresses inside the area defined together with IMEM_BOUND_LO register are either allowed in user mode or blocked while in user mode (allowing accesses outside the area only) depending on memory protection flags in MEM_CONF register. In super user mode the whole address space is accessible.
notes:	See Chapter 6 for more details about programming considerations.

symbol:	**MEM_CONF[1..0]**
offset:	**1CH**
reset value:	**3H**

description:	This register contains flags which control the protection of address spaces defined by the contents of registers DMEM_BOUND_LO, DMEM_BOUND_HI, IMEM_BOUND_LO, IMEM_BOUND_HI. Flag in the bit position 0 controls protection of instruction memory and flag in the bit position 1 controls protection of data memory. If the respective flag is high ('1') the address space between the low and high boundaries (boundaries included) is not allowed to be accessed in user mode. If the flag is low ('0') then only the address space between the limits (boundaries included) is allowed to be accessed in user mode.
notes:	See Chapter 6 for more details about programming considerations.

symbol:	**SYSTEM_ADDR[31..0]**
offset:	**1DH**
reset value:	**00000001H**
description:	The contents of this register defines the entry address of system call handler. When executing scall instruction the address in this register will be loaded to program counter.
notes:	

symbol:	**EXCEP_ADDR[31..0]**
offset:	**1EH**
reset value:	**00000001H**
description:	The contents of this register defines the entry address of an exception handler. When an instruction causes an illegal event the address in this register will be loaded to program counter.
notes:	See Section 1.4 for more information about exceptions.

symbol:	**BUS_CONF[11..0]**
fields:	**CBUS_WC[11..8], DBUS_WC[7..4], IBUS_WC[3..0],**
offset:	**1FH**
reset value:	**FFFH**
description:	This register is used to set the amount of wait cycles per bus access. Data memory, instruction memory and coprocessor buses can be configured separately. The number of wait cycles can be set to a value in range 0 to 15. Bit fields are associated to different buses as follows: **CBUS_WC –** coprocessor bus, **DBUS_WC** – data memory bus, **IBUS_WC** – instruction memory bus. For maximum performance, number of access cycles (start cycle + wait cycles) should be set to smallest possible value. With zero wait cycles; the memory/coprocessor in question has to be able to respond in shorter time than one clock cycle (asynchronous operation).

notes: See COFFEE interface document.

symbol: **COP_CONF [27..0]**
fields: **C3_IF[27..26], C2_IF[25..24], C1_IF[23..22], C0_IF[21..20], C3_IR[19..15], C2_IR[14..10], C1_IR[9..5], C0_IR[4..0]**
offset: **20H**
reset value: **0000000H**
description: This register is used to configure the behaviour of coprocessor interface. The coprocessor interface can operate in COFFEE native mode or MIPS compliant mode. The mode can be selected for each coprocessor separately: **C3_IF –** interface mode of coprocessor 3, **C2_IF –** interface mode of coprocessor 2, **C1_IF –** interface mode of coprocessor 1, **C0_IF –** interface mode of coprocessor 0. Use value 0 for COFFEE native mode and value 1 for MIPS mode.

Fields **C0_IR** through **C3_IR** specify index of the instruction register of the coprocessor in question. When COFFEE core encounters a coprocessor instruction it writes the instruction word to coprocessor bus and drives `cop_rgi` signal according to corresponding **C*X*_IR** field. A value from 0 to 31 can be specified. Fields are associated to coprocessors as follows: **C3_IR** – coprocessor 3 instruction register, **C2_IR** – coprocessor 2 instruction register, **C1_IR** – coprocessor 1 instruction register, **C0_IR** – coprocessor 0 instruction register.
notes: In COFFEE core version 1.0 only COFFEE native mode is supported (C*X*_IF fields are ignored)

symbol: **TMR0_CNT[31..0]**
offset: **21H**
reset value: **00000000H**
description: This register contains the current value of the internal timer counter 0. Can be used to set initial value to counter 0.
notes: See document about timers.

symbol: **TMR0_MAX_CNT[31..0]**
offset: **22H**
reset value: **00000000H**
description: This register is used to define maximum value for timer counter 0. After reaching maximum value the counter will be loaded with zero. A value greater than defined by this register can be written to TMR0_CNT in which case the counter will count to FFFFFFFFH before starting from zero.

notes: See document about timers.

symbol: **TMR1_CNT[31..0]**
offset: **23H**
reset value: **00000000H**
description: This register contains the current value of the internal timer counter 1. Can be used to set initial value to counter 1.
notes: See document about timers.

symbol: **TMR1_MAX_CNT[31..0]**
offset: **24H**
reset value: **00000000H**
description: This register is used to define maximum value for timer counter 1. After reaching maximum value the counter will be loaded with zero. A value greater than defined by this register can be written to TMR1_CNT in which case the counter will count to FFFFFFFFH before starting from zero.
notes: See document about timers.

symbol: **TMR_CONF[31..0]**
fields: **TMR1_CONF[31..16], TMR0_CONF[15..0]**
offset: **25H**
reset value: **00000000H**
description: This register is used to configure both internal timers: timer0 and timer1. See the timer document for explanation of bit-fields in **TMR1_CONF** and **TMR0_CONF**.
notes: See document about timers.

symbol: **RETI_ADDR[31..0]**
offset: **26H**
reset value: **FFFFFFFFH**
description: The address in this register will be loaded to program counter when executing `reti` instruction. When entering an interrupt service routine this register contains a valid return address by default. Return to different address can be forced by writing the desired return address to this register before executing `reti`.
notes: Interrupts should be disabled when writing to this register. See Section 1.5 for more information about interrupts.

symbol: **RETI_PSR[7..0]**
offset: **27H**
reset value: **0EH**
description: The contents of this register will be written to PSR register when executing `reti` instruction. When entering an interrupt service routine this register contains PSR flags

from the interrupted context. Return with modified flags can be forced by writing the desired flags to this register before executing `reti`.

notes: Interrupts should be disabled when writing to this register. See Section 1.5 for more information about interrupts.

symbol: **RETI_CR0[2..0]**
offset: **28H**
reset value: **0H**
description: The contents of this register will be written to flag register C0 when executing `reti` instruction. When entering an interrupt service routine this register contains C0 flags from the interrupted context. Return with modified flags can be forced by writing the desired flags to this register before executing `reti`.

notes: Interrupts should be disabled when writing to this register. See Section 1.5 for more information about interrupts.

1.1.2 Coprocessor Registers

Milk coprocessor has 8 general purpose 32-bit registers for arithmetic operands and results storage. Two special purpose registers are present in the architecture: status register and control register.

1.1.2.1 Status Register

It' s a 32-bit register shown in Figure 1-2.

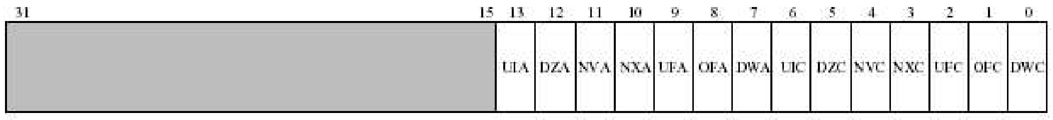


Figure 1-2. Coprocessor status register

Bits 31..14 are not used in the current implementation, and it' s assumed they all are zeroes.
Bits 13..7 are the flag bits related to floating-point exceptions, and they refer to the whole computation since last reset or last writing from user.
Bits 6..0 are the same flags, but they refer to the last executed instruction only.

1.1.2.2 Control Register

The content of the control register is shown in Figure 1-3.

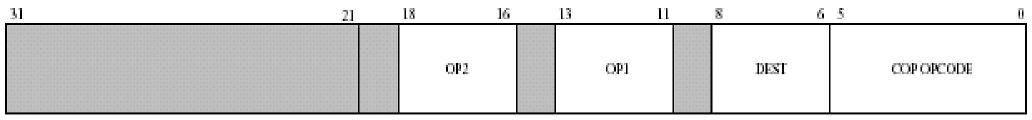


Figure 1-3. Coprocessor control register

Bits 31..26 contain the encoding of the cop instruction of COFFEE™ RISC core.
Bits 25..24 are used to index one among the 4 coprocessors that can be attached to COFFEE™ RISC core.
Bits 23..22 are unused.
Bit 21 specifies the floating-point precision. Milk coprocessor currently supports only single precision, and this bit is always 0.
Bits 20..16 are the address of the second operand' s source register. When the current instruction supports only on operand this field is ignored. Note that since only 8 registers are present in the architecture, bits 20 and 19 are not used.
Bits 15..11 are the address of the first operand' s source register. Note that since only 8 registers are present in the architecture, bits 15 and 14 are not used.
Bits 10..6 are the address of the destination' s register. Note that since only 8 registers are present in the architecture, bits 10 and 9 are not used.
Bits 5..0 are the opcode of the current instruction performed by Milk.

1.2 Bit and Byte Ordering

A system' s byte ordering scheme, or endian scheme, affects memory organization and defines the relationship between address and byte position of data in memory:

- Big-endian systems store the sign bit in the lowest address byte
- Little-endian systems store the sign bit in the highest address byte

COFFEE™ RISC uses the big-endian byte scheme. Byte ordering is as follows:

- The bytes of a **longword** (64-bit) are numbered from 0 to 7. Byte 0 holds the sign and most significant bits
- The bytes of a **word** (32-bit) are numbered from 0 to 3. Byte 0 holds the sign and most significant bits
- The bytes of a **halfword** (16-bit) are numbered from 0 to 1. Byte 0 holds the sign and most significant bits

The bits of each byte are numbered from 7 to 0, using the format shown in Figure 1-4.

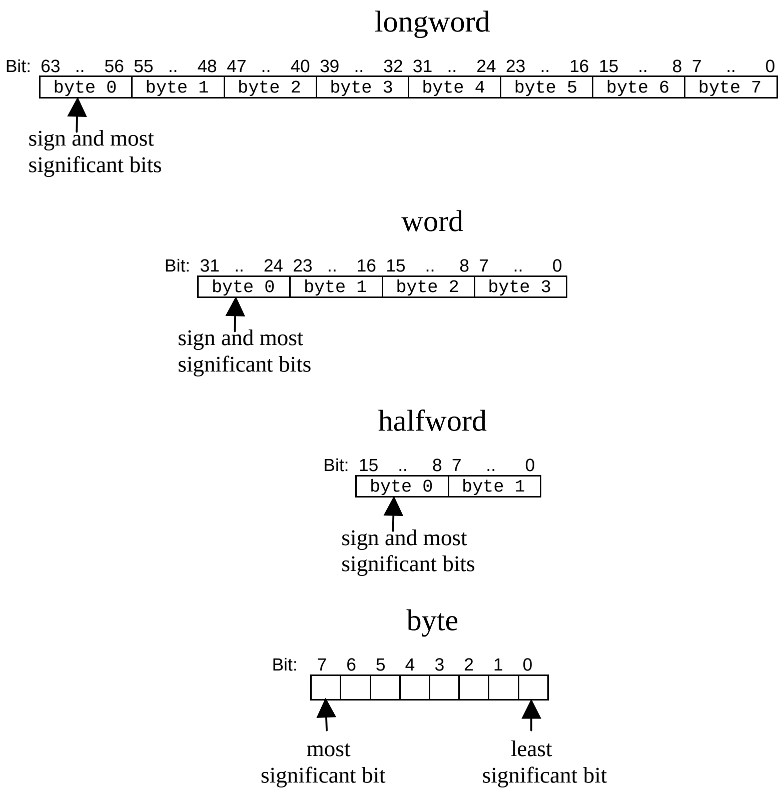

longword
Bit: 63 .. 56 55 .. 48 47 .. 40 39 .. 32 31 .. 24 23 .. 16 15 .. 8 7 .. 0
byte 0 | byte 1 | byte 2 | byte 3 | byte 4 | byte 5 | byte 6 | byte 7
sign and most significant bits
word
Bit: 31 .. 24 23 .. 16 15 .. 8 7 .. 0
byte 0 | byte 1 | byte 2 | byte 3
sign and most significant bits
halfword
Bit: 15 .. 8 7 .. 0
byte 0 | byte 1
sign and most significant bits
byte
Bit: 7 6 5 4 3 2 1 0
most significant bit
least significant bit

Figure 1-4. Bit and byte order

1.3 Addressing

COFFEE core can only address full words. This is alleviated by providing special instructions for fast extraction and merging of bytes/halfwords. Based addressing is supported by hardware while others must be synthesized by software. Two instructions are provided for accessing data in memory: `ld` for loading a word from memory and `st` for storing a word to memory. See Chapter 3 for more information about main instruction set.

1.4 Exceptions

In this document an *exception* means an event that will halt the processing in the current context immediately and cause the core to switch to an exception handling routine. An exception is considered an error condition and has to be dealt with immediately. Exceptions are listed in Table 1-2.
An instruction causing an exception is canceled and execution of an exception handler is started at an address defined in CCB register EXCEP_ADDR. Before switching to the exception handler status information is saved to following CCB registers:

EXCEPTION_PC – memory address of the violating instruction,
EXCEPTION_PSR – PSR flags used to when decoding violating instruction, EXCEPTION_CS – Exception code, see table below.
The exception handler will be started in superuser mode, interrupts disabled.

Note that very often in literature an exception means interrupting the processor in general. See also Section 1.5 for information about interrupts.

For more information about exceptions see additional exception documentation.

Code	Name	Description
0	Instruction address violation[1]	While in user mode, instruction is fetched from memory address not allowed for user
1	Unknown opcode	Version 1.0 of COFFEE™ RISC does not have any unused opcodes which makes this obsolete
2	Illegal instruction	While in 16-bit mode, trying to execute an instruction which is valid only in 32-bit mode or trying to execute a superuser only instruction in user mode
3	Miss aligned jump address[2]	Calculated jump target is not aligned to word (32-bit mode) or halfword (16-bit mode) boundary
4	Jump address overflow	A PC relative jump below the bottom of the memory or above the top of the memory
5	Miss aligned instruction address[3]	Instruction address is not aligned according to mode, this can be caused by: • External boot address was not aligned to word boundary • An interrupt vector is not properly aligned or interrupt mode is not correctly set • System entry address is not aligned to word boundary
224...255	trap[4]	Processor encountered a trap instruction
6	Arithmetic overflow	The result of a signed arithmetic operation exceeds 2^{31} - 1 or falls below -2^{31}
7	Data address violation	While in user mode, a data address refers to memory address nor allowed for user
8	Data address overflow	Trying to index data below the bottom or above the top of the memory
9	Illegal jump	Trying to jump to protected instruction memory area while in user mode
10...15		Reserved for future extensions

Table 1-2. Exception types and codes

Notes for Table 1-2:

[1] If sequential execution traverses the boundary of the protected instruction memory area, the address of the instruction pointed to is saved.

[2] A jump between memory areas using different encoding will result in unpredictable behaviour.

[3] In this case, the address is saved, since it cannot be known which instruction (if any) caused the exception.

[4] For software exceptions (such as division by zero, or array bounds exceeded). Exception address will point to `trap` instruction. Note, that you cannot generate hardware exceptions using `trap` instruction because trap code will be padded with ones.

1.5 Interrupts

In this document an interrupt is defined as an event that causes hardware assisted context switch because an external/internal device is requesting time from CPU (Central Processing Unit). This is the normal way to interrupt a processor. Interrupt requests can originate, for example, from a timer or an external IO (Input/Output) device, coprocessor etc. This section covers the built-in interrupt controller of COFFEE™ core.

COFFEE™ core supports connecting eight external interrupt sources directly. If coprocessors are not connected, four inputs reserved for coprocessor exception signalling can be used as interrupt request lines giving possibility to connect twelve sources directly. Built-in timers can also be configured to generate interrupts. This feature can be used for example to switch execution to an operating system kernel in multitasking systems.

All interrupts are vectored. Interrupt vectors reside in CCB. With built-in interrupt controller the entry address of an interrupt service routine is the corresponding vector directly. If an external controller is used the entry address is combination of the vector and an offset given externally: ISR_ENTRY = BASE + (OFFSET x 16), where

BASE = EXT_INT*X*_VEC[31..12],
OFFSET = provided by an external controller,
ISR_ENTRY = entry address of an interrupt service routine.

Once an interrupt request is detected, it is saved in a register called INT_PEND, which is visible via CCB. In order to interrupt the core, a pending request has to pass priority check and masking. To pass, the following conditions have to be valid: IE flag in processor status register must be set, Interrupt mask register (INT_MASK) has to have a high bit (‘1’) in the corresponding position, no interrupts with higher priority are pending or in service, and instructions currently on pipeline do not cause exceptions. Once a pending request gets through, the control unit of COFFEE™ core will initiate context switch as soon as possible.

The following steps are taken when switching to an interrupt service routine. Return address, processor status register and condition register C0 are saved to hardware stack. (The top of the hardware stack is visible as three separate registers in CCB). The start address of an interrupt service routine is calculated and written to the program counter. The bit corresponding to the interrupt source is set in INT_SERV register and cleared from INT_PEND register. Further interrupts are disabled by clearing IE flag from PSR. Signal INT_ACK is pulsed to inform an external interrupt controller that a request got through and is now in service. Finally, execution of an interrupt service routine is started in mode defined by CCB registers INT_MODE_IL and INT_MODE_UM.

Returning from an interrupt service routine is done by executing `reti` instruction. Execution of `reti` instruction causes the state of the core to be restored from hardware stack. By default, execution resumes from the address, which was saved to hardware stack when entering service routine. If execution is desired to be resumed from a different context the hardware stack can be modified by writing suitable values to CCB registers RETI_ADDR, RETI_PSR and RETI_CR0. Signal INT_DONE is pulsed to inform an external interrupt controller that handling the latest acknowledged request has ended. The corresponding bit is cleared from INT_SERV register.

Priorities between interrupt sources can be set by software via CCB registers. Interrupt sources can be masked individually via CCB mask register and disabled or enabled all at once using `di` and `ei` instructions. *If internal interrupt handler is used, the priorities between sources can be set by software, with external handler, priorities will be fixed according to Table 1-3*. Note that priorities for coprocessor exceptions/interrupts can always be set by software. If multiple sources have the same priority, resolving is performed internally in the following order (COP0_INT having the highest priority):
COP0_INT, COP1_INT, COP2_INT, COP3_INT,
EXT_INT0, EXT_INT1, EXT_INT2, EXT_INT3,
EXT_INT4, EXT_INT5, EXT_INT6, EXT_INT7.

A request with higher priority can interrupt the current service routine if interrupts have been re-enabled in the routine with `ei` instruction (nesting of interrupts).

Priority	Name
Software controlled	Coprocessor number 0 exception/interrupt
	Coprocessor number 1 exception/interrupt
	Coprocessor number 2 exception/interrupt
	Coprocessor number 3 exception/interrupt
15	External interrupt 0
15	External interrupt 1
15	External interrupt 2
15	External interrupt 3
15	External interrupt 4
15	External interrupt 5
15	External interrupt 6
15	External interrupt 7

Table 1-3. Interrupt priorities if external handler is used, 0 - highest

Do not do this!

Do not change interrupt priorities while in interrupt service routine if you use nested interrupts (unless you are 100% sure that a new request from a source cannot arise before a service routine is finished). In extreme cases this can lead to hardware stack overflow if interrupt nesting level is twelve and priorities are changed so that multiple requests from a single source can be active simultaneously. Normally an interrupt service routine cannot be interrupted by a new request from the same source because of priority resolving.

In Table 1-4 is a summary of the registers of the built-in interrupt controller. All the registers are accessed via CCB. See Section 1.1 for more information about registers.

For more information about interrupts see additional interrupt documentation.

Symbol	Usage
COP0_INT_VEC	Entry address of an interrupt service routine for coprocessor 0.
COP1_INT_VEC	Entry address of an interrupt service routine for coprocessor 1.
COP2_INT_VEC	Entry address of an interrupt service routine for coprocessor 2.
COP3_INT_VEC	Entry address of an interrupt service routine for coprocessor 3.
EXT_INT0_VEC	Base/entry address of an interrupt service routine for interrupt 0.
EXT_INT1_VEC	Base/entry address of an interrupt service routine for interrupt 1.
EXT_INT2_VEC	Base/entry address of an interrupt service routine for interrupt 2.
EXT_INT3_VEC	Base/entry address of an interrupt service routine for interrupt 3.
EXT_INT4_VEC	Base/entry address of an interrupt service routine for interrupt 4.
EXT_INT5_VEC	Base/entry address of an interrupt service routine for interrupt 5.
EXT_INT6_VEC	Base/entry address of an interrupt service routine for interrupt 6.
EXT_INT7_VEC	Base/entry address of an interrupt service routine for interrupt 7.
INT_MODE_IL	Instruction decoding mode flags for interrupt routines.
INT_MODE_UM	User mode flags for interrupt routines.
INT_MASK	Mask register for blocking requests.
INT_SERV	Interrupt service status bits (read-only).
INT_PEND	Pending interrupt requests (read-only).
EXT_INT_PRI	Register for defining priorities of interrupt requests.
COP_INT_PRI	Register for defining priorities of interrupt requests from coprocessors.
RETI_ADDR	Top of hardware stack, program counter of an interrupted context.
RETI_PSR	Top of hardware stack, processor status of an interrupted context.
RETI_CR0	Top of hardware stack, condition register C0 of an interrupted context.

Table 1-4. Build-in interrupt controller register

2. LEXICAL CONVENTIONS

This chapter describes lexical conventions associated with the following items:

- Blank and Tab Characters (Section 2.1)
- Comments (Section 2.2)
- Identifiers (Section 2.3)
- Constants (Section 2.4)
- Physical lines (Section 2.5)
- Statements (Section 2.6)
- Expressions (Section 2.7)
- Macros (Section 2.8)
- Conditional Execution (Section 2.9)
- Sections (Section 2.10)
- Location Counters (Section 2.11)
- Relocations (Section 2.12)

2.1 Blank and Tab Characters

You can use blank and tab characters anywhere between operators, identifiers, and constants. Adjacent identifiers or constants that are not otherwise separated must be separated by a blank or tab.
These characters can also be used within character constants; however, they are not allowed within operators and identifiers

2.2 Comments

The double slash (//) and semicolon (;) introduces a comment. Comments that start with a '//' (or ';') extend through the end of the line on which they appear.
Block comments are not supported.

2.3 Identifiers

An identifier consists of a case-sensitive sequence of alphanumeric characters (A-Z, a-z, 0-9) and the following special character:

- **.** (period)

Identifiers can be up to 31 characters long, and the first character cannot be numeric (0-9).
If an undefined identifier is referenced, the assembler assumes that the identifier is an external symbol. The assembler treats the identifier like a name specified by a `.global` directive (see Chapter 5 for more information about directives).

If the identifier is defined to the assembler and the identifier has not been specified as global, the assembler assumes that the identifier is a local symbol.

2.4 Constants

The assembler supports the following constants:

- Scalar constants (Section 2.4.1)
- Floating-point constants (Section 2.4.2)
- String constants (Section 2.4.3)

2.4.1 Scalar Constants

The assembler interprets all scalar constants as two' s complement numbers. Scalar constants can be any of the digits *0123456789abcdefABCDEF*.
Scalar constants can be decimal, binary, hexadecimal, or octal constants:

- Decimal constants consist of a sequence of decimal digits (*0*-*9*) without a leading zero.
- Binary constants consist of the characters *0b* (or *0B*) followed by a sequence of binary digits (*01*).
- Hexadecimal constants consist of the characters *0x* (or *0X*) followed by a sequence of hexadecimal digits (*0-9abcdefABCDEF*).
- Octal constants consist of the characters *0c* (or *0C*) followed by a sequence of octal digits (*0*-*7*).

2.4.2 Floating-Point Constants

Floating-point constants can appear only in floating-point directives (see Chapter 5 for more information about directives) and in the coprocessor floating-point instructions (see Chapter 4 for more information about coprocessor instructions). Floating-point constant should be defined like follows: digit *zero* followed by *f/F* followed by *sign* (optional) followed by *integer1* (represents fraction part) followed by *e/E* followed by *sign* of exponent (optional) and finally an *integer2* representing exponent:

```
0f|F[+|-]<integer1>e|E[+|-]<integer2>
```

For example, the number .02173 should be represented as follows:

```
.float 0F2173E-5
```

Hexadecimal floating-point constants are not supported.

The assembler does not use any rounding mode to convert floating-point constants.

2.4.3 String Constants

All characters except the newline character are allowed in string constants. String constants begin and end with double quotation marks (”).
The assembler observes some of the backslash conventions used by the C language. Table 2-1 shows the assembler' s backslash conventions.

Convention	**Meaning**
\n	Newline (0x0a)
\0	End of string (0x00)
\r	Carriage return (0x0d)
\t	Horizontal tab (0x09)
\\	Backslash (0x05)
\”	Quotation mark (0x22)

Table 2-1. Backslash conventions

2.5 Multiple Lines per Physical Line

You cannot include multiple statements on the same line.

2.6 Statements

The assembler supports the following types of statements:
- Null statements
- Keyword statements

Each keyword statement can include an optional label, an operation code (mnemonic or directive), and zero or more operands (with an optional comment following the last operand on the statement):

```
[label:] opcode operand [// | ; comment]
```

2.6.1 Labels

A label definition consists of an identifier followed by a colon (:). (See Section 2.3 for the rules governing identifiers.) Label definitions assign the current value and type of the location counter to the name. An error results when the name is already defined.
A label definition always ends with a colon. You can put a label definition on a line by itself.
Numeric labels are not supported.

2.6.2 Null Statement

A null statement is an empty statement that the assembler ignores. Null statements can have label definitions. For example:

```
label:     // some comment
```

2.6.3 Keyword Statement

A keyword statement contains a predefined keyword. The syntax for the rest of the statement depends on the keyword. Keywords are either assembler instructions (mnemonics) or directives.
Assembler instructions in the main instruction set and the coprocessor instruction set are described in Chapter 3 and Chapter 4, respectively. Assembler directives are described in Chapter 5.

2.7 Expressions

An expression is a sequence of symbols and operations that represents a value. An expression specifies a numeric value. This value can be an address, immediate value, or constant. Arguments can be constants or symbols.

operator	description
*	multiplication
/	division
+	addition
-	subtraction
()	grouping parenthesis

Table 2-2. Supported operators in expressions

2.8 Macros

It is possible to define macro with `.macro` directive. Macro should finish with `.endm` directive.
In second line of macro it is possible to define local macro labels using directive `.local`.
Macro can have parameters. Two macros cannot have the same name. Number of call parameters and defined parameters should be the same. See Chapter 5 for more information about directives.

2.9 Conditional Execution

Conditional execution syntax is as follows:

```
if (cond, cr) instruction
```

where *cond* is as specified in the Table 2-3 and *cr* is c0, c1, c2, c3, c4, c5, c6 or c7.

Conditional execution isn't allowed in 16-bit mode. Just if condition register is C0, conditional execution is not a syntax error in 16-bit mode, but is expanded like this:

```
bcond      4
nop
instruction
```

mnemonic	condition	explanation	code	flags
c	carry	Carry out of MSB	000	C = 1
eq	=	equal	011	Z = 1
gt	>	greater than	100	Z = 0 & N = 0
lt	<	less than	101	Z = 0 & N = 1
ne	≠	not equal	110	Z = 0
elt	≤	equal or less than	010	Z = 1 or N = 1
egt	≥	equal or greater than	001	Z = 1 or N = 0
nc	!carry	No carry-out	111	C = 0

Table 2-3. Condition codes and mnemonics

2.10 Sections

Default sections and their usual meanings:

- **.bss** (block started by symbol) – zero initialized data (and uninitialized data)
- **.text** – PC relative stuff (might be code, might be data)
- **.data** – initialized data
- **.rdata** – read-only data

User is able to define additional sections using `.section` directive. These could be used to allocate some "special" data or code. See Chapter 5 for more information about directives.
Subsections (e.g., `.text 0` ... `.text N`) aren't supported.

Absolute section can be defined like this:

```
.section OS_SEC, d, 0xABCD0000
.section OS_SEC, 0xABCD0000
.data 0x10000000
```

When an assembler sees one of the section directives: `.bss`, `.text`, `.data`, `.rdata` or `.section` it switches to a location counter of that particular section (also to a working mode of that particular section).

If a section was not defined before in the current file, the location counter in question is set to zero.

If none section is defined in 1st line, it is count to be **text section**.
Note: it is needed to define section mode using `.codeXX` directive immediately after section description directive. This creates internal subsections depending on coding mode. All subsections in output file are in the same order like in source file and have own section header (See Section 7.2.3 for more information about section header).
If mode isn't set, it is assumed to be 32 -bit mode, but then Instruction Simulator will not work properly.

Section order in output file is shown in Figure 7-2.
See Chapter 7 for more information about COFF output file.

2.11 Location Counter

The smallest addressable unit is assumed to be one byte, which means, that any location counter (each section has a location counter) is incremented by an amount equal to the amount of bytes produced by an assembly language statement. For example, the following statement produces 12 bytes and increments the location counter by 12:

```
          .ascii     "Hello world\0"
```

Following statement increments the locations counter by 4 if 32-bit encoding is used and by 2 if 16-bit encoding is used:

```
          addi       R1, 0xff
```

COFFEE™ core does not support byte accesses even though software tools expect it to! To make this work we throw away two address bits and say goodbye to 16GB address space where each consecutive address refers to 32-bit word. **Note:** there are SEVERE LIMITATIONS.

The assembler is not expected to automatically align data allocations; it gives error messages of miss-aligned cases. Words should start on word boundary, halfwords on halfword boundary. Byte can be anywhere (byte boundary). 32-bit instructions should start on word boundary. 16-bit instruction should start on halfword boundary.
See Chapter 5 for more information about directives `.org` and `.align`.

Assembler does some alignment on end of section.

2.12 Relocations

It is impossible to specify a relocation type explicitly in assembly code. Assembler sets all types of relocation internally and produces the special relocation information (assembler supports COFF format).
All relocation references are done with assumptions that all sections starts on address 0x00.
See Section 7.2.5 for more details about relocation information.

3. MAIN INSTRUCTION SET

This chapter describes instruction notation and discusses assembler instructions for the main processor. Chapter 4 describes coprocessor notation and instructions.
Section 3.1 contains instruction set summary tables.
The assembler' s main instruction set contains the following classes of instructions:

- Integer arithmetic instructions (Section 3.2)
- Byte and bit field manipulation instructions (Section 3.3)
- Boolean bitwise operation instructions (Section 3.4)
- Branch (conditional jump) instructions (Section 3.5)
- Jump instructions (Section 3.6)
- Integer comparison instructions (Section 3.7)
- Shift instructions (Section 3.8)
- Memory load and store, data moving instructions (Section 3.9)
- Coprocessor instructions (Section 3.10)
- Miscellaneous instructions (Section 3.11)
- Pseudo instructions (Section 3.12)

The abbreviations used this chapter are listed in Table 3-1.

Abbreviation	Description
creg	Condition register specifier creg ∈ {c0,c1,c2,c3,c4,c5,c6,c7}
cond	Condition specifier, see table 2-3. cond ∈ {c,eq,gt,lt,ne,elt,egt,nc}
dreg, sreg, sreg1, sreg2,	Register specifiers: dreg – destination register ∈ reg32 sreg, sreg1, sreg2 – source registers ∈ reg32 reg32 = {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16, r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31}
dr, sr, sr1, sr2	Register specifiers: dr – destination register ∈ reg8 sreg, sreg1, sreg2 – source registers ∈ reg8 reg8 = {r24,r25,r26,r27,r28,r29,r30,r31}
imm, imm1, imm2	Scalar or symbolic constant or an expression revealing a constant. See Table 3-14 for allowed values
cp_sreg, cp_dreg	coprocessor source and destination register specifiers respectively ∈ {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16, r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31}

Table 3-1. Abbreviations used in main instruction set

Notes about instruction definitions

16-bit mode refers to instruction word length. Data is manipulated in 32-bit words except with 16-bit multiplication instructions.

If the syntax of an instruction is different in 16-bit mode than in 32-bit mode then both syntaxes are presented: First the 32-bit version and then 16-bit version. If both syntaxes are similar (or the particular instruction is not defined in 16-bit mode) then only one is presented.

Optional parameters for conditional execution are enclosed in brackets.

Conditional execution is not allowed in 16-bit mode.

3.1 Summary of Machine Instructions

Tables from Table 3-4 to Table 3-11 presents a summary of machine instructions implemented in COFFEE™ core. The exact behaviour of instructions is illustrated using RTN notation (Register Transfer Notation), which is explained in Table 3-2.

Operator	Description
$\leftarrow$	Register transfer. Left hand side of the operator is target and right hand side is source.
[ ]	Memory index. Selects one item or a range of items. Most often one word.
< >	Bit index. Selects a bit or a range of bits.
n..m	Index range from n to m. Either n downto m or n to m.
$\rightarrow$	Condition operator. If value on left hand side is true, action or value on right hand side is yielded.
:=	Substitution (of dummy variables).
#	Concatenation. Bits on right are appended to bits on left.
:	Parallel separator. Used to list operations which are performed in parallel.
;	Sequential separator. Used to list operations which are performed sequentially. Left hand side performed first.
@	Repetition. Value on right hand side is repeated as many times as specified by value on left hand side. Values are concatenated.
{ }	Operation modifier. Refines preceding operation.
( )	Operation or value grouping. (evaluation order)
$= \neq < \leq > \geq$	Comparison. Evaluates to true (1) or false (0).
$+ - \times \div$	Arithmetic operators: addition, subtraction, multiplication, division.
$\wedge \vee \neg \oplus \equiv$	Logical operators: AND, OR, NOT, EXCLUSIVE OR, EQUIVALENCE.
<< >>	Left shift and right shift operators respectively.

Table 3-2. RTN notations used in Summary Tables

mnemonic	explanation
dr, sr, cr	Destination register index, source register index and condition register index respectively.
imm	Immediate constant embedded in instruction word, zero-extended by hardware.
simm	Immediate constant embedded in instruction word, sign-extended by hardware.
R	Currently visible register bank, a set of 32 registers or coprocessor register bank.
M	Ideal data memory which fills the 4GB address space, word addressed.
C	Condition register bank, a set of eight three-bit wide registers.
carry	Carry flag evaluated by compare instructions and some arithmetic instructions.
neg	Negative flag evaluated by compare instructions and some arithmetic instructions.
zero	Zero flag evaluated by compare instructions and some arithmetic instructions.
M64	Intermediate register, which contains a 64-bit product of previous 32-bit multiplication.
HWS	Hardware stack. Top of stack: HWS[0].
notes	
Symbols which are not defined in this table are dummy variables (or defined earlier in this manual?). If a bit field on right hand side of '←' operator is shorter than the destination on left hand side, the bit field is padded with zeros from left. The descending order of significance is from left to right (MSB equals bit index 31). Bit indexes of condition flags are Z: 2, N: 1, C: 0.	

Table 3-3. Notations used in Summary Tables

Mnemonic and operands	**Description**	**Formal definition (RTN)**

add dr, sr1, sr2	add signed integers	R[dr] ← R[sr1] + R[sr2]
		C[0] ← zero#neg#carry

addi dr, sr, simm		R[dr] ← R[sr]+simm
		C[0] ← zero#neg#carry

addiu dr, sr, imm	add unsigned integers	R[dr] ← R[sr]+imm
		C[0] ← zero#neg#carry

addu dr, sr1, sr2		R[dr] ← R[sr1]+R[sr2]
		C[0] ← zero#neg# carry

mulhi dr	evaluate upper 32 bits of previous integer multiplication	R[dr] ← M64<63..32>
muli dr, sr, simm	multiply signed integers	R[dr] ← R[sr1] × simm
muls dr, sr1, sr2		R[dr] ← R[sr1] × R[sr2]
mulu dr, sr1, sr2	multiply unsigned integers	R[dr] ← R[sr1] × R[sr2]
mulus dr, sr1, sr2	multiply unsigned integer with signed integer	R[dr] ← R[sr1] × R[sr2]
muls_16 dr, sr1, sr2	multiply signed integers (16-bit operands)	R[dr] ← R[sr1]<15..0> × R[sr2]<15..0>
mulu_16 dr, sr1, sr2	multiply unsigned integers (16-bit operands)	R[dr] ← R[sr1]<15..0> × R[sr2]<15..0>
mulus_16 dr, sr1, sr2	multiply unsigned integer with signed integer (16-bit operands)	R[dr] ← R[sr1]<15..0> × R[sr2]<15..0>

sub dr, sr1, sr2	subtract signed integers	R[dr] ← R[sr1] - R[sr2]
		C[0] ← zero#neg#carry

subu dr, sr1, sr2	subtract unsigned integers	R[dr] ← R[sr1] - R[sr2]
		C[0] ← zero#neg#carry

Table 3-4. Summary of integer arithmetic instructions

Mnemonic and operands	Description	Formal definition (RTN)
exb dr, sr, imm	extract byte from register	(imm = 0) $\rightarrow$ R[dr] $\leftarrow$ R[sr]<7..0> (imm = 1) $\rightarrow$ R[dr] $\leftarrow$ R[sr]<15..8> (imm = 2) $\rightarrow$ R[dr] $\leftarrow$ R[sr]<23..16> (imm = 3) $\rightarrow$ R[dr] $\leftarrow$ R[sr]<31..24>
exh dr, sr, imm	extract halfword from register	(imm = 0) $\rightarrow$ R[dr] $\leftarrow$ R[sr]<15..0> (imm = 1) $\rightarrow$ R[dr] $\leftarrow$ R[sr]<31..16>
exbf dr, sr1, sr2	extract arbitrary bit field from register	L := R[sr2]<10..5>; P := R[sr2]<4..0>; R[dr] $\leftarrow$ R[sr1]<P+L - 1..P>
exbfi dr, sr, imm1, imm2		L := imm1; P := imm2; R[dr] $\leftarrow$ R[sr1]<P + L - 1..P>
lli dr, imm	load lower part of register	R[dr] $\leftarrow$ 16@0#imm
lui dr, imm	load upper part of register	R[dr]<31..16> $\leftarrow$ imm

sext dr, sr1, sr2	sign extend value in register	P := R[sr2];
		R[dr]<31..P> $\leftarrow$ R[sr1]<P>@(32 – P):
		R[dr]<P – 1..0> $\leftarrow$ R[sr1]<P – 1..0>

sexti dr, sr, imm		P := imm;
		R[dr]<31..P> $\leftarrow$ R[sr]<P>@(32 – P):
		R[dr]<P – 1..0> $\leftarrow$ R[sr]<P – 1..0>

conb dr, sr1, sr2	concatenate bytes	R[dr] $\leftarrow$ 0@16 # R[sr1]<7..0> # R[sr2]<7..0>
conh dr, sr1, sr2	concatenate halfwords	R[dr] $\leftarrow$ R[sr1]<15..0> # R[sr2]<15..0>

Table 3-5. Summary of byte and bit field manipulation instructions

Mnemonic and operands	Description	Formal definition (RTN)
and dr, sr1, sr2	bitwise AND	R[dr] $\leftarrow$ R[sr1] $\wedge$ R[sr2]
andi dr, sr, imm		R[dr] $\leftarrow$ R[sr] $\wedge$ imm
not dr, sr	bitwise NOT	R[dr] $\leftarrow$ $\neg$R[sr]
or dr, sr1, sr2	bitwise OR	R[dr] $\leftarrow$ R[sr1] $\vee$ R[sr2]
ori dr, sr, imm		R[dr] $\leftarrow$ R[sr] $\vee$ imm
xor dr, sr1, sr2	bitwise XOR	R[dr] $\leftarrow$ R[sr1] $\oplus$ R[sr2]

Table 3-6. Summary of Boolean bitwise operation instructions

Mnemonic and operands	Description	Formal definition(RTN)
bc cr, simm		(C[cr]<0> = 1) → PC ← PC+(simm#0)
begt cr, simm		(C[cr]<2> = 1 ∨ C[cr]<1> = 0) → PC ← PC+(simm # 0)
belt cr, simm		(C[cr]<2> = 1 ∨ C[cr]<1> = 1) → PC ← PC+(simm#0)
beq cr, simm	branch on condition	(C[cr]<2> = 1) → PC ← PC+(simm#0)
bgt cr, simm		(C[cr]<2> = 0 ∧ C[cr]<1> = 0) → PC ← PC+(simm#0)
blt cr, simm		(C[cr]<1> = 1) → PC ← PC+(simm#0)
bnc cr, simm		(C[cr]<0> = 0) → PC ← PC+(simm#0)
bne cr, simm		(C[cr]<2> = 0) → PC ← PC+(simm#0)

jal simm	jump and save return address	(PSR<3> = 1) → INCREMENT:=4;
		(PSR<3> = 0) → INCREMENT:=2;
		R[31] ← PC+INCREMENT;PC ← PC+(simm#0)

jalr sr		(PSR<3> = 1) → INCREMENT:=4;
		(PSR<3> = 0) → INCREMENT:=2;
		R[31] ← PC+INCREMENT;PC ← R[sr]

jmp simm	jump	PC ← PC+(simm#0)
jmpr sr		PC ← R[sr]

Table 3-7. Summary of jump instructions

Mnemonic and operands	Description	Formal definition (RTN)
cmp cr, sr1, sr2	Compare contents of registers. When evaluating carry flag, unsigned comparison is used.	(R[sr1] = R[sr2]) → C[cr]<2> ← 1 (R[sr1] ≠ R[sr2]) → C[cr]<2> ← 0 (R[sr1] < R[sr2]) → C[cr]<1> ← 1 (R[sr1] ≥ R[sr2]) → C[cr]<1> ← 0 (R[sr1] - R[sr2] ≥ 2^{32}) → C[cr]<0> ← 1 (R[sr1] - R[sr2] < 2^{32}) → C[cr]<0> ← 0
cmpi cr, sr, simm	Compare an immediate to register operand. When evaluating carry flag, unsigned comparison is used.	(R[sr1] = simm) → C[cr]<2> ← 1 (R[sr1] ≠ simm) → C[cr]<2> ← 0 (R[sr1] < simm) → C[cr]<1> ← 1 (R[sr1] ≥ simm) → C[cr]<1> ← 0 (R[sr1] - simm ≥ 2^{32}) → C[cr]<0> ← 1 (R[sr1] - simm < 2^{32}) → C[cr]<0> ← 0

Table 3-8. Summary of integer comparision instructions

Mnemonic and operands	Description	Formal definition(RTN)

sll dr, sr1, sr2	Logical shift left.	P := R[sr2]<5..0>; R[dr] ← R[sr1]<31 - P..0> # 0@P;
		C[0] ← zero#neg#carry

slli dr, sr, imm		P := imm; R[dr] ← R[sr1]<31 - P..0> # 0@P;
		C[0] ← zero # neg # carry

sra dr, sr1, sr2	Arithmetic shift right.	P := R[sr2]<5..0>; R[dr] ← R[sr1]<31>@P#R[sr1]<31..P>
srai dr, sr, imm		P := imm; R[dr] ← R[sr1]<31>@P#R[sr1]<31..P>
srl dr, sr1, sr2	Logical shift right.	P := R[sr2]<5..0>; R[dr] ← 0@P#R [sr1]<31..P>
srli dr, sr, imm		P := imm; R[dr] ← 0@P#R[sr1]<31..P>

Table 3-9. Summary of shift instructions

Mnemonic and operands	Description	Formal definition(RTN)
ld dr, sr, simm	Load word from memory.	R[dr] ← M[sr+simm]
st sr1, sr2, simm	Store word to memory	M[sr2 + simm] ← R[sr1]
mov dr, sr	Register to register move	R[dr] ← R[sr]
movfc imm, dr, sr	Move data from coprocessor register. dr – destination index at COFFEE core. sr – source index at coprocessor.	R[dr] ← R[sr]
movtc imm, dr, sr	Move data to coprocessor register. dr – destination index at coprocessor. sr – source index at COFFEE core.	R[dr] ← R[sr]

Table 3-10. Summary of load, store and data moving instructions

Mnemonic and operands	**Description**	**Formal definition(RTN)**
chrs imm	Change visible register set.	PSR<1> ← imm<0> PSR<2> ← imm<1>
di	Disable interrupts.	PSR<4> ← 0
ei	Enable interrupts.	PSR<4> ← 1
swm imm	Switch instruction decoding mode.	(imm = 16) → PSR<3> ← 0:PSR<5> ← 0:PSR<6> ← 0 (imm = 32) → PSR<3> ← 1:PSR<5> ← 0:PSR<6> ← 0

reti	Return from interrupt service routine.	PC ← HWS[0]<31..0>:
		PSR ← HWS[0]<39..32>:
		C[0] ← HWS[0]<42..40>

retu	Return to user mode.	PC ← R[31]
		PSR ← SPSR

scall	System call.	SPSR ← PSR;PSR<0> ← 0:PSR<1> ← 1:PSR<2> ← 1:
		PSR<3> ← 1: PSR<4> ← 0;
		(SPSR<3> = 1) → INCREMENT:=4;
		(SPSR<3> = 0) → INCREMENT:=2;
		R[31] ← PC+INCREMENT;PC ← CCB[29]

rcon sr	Restore condition register bank.	C ← R[sr]<23 ..0>
scon dr	Save condition register bank.	R[dr] ← 0@8#C[7]#C[6]#C[5]#C[4] # C[3] # C[2] # C[1] # C[0]

trap imm	Software exception.	CCB[21] ← 1@3#imm:
		CCB[22] ← address of trap instruction:
		CCB[23] ← PSR:
		PSR<0> ← 0:PSR<1> ← 1:PSR<2> ← 1:PSR <3> ← 1: PSR<4> ← 0:
		PC ← CCB[32]
		See chapter 4.7.2 for details.

nop	No operation.	-

Table 3-11. Summary of miscellaneous instructions

3.2 Integer Arithmetic Instructions

add

syntax: `(cond, creg ) add dreg, sreg1, sreg2`
`add dr, sr`

description: The contents of the source registers *sregi* are summed together and the result is placed to the destination register *dreg*. Exception is raised if the result exceeds 2^{31}-1 or falls below -2^{31}. In 16-bit mode the register *dr* is the second source and the destination.

notes: Operation is carried out using twos complement arithmetic.

flags: Z, N, C (creg0)

addi

syntax: `(cond, creg ) addi dreg, sreg1, imm`
`addi dr, imm`

description: The immediate constant is sign extended and summed with the contents of the source register *sreg1*. The result is placed to the destination register *dreg*. Exception is raised if the result exceeds 2^{31}-1 or falls below -2^{31}. In 16-bit mode the register *dr* is the first source register and the destination.

notes: Operation is carried out using twos complement arithmetic. See the permitted values for the immediate in the Table 3-14.

flags: Z, N, C (creg0)

addiu

syntax: `(cond, creg ) addiu dreg, sreg1, imm`
`addiu dr, imm`

description: The immediate constant is zero extended and summed with the contents of the source register *sreg1*. The result is placed to the destination register *dreg*. Overflow is ignored. In 16-bit mode the register dr is the first source register and the destination.

flags: Z, N, C (creg0)

notes: The register operand can also be 'negative' even though the instruction is supposed to be '*add with immediate, unsigned operands*' . The only difference to `addi` is that possible overflow condition is ignored. In general addition procedure is exactly the same for both kinds of operands (2C or unsigned) only the result is interpreted differently (in this case by the programmer or compiler). Flags are set as expected when using 2C arithmetic. See the permitted values for the immediate in the Table 3-14.

addu

syntax: `(cond, creg ) addu dreg, sreg1, sreg2`
`addu dr, sr`

description: The contents of the source registers *sregi* are summed together and the result is placed to the destination register *dreg*.

Overflow is ignored. In 16 bit mode the register dr is the second source and the destination.
flags: C, N, Z (CREG 0)
notes: Addition wider than 32 bits can be carried out as follows: Add the lower 32 bits with `addu` and add one to the upper 32 bits if carry was set in condition register creg0 as a result of the first addition. The register operands can also be 'negative' even though the instruction is supposed to be '*add, unsigned operands*' . The only difference to `add` is that possible overflow condition is ignored. In general addition procedure is exactly the same for both kinds of operands (2C or unsigned) only the result is interpreted differently (in this case by the programmer or compiler). Flags are set as expected when using 2C arithmetic.

mulhi
syntax: `(cond, creg) mulhi dreg`
description: Returns the upper 32 bits of a 64-bit product based on the previous instruction which has to be one of the instructions `mulu`, `muls`, `muli` or `mulus`.
notes: See also `mulu`, `muli`, `muls` and `mulus`.

muli
syntax: `(cond, creg) muli dreg, sreg1, imm`
`muli dr, imm`
description: Multiplies the contents of the source register *sreg1* with the sign extended immediate *imm* and places the result to the destination register *dreg*. The operands are assumed to be signed integers (2C). In 16-bit mode *dr* is the source and the destination register.
notes: See `mulhi` for recovering the upper 32 bits of a product longer than 32-bit. See the permitted values for the immediate in Table 3-14.

muls
syntax: `(cond, creg) muls dreg, sreg1, sreg2`
`muls dr, sr`
description: Multiplies the contents of the source register *sreg1* with the source register *sreg2* and places the lower 32 bits of the result to the destination register *dreg*. The operands are assumed to be signed integers (2C). In 16-bit mode *dr* is the second source register and the destination.
notes: See `mulhi` for recovering the upper 32 bits of a product longer than 32-bit.

muls_16
syntax: `(cond, creg) muls_16 dreg, sreg1, sreg2`
`muls_16 dr, sr`
description: Multiplies the lower halfword of the source register *sreg1* with the lower halfword of the source register *sreg2* and places

the result to the destination register *dreg*. The operands are assumed to be signed integers (2C). In 16-bit mode *dr* is the second source register and the destination.

mulu

syntax: `(cond, creg) mulu dreg, sreg1, sreg2`
`mulu dr, sr`

description: Multiplies the contents of the source register *sreg1* with the source register *sreg2* and places the lower 32 bits of the result to the destination register *dreg*. The operands are assumed to be unsigned integers). In 16-bit mode *dr* is the second source register and the destination.

notes: See `mulhi` for recovering the upper 32 bits of a product longer than 32-bit.

mulu_16

syntax: `(cond, creg) mulu_16 dreg, sreg1, sreg2`
`mulu_16 dr, sr`

description: Multiplies the lower halfword of the source register *sreg1* with the lower halfword of the source register *sreg2* and places the result to the destination register *dreg*. The operands are assumed to be unsigned integers. In 16-bit mode *dr* is the second source register and the destination.

mulus

syntax: `(cond, creg) mulus dreg, sreg1, sreg2`
`mulus dr, sr`

description: Multiplies the contents of the source register *sreg1* with the source register *sreg2* and places the lower 32 bits of the result to the destination register *dreg*. The operand in register *sreg1* is assumed to be an unsigned integer and the operand in register *sreg2* is assumed to be a signed integer. In 16-bit mode *dr* is the second source register and the destination.

notes: See `mulhi` for recovering the upper 32 bits of a product longer than 32-bit.

mulus_16

syntax: `(cond, creg) mulus_16 dreg, sreg1, sreg2`
`mulus_16 dr, sr`

description: Multiplies the lower halfword of the source register *sreg1* with the lower halfword of the source register *sreg2* and places the result to the destination register *dreg*. The operand in register *sreg1* is assumed to be an unsigned integer and the operand in register *sreg2* is assumed to be a signed integer. In 16-bit mode *dr* is the second source register and the destination.

sub

syntax: `(cond, creg) sub dreg, sreg1, sreg2`
`sub dr, sr`

description: The contents of the source register *sreg2* is subtracted from the contents of the source register *sreg1* and the result is placed to the destination register *dreg*. Exception is raised if the result exceeds 2^{31}-1 or falls below -2^{31}. In 16-bit mode *dr* is the second source register and the destination.
notes: Operation is carried out using twos complement arithmetic
flags: Z, C, N

subu
syntax: `(cond, creg) subu dreg, sreg1, sreg2`
`subu dr, sr`
description: The contents of the source register *sreg2* is subtracted from the contents of the source register *sreg1* and the result is placed to the destination register *dreg*. In 16-bit mode *dr* is the second source register and the destination.
flags: Z, C, N
notes: Over/underflow is ignored.

3.3 Byte and Bit Field Manipulation Instructions

conb
syntax: `(cond, creg) conb dreg, sreg1, sreg2`
`conb dr, sr`
description: Concatenates the least significant bytes from the source registers to form a halfword. The least significant byte from the register *sreg1* becomes the most significant byte of the halfword and the least significant byte from the register *sreg2* becomes the least significant byte of the halfword. The resulting halfword is saved to the destination register *dreg*. The upper halfword of the result is filled with zeros. In 16-bit mode *dr* corresponds to the second source register *sreg2* (and the destination) and *sr* corresponds to *sreg1*.
notes: Note that ordering of operands is different in 16-bit mode from that of 32-bit mode.

conh
syntax: `(cond, creg) conh dreg, sreg2, sreg1`
`conh dr, sr`
description: Concatenates the least significant halfwords from the source registers to form a word. The least significant halfword from the register *sreg2* becomes the most significant halfword of the word and the least significant halfword from the register *sreg1* becomes the least significant halfword of the word. The resulting word is saved to the destination register *dreg*. In 16-bit mode *dr* corresponds to the second source register *sreg2* (and the destination) and *sr* corresponds to *sreg1*.
notes: Note that ordering of operands is different in 16-bit mode from that of 32-bit mode.

exb
syntax: `(cond, creg) exb dreg, sreg, imm`
description: Extracts the byte specified by the immediate *imm* from the source register *sreg/sr* and places it to the least significant end of the destination register *dreg/dr*. The upper three bytes in the destination register are cleared. The extracted byte is specified according to the Table 3-12.

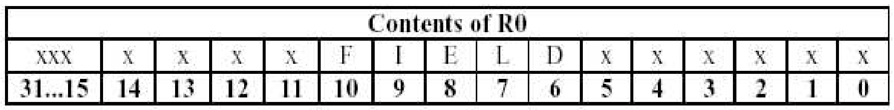
Contents of a source register			
high end		low byte	
byte3	byte2	byte1	byte0

0	byte0
1	byte1
2	byte2
3	byte3

Table 3-12. Extracted byte specification

notes: See the permitted values for the immediate in Table 3-14.

exbf
syntax: `(cond, creg) exbf dreg, sreg1, sreg2`
`exbf dr, sr`
description: Operates like `exbfi`, but the two immediates defining the extracted field are combined and read from the least significant end of the source register *sreg2*: bits 10 down to 5 define the length of the field and bits 4 down to 0 define the LSB position. In the 16-bit mode *dr* is the second source and the destination.
notes: Example
Suppose that the bitfield shown bellow should be extracted from register R0 shown in Figure 3-1 (could be for example a sub address field in a message frame).



Contents of R0															
xxx	x	x	x	x	F	I	E	L	D	x	x	x	x	x	x
31...15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3-1. Content of R0

Now the length of the bitfield is `5 = 000101` and LSB position is `6 = 00110`. To extract the bitfield we have to place a constant `000101 00110 = 000 1010 0110 = 0A6h` in second source register (say R2). The following code could be used to place the result in R3:

```
lli R2, 0a6h
exbf R3, R0, R2
```

If we assume that the length of the bitfield in question is contained in register R1 and the LSB position is in register R2. The following code could be used to extract the bitfield to R3:

```
// shift the length to bits 10 downto 5
slli R1, R1, 5
or R2, R2, R1 // combine length and position
exbf R3, R0, R2
```

See also `exbfi`.

exbfi
syntax: `exbfi dreg, sreg1, imm1, imm2`
description: Extracts a bitfield of arbitrary length and position from the source register *sreg1* and places it to the low end of the destination register *dreg*. Bitfield length and position are defined by the immediates *imm1 and imm2* as follows: *imm1* defines the length of the bitfield. Immediate *imm2* specifies the LSB position of the extracted bitfield in the source register. If the extracted bitfield is shorter than 32 bits, the extra bit positions in the destination register are filled with zeros.
notes: Can be used only in 32-bit mode. This instruction cannot be executed conditionally. See the permitted values for the immediate in Table 3-14.

exh
syntax: `(cond, creg) exh dreg, sreg1, imm`
description: Extracts the halfword specified by the immediate *imm* from the source register *sreg1/sr* and places it to the least significant end of the destination register *dreg/dr*. The upper halfword in the destination register is cleared. If *imm* = 0, then the least significant halfword is extracted, otherwise the most significant halfword is extracted.

lli
syntax: `lli dreg, imm`
description: Loads the lower halfword of the destination register *dreg* with the immediate *imm*. The upper half of the destination register is cleared.
notes: Can be used only in 32-bit mode. This instruction cannot be executed conditionally. See the permitted values for the immediate in Table 3-14.

lui
syntax: `lui dreg, imm`
description: Loads the upper halfword of the destination register *dreg* with the immediate *imm*. The lower half of the destination register is preserved.

notes: Can be used only in 32-bit mode. This instruction cannot be executed conditionally. See the permitted values for the immediate in Table 3-14.

sext

syntax: `(cond, creg) sext dreg, sreg1, sreg2`
`sext dr, sr`

description: Works as `sexti`, but the position of the sign bit is evaluated using the five least significant bits from the source register *sreg2*. In 16-bit mode dr is the second source register and the destination.

notes: See also `sexti`.

sexti

syntax: `(cond, creg) sexti dreg, sreg, imm`
`sexti dr, imm`

description: Sign extends the operand in the source register *sreg* and places the result to the destination register *dreg*. The position of the sign bit is specified by the immediate *imm* (0 corresponds to LSB and 31 corresponds to MSB). In 16-bit mode *dr* is the source register and the destination.

notes: See the permitted values for the immediate in Table 3-14.

3.4 Boolean Bitwise Operation Instructions

and

syntax: `(cond, creg ) and dreg, sreg1, sreg2`
`and dr, sr`

description: Bitwise Boolean AND operation is performed to the contents of the source registers *sregi*. The result is placed to the destination register *dreg*.In 16-bit mode the register *dr* is the second source and the destination.

andi

syntax: `(cond, creg ) andi dreg, sreg1, imm`
`andi dr, imm`

description: The immediate constant is zero extended. Bitwise Boolean AND operation is performed to the extended immediate and the contents of the source register *sreg1*. The result is placed to the destination register *dreg*. In 16 bit mode the register *dr* is the register source and the destination.

notes: See the permitted values for the immediate in Table 3-14.

not

syntax: `(cond, creg) not dreg, sreg1`

description: Performs a bitwise Boolean NOT operation to the contents of the source register *sreg1/sr* and places the result to the destination register *dreg/dr*.

or

syntax: `(cond, creg) or dreg, sreg1, sreg2`
```
or dr, sr
```
description: Performs a bitwise Boolean OR operation to the contents of the source registers *sregi* and places the result to the destination register *dreg*. In 16-bit mode *dr* is the second source and the destination register.

ori

syntax: `(cond, creg) ori dreg, sreg1, imm`
```
ori dr, imm
```
description: Performs a bitwise Boolean OR operation to the contents of the source register *sreg1* and zero extended immediate *imm*. The result is placed to the destination register *dreg*. In 16-bit mode *dr* is the source and the destination register.

notes: See the permitted values for the immediate in Table 3-14.

xor

syntax: `(cond, creg) xor dreg, sreg1, sreg2`
```
xor dr, sr
```
description: Performs a bitwise XOR operation to the contents of the source registers *sreg1* and *sreg2*. The result is placed to the destination register *dreg*. In 16-bit mode the bitwise XOR is performed to the contents of *dr* and *sr* and the result is placed into *dr*.

3.5 Branch (Conditional Jump) Instructions

bc

syntax: `bc creg, imm`
```
bc imm
```
description: If the carry flag in the condition register *creg* is high, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is allways creg0.

notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

begt

syntax: `begt creg, imm`
```
begt imm
```
description: If the flags in the condition register *creg* indicate that the condition eqt (equal or greater than) is true, program execution branches to target address specified by the immediate *imm*. The target

address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is always creg0.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

belt
syntax: `belt creg, imm`
```
belt imm
```
description: If the flags in the condition register *creg* indicate that the condition elt (equal or less than) is true, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is always creg0
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

beq
syntax: `beq creg, imm`
```
beq imm
```
description: If the flags in the condition register *creg* indicate that the condition eq (equal) is true, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is always creg0.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

bgt
syntax: `bgt creg, imm`
```
bgt imm
```
description: If the flags in the condition register *creg* indicate that the condition gt (greater than) is true, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the

contents of the program counter PC. In 16-bit mode the condition register used is always creg0.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

blt
syntax: `blt creg, imm`
`blt imm`
description: If the flags in the condition register *creg* indicate that the condition lt (less than) is true, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is always creg0.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

bne
syntax: `bne creg, imm`
`bne imm`
description: If the flags in the condition register *creg* indicate that the condition ne (not equal) is true, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is always creg0.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

bnc
syntax: `bnc creg, imm`
`bnc imm`
description: If the carry flag in the condition register *creg* is low, program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. In 16-bit mode the condition register used is always creg0.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot).

The branch offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

3.6 Jump Instructions

jal
syntax: `jal imm`
description: Program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC. Link address is saved to register R31/SR31. The link address is the address of the next instruction after branch slot instruction.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The jump offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

jalr
syntax: `(cond, creg) jalr sreg1`
description: Program execution branches to target address specified by the contents of the source register *sreg1/sr*. Link address is saved to register R31/SR31. The link address is the address of the next instruction after branch slot instruction.
notes: The instruction following this instruction is always executed (branch slot). Conditional jumps (branches) that can reach the whole address space can be synthesized by executing this instruction conditionally. Note that the address in the source register should be aligned to word boundary if in 32-bit mode or halfword boundary if in 16-bit mode.

jmp
syntax: `jmp imm`
description: Program execution branches to target address specified by the immediate *imm*. The target address is calculated as follows: The immediate offset *imm* is shifted left by one bit and sign extended. The sign extended offset is added to the contents of the program counter PC.
notes: This instruction cannot be executed conditionally. The instruction following this instruction is always executed (branch slot). The jump offset is calculated relative to the instruction in the slot. See the permitted values for the immediate in Table 3-14.

jmpr
syntax: `(cond, creg) jmpr sreg1`

description: Program execution branches to target address specified by the contents of the source register *sreg1/sr*.
notes: The instruction following this instruction is always executed (branch slot). Conditional jumps (branches) that can reach the whole address space can be synthesized by executing this instruction conditionally. Note that the address in the source register should be aligned to word boundary if in 32-bit mode or halfword boundary if in 16-bit mode.

3.7 Integer Comparison Instructions

cmp
syntax:
```
cmp creg, sreg1, sreg2
cmp sr1, sr2
```
description: The contents of the source registers *sregi/sri* are compared as if they were signed numbers. The operation is logically done by subtracting the contents of *sreg2/sr2* from the contents of *sreg1/sr1*. Flags N, Z and C are set or cleared accordingly and saved to the condition register *creg*. In 16-bit mode the condition register is always creg0.
flags: N, Z, C
notes: The logical subtraction *sreg1- sreg2/sr1 - sr2* does not overflow, that is, the flags are always set correctly independently of the result of the subtraction. This instruction cannot be executed conditionally.

cmpi
syntax:
```
cmpi creg, sreg1, imm
cmpi sr, imm
```
description: The immediate constant *imm* is sign extended and compared to the contents of the source register *sreg1/sr1* as if they were signed numbers. The operation is logically done by subtracting the immediate *imm* from the contents of *sreg1/sr1*. Flags N, Z and C are set or cleared accordingly and saved to the condition register *creg*. In 16 bit mode the condition register is always creg0.
flags: N, Z, C
notes: The logical subtraction *sreg1- imm/sr - imm* does not overflow, that is, the flags are always set correctly independently of the result of the subtraction. This instruction cannot be executed conditionally. See the permitted values for the immediate in Table 3-14.

3.8 Shift Instructions

sll
syntax:
```
(cond, creg) sll dreg, sreg1, sreg2
sll dr sr
```

description: Performs the logical shift left to the contents of the source register *sreg1/sr* and places the result to the destination register *dreg/dr*. The six least significant bits in the source register *sreg2* specify the amount of shift. The last 'dropped' bit (bit 32) is saved as carry flag in register creg0. In 16-bit mode *dr* is the second source register and the destination.
flags: C, N, Z
notes: If the unsigned integer formed by the six least significant bits in the source register *sreg2* imply a shift of more than 32 positions then the result will be a shift of 32 positions (which is zero).

slli
syntax:
```
(cond, creg) slli dreg, sreg1, imm
 slli dr, imm
```
description: Performs the logical shift left to the contents of the source register *sreg1* and places the result to the destination register *dreg*. The immediate *imm* specifies the amount of shift. The last 'dropped' bit (bit 32) is saved as carry flag in register creg0. In 16-bit mode *dr* is the source register and the destination.
notes: See the permitted values for the immediate in Table 3-14.
flags: C, N, Z

sra
syntax:
```
(cond, creg) sra dreg, sreg1, sreg2
 sra dr sr
```
description: Performs the arithmetic shift right to the contents of the source register *sreg1/sr* and places the result to the destination register *dreg/dr*. The six least significant bits in the source register *sreg2* specify the amount of shift. In 16-bit mode *dr* is the second source register and the destination.
notes: If the unsigned integer formed by the six least significant bits in the source register *sreg2* imply a shift of more than 32 positions then the result will be a shift of 32 positions.

srai
syntax:
```
(cond, creg) srai dreg, sreg1, imm
 srai dr, imm
```
description: Performs the arithmetic shift right to the contents of the source register *sreg1* and places the result to the destination register *dreg*. The immediate *imm* specifies the amount of shift. In 16-bit mode *dr* is the source register and the destination.
notes: See the permitted values for the immediate in Table 3-14.

srl
syntax:
```
(cond, creg) srl dreg, sreg1, sreg2
  srl dr sr
```
description: Performs the logical shift right to the contents of the source register *sreg1/sr* and places the result to the destination register

dreg/dr. The six least significant bits in the source register *sreg2* specify the amount of shift. In 16-bit mode *dr* is the second source register and the destination.
notes: If the unsigned integer formed by the six least significant bits in the source register *sreg2* imply a shift of more than 32 positions then the result will be a shift of 32 positions.

srli
syntax: `(cond, creg) srli dreg, sreg1, imm`
```
srli dr, imm
```
description: Performs the logical shift right to the contents of the source register *sreg1* and places the result to the destination register *dreg*. The immediate *imm* specifies the amount of shift. In 16-bit mode *dr* is the source register and the destination.
notes: See the permitted values for the immediate in Table 3-14.

3.9 Memory Load and Store, Data Moving Instructions

ld
syntax: `(cond, creg) ld dreg, sreg1, imm`
description: Loads a 32-bit data word from memory to the destination register *dreg/dr*. The address of the data is calculated as follows: The immediate offset *imm* is sign extended and added to the contents of the source register *sreg1/sr*. The address is not auto-aligned (two least significant bits of the resulting address are driven to address bus).
notes: The result of the address calculation doesn't have to be aligned to word boundary. The two least significant bits can be used for example as byte index if narrower bus is used. Also the smallest addressable unit can be 32-bit word giving 16GB address range! See the permitted values for the immediate in Table 3-14.

mov
syntax: `(cond, creg) mov dreg, sreg1`
description: Copies the contents of the source register *sreg1/sr* to the destination register *dreg/dr*.

st
syntax: `(cond, creg) st sreg2, sreg1, imm`
description: Stores the data in the source register *sreg2/sr2* to memory location whose address is calculated as follows: The immediate offset *imm* is sign extended and added to the contents of the source register *sreg1/sr1*. The address is not auto-aligned (two least significant bits of the resulting address are driven to address bus).
notes: The two least significant bits can be used for example as byte index if narrower bus is used. Also the smallest addressable unit can be 32-bit word giving 16GB address range! See the permitted values for the immediate in Table 3-14.

3.10 Coprocessor instructions

movfc
syntax: (cond, creg) movfc imm, dreg, cp_sreg
description: Copies the contents of one of the registers in the coprocessor number imm to the destination register dreg/dr. The immediate imm is used to specify one of the four possible coprocessors: 0, 1, 2 or 3. Cp_reg is an index to the coprocessor register file.

movtc
syntax: (cond, creg) movtc imm, cp_dreg, sreg1
description: Copies the contents of the source register sreg1/sr to the coprocessor register cp_dreg. The immediate imm is used to specify one of the four possible coprocessors: 0, 1, 2 or 3.

3.11 Miscellaneous Instructions

chrs
syntax: chrs imm
description: Specifies which register set is used for reading or writing. The source register(s) and the destination register don't have to reside in the same set. The register sets to be used are specified by the immediate *imm* according to the Table 3-13.

imm	Write	Read
0 (00b)	Set1 (user set)	Set1 (user set)
1 (01b)	Set1 (user set)	Set 2 (superuser set)
2 (10b)	Set 2 (superuser set)	Set1 (user set)
3 (11b)	Set 2 (superuser set)	Set 2 (superuser set)

Table 3-13. Register set definition for writing and reading

notes: When execution in the super user mode begins the default register set for reading and writing is the super user set (set 2). When returning back to the user mode the default register set is the user set (set 1). This command is allowed only in super user mode. An exception is raised on an attempt to use this command in user mode. As a result, the user cannot see the register set intended only for super user. Not allowed to be executed conditionally.

di
syntax: `di`
description: Disables maskable interrupts.
notes: Not permitted to be executed conditionally. An exception is raised on an attempt to use this command in user mode. See Section

1.4 and Section 1.5 about exceptions and interrupts for definitions and details.

ei
syntax: `ei`
description: Enables maskable interrupts.
notes: Not permitted to be executed conditionally. An exception is raised on an attempt to use this command in user mode. See Section 1.4 and Section 1.5 about exceptions and interrupts for definitions and details.

reti
syntax: `reti`
description: Used for returning from an interrupt service routine. Loads PC, CR0 and PSR from the hardware stack and signals to the external (and internal) interrupt handler that the servicing of the last interrupt request was completed.
notes: Not allowed to be executed conditionally. `Reti` instruction has to be followed by two `nop`s!

retu
syntax: `retu`
description: Used for returning or moving from system code/superuser mode to user mode. Execution of user code starts from a address in register PR31. Status flags are copied from the register SPSR. (They should be set appropriately before issuing `retu`). Available only in superuser mode.

scall
syntax: `(cond, creg) scall`
description: System call transfers the processor to the superuser mode and execution of instructions begins at address defined in register SYSTEM_ADDR. The link address is saved in to the register PR31 (link register of SET2). The link address is the address of the instruction following `nop` (see notes below). The state of the processor before `scall` is copied to the register SPSR.
notes: When transferring the control to superuser code the default settings are 32-bit mode, interrupts disabled and superuser register set (both read and write). As with branches and jumps also this instruction has a branch slot, which in this case has to be filled with a `nop` instruction. See `retu`.

swm
syntax: `swm imm`
description: Changes the instruction decoding mode. The value of the immediate *imm* specifies the mode: *imm* = 16 => switch to 16-bit

mode, *imm* = 32=> switch to 32-bit mode. Other values are reserved for future extensions.
flags: IL
notes: This instruction is not allowed to be executed conditionally. See the permitted values for the immediate in Table 3-14. This instruction has to be followed by two `nop` instructions!

nop
syntax: `nop`
description: Idle command that does not alter the state of the processor.
notes: See the list of instructions which require a succeeding `nop`. This instruction cannot be executed conditionally (even if it could it wouldn't have any effect anyway).

rcon
syntax: `rcon sreg1`
description: Restores the contents of all the condition registers from the source register *sreg1*.
notes: This instruction is not allowed to be executed conditionally.

scon
syntax: `scon dreg`
description: Saves the contents of all the condition registers to the (low end of) destination register *dreg.*
notes: This instruction is not allowed to be executed conditionally.

trap
syntax: `trap imm`
description: Generates a software trap. Execution is started at the address of exception handler routine defined in the CCB register EXCEP_ADDR. The address of the trap instruction is saved in the EPC register and the exception code in exception cause register (ECS).
notes: See Section 1.4 to get more information about exceptions and about the code.

3.12 Pseudo Instructions

dec
syntax: `dec dr`
description: Word decrement.
pseudo code:
32-bit mode:

```
        addiu dr, dr, -1
```

16-bit mode:

```
          addiu dr, -1
```

decb
syntax: `decb dr`
description: Byte decrement = modulo 256 decrement.
pseudo code:
32-bit mode:
```
          addiu dr, dr, -1
          andi dr, dr, 0xff
```

16-bit mode:
```
          addiu dr, -1
          slli dr, 24
          srli dr, 24
```

inc
syntax: `inc dr`
descrption: Word increment.
pseudo code:
32-bit mode:
```
          addiu dr, dr, 1
```

16-bit mode:
```
          addiu dr, 1
```

incb
syntax: `incb dr`
description: Byte increment = modulo 256 increment.
pseudo code:
32-bit mode:
```
          addiu dr, dr, 1
          andi dr, dr, 0xff
```

16-bit mode:
```
          addiu dr, 1
          slli dr, 24
          srli dr, 24
```

ldra
syntax: `ldra dr, limm`
description: Load register with address.
pseudo code:
32-bit mode:
```
          lli dr, imm & 0xffff
          lui dr, imm >> 16
```

16-bit mode
```
          xor dr, dr
          ori dr, imm >> 25
          slli dr, 7
          ori dr, (imm >> 18) & 0x7f
          slli dr, 7
          ori dr, (imm >> 11) & 0x7f
          slli dr, 7
          ori dr, (imm >> 4) & 0x7f
          slli dr, 4
          ori dr, imm & 0xf
```

ldri
syntax: `ldri dr, limm`
description: Load register with long immediate or constant.
pseudo code:

```
32-bit mode:
          lli dr, imm & 0xffff
          if(imm > 65535)
                    lui dr, imm >> 16

16-bit mode:
if(limm == 0)
          xor dr, dr
else
{
          if(imm[31:25] != 0)
          {
                    ori dr, imm >> 25
                    slli dr, 7
          }
          if(imm[31:18] != 0)
          {
                    ori dr, (imm >> 18) & 0x7f
                    slli dr, 7
          }
          if(imm[31:11] != 0)
          {
                    ori dr, (imm >> 11) & 0x7f
                    slli dr, 7
          }
          if(imm[31:4] != 0)
          {
                    ori dr, (imm >> 4) & 0x7f
                    slli dr, 4
          }
          ori dr, imm & 0xf
}
```

instruction	Permitted values for *imm*			notes
	16 bit	32 bit		
		conditional	unconditional	
addi	$-2^6 \ldots 2^6-1$	$-2^8 \ldots 2^8-1$	$-2^{14} \ldots 2^{14}-1$	
addiu	$0 \ldots 2^7-1$	$0 \ldots 2^9-1$	$0 \ldots 2^{15}-1$	
andi	$0 \ldots 2^7-1$	$0 \ldots 2^9-1$	$0 \ldots 2^{15}-1$	
bxx [1]	$-2^9 \ldots 2^9-1$	-	$-2^{21} \ldots 2^{21}-1$	Should be even in 32bit mode
chrs	0...3	-	0..3	
cmpi	$-2^6 \ldots 2^6-1$	-	$-2^{16} \ldots 2^{16}-1$	
exb	0...3	0..3	0...3	
exbfi [3]	-	-	imm1: 0...32 imm2: 0...31	Only 32 bit mode
exh	0 or 1	0 or 1	0 or 1	
jal	$-2^9 \ldots 2^9-1$	-	$-2^{24} \ldots 2^{24}-1$	Should be even in 32bit mode
jmp	$-2^9 \ldots 2^9-1$	-	$-2^{24} \ldots 2^{24}-1$	Should be even in 32bit mode
ld	-8...7	$-2^8 \ldots 2^8-1$	$-2^{14} \ldots 2^{14}-1$	
lli	-	-	$0 \ldots 2^{16}-1$ (or $-2^{15} \ldots 2^{15}-1$)	Only 32 bit mode
lui	-	-	$0 \ldots 2^{16}-1$ (or $-2^{15} \ldots 2^{15}-1$)	Only 32 bit mode
movfc	0...3	0..3	0..3	
movtc	0...3	0..3	0..3	
muli	$-2^6 \ldots 2^6-1$	$-2^8 \ldots 2^8-1$	$-2^{14} \ldots 2^{14}-1$	
ori	$0 \ldots 2^7-1$	$0 \ldots 2^9-1$	$0 \ldots 2^{15}-1$	
sexti	0...31	0...31	0...31	
slli	0...32	0...32	0...32	
srai	0...32	0...32	0...32	
srli	0...32	0...32	0...32	
st	-8...7	$-2^8 \ldots 2^8-1$	$-2^{14} \ldots 2^{14}-1$	
swm[2]	16 or 32	-	16 or 32	
trap	-	-	0...31	

Table 3-14. Permitted values for immediate constant

assembly instruction/variant	**machine instructions**	
	16 bit output	**32 bit output**

add dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	add dr, sr1
		add dr, sr1, sr2

add dr, sr	**add** dr, sr	**add** dr, sr, dr

addi dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	addi dr, imm
		addi dr, sr1, imm

addi dr, imm	**addi** dr, imm	**addi** dr, dr, imm

addiu dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	addiu dr, imm
		addiu dr, sr1, imm

addiu dr, imm	**addiu** dr, imm	**addiu** dr, dr, imm

addu dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	addu dr, sr1
		addu dr, sr1, sr2

addu dr, sr	**addu** dr, sr	**addu** dr, sr, dr

and dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	and dr, sr1
		and dr, sr1, sr2

and dr, sr	**and** dr, sr	**and** dr, sr, dr

andi dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	andi dr, imm
		andi dr, sr1, imm

andi dr, imm	**andi** dr, imm	**andi** dr, dr, imm

bc cr, imm	*if (cr == c0)*
	   bc imm
	else
	   error
		bc cr, imm

bc imm	**bc** imm	**bc** c0, imm

begt cr, imm	*if (cr == c0)*
	   begt imm
	else
	   error
		begt cr, imm

begt imm	**begt** imm	**begt** c0, imm

belt cr, imm	*if (cr == c0)*
	   belt imm
	else
	   error
		belt cr, imm

belt imm	**belt** imm	**belt** c0, imm

beq cr, imm	*if (cr == c0)*
	   beq imm
	else
	   error
		beq cr, imm

beq imm	**beq** imm	**beq** c0, imm

bgt cr, imm	*if (cr == c0)*
	    bgt imm
	else
	    error
		bgt cr, imm

bgt imm	**bgt** imm	**bgt** c0, imm

blt cr, imm	*if (cr == c0)*
	    blt imm
	else
	    error
		blt cr, imm

blt imm	**blt** imm	**blt** c0, imm

bnc cr, imm	*if (cr == c0)*
	    bnc imm
	else
	    error
		bnc cr, imm

bnc imm	**bnc** imm	**bnc** c0, imm

bne cr, imm	*if (cr == c0)*
	    bne imm
	else
	    error
		bne cr, imm

bne imm	**bne** imm	**bne** c0, imm
chrs imm	**chrs** imm	**chrs** imm

cmp cr, sr1, sr2	*if (cr == c0)*
	    cmp sr1, sr2
	else
	    error
		cmp cr, sr1, sr2

cmp sr1, sr2	**cmp** sr1, sr2	**cmp** c0, sr1, sr2

cmpi cr, sr1, imm	*if (cr == c0)*
	    cmpi sr1, imm
	else
	    error
		cmpi cr, sr1, imm

cmpi sr, imm	**cmpi** sr, imm	**cmpi** c0, sr, imm

conb dr, sr1, sr2	*if (sr2 != dr)*
	    mov dr, sr2
	conb dr, sr1
		conb dr, sr1, sr2

conb dr, sr	**conb** dr, sr **conh** dr, sr **slli** dr, 8 **srli** dr, 16	**conb** dr, dr, sr

conh dr, sr2, sr1	*if (sr2 != dr)*
	    mov dr, sr2
	conh dr, sr1
		conh dr, sr2, sr1

conh dr, sr	**conh** dr, sr	**conh** dr, dr, sr
decb dr	**addi** dr, -1 **slli** dr, 24 **srli** dr, 24	**addi** dr, dr, -1 **andi** dr, dr, 0xff
dec dr	**addi** dr, -1	**addi** dr, dr -1
di	**di**	**di**

ei	**ei**	**ei**
exb dr, sr, imm	**exb** dr, sr, imm	**exb** dr, sr, imm
exb dr, imm	**exb** dr, dr, imm	**exb** dr, dr, imm
exbf dr, sr1, sr2	*if (sr2 != dr)* **mov** dr, sr2 **exbf** dr, sr1	**exbf** dr, sr1, sr2
exbf dr, sr	**exbf** dr, sr	**exbf** dr, sr, dr
exbfi dr, sr1, imm1, imm2	*if (sr1 != dr)* **mov** dr, sr1 **slli** dr, (32 – (imm1 + imm2)) **srli** dr, (32 – imm1)	**exbfi** dr, sr1, imm1, imm2
exbfi dr, imm1, imm2	**slli** dr, (32 – (imm1 + imm2)) **srli** dr, (32 – imm1)	**exbfi** dr, dr, imm1, imm2
exh dr, sr, imm	**exh** dr, sr, imm	**exh** dr, sr, imm
exh dr, imm	**exh** dr, dr, imm	**exh** dr, dr, imm
incb dr	**addiu** dr, 1 **slli** dr, 24 **srli** dr, 24	**addiu** dr, dr, 1 **andi** dr, dr, 0xff
inc dr	**addiu** dr, 1	**addiu** dr, dr 1
jal imm	**jal** imm	**jal** imm
jalr sr	**jalr** sr	**jalr** sr
jmp imm	**jmp** imm	**jmp** imm
jmpr sr	**jmpr** sr	**jmpr** sr
ld dr, sr, imm	**ld** dr, sr, imm	**ld** dr, sr, imm
ld dr, sr	**ld** dr, sr, 0	**ld** dr, sr, 0

ldri dr, limm	**xor** dr, dr
	if (limm[31:25] != 0){
	ori dr, limm >> 25
	slli dr, 7
	}
	if (limm[31:18] != 0){
	ori dr, (limm >> 18) & 0x7f
	slli dr, 7
	}
	if (limm[31:11] != 0){
	ori dr, (limm >> 11) & 0x7f
	slli dr, 7
	}
	if (limm[31:4] != 0){
	ori dr, (limm >> 4) & 0x7f
	slli dr, 4
	}
	if (limm[3:0] != 0){
	ori dr, limm & 0xf
	}
		lli dr, limm & 0xffff
		if (limm > 65535)
		lui dr, limm >> 16

ldra dr, limm	**xor** dr, dr **ori** dr, limm >> 25 **slli** dr, 7 **ori** dr, (limm >> 18) & 0x7f **slli** dr, 7 **ori** dr, (limm >> 11) & 0x7f **slli** dr, 7 **ori** dr, (limm >> 4) & 0x7f **slli** dr, 4 **ori** dr, limm & 0xf	**lli** dr, limm & 0xffff **lui** dr, limm >> 16
ldra dr, limm + imm	**xor** dr, dr **ori** dr, limm >> 25 **slli** dr, 7 **ori** dr, (limm >> 18) & 0x7f **slli** dr, 7 **ori** dr, (limm >> 11) & 0x7f **slli** dr, 7 **ori** dr, (limm >> 4) & 0x7f **slli** dr, 4 **ori** dr, limm & 0xf **addi** dr, imm	**lli** dr, limm & 0xffff **lui** dr, limm >> 16 **addi** dr, imm
lli dr, imm	**xor** dr, dr **ori** dr, (imm >> 9) **slli** dr, 7 **ori** dr, ((imm >> 2) & 0x7f) **slli** dr, 2 **ori** dr, (imm & 0x3)	**lli** dr, imm
lui dr, imm	**swm** 32 **nop** **nop** **.align** 2 **.code**32 **lui** dr, imm **swm** 16 **nop** **nop** **.code**16	**lui** dr, imm
mov dr, sr1	**mov** dr, sr1	**mov** dr, sr1
movfc imm, dr, cpr	**movfc** imm, dr, cpr	**movfc** imm, dr, cpr
movtc imm, cpr, sr1	**movtc** imm, cpr, sr1	**movtc** imm, cpr, sr1
mulhi dr	**mulhi** dr	**mulhi** dr

muli dr, sr1, imm	*if (sr1 != dr)*
	    mov dr, sr1
	muli dr, imm
		muli dr, sr1, imm

muli dr, imm	**muli** dr, imm	**muli** dr, dr, imm

muls dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	muls dr, sr1
		muls dr, sr1, sr2

muls dr, sr	**muls** dr, sr	**muls** dr, sr, dr

muls_16 dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	muls_16 dr, sr1
		muls_16 dr, sr1, sr2

muls_16 dr, sr	**muls_16** dr, sr	**muls_16** dr, sr, dr

mulu dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	mulu dr, sr1
		mulu dr, sr1, sr2

mulu dr, sr	**mulu** dr, sr	**mulu** dr, dr, sr

mulu_16 dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	mulu_16 dr, sr1
		mulu_16 dr, sr1, sr2

mulu_16 dr, sr	**mulu_16** dr, sr	**mulu_16** dr, sr, dr

mulus dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	mulus dr, sr1
		mulus dr, sr1, sr2

mulus dr, sr	**mulu** dr, sr	**mulus** dr, dr, sr
mulsu dr, sr	**mulus** dr, sr	**mulus** dr, sr, dr

mulus_16 dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	mulus_16 dr, sr1
		mulus_16 dr, sr1, sr2

mulus_16 dr, sr	**not allowed**	**not allowed**
mulsu_16 dr, sr	**mulus_16** dr, sr	**mulus_16** dr, sr, dr
nop	**nop**	**nop**
not dr, sr1	**not** dr, sr1	**not** dr, sr1

or dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	or dr, sr1
		or dr, sr1, sr2

or dr, sr	**or** dr, sr	**or** dr, sr, dr

ori dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	ori dr, imm
		ori dr, sr1, imm

ori dr, imm	**ori** dr, imm	**ori** dr, dr, imm
rcon sr	**rcon** sr	**rcon** sr
reti	**reti**	**reti**
retu	**retu**	**retu**
scall	**scall**	**scall**
scon dr	**scon** dr	**scon** dr

sext dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	sext dr, sr1
		sext dr, sr1, sr2

sext dr, sr	**sext** dr, sr	**sext** dr, sr, dr

sexti dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	sexti dr, imm
		sexti dr, sr1, imm

sexti dr, imm	**sexti** dr, imm	**sexti** dr, dr, imm

sll dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	sll dr, sr1
		sll dr, sr1, sr2

sll dr, sr	**sll** dr, sr	**sll** dr, sr, dr

slli dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	slli dr, imm
		slli dr, sr1, imm

slli dr, imm	**slli** dr, imm	**slli** dr, dr, imm

sra dr, sr1, sr2	*if(sr2 != dr)*
	   mov dr, sr2
	sra dr, sr1
		sra dr, sr1, sr2

sra dr, sr	**sra** dr, sr	**sra** dr, sr, dr

srai dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	srai dr, imm
		srai dr, sr1, imm

srai dr, imm	**srai** dr, imm	**srai** dr, dr, imm

srl dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	srl dr, sr1
		srl dr, sr1, sr2

srl dr, sr	**srl** dr, sr	**srl** dr, sr, dr

srli dr, sr1, imm	*if (sr1 != dr)*
	   mov dr, sr1
	srli dr, imm
		srli dr, sr1, imm

srli dr, imm	**srli** dr, imm	**srli** dr, dr, imm
st sr2, sr1, imm	**st** sr2, sr1, imm	**st** sr2, sr1, imm
st sr2, sr1	**st** sr2, sr1, 0	**st** sr2, sr1, 0

sub dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	sub dr, sr1
		sub dr, sr1, sr2

sub dr, sr	**subu** dr, sr **not** dr, dr **addi** dr, 1	**sub** dr, dr, sr

subu dr, sr1, sr2	*if (sr2 != dr)*
	   mov dr, sr2
	subu dr, sr1
		subu dr, sr1, sr2

subu dr, sr	**subu** dr, sr **not** dr, dr **addiu** dr, 1	**subu** dr, dr, sr
swm imm	**swm** imm	**swm** imm
trap imm	**trap** imm	**trap** imm

xor dr, sr1, sr2	*if (sr2 != dr)*
	    mov dr, sr2
	xor dr, sr1
		xor dr, sr1, sr2

xor dr, sr	**xor** dr, sr	**xor** dr, sr, dr

Table 3-15. Instruction mapping in 16-bit and 32-bit mode

4. COPROCESSOR INSTRUCTION SET

This chapter describes the machine instructions implemented in Milk coprocessor. The instruction set syntax is made of an instruction mnemonic followed by destination and source registers.

The abbreviations used this chapter are listed in Table 4-1.

Abbreviation	Description
dr	Destination register, number in the range 0..31
sr1, sr2	Source register, number in the range 0..31 (in case that the instruction only needs one operand, it' s simply named sr)
opc	Opcode of Milk instructions

Table 4-1. Abbreviations used in coprocessor instruction set

Note that each of the supported instructions' mnemonics ends with a number (coprocessor index), for example, `fadd0`, `fadd1`, `fmul1`, etc. This is due to the fact that COFFEE™ RISC core supports up to 4 coprocessors, so any of them could be a floating-point unit (FPU), and the one who should actually perform the operation is that indexed by the number specified by the number at the end of the mnemonic. This way, `fadd0` is a floating-point addiction to be executed by coprocessor number 0, `fadd1` is a floating-point addiction to be executed by coprocessor number 1, and so on (for this reason, in the following pages are explained the instructions related to coprocessor number 0, because the instructions related to the other ones are exactly the same for meaning and syntax, and differs only for the coprocessor index).

fadd0

syntax: `fadd0 dr, sr1, sr2`
description: Single-precision floating-point (algebraic) addiction to be executed by coprocessor number 0. The contents of the source registers sr1 and sr2 are added together and the result is placed to the destination register dr. Overflow exception is raised if the result' s exponent exceeds 127. Underflow exception is raised if the result' s exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever both operands are infinites with opposite signs, or when at least one of the operands is a SNaN.

fsub0

syntax: `fsub0 dr, sr1, sr2`

description: Single-precision floating-point (algebraic) subtraction to be executed by coprocessor number 0. The contents of the source register sr2 is subtracted by the one in sr1 are subtracted and the result is placed to the destination register dr. Overflow exception is raised if the result' s exponent exceeds 127.Underflow exception is raised if the result' s exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever both operands are infinites with opposite signs, or when at least one of the operands is a SNaN.

fmul0

syntax: `fmul0 dr, sr1, sr2`

description: Single-precision floating-point multiplication to be executed by coprocessor number 0. The contents of the source registers sr1 and sr2 are multiplied and the result is placed to the destination register dr. Overflow exception is raised if one of the operands is infinite and the other is a finite number, or if the result' s exponent exceeds 127. Underflow exception is raised if the result' s exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever one operand is infinite and the other one is null, or when at least one of the operands is a SNaN.

fdiv0

syntax: `fdiv0 dr, sr1, sr2`

description: Single-precision floating-point division to be executed by coprocessor number 0. The content of the source register sr2 is divided by the one in sr1 and the result is placed to the destination register dr. Overflow exception is raised if dividend is infinite and divisor is zero, or if the result' s exponent exceeds 127. Underflow exception is raised if the result' s exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever both operands are infinite or both are null, or when at least one of the operands is a SNaN. Division by zero exception is raised when a finite non-null number is divided by a null divisor.

fsqrt0

syntax: `fsqrt0 dr, sr`

description: Single-precision floating-point square-root to be executed by coprocessor number 0. The content of the source register sr is square-rooted and the result is placed to the destination register dr. Invalid operation exception occurs whenever radicand is negative, or when it' s a SNaN.

fabs0

syntax: `fabs0 dr, sr`
description: Single-precision floating-point absolute value (ABS) to be executed by coprocessor number 0. The absolute value of the content of the source register sr is placed to the destination register dr. Invalid operation exception occurs whenever operand is a NaN.

fmov0

syntax: `fmov0 dr, sr`
description: The operand has to be moved to another register by coprocessor number 0. The value of the content of the source register sr is moved to the destination register dr.

fneg0

syntax: fneg0 dr, sr
description: Single-precision floating-point sign inversion to be executed by coprocessor number 0. The value of the content of the source register sr is inverted in sign and placed to the destination register dr. Invalid operation exception occurs whenever operand is a NaN.

fnop0

syntax: `fnop0`
description: No operation is executed.

fcvt.s0

syntax: `fcvt.s0 dr, sr`
description: Integer to single-precision floating-point conversion to be executed by coprocessor number 0. The value of the content of the source register (considered as an integer) sr is converted into single precision floating-point format and placed to the destination register dr.

fcvt.w0

syntax: fcvt.w0 dr, sr
description: Single-precision executed by coprocessor of the source floating-point and placed to the exception is generated large argument (≤ -2147483649.0). Denormal numbers are flattened to zero, and inexact result exception is raised.

fcCONDITION0

syntax: `fcCONDITION0 dr, sr1, sr2`
description: Comparison to be executed by coprocessor number 0. The contents of the source registers sr1 and sr2 are compared according to the condition specified in the name of the instruction and the result is placed to the destination register dr. Invalid operation exception occurs when at least one of the operands is a NaN and MSB in the opcode is set; result is unordered. NaN compares unordered with everything including itself. Sign of zero is ignored, so +0 = -0.
note: for more details about `CONDITION` look in Milk documentation.

5. ASSEMBLER DIRECTIVES

Assembler directives are instructions to the assembler to perform various bookkeeping tasks, storage reservation, and other control functions. To distinguish them from other instructions, directive names begin with a period.
Directives should by in lower-case (case sensitive).

.align *N*

Pad the location counter (in the current section) to a particular storage boundary. *N* defines the number of zero bits in LSB end of location counter:
N = 0 => byte boundary (8-bits)(no padding)
N = 1 => halfword boundary (16-bits)
N = 2 => word boundary (32-bits)
This directive has no effect if location counter is already aligned properly.

.ascii "*some text here*"

Assemble text into consecutive addresses, one character per byte. You can optionally use the backslash escape characters. No trailing zero is added to terminate the string. ASCII 8-bit conversion is used.

.byte [*b1, b2, b3,...,bn*]

Assemble bytes *b1...bn* to consecutive addresses and increment location counter after each byte. If no arguments are given, location counter is incremented by one.

.bss

Start or continue **bss** section. In practice a **bss** section can contain only allocation of zero initialized or uninitialized data, like this:

```
my_variable_in_bss_section:      .word      0
your_variable_in_bss_section:    .word
```

.code*16|32|N*

This directive is used to switch instruction encoding mode. With selector '32' the assembler will switch to 32 bit mode outputting 32-bit machine instructions, with selector '16' 16-bit instructions are output. *N* accommodates for future extensions to instruction set architecture.
This directive should follow the section description directive.

Note: before using one more time `.codeXX` directive in the section, make sure `SWM XX` instruction was in use before.

.data

Start or continue data section.

.double [*n1*,*n2*,*n3*,*n4*,...,*nn*]

Assemble double precision (64-bit presentation) floating point numbers. Each number reserves eight bytes, so location counter is incremented by eight after each number. IEEE Standard 754 is followed. If no arguments are given, location counter is incremented by eight and zeros allocated. No rounding is done.

.equ *SYMBOL, EXPRESSION*

This directive sets the value of *SYMBOL* to *EXPRESSION*. To define name aliases is used syntax `SYMBOL = VALUE`.
Constants are global for whole code.

.err [*"Error message"*]

When the assembler encounters this directive, it prints the string in quotes (if any given) and stops assembly process.

.extern *SYMBOL*[, *SYMBOL_2*, .., *SYMBOL_N*]

Define a symbol to be external. Assembler treats all undefined symbols as external but it produces warning message if some symbol was used as external, but was not declared with **.external** directive.

.fill *REPEAT*, *VALUE*[, *SIZE*]

Fill *REPEAT* x *SIZE* memory locations with *VALUE*. Location counter will be incremented by an amount of *REPEAT* x *SIZE*. *SIZE* is size in bytes; allowed values are *1*, *2*, *4* or *8*. If size is not specified, one byte is assumed.

.float [*n1*,*n2*,*n3*,*n4*,...,*nn*]

Assemble single precision (32-bit presentation) floating point numbers. Each number reserves four byte, so location counter is incremented by four after each number. IEEE Standard 754 is followed. If no arguments are given, location counter is incremented by four and zeros allocated. No rounding is done.

.global *SYMBOL*[, *SYMBOL_2*, .., *SYMBOL_N*]

Define a symbol to be visible outside current source file. Allows linking other modules with current module.

.hword [*n1,n2,n3,n4,...,nn*]

Assemble halfwords (16-bit integers) *n1...nn* and increment location counter by two per argument. If no arguments are given, location counter is incremented and zeros assembled.

.include "*filename*"

Include code from specified file. Is possible define path with filename or include with `-I` argument in calling line.

.local *label*[*, label2, …, labelN*]

Define local labels for macro. It supposed be just in 2nd macro line. In code it appears with the same name plus number of macro use, e.g. in 1st time macro call it will be *label1,* in 2nd – *label2.*

.lword [*n1,n2,n3,n4,...,nn*]

Assemble long words (64-bit integers) *n1...nn* and increment location counter by eight per argument. If no arguments are given, location counter is incremented by eight and zeros assembled.

.macro *macro_name*[(*arg1, arg2,..., argn*)]

Start macro definition. Macros can have local labels defined with **.local** directive in 2nd line (immediately after macro name). Also is possible to define constants inside macro or use already defined constants. Use of any another directive inside macro is not allowed.

.endm

Mark the end of a macro definition. If **.endm** will not be found after 100 lines, warning message is produced.

.org *new_lc_value*[*, fill_byte*]

Define a new value for current location counter. You can only advance location counter. It is not possible to go backwards. The skipped bytes are filled with *fill_byte*, which by default is zero. Note that you cannot use a label as *new_lc_value* or you cannot use an expression as *new_lc_value*.

.proc [*name*]

Start of procedure. Ignored like comment.

.endproc [*name*]

End of procedure. Ignored like comment.

.rdata

Start or continue read-only data section.

.space *N*

Reserve *N* bytes of space (increment the location counter by *N*). Zeros are assembled?.

.section *NAME*[, *TYPE*, *absolute_section_place*]

Use the **.section** directive to assemble the following code into a section named *NAME*. Section type (*TYPE*) can be one of the following: *b*, *x*, *d*, *r* or *nothing*. Explanations of section types are in Table 5-1.
If *absolute_section_place* is set, section is defined to be absolute.

Convention	Meaning
x	Executable section (executable text) (loaded to instruction memory area anyway, may contain PC relative data).
r	Read-only data section.
d	Data section (initialized data) (read, write).
b	Bss section (uninitialized data).
nothing	Regular section (allocated, relocated, loaded). In current version is the same like executable text section.

Table 5-1. Section type conventions

.text

Start or continue text section.

.word [*n1,n2,n3,n4,...,nn*]
.word "*Hello world!*"

Assemble words (32-bit integers) *n1...nn* and increment location counter by four after each parameter/character. If no arguments are given, location counter is incremented by four and zeros assembled. The second version allocates space for a string and places ASCII codes of the string to consecutive **words (3 zero bytes are added before each character)**.

6. PROGRAMMING CONSIDERATIONS

This chapter gives rules and examples to follow when designing an assembly language program.
The chapter addresses topic:

- The use of registers, section and location counters, and stack frames (Section 6.1)

This chapter does not address coding issues related to performance or optimization.

6.1 General Coding Concerns

This section describes some general areas of concern to the assembly language programmer:

- Usage of registers (Section 6.1.1)
- Control of section and location counters with directives (Section 6.1.2)

Another general coding consideration is the use of data structures to communicate between high-level language procedures and assembly procedures. In most cases, this communication is handled by means of simple variables: pointers, integers, Booleans, and single- and double-precision real numbers. Describing the details of the various high-level data structures that can also be used – arrays, records, sets, and so on – is beyond the scope of this manual.

6.1.1 Register Use

The main processor has 2 sets of 32 32-bit integer registers. The uses and restrictions of these registers are described in Table 1-1.

Register usage of a privileged user

When processor starts executing instructions after boot (see interface document) following conditions are assumed: 32 bit instruction word length, super user mode, register set SET2 for reading and writing and all interrupts (also `cop` exceptions) disabled. Boot code has the responsibility to initialize the special purpose registers to guarantee proper handling of interrupts and coprocessor exceptions. User mode can be entered by issuing the command `retu` (see Chapter 3 for more information about instructions). Before passing the control, registers SPSR and PR31 must be set appropriately. Executing `retu` causes

PSR to be overwritten by SPSR (not all flags though) and PC (program counter) overwritten by PR31. That is, execution will start at address saved to PR31 and with status flags saved in SPSR.
When an application program issues the command `scall` (requesting some system/kernel service, for example), SPSR is overwritten with PSR and PR31 is overwritten with link address (an address to return when resuming application code). In practice this means that super user is able to see the state in which the user was before calling system code and is able to resume execution from the correct address. Also the super user has full control over the user and the possibility to read and alter the status bits of the user. An application program can pass parameters to privileged software (and the other way around) in some general purpose registers RXX , if desired , since privileged software can read and write both sets of registers with the help of `chrs` command. For more information about instructions `scall`, `retu` and `chrs` see Chapter 3.

Register limitations in 16-bit mode

In 16-bit mode only the last eight registers from both sets are available, that is registers R24...R31 from set 1 and PR24...PR31 from set 2. Assembler provided straightforward notions to access registers are listed in Table 6-1.
Condition registers C1...C7 are disabled in 16 bit mode. Register C0 is always used (automatically selected) with conditional branches and arithmetic.

Register name	**Software used name**	**Description**
	32-bit mode	
R0..R31	R0..R31 | r0..r31	Set 1 registers
PR0..PR31	R0..R31 | r0..r31	Set 2 registers
C0 .. C7	C0..C7 | c0..c7	Condition registers
CR0..CR31	CR0..CR31 | cr0..cr31	Coprocessor registers
	16-bit mode	
R24..R31	R0..R7 | r0..r7	Set 1 registers
PR24..PR31	R0..R7 | r0..r7	Set 2 registers
C0	C0 | c0	Condition registers
CR0..CR31	CR0..CR31 | cr0..cr31	Coprocessor registers

Table 6-1. Register name and software used name mapping

6.1.2 Using Directives to Control Sections and Location Counters

See Section 2.10 and Section 2.11 for details about sections and location counters. Also see Chapter 5 for more information about directives `.org` and `.align`.

7. OBJECT FILES

7.1 Object File Overview

Compilers and assemblers create object files containing the generated binary code and data for a source file. Linkers combine multiple object files into one file; loaders take object files and load them into memory.
Note, in this document we speak about object files in COFF format.

What goes into an object file?

An object file contains basic information:

1. Header information: This is overall information about the file, such as the size of code, the name of source file it was translated from, and the creation date.
2. Object code: This is binary instructions and data generated by a compiler or assembler.
3. Relocation information: This is a list of the places in the object code that have to be fixed up when the linker changes the address of the object code.
4. Symbols: These include global symbols defined in this module and symbols to be imported from other modules or defined by the linker.
5. Debugging information: This includes other information about the object code that is not needed for linking but is useful to a debugger (such as source file and line number information, local symbols, and descriptions of data structures used by the object code such as C structure definitions)

Figure 7-1 shows the overall structure of the object file.

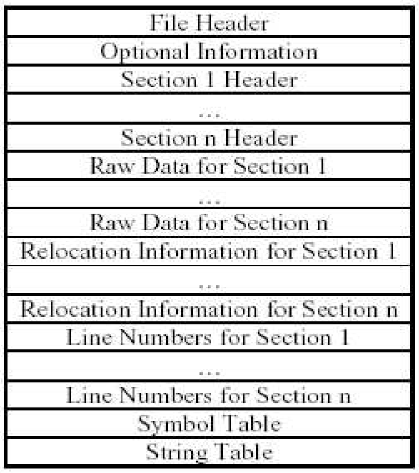


Figure 7-1. The structure of the object file

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and it is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

Designing an object format

The design of an object format is a compromise driven by the various uses to which an object file will put. A file may be **linkable**, used as input by link editor or linking loader; **executable**, capable of being loaded into memory and run program; **loadable**, capable of being loaded into memory as a library along with a program; or any combination of the three.

A linkable file contains extensive symbol and relocation information needed by the linker along with the object code. The object code is often divided up into many small logical segments that will be treated differently by the linker. An executable file contains object code – usually page aligned to permit the file to be mapped into the address space – but doesn't need any symbols (unless it will do run-time dynamic linking) and needs little or no relocation information. The object code is a single large segment or a small set of segments that reflect the hardware execution environment (most often read-only vs. read/write pages). Depending on the details of a system's run-time environment, a loadable file may consist solely of object code, or it may contain complete symbol and relocation information to permit run-time symbol linking.

There is some conflict among these applications. The logically oriented grouping of linkable segments rarely matches the hardware-oriented grouping of executable segments. Particularly on smaller computers, linkable files are read and written by linker a piece at a time, while executable files are loaded as a whole into main memory (DOS linkable OMF and executable EXE).

The basic elements of the COFF definition

A simple abstraction is essential to the COFF concept, an abstraction that identifies the most seminal, common denominators of all operating systems.

The COFF system **maps** the three abstract elements of a program: machine code, initialized data, and uninitialized data, to three corresponding special **sections** in the COFF file:

- The **text section**
- The **data section**
- The **bss section**

The COFF file also includes areas for relocation information and symbolic debug information. All this information is organized as a data structure.
The COFF defined data structure includes an organized system of pointers that allow efficient access to, and manipulation of, any of the three sections, as well as the symbolic debug information and relocation information areas that contain useful information to the linker.

The COFF definition creates two major benefits: enhanced portability and system extensibility.

7.2 Object File Content

A section is the smallest portion of the object file that is relocated and treated as one separate and distinct entity. Text sections contain executable machine code and the operating system treats them as write protected. Data sections contain initialized program code and are readable and writable. Bss sections basically contain information on how large the uninitialized data area is. The **bss section** is usually made a contiguous with the **data section** when the program is loaded into memory.

Software defined data structures are also easily extensible. Though the **text**, **data** and **bss** sections are special, they are not sacrosanct. If necessary, it is possible to add sections to the COFF definition.

Also the assembler's **.section** directive was created in response to the need of a special section. The **.section** allows the specification of a section name, and the section content's type. User defined sections follows main sections.

COFF file headers

The roughly fifty or so bytes at the beginning of the COFF file contain the COFF file headers. The COFF file headers hold, among other things, the information indicating whether or not a file is executable and general run-time parameters. The header is also the beginning point for the system of pointers that relate the different structures of the COFF file.

There are two COFF headers; both are defined as structures that contain pertinent COFF information fields. The first is called the **file header**, and the second (which may or may not be present) is called the **optional header**.

7.2.1 The File Header

The first of the two COFF headers are usually simply referred to as the **file header** and contains general information such as a file time stamp and a magic number.

The file header has 20 bytes of information as shown in Table 7-1.

Bytes	**Name**	**Description**
0-1	f_magic	Magic number for target machine (0xC00F for COFFEE ™ RISC core)
2-3	f_nscns	Number of sections contained within this file (main and subsections)
4-7	f_timdat	Time and date stamp indication when the file was created, expressed seconds since 00:00:00 GTM, January 1, 1970
8-11	f_symptr	File pointer containing the starting address of the symbol table
12-15	f_nsyms	Number of entries in the symbol table
16-17	f_opthdr	Number of bytes in the optional header
18-19	f_flag	Flags

Table 7-1. File header information

In general, `f_nscns` field says how many section headers are following file header (and optional header).
File pointer is the byte offset to the start of the symbol table from beginning of the file. The flags describe the type of the object file. Currently defined flags are presented in Table 7-2.

Flag	Name	Description
0x0001	F_RELFLG	If set, there is no relocation information in this file. This is usually clear for objects and set for executables
0x0004	F_LNNO	If set, all line number information has been removed from the file (or was never added in the first place)
0x0008	F_LSYMS	If set, all local symbols have been removed from the file (or were never added in the first place)

Table 7-2. Currently defined flags

7.2.2 The Optional Header

The second COFF header is known by at least four names: optional header, standard header, system `a.out` header, and auxiliary header. In this document we choose to call the second header the **optional header**.

Most of the fields in the optional header provide run-time information about the COFF file. And since only executable files need run-time information, it is the linker that fills in the appropriate values. Typically, assembler-created object files do not contain the optional header, but if the optional header is present, most of its values are meaningless (and not necessary initialized to zero).

`Crasm` (COFFEE ™ RISC Assembler) provides COFF object file without optional header (`f_opthdr` is always zero).

7.2.3 Section Headers

Section headers follow the optional header. The position of the first section header is found by adding the size of the **file header** to the value found in the `f_opthdr` that represent the size of optional header.

The section header contains two fields that play a key role in the process of relocation: `s_relptr`, the pointer to the relocation entries; and `s_scnptr`, the pointer to the section raw data.

Subsections headers can follow section header. Amount of subsection headers is set in **s_flag** field. All subsections are part of main section, just divided by coding mode. Mode is set in **s_flag** field.

Each section header has 40 bytes of information as shown in Table 7-3.

Bytes	Name	Description
0-7	s_name	8-character null padded section name

8-11	s_paddr	Physical address of section. For unlinked objects, this address is relative to the object's address space (i.e. first section is always at offset zero)
12-15	s_vaddr	Virtual address of section. Always the same value as s_paddr
16-19	s_size	Section size in bytes. You should always read this many bytes from the file, beginning s_scnptr bytes from beginning of the object. Zero if section is empty
20-23	s_scnptr	File pointer to raw data for this section
24-27	s_relptr	File pointer to relocation entries for this section
28-31	s_lnnoptr	File pointer to line number entries for this section
32-33	s_nreloc	Number of relocation entries for this section. Beware files with more than 65535 entries; this field truncates the value with no other way to get the "real" value
34-35	s_nlnno	Number of line number entries for this section. Beware files with more than 65535 entries; this field truncates the value with no other way to get the "real" value
36-39	s_flags	Flags

Table 7-3. Section header information

The size of a main section is padded to a multiple of 4 bytes.
Long names of sections are kept in **string table**; in that case `s_name` field starts with slash ('/'), and has offset to string table where the name is located.
Flags describe section contents and determine how the linker and system loader handle the section.
Detailed explanation of s_flag bytes is in Table 7-4.

Byte	**Description**
1	Section mode description
2-3	Amount of subsections in main section
4	Section contents description

Table 7-4. Detailed s_flag explanation

Possible values of section mode are in Table 7-5. Possible values of section contents are in Table 7-6.

Flag	**Description**
0x00	Main section (mode unimportant)
0x10	32-bit mode subsection
0x01	16-bit mode subsection

Table 7-5. Section mode flags

Flag	Name	Description
0x10	STYP_RDATA	Section contains only read-only data
0x20	STYP_TEXT	Section contains executable text; text sections are allocated, relocated, and loaded
0x40	STYP_DATA	Section contains initialized data; data sections are allocated, relocated, and loaded
0x80	STYP_BSS	Section contains only uninitialized data; bss sections are only allocated

Table 7-6. Section contents flags

7.2.4 Section Data

The raw data for each section begins at a 4-byte boundary in the file. Section data are in the same sequence as sections headers. Each section data can be found by using `s_scnpt` **pointer** value from that section header.

Predefined section header and section raw data sequence is presented in Figure 7-2 (it is similar to OMAGIC definition).

.text	Text segment
.rdata	
user defined text or rdata section 1	
…	
user defined text or rdata section n	
.data	Data segment
user defined data section 1	
…	
user defined data section n	
.bss	Bss segment
user defined bss section 1	
…	
user defined data section 1	

Figure 7-2. Predefined section raw data sequence

7.2.5 Section Relocation Information

A relocation entry is created by the assembler for every instance of an address reference that requires patching by the linker. The relocation entry's field values identify the area in raw data that needs patching and associates that area with a symbol table entry that defines the run-time address – the value used to patch the raw data. Note, some instructions can have 2 relocation entries.

Each relocation entry has 10 bytes of information as shown in Table 7-7.

Bytes	Name	Description
0-3	r_vaddr	(Virtual) address of relocation. This is a byte-offset value relative to the start of its raw data
4-7	r_symndx	Pointer to appropriate symbol table entry that contains run-time address information (counted from 0). The symbol table entry is accessed by adding this value, **r_symndx**, to the value of **f_symptr**.
8-9	r_type	Type of relocation

Table 7-7. Relocation entry information

The `r_type` field tells the linker which algorithm to use during the address calculation process. Currently defined types are in presented in Table 7-8.

Type	Bit form	Description
0x67C8	011 00111 11001 000	Start in 3rd bit, 7 least significant bits after shifting to right by 25; simple relocation independent on mode
0x6790	011 00111 10010 000	Start in 3rd bit, 7 least significant bits after shifting to right by 18; simple relocation independent on mode
0x6758	011 00111 01011 000	Start in 3rd bit, 7 least significant bits after shifting to right by 11; simple relocation independent on mode
0x6720	011 00111 00100 000	Start in 3rd bit, 7 least significant bits after shifting to right by 4; simple relocation independent on mode
0x6400	011 00100 00000 000	Start in 3rd bit, 4 least significant bits, no shifting; simple relocation independent on mode
0x4F00	010 01111 00000 000	Start in 2nd bit, 15 least significant bits, no shifting; simple relocation independent on mode
0x2178	001 00001 01111 000	Start in 1st bit, 1 least significant bit after shifting to right by 15; simple relocation independent on mode
0x4F80	010 01111 10000 000	Start in 2nd bit, 15 least significant bits after shifting to right by 16; simple relocation independent on mode
0x21F8	001 00001 11111 000	Start in 1st bit, 1 least significant bit after shifting to right by 31; simple relocation independent on mode
0x0000	000 00000 00000 000	Start in 0 bit, 32 least significant bits (zero length is a nonsense, so it should be assume as 32-bit long), no shifting; simple relocation of external or internal defined word independent on mode
0x1606	000 10110 00000 110	Start in 0 bit, 22 least significant bits, no shifting; 32-bit mode external PC relative relocation
0x1906	000 11001 00000 110	Start in 0 bit, 25 least significant bits, no shifting; 32-bit mode external PC relative relocation
0x0A05	000 01010	Start in 0 bit, 10 least significant bits, no shifting; 16-bit

	00000 101	mode external PC relative relocation

Table 7-8. Currently defined relocation types

Detailed explanation of **bit form** of relocation type is in Table 7-9.

Bit	Description
15-13	Bit position where relocation starts; in byte specified by r_vaddr
12-8	Length of immediate (address) value in bits; least significant bits
7-3	Length of shift to left in bits
2-0	Relocation mode

Table 7-9. Explanation of a bit form in relocation type

7.2.6 Line Numbers Information

Line number information is a special part of the COFF file that contains line number structures. The line number structure associates every line in the source file that represents machine code with its relevant address in the text section. Line number structures allow creation of breakpoints by symbolic definition, and support source code trace of program execution.

`Crasm` (COFFEE ™ RISC Assembler) provides COFF object file without line numbers (`s_lnnoptr`, `s_nlnno` and `x_nlinno` fields are always zero).

7.2.7 Symbol Table Information

Though the symbol table entry is not an excessive large structure, the information it contains is the most complex of the COFF definition. This is because of the complex nature of debug information. All symbols have a symbol table entry, but not all have relocation information.

COFF defines a dual role for the symbol table: defining the run-time address for the relocation process, and providing symbolic debug information. For the moment, debug information aspect is ignored and instead the explanation concentrates only on those parts of the symbol table entry that play a role in the relocation process.

Symbols appear in the sequence show in Figure 7-3(order is very important only for debug information).

Static symbols and labels

Defined global symbols
Undefined global symbols

Figure 7-3. Symbols appearing sequence

The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Table 7-10. Note that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Name	Description
0-7	n_name	8-character null padded section name or an index to a symbol in the string table
8-11	n_value	Relocatable address of the symbol. This value is placed into the area in the section's raw data pointed to by the relocation structure's r_vaddr value
12-13	n_scnum	Section number where the symbol is defined. The first section is section one
14-15	n_type	Basic and derived type specification. Currently not in use and always is 0
16	n_sclass	Storage class of symbol. Tells where and what the symbol represents
17	n_numaux	Number of following auxiliary entries

Table 7-10. Symbol table information

7.2.7.1 Symbol name

The first 8 bytes in the symbol table entry can have two meanings. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the first byte is zero, and the second one is the offset (relative to the beginning of the string table) of the name in the string table as shown in Table 7-11.

Bytes	**Name**	**Description**
0-3	n_zeroes	Zero in this field indicates the name is in the string table
4-7	n_offset	Offset of the name in the string table

Table 7-11. Description of a symbol name

7.2.7.2 Symbol value

The meaning of a symbol value depends on its storage class. Crasm (COFFEE ™ RISC Assembler) used storages classes are listed in chapter **Storage Class**. In all cases value has a meaning **relocatable address**.

Relocatable symbols have a value equal to the virtual address of the symbol (relative to the beginning of section raw data). When the linker relocates the section, the value of theses symbols changes.

7.2.7.3 Section number

The meaning of `n_scnum` field is summarized in Table 7-12.

Value	Name	Description
-1	N_ABS	Absolute symbol
0	N_UNDEF	Undefined (external) symbol
> 1	N_SCNUM	Section number

Table 7-12. The meaning of n_scnum field

The subsections are counted as sections too, because they section headers are listed. Section numbers are directly connected with section headers: `n_scnum = 1` links to 1st section header.
Subsections aren't listed in Symbol Table.

7.2.7.4 Storage class

The storage class field `n_sclass` has one of the values described in Table 7-13.

Value	Name	Description
0x00	C_NULL	–
0x02	C_EXT	External (public) symbol
0x03	C_STAT	Static (private) symbol
0x06	C_LABEL	Label

Table 7-13. Storage class field n_sclass values

7.2.7.5 Auxiliary entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry (18 bytes). However, unlike symbol table entries, the format of an auxiliary table entry depends on symbol type and storage class.

`Crasm` (COFFEE ™ RISC Assembler) uses auxiliary entries just for sections. The format is shown in Table 7-14.

Bytes	Name	Description
0-3	x_scnlen	Section length
4-5	x_nreloc	Number of relocation entries
6-7	x_nlinno	Number of line numbers
8-17	–	Unused (padded with zeros)

Table 7-14. The format of a auxiliary table entry

7.2.8 String Table Information

The string table is the final component of the symbolic system. If a symbol exceeds eight characters, the name field in the symbol table structure does not contain the name, but instead it is an offset in the string table. The string table consists of null-terminated strings; therefore it can support symbol names of any length.

The first four bytes of the string table is the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4.
An empty string table always has the first four bytes used for defining the length, but the length value in this case is 0.

7.3 Assembler and Linker Process of Relocation

This section presents a step-by-step walkthrough of a simple relocation process.
The simple relocation case occurs only when one source file is compiled and linked. This is not very realistic, since most applications consist of several source files that have external symbolic references. Though lacking realism, the simple relocation case is the best way to explain the relocation process.

Relocatable code before linking

The following example shows address encoding of a machine code symbolic access to data defined in the data section:

```
Address      Address                Opcode                        Source
in code      in section
------------------------------------------------------------------------
                                                                .TEXT
00000000     00000000               0100101000000100           xor r4, r4
00000002     00000002               1010100001101100           ori r4, @label
00000004     00000004               0011010001110100           slli r4, 7
00000006     00000006               1010101001101100           ori r4, @label
00000008     00000008               0011010001110100           slli r4, 7
```

```
0000000A     0000000A           1010101011001100         ori r4, @label
0000000C     0000000C           0011010001110100         slli r4, 7
0000000E     0000000E           1010101100001100         ori r4, @label
00000010     00000010           0011010001000100         slli r4, 4
00000012     00000012           1010100001110100         ori r4, @label
                                                         .DATA
00000014     00000000           ...                         <data>
...           ...               ...                         ...
1B36CE32     1B36CE1E           01100001                    label: a
...           ...               ...                         ...
1B36CE34     1B36CE20                                       <data>
```

Note, this code is produced by the assembler. In original source code text section looks like this:

```
.text
.code16
ldra r4, label
```

The object file created by the assembler results in the `ori r4, @label` instructions finding *label* at address **0x1B36CE1E**. Whole address is too long to fit into `ORI` instruction, so it is translated into 32-bit binary (**0b00011011001101101100111000011110**), divided into separate parts and written into separate `ORI` instructions (the emboldened portion of the opcode).

```
00011011001101101100111000011110     - original address (0x1B36CE1E)
0001101                              - 1st ORI
       1001101                       - 2nd ORI
              1011001                - 3rd ORI
                     1100001         - 4th ORI
                            1110     - 5th ORI
```

There is no need to regenerate original address value from `ORI` (`LUI` or `LLI`) instructions (using relocation type `r_type` field from relocation entries table), because each instruction relocation entry has index to symbol table entry (`n_symndx`) where this value is located in `n_value` field.

Linking this object file causes the relocation process to be performed. For the moment, assume that linked executable files have the text section starting at 0x0, and the data section starting at 0x100. This means that the linker updates (or relocates) the `ori r4, @label` instructions symbolic reference to `label` with the correct run-time address.

Each `ORI` instruction has different relocation entry, but they point to the same symbol table entry.

The 1st `ORI` instruction relocation information is following:
`00000003 00000004 67C8`
<u>That means:</u>

`r_vaddr = 0x3`
`r_symndx = 0x4`
`r_type = 0x67C8` (Start in 3rd bit, 7 least significant bits after shifting to right by 25; simple relocation)

Linker should understand current line as follows – the address value from symbol table entry 4 should be shifted to right by 25 bits and 7 least significant bits are written into byte 0x3 (relative to text section – because this relocation entry depends to text section) begin with 3rd bit position.

The 2nd `ORI` instruction relocation information is following:
`00000007 00000004 6790`
<u>That means:</u>
`r_vaddr = 0x7`
`r_symndx = 0x4`
`r_type = 0x6790` (Start in 3rd bit, 7 least significant bits after shifting to right by 18; simple relocation)

Linker should understand current line as follows – the address value from symbol table entry 4 should be shifted to right by 18 bits and 7 least significant bits are written into byte 0x7 (relative to text section – because this relocation entry depends to text section) begin with 3rd bit position.

The 3rd `ORI` instruction relocation information is following:
`0000000B 00000004 6758`
<u>That means:</u>
`r_vaddr = 0xB`
`r_symndx = 0x4`
`r_type = 0x6758` (Start in 3rd bit, 7 least significant bits after shifting to right by 11; simple relocation)

Linker should understand current line as follows – the address value from symbol table entry 4 should be shifted to right by 11 bits and 7 least significant bits are written into byte 0xB (relative to text section – because this relocation entry depends to text section) begin with 3rd bit position.

The 4th `ORI` instruction relocation information is following:
`0000000F 00000004 6720`
<u>That means:</u>
`r_vaddr = 0xF`
`r_symndx = 0x4`
`r_type = 0x6720` (Start in 3rd bit, 7 least significant bits after shifting to right by 4; simple relocation)

Linker should understand current line as follows – the address value from symbol table entry 4 should be shifted to right by 4 bits and 7 least significant bits are written into byte 0xF (relative to text section –

because this relocation entry depends to text section) begin with 3rd bit position.

The 5th `ORI` instruction relocation information is following:
`00000013 00000004 6700`
<u>That means:</u>
`r_vaddr = 0x13`
`r_symndx = 0x4`
`r_type = 0x6700` (Start in 3rd bit, 4 least significant bits, no shifting; simple relocation)

Linker should understand current line using following instructions – the from address value from symbol table entry 4 (without shifting) 4 least significant bits are written into byte 0x13 (relative to text section – because this relocation entry depends to text section) begin with 3rd bit position.

The symbol information (from symbol table entry 4) is following:
`6c6162656c000000 1B36CE1E 0002 0000 06 00`
<u>That means:</u>
`n_name = label`
`n_value = 0x1B36CE1E`
(or `0b00011011001101101100111000011110`)
`n_scnum = 0x2` (points to .data section)
`n_type = 0x0`
`n_sclass = 0x6` (`C_LABEL`)
`n_numaux = 0x0` (no auxiliary entries)

Linker algorithm

One way how linker can calculate relocated address:

1. Linker sets the run-time start addresses for sections.
2. Linker gets current relocation address from symbol table entry (which is indexed by `r_symndx`) `n_value` field.
3. Linker calculates new relocation address by adding run-time start address of section (which number is in symbol entry `n_scnum` field) and current relocation address (because it is relative offset of the symbol within the section).
4. Now linker needs to do manipulations with new calculated address. This is needed because address is assumed to be 32-bits long but places for immediate values in instructions are less. Manipulations are described in `r_type`:
   - Linker needs shift to left new value by so many bits as it is set in 7..3 bits from `r_type`;
   - Linker takes so many least significant bits as it is set in 12..8 bits from `r_type`;
5. Linker gets byte address in raw data with pointer `r_vaddr` and exact bit position is set in 15..13 bits from `r_type`. That is

starting point for writing bits that it took in step 4. Note, bits are needed to be written form right to left starting from least significant.

Example of 2nd `ORI` instruction:

1. The new address of data section is `0x100`
2. Symbol table entry `4` → `6c6162656c000000 1B36CE1E 0002 0000 06 00`
   IMM address is `0x1B36CE1E` (`n_value` fielf).
3. New address value = `0x1B36CE1E` + `0x100` = `0x1B36CF1E`
4. `1B36CF1E` in 32-bit binary is `0b00011011001101101100111100011110`
   Relocation entry for this instruction → `00000007 00000004 6712`
   `r_type = 0x6790` can be written:
   ```
   011 00111 & 10010 000
   3  |  7   |  18  | 0
   ```
   Linker should understand whole line as follows – the address of IMM from symbol table entry 4 (note: entry counting starts form 0, auxiliary entries are counted too) should be shifted to right by 18 bits and 7 least significant bits are written into byte 0x7 (relative to text section because this relocation depends to text section) begin with 3rd bit position.
   - We need shift value 0b00011011001101101100111100011110 to right by 18. We get 0b00011011001101
   - For as important are just 7 least significant bits, so we get 0b1001101
5. New value from step 4 is written into 0x7 byte begin with 3rd bit position.

```
00000006        10101010 01101100            ori r4, @label
                 6th byte 7th byte
00000006        1010101001101 100            ori r4, @label
                            ^
                            |
                            3rd bit
00000006        101010 1001101 100            ori r4, @label
                           <-
                            7 bits long IMM value
```

Relocatable code after linking

The relocated code looks like this:

```
Address                 Opcode                      Source
--------------------------------------------------------------------------
                                                 .TEXT
00000000                0100101000000100            xor r4, r4
00000002                1010100001101100            ori r4, @label
00000004                0011010001110100            slli r4, 7
00000006                1010101001101100            ori r4, @label
00000008                0011010001110100            slli r4, 7
0000000A                1010101011001100            ori r4, @label
0000000C                0011010001110100            slli r4, 7
0000000E                1010101110001100            ori r4, @label
00000010                0011010001000100            slli r4, 4
00000012                1010100001110100            ori r4, @label
                                                 .DATA
00000100                                            <data>
...                     ...                         ...
1B36CF1E                01100001                    label: a
...                     ...                         ...
1B36CF20                                            <data>
```

After linking expressions like **address in code (relative to start of whole source code/file)** and **address in section (relative to start of section)** isn't used. Now the address depends on new value where section is relocated.
The new address of the *label* is **0x1B36CF1E** (old value plus new address of data section start), in 32-bit binary it is **0b00011011001101101100111000011110**. This value is shifted and parted according to defined relocation types `r_type` (the emboldened portion of the opcode).

```
00011011001101101100111100011110     - original address (0x1B36CF1E)
0001101                              - 1st ORI
       1001101                       - 2nd ORI
              1011001                - 3rd ORI
                     1110001         - 4th ORI
                            1110     - 5th ORI
```

7.4 Object-File Formats (OMAGIC, NMAGIC, ZMAGIC)

The optional header stores run-time information about the object. Its magic number field indicates how the file is to be organized in virtual memory.

The possible image formats are:

- **Impure Format (OMAGIC)** (Section 7.4.1)

OMAGIC files are typically relocatable object files. They are referred to as "impure" because the text segment is w ritable.

- **Shared Text Format (NMAGIC)** (Section 7.4.2)
  NMAGIC files are static executables that use a different organization from the default ZMAGIC layout. The NMAGIC format is historical and offers no special advantages. In an NMAGIC file, the text segment is shared.
- **Demand Paged Format (ZMAGIC)** (Section 7.4.3)
  ZMAGIC files are executable files or shared libraries. This format is referred to as demand-paged because its segments are blocked on page boundaries, allowing the operating system to page in text and data as needed by running process.

The ordering of section within segment is flexible. All following figures depict the default ordering as laid out by the linker.
The default segment ordering, which places the text segment before the data segment, is flexible. However, the bss segment is required to contiguously follow the data segment, wherever the data segment is located.

All three formats are constrained by the following restrictions:

- Segments must not overlap
- The bss segment must follow the data segment

7.4.1 Impure Format (OMAGIC) Files

The typically **OMAGIC** format is shown in Figure 7-4.

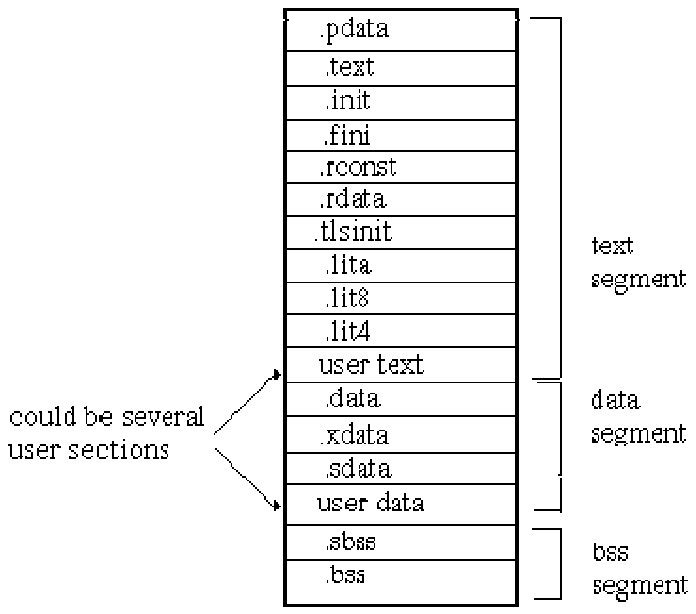


Figure 7-4. OMAGIC layout

Features:

- Segments must not overlap
- The bss segment must follow the data segment
- Starting section addresses are aligned on a 16-byte boundary
- Pre-link OMAGIC objects are zero-based, with the data segment contiguous to the text segment
- May contain relocation information
- Cannot be a shared object

OMAGIC layout is most commonly used for pre-link object files produced by compilers. Post-link OMAGIC files tend to be used for special purposes such as loadable device drivers or on input objects.

OMAGIC files can also be executable. A programmer might also choose to use an OMAGIC format for self-modifying programs or for any other application that has a reason to write to the text segment.

7.4.2 Shared Text (NMAGIC) Files

The **NMAGIC** file format is of historical interest only. The typically **NMAGIC** format is shown in Figure 7-5.

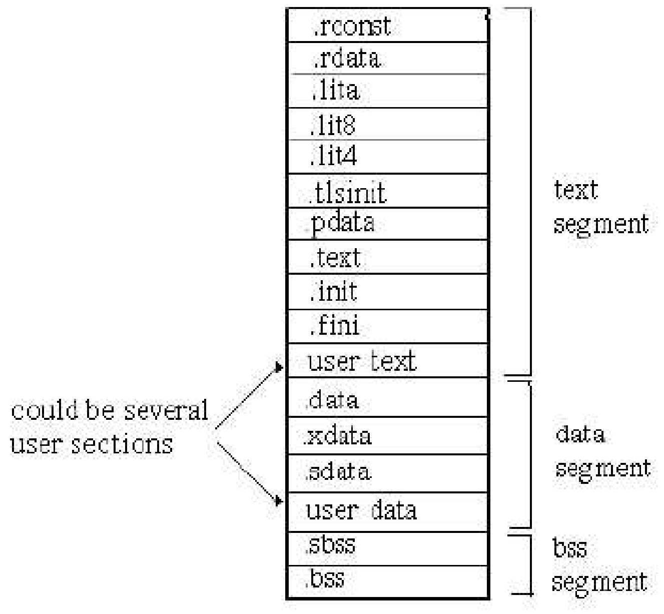


Figure 7-5. NMAGIC layout

Features:

- Segments must not overlap
- The bss segment must follow the data segment

- Text and data segment addresses fall on page-size boundaries. The bss segment is aligned on a 16-byte boundary
- Cannot contain relocation information
- Cannot be a shared object

7.4.3 Demand Paged (ZMAGIC) Files

The **ZMAGIC** format can have 2 different layouts:

- Layout for shared objects shown in Figure 7-6.

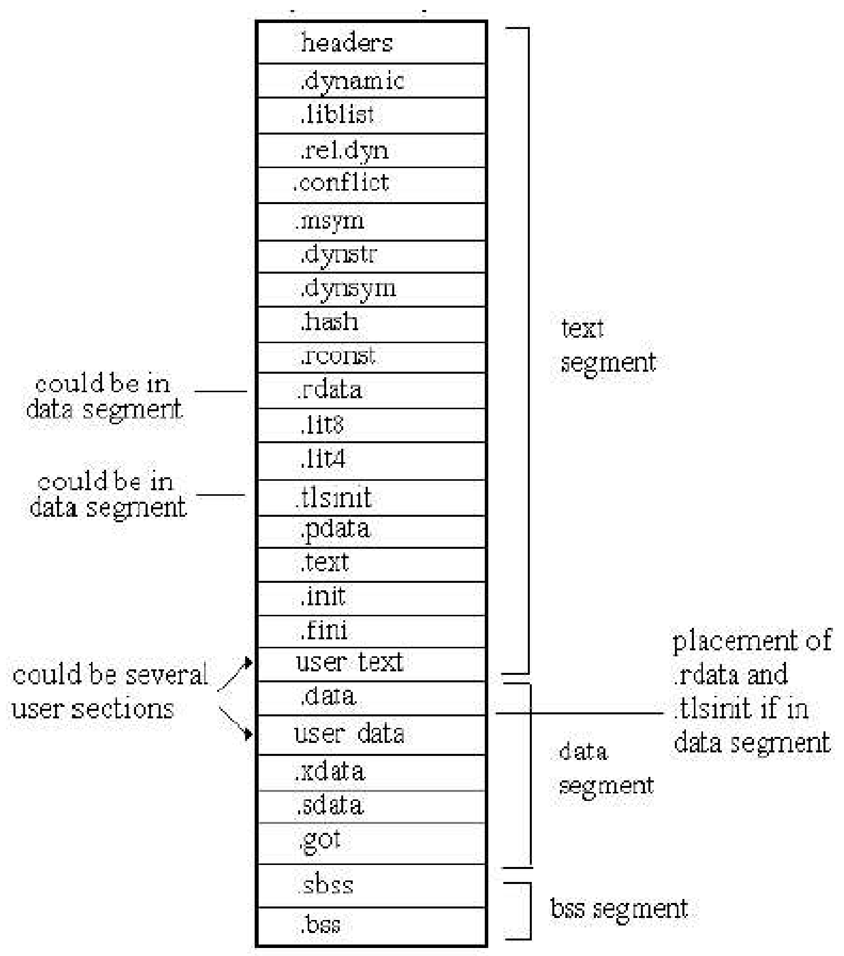


Figure 7-6. ZMAGIC dynamic layout

- Layout for static executable objects shown in Figure 7-7.

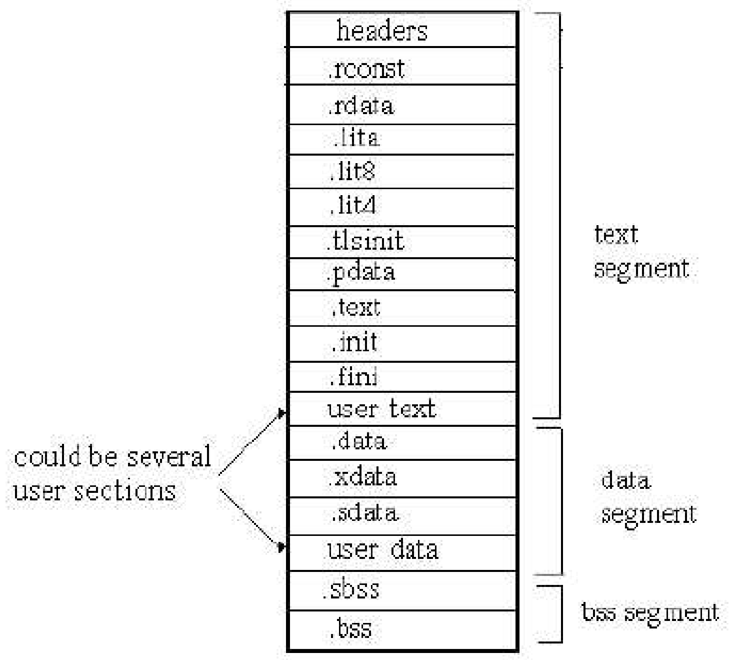


Figure 7-7. ZMAGIC static layout

Features:

- Segments must not overlap
- The bss segment must follow the data segment
- Text and data segments are blocked; the blocking factor is the page size
- Can be either a shared or nonshared object
- Cannot contain relocation information, but shared objects may contain dynamic relocation information

The `.rdata` and `.tlsinit` sections are shown as a part of the text segment. However, it is possible that one or both of those sections might be in the data segment. They are placed in the data segment only if they contain dynamic relocations.

8. REFERENCES

1. Linkers and Loaders, John R.Levine.
2. Understanding and Using COFF, Gintaras R.Gircys.
3. Developer's Topics, Chapter 7 – Common Object File Format.