COFFEE Core USER MANUAL

July 2007

Contents

6. Instruction Set Specifications
    6.1. General information
    6.2. Instruction definition
    6.3. Instruction execution cycle times
    6.4. ISA summary

1. Interface specification of the COFFEE RISC Core

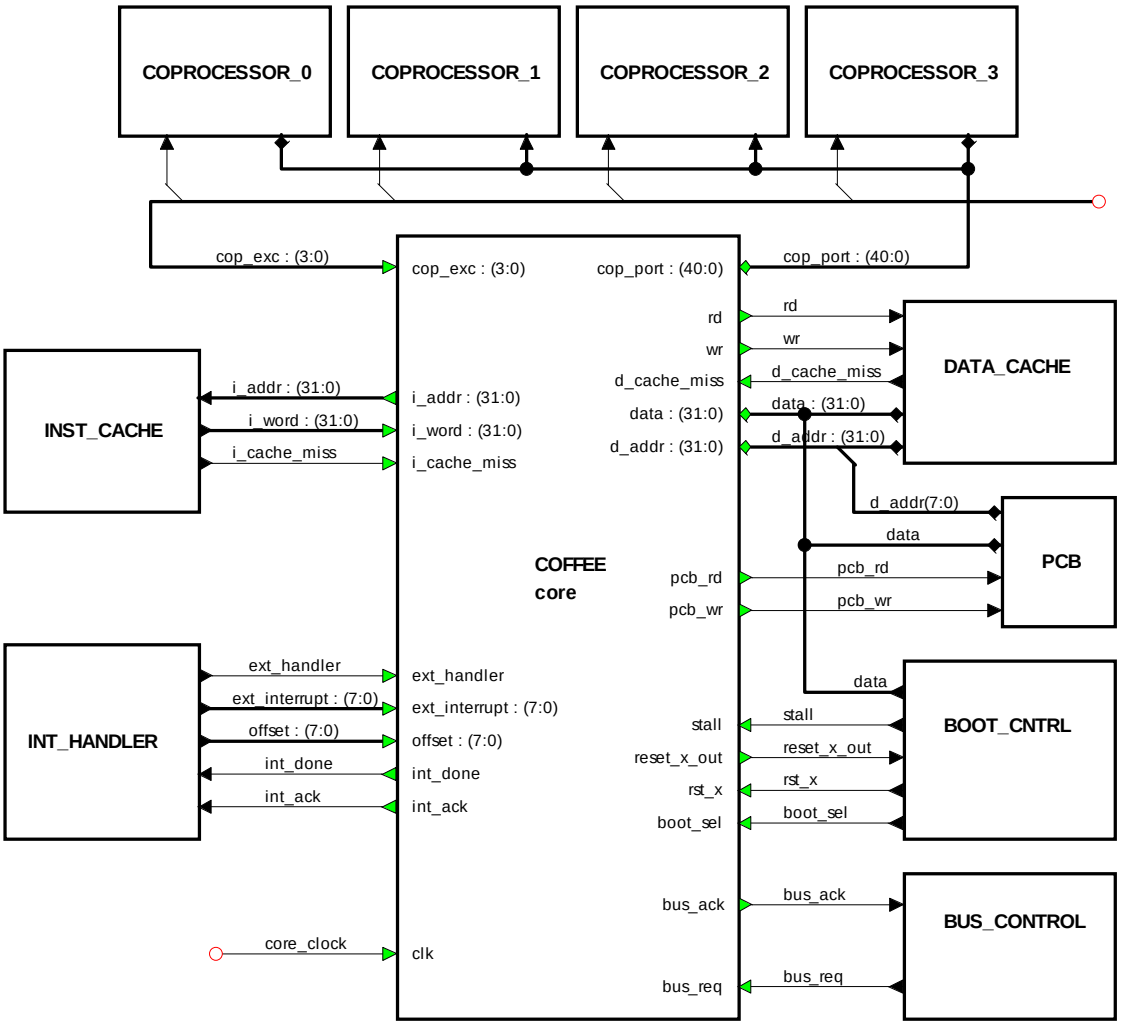

Figure 1.1 Core Interface

1.1. Shared data bus

COFFEE processor core allows sharing its data memory bus with other devices. In simple applications all peripherals and data memory can be connected directly to COFFEE core data bus. Multiprocessor systems with shared memory are also possible. However it should be noted that having too many devices communicating via shared bus eventually reduces performance since core will be stalled every time it needs the bus but it's not available.

Handshaking and accessing

Before accessing the bus, external device has to signal a request to COFFEE core. If multiple devices are connected, a bus arbiter must be used. Here, we assume that arbiter or some other device is communicating with core. Two signals are used for handshaking: *bus_req* (input to core) and *bus_ack* (output from core). Handshaking scheme is basically requesting and passing '*token*', that is the ownership of the bus.

An external device has to request the bus by asserting *bus_req* signal. It can access the bus when core responds by asserting *bus_ack* signal. This will happen immediately if core is not accessing the bus or after current access if the bus is reserved by core. An external device must hold *bus_req* signal active throughout its access. It can perform several consecutive data transfers if core does not request the bus back (which is seen from *bus_ack* -signal going low).When core requests the bus by driving *bus_ack* -low an external device can finish current access, but must pass the token to core(this is only a recommendation, solution will depend on application). This is signaled by driving *bus_req* low. If an external device has finished its access and has no further transfers, the token is automatically passed to core even if it does not request it. This happens when the device lowers *bus_req* signal. This guarantees that the core can access the bus immediately if there's no traffic (in other words, core has the token by default). After reset, core owns the token.

This simple scheme should be adequate for most applications: it's fair since the bus cannot be locked by one device and it's predictable because the bus will be available as soon as an active access is finished. Also the core, which is assumed to use the bus mostly, does not have to request the token if there's no traffic on bus.

When an external device owns the token, core floats both address and data lines, read and write controls will be low (signals *rd*, *wr*, *pcb_rd* and *pcb_wr*). Likewise any other device must float the bus when core owns the token. Note that when core has the token but is not using the bus (bus idle cycle), it holds values of the last access on data and address lines (power saving feature).

Note, that read and write signals from multiple devices must be connected via OR gate because they are not tri-state signals. For detailed description of data memory interface signals, see COFFEE interface.

About timing

Bus_req is an asynchronous input while *bus_ack* is a synchronous output. Appropriate timing constraints must be applied to *bus_req* to get correct results from synthesis if core is synthesized separately.

Timing diagrams

Figures T.5 through T.9 illustrate the timing of signals of the shared bus interface. Table 1 explains mnemonics and keywords. All delays are relative to previous rising edge of the clock CLK.

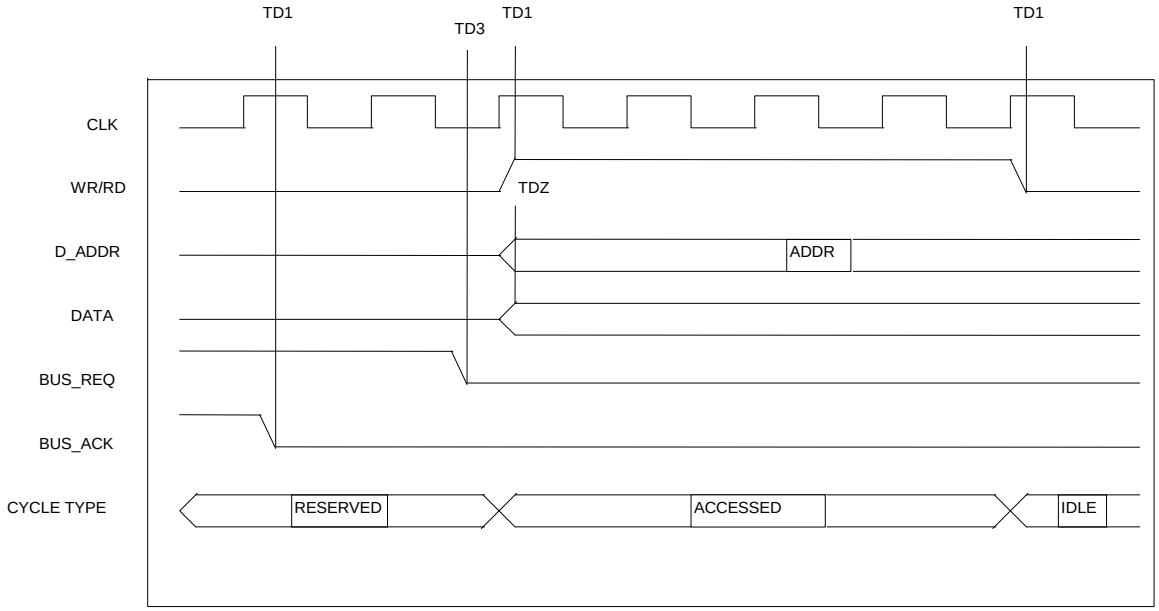

FIGURE T.5, DATA BUS STATE TRANSITION: RESERVED => ACCESSED

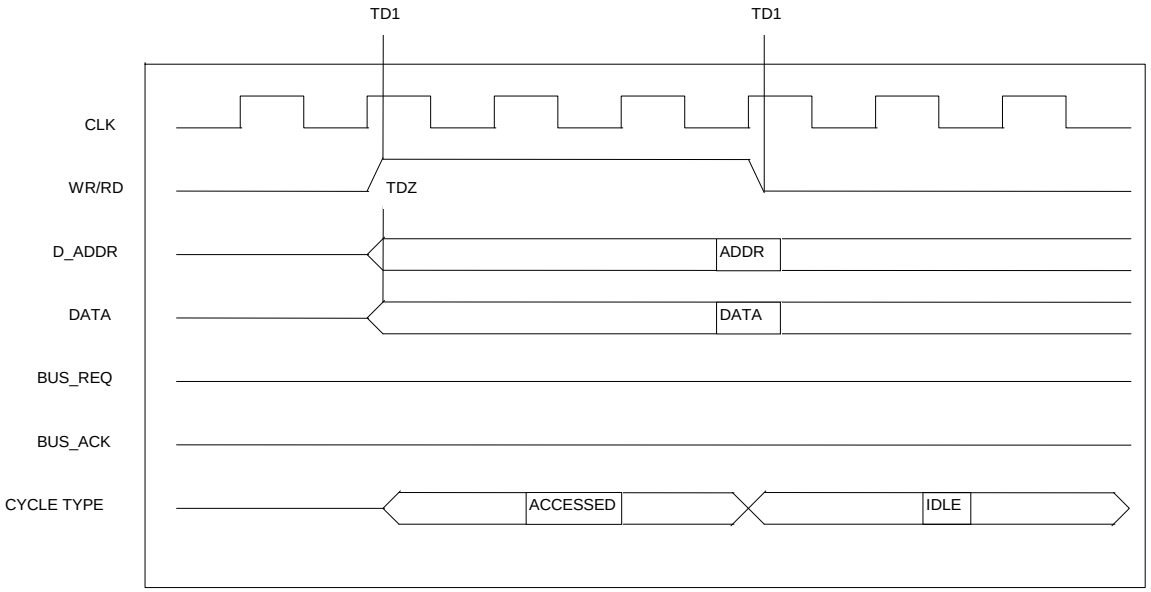

FIGURE T.7, DATA BUS STATE TRANSITION: ACCESSED => IDLE

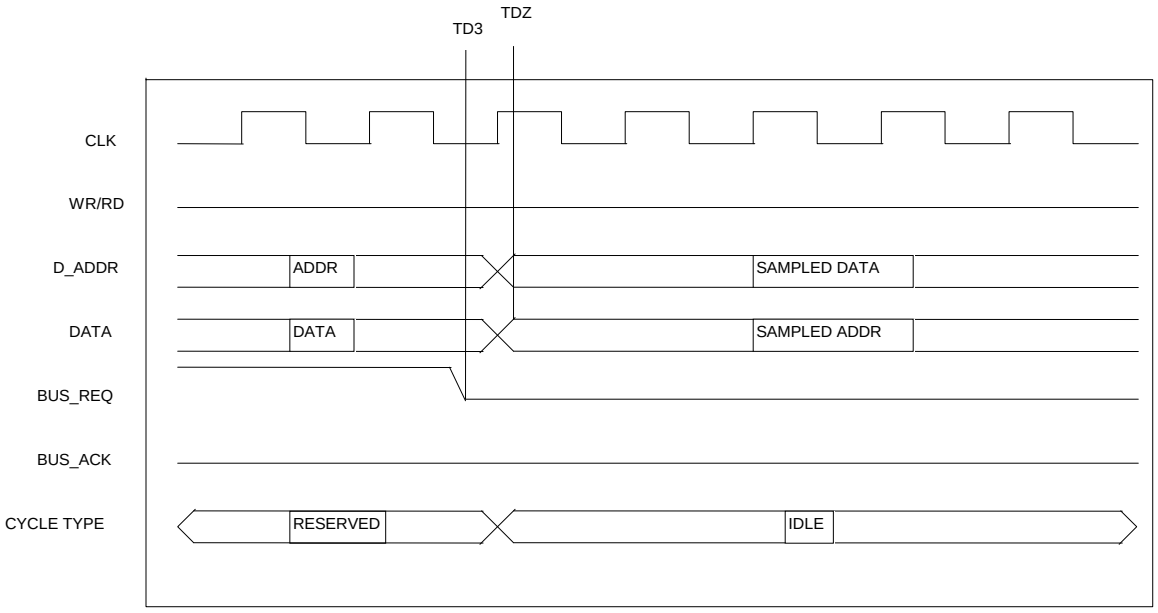

FIGURE T.6, DATA BUS STATE TRANSITION: RESERVED => IDLE

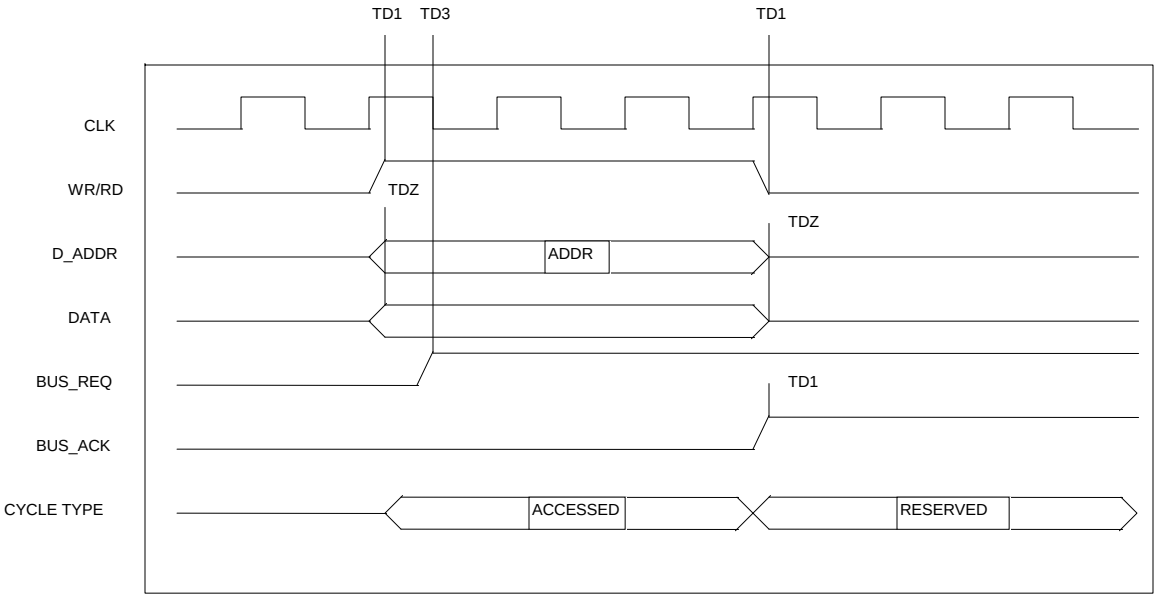

FIGURE T.8, DATA BUS STATE TRANSITION: ACCESSED => RESERVED

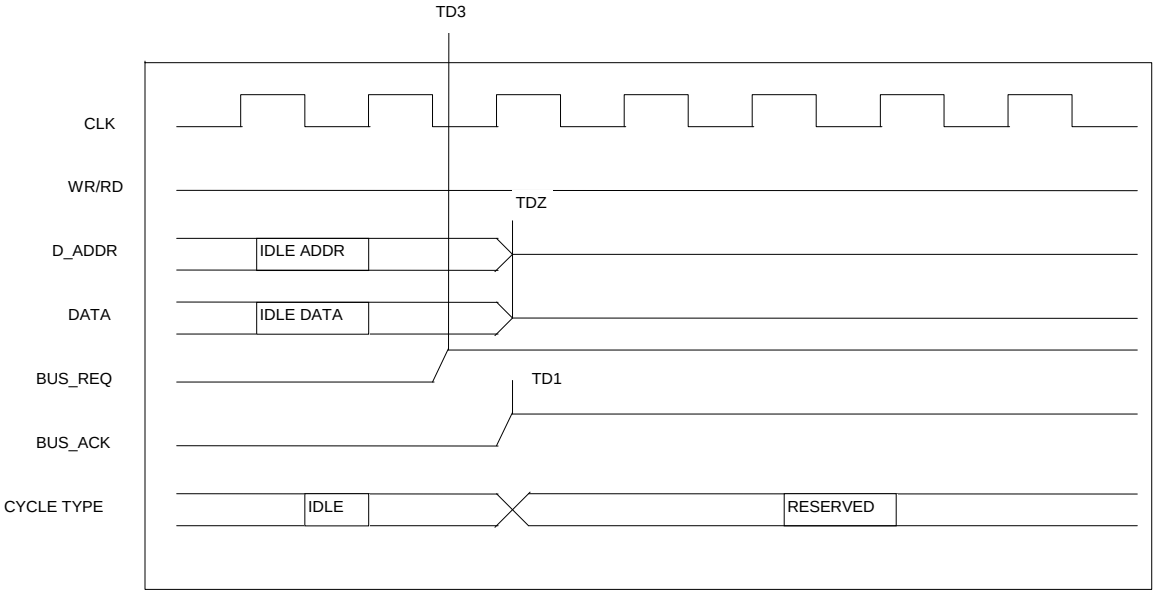

FIGURE T.9, DATA BUS STATE TRANSITION: IDLE => RESERVED

Table 1.1 Mnemonics and keywords used in figures T.5 through T.9

Keyword	Explanation	Notes
TD1	Delay from rising clock edge to the moment when data is valid on output of a D flip-flop.	Technology dependent.
TD3	Maximum delay of the *bus_req* –signal. Refer to results of preliminary synthesis.	Suitable synthesis constraints for *bus_req* –input must be set. See interface specification.
TDZ	TD1 + delay of a tri-state gate	See synthesis notes about handling tri-state control signals.
IDLE DATA	Data and address which are driven to bus by core when bus is in idle state. Should be the values from previous access unless the bus is floated during the last RESERVED –cycle.	
IDLE ADDR		
SAMPLED DATA		
SAMPLED ADDR		
ADDR	Valid address or data of an active access.	
DATA		
IDLE	Idle cycle of the bus, no active accesses. Core drives last values on bus.	
ACCESSED	Core owns the bus and is performing write or read access.	
RESERVED	An external device owns the bus.	

1.2. Interfacing coprocessors

Table 1.2 Coprocessor interfacing signals.

Signal	Direction from the core side	Purpose/description when active
COP_EXC[3..0]: COP_EXC(3) – COP 3 COP_EXC(2) – COP 2 COP_EXC(1) – COP 1 COP_EXC(0) – COP 0	In	Coprocessor exception. Coprocessor can interrupt the core by activating this signal.
COP_PORT(40): WR_COP	out	Write to cop. Write access to coprocessor register file.
COP_PORT(39): RD_COP	out	Read from cop. Read access to coprocessor register file.
COP_PORT[38..37]: C_INDX	out	Coprocessor index used to address one of the four possible coprocessors
COP_PORT[36..32]: R_INDX	out	Register index used to select the right register from the accessed coprocessor register bank.
COP_PORT[31..0]: DATA	inout	Data to/from the coprocessor.
stall	in	Freezes the whole core! This signal does not strictly speaking belong to coprocessor interface but can be used if no other solution is available.

2. Registers

2.1. General

COFFEE has two different register sets. 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's a total of 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.

Besides the register bank described here, COFFEE has another register bank, CCB (core control block), which is mapped to memory (accessed using load and store instructions). 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.

Table 2.1 Registers

SET 1			**SET 2**		
R0	GPR	32 bits	PR0	GPR	32 bits
R1	GPR	32 bits	PR1	GPR	32 bits
...			...		
R28	GPR	32 bits	PR28	GPR	32 bits
R29	GPR	32 bits	PR29	PSR	8 bits
R30	GPR	32 bits	PR30	SPSR	32 bits
R31	GPR/LR	32 bits	PR31	GPR/LR	32 bits

2.2. SET 1: General Purpose Registers

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 behavior is taken into account. All general purpose registers (and the link register) are 32 bits wide.

2.3. SET 2: General Purpose Registers

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 behavior is taken into account. All general purpose registers (and the link register) are 32 bits wide.

2.4. SET 2: Special Purpose Registers

There's two special purpose registers in SET 2: PSR and SPSR. PSR is eight bits 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 and contains the flags explained below. Bits 7 .. 5 are reserved for future extensions.

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

- 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 be also 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 .. 5 are writable but only bits 7 .. 0 are saved in case of *scall*.

2.5. Condition Registers

There are eight three 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 chapters ‘conditional execution’ and ‘instruction specifications’ for more information.

2.6. CCB registers

Note, that ‘byte’ addresses (that is consecutive addresses) are used in table below. 256 consecutive addresses are reserved for core configuration block. Addresses beyond CCB_BASE + FFh can be configured to point to an external peripheral configuration block (PCB), if present.

Registers which are shorter than 32 bits:

- LSB of a GPR corresponds to LSB of the short register in CCB.
- Unused bits read as zeros.
- For code compatibility with future versions, you should write unused bits as you would if there were more bits (interrupt masking, for example).

Core control block (CCB)

Offset	Mnemonic	Width	Description/usage	Notes
00h	CCB_BASE	32	The content 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_ ASE] + 255 map to CCB registers.	The base address has to be aligned to 256B boundary, that is, bits 7 .. 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
01h	PCB_BASE	32	The content of this register defines the first address of peripheral address space. All memory accesses in range [PCB_BASE] to [PCB_END] map to an external device(s) connected to data memory bus. Accesses to peripheral devices activate *pcb_rd* and *pcb_wr* signals instead of rd and wr signals.	See note 2 below
02h	PCB_END	32	The content of this register defines the last address of peripheral address space. All memory accesses in range [PCB_BASE] to [PCB_END] map to an external device(s) connected to data memory bus. Accesses to peripheral devices activate *pcb_rd* and *pcb_wr* signals instead of rd and wr signals.	
03h	PCB_AMASK	32	This register is used to define a mask for peripheral addresses. The address driven on address bus in constructed by masking the actual address with the contents of this register.	
04h	COP0_INT_VEC	32	Co-processor 0 interrupt service	See interrupts

			routine start address	
05h	COP1_INT_VEC	32	Co-processor 1 interrupt service routine start address	See interrupts
06h	COP2_INT_VEC	32	Co-processor 2 interrupt service routine start address	See interrupts
07h	COP3_INT_VEC	32	Co-processor 3 interrupt service routine start address	See interrupts
08h	EXT_INT0_VEC	32	External interrupt 0 service routine base address	See interrupts
09h	EXT_INT1_VEC	32	External interrupt 1 service routine base address	See interrupts
Ah	EXT_INT2_VEC	32	External interrupt 2 service routine base address	See interrupts
Bh	EXT_INT3_VEC	32	External interrupt 3 service routine base address	See interrupts
Ch	EXT_INT4_VEC	32	External interrupt 4 service routine base address	See interrupts
Dh	EXT_INT5_VEC	32	External interrupt 5 service routine base address	See interrupts
Eh	EXT_INT6_VEC	32	External interrupt 6 service routine base address	See interrupts
Fh	EXT_INT7_VEC	32	External interrupt 7 service routine base address	See interrupts
10h	INT_MODE_IL	12	The content 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 associations: See note 3 below. See interrupts and processor status register.

11h	INT_MODE_UM	12	The content 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.	
12h	INT_MASK	12	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’) allows requests to pass through.	
13h	INT_SERV	12	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 are 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.	Read only. See interrupts.

14h	INT_PEND	12	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.	
15h	EXT_INT_PRI	32	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. Bit field **PRIX** is associated with external interrupt number *X*. X varies from 0 to 7. Interrupt priorities: Bits 31 .. 28 : INT 7 priority Bits 27 .. 24 : INT 6 priority ... Bits 7 .. 4 : INT 1 priority Bits 3 .. 0 : INT 0 priority	0 – highest priority 15– lowest priority Priorities for external interrupts can only be set if external handler is not used.
16h	COP_INT_PRI	16	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. Bit field **PRIX** is associated with coprocessor number *X*. X varies from 0 to 3. Bits 15 .. 12 : COP3 priority Bits 11 .. 8 : COP2 priority Bits 7 .. 4 : COP1 priority Bits 3 .. 0 : COP0 priority	

17h	EXCEPTION_CS	8	This is a read-only register which is used to report the cause of an exception to an exception handler.	Read only. See exceptions.
18h	EXCEPTION_PC	32	This is a read-only register which is used to report the memory address of the instruction which caused an exception. It can be used by exception handler.	
19h	EXCEPTION_PSR	8	Contains a copy of processor status flags (PSR) which were valid when the instruction causing an exception was decoded. It can be used by exception handler.	
1Ah	DMEM_BOUND_LO	32	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.	The CCB block can be protected from user level code by mapping it to protected address space. See programming considerations.
1Bh	DMEM_BOUND_HI	32	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.	

1Ch	IMEM_BOUND_LO	32	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	
1Dh	IMEM_BOUND_HI	32	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.	

1Eh	MEM_CONF	2	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, and 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.	See programming considerations
1Fh	SYSTEM_ADDR	32	The content 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.	See instruction specifications: *scall*
20h	EXCEP_ADDR	32	The content 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.	See exceptions
21h	BUS_CONF	12	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	See COFFEE interface

			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). Bit fields: Bits 11 .. 8: CBUS_WC Bits 7 .. 4: DBUS_WC Bits 3 .. 0: IBUS_WC	
22h	COP_CONF	28	This register is used to configure the behavior 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	In COFFEE core version 1.0 only COFFEE native mode is supported (C*X*_IF fields are ignored)

			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. Bit fields Bits 27 .. 26: C3_IF Bits 25 .. 24: C2_IF Bits 23 .. 22: C1_IF Bits 21 .. 20: C0_IF Bits 19 .. 15: C3_IR Bits 14 .. 10: C2_IR Bits 9 .. 5 : C1_IR Bits 4 .. 0 : C0_IR	
23h	TMR0_CNT	32	This register contains the current value of the internal timer counter 0. Can be used to set initial value to counter 0.	See chapter about timers.
24h	TMR0_MAX_CNT	32	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.	
25h	TMR1_CNT	32	This register contains the current value of the internal timer counter 1. It can be used to set initial value to counter 1.	

26h	TMR1_MAX_CNT	32	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	
27h	TMR_CONF	32	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. Bits 31 .. 16 : TMR1_CONF Bits 15 .. 0: TMR0_CONF	
28h	RETI_ADDR	32	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*.	Interrupts should be disabled when writing to this register. See document about interrupts.
29h	RETI_PSR	8	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*.	Interrupts should be disabled when writing to this register. See document about interrupts.

2Ah	RETI_CR0	3	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.	Interrupts should be disabled when writing to this register. See document about interrupts.
2Bh	FPU_STATUS	32	A copy of the status flags sampled from floating point coprocessor (Milk).	See Milk documentation about status flags and reset values.
29.ffh	CORE_VER_ID	32	This read-only register contains version number of the processor core.	Reading from any of the unused offsets (2B...FE) returns the version identification number from this register.

[2] Address range ([CCB_BASE] + 100h) to [CCB_END] is used to access an external configuration block directly. This makes it possible to connect peripherals directly to data cache bus instead of system bus.

[3] Bit index and interrupt source associations:

bit	**source**	**bit**	**source**	**bit**	**source**
0	coprocessor 0 interrupt (exception)	4	ext interrupt 0	8	ext interrupt 4
1	coprocessor 1 interrupt (exception)	5	ext interrupt 1	9	ext interrupt 5
2	coprocessor 2 interrupt (exception)	6	ext interrupt 2	10	ext interrupt 6
3	coprocessor 3 interrupt (exception)	7	ext interrupt 3	11	ext interrupt 7

[4] Memory protection can be dynamically configured which is convenient in multitasking system. Most secure way is to set the limits always when switching task and to allow one task to access only address space reserved for it (data and instruction memory). If different tasks share global data(dangerous!) address spaces can overlap. In most cases communication between tasks should follow schemes offered by operating system. In simple systems only vital part of the the memory might be protected and the rest of the memory is ‘free’ to everyone. In both cases it is recommended that CCB is mapped to protected area!

2.7. 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 'instruction definitions' for details). Before passing the control, registers SPSR and PR31 must be set appropriately. Execution of *retu instruction* 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, 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 'instruction definitions'.

2.8. 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. Registers are mapped so that referring to register R0/PR0 in 16 bit mode means referring to register R24/PR24 in 32 bit mode and in general referring to Rx/PRx in 16 bit mode means referring to R(x+24)/PR(x+24) in 32 bit mode where x is an integer in the range 0...7. Of course, assembler should provide straight forward notion to access registers.

Condition registers C1...C7 are disabled in 16 bit mode. Register C0 is always used (automatically selected) with conditional branches and arithmetic.

2.9. Register values after reset

PSR start value is 0000 1110b. SPSR is set to 0000 0009h. The other registers in RF and CR are set to zero upon reset.

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

CCB (internal) register values after reset

Mnemonic	Reset Value	Notes
CCB_BASE	0001 0000h	64KB offset from the 'start'. Depending on the actual memory implementation, data and instruction cache may or may not point to the same physical memory.
PCB_BASE	0001 0100h	
PCB_END	0001 01FFh	
PCB_AMASK	0000 00FFh	Must be set if an external configuration block is present.
COP0_INT_VEC	0000 0001h	
COP1_INT_VEC	0000 0001h	
COP2_INT_VEC	0000 0001h	
COP3_INT_VEC	0000 0001h	
EXT_INT0_VEC	0000 0001h	
EXT_INT1_VEC	0000 0001h	
EXT_INT2_VEC	0000 0001h	
EXT_INT3_VEC	0000 0001h	
EXT_INT4_VEC	0000 0001h	
EXT_INT5_VEC	0000 0001h	
EXT_INT6_VEC	0000 0001h	
EXT_INT7_VEC	0000 0001h	
INT_MODE_IL	FFFh	32 bit mode for all routines
INT_MODE_UM	FFFh	Super user mode for all routines
INT_MASK	000h	All interrupts disabled
INT_SERV	000h	
INT_PEND	000h	
EXT_INT_PRI	0000 0000h	
COP_INT_PRI	0000h	
EXCEPTION_CS	00h	
EXCEPTION_PC	0000 0000h	
EXCEPTION_PSR	00h	
DMEM_BOUND_LO	0000 0000h	All the address space reserved for super user. Cannot run in user mode before configuring these register appropriately.
DMEM_BOUND_HI	FFFF FFFFh	
IMEM_BOUND_LO	0000 0000h	
IMEM_BOUND_HI	FFFF FFFFh	
MEM_CONF	3h	
SYSTEM_ADDR	00000000h	

EXCEP_ADDR	00000000h	
BUS_CONF	FFFh	
COP_CONF	000 0000h	
TMR0_CNT	0000 0000h	
TMR0_MAX_CNT	0000 0000h	
TMR1_CNT	0000 0000h	
TMR1_MAX_CNT	0000 0000h	
TMR_CONF	0000 0000h	
RETI_ADDR	0000 0001h	
RETI_PSR	09h	
RETI_CR0	0h	
FPU_STATUS	-	
CORE_VER_ID	Version dependent	

3. Timers

COFFEE core has two independent built-in timers. Both timers are 32 bit wide and both have separate 8 bit divisor. Timers can be configured as watchdog timers or timer tick generators for system. Timer registers are accessible via CCB (core configuration block) and can be configured using load and store instructions.

3.1. Timer registers

Table 3.1 Timer configuration and control registers

Register mnemonic	Bit field mnemonic	Bits	Explanation
TMR0_CNT		[31:0]	Current value of the timer0 counter. Can be set to arbitrary value.
TMR0_MAX_CNT		[31:0]	The maximum value of timer0 counter. Depending on CONT –bit, the timer will stop at maximum value or restart from zero. Note that, you can set a value greater than maximum count in TMR0_CNT –register in which case the timer counter will count to 0xffffffff and start over from zero.
TMR1_CNT		[31:0]	Current value of the timer1 counter. Can be set to arbitrary value.
TMR1_MAX_CNT		[31:0]	The maximum value of timer1 counter. Depending on CONT –bit, the timer will stop at maximum value or restart from zero. Note that, you can set a value greater than maximum count in

			TMR1_CNT –register in which case the timer counter will count to 0xffffffff and start over from zero.
TMR_CONF	TMR1_CONF	[31:16]	Configuration bits for timer1. See table 2 for bit field definitions.
	TMR0_CONF	[15:0]	Configuration bits for timer0. See table 2 for bit field definitions.

Table 3.2 Bit fields of configuration registers TMR1_CONF and TMR0_CONF

Bit field	**bits**	**explanation**
EN	31/15	EN = 1 enables timer. A timer can be stopped at any moment by writing EN = 0. Clearing EN bit will zero timer divider => timer will be incremented [DIV] + 1 clock cycles after enabling it.
CONT	30/14	CONT = 1: Continuous mode. Timer counter will start from zero after reaching maximum count defined in TMRx_MAX_CNT –register. CONT = 0: Timer counter will stop at maximum count.
GINT	29/13	GINT = 1: Generate an interrupt when maximum count is reached. GINT = 0: Do not generate interrupts.
WDOG	28/12	WDOG = 1: Enable watchdog function. If the timer reaches maximum count defined in TMRx_MAX_CNT the core will be reset.
-	27/11	Reserved, 0 or 1 can be written.
INTN	[26:24]/[10:8]	Bit field defining which interrupt to associate the timer with: “000” => EXT_INT0 ... “111” => EXT_INT7
DIV	[23:16]/ [7:0]	Divider value which defines how many clock cycles corresponds to one timer cycle: A timer counter will be incremented every [DIV] + 1 cycles, that is a zero value in DIV field sets the timer frequency to be the same as clock frequency of the core.

Note: The timer divider is useful when clock frequency is reduced in order to save power. Only the DIV field has to be touched in order to maintain timing.

4. Processor Operating Modes

4.1. 16 bit mode and 32 bit decoding modes

16 bit mode refers to length of the instruction word. When in this mode, core expects to get instruction words encoded in 16 bits. Mode can be switched on the fly using *swm*

instruction. Of course when running actual code, the encoding really has to change after *swm* instruction (See document instruction execution cycle times).

4.2. Limitations in 16 bit mode

- Only 8 registers per set available: registers 24...31 mapped as registers 0...7
- Conditional execution is not available
- Only one condition register (CR0) in use
- Immediate constants are shorter, see instruction specifications.
- Instructions lui, lli, exbfi and cop not available (available as pseudo –operations if supported by assembler).
- 2nd source register and destination register shared.

4.3. Super user mode

The core can operate in *super user* mode or *user* mode. In super user mode, core can access the whole memory space and both register banks. In user mode, access to protected memory areas (software configurable) is denied and only 1st register bank is accessible. It's possible to switch from super user mode to user mode but not vice versa, except using scall instruction which transfers execution to system code. System code entry address must be configured in startup code. Interrupt service routines can be run in both modes. This can also be configured by startup code. Core boots in super user mode, which makes possible to do the necessary configurations before starting application in user mode.

4.4. Resetting the processor

After powering up the core, *rst_x* pin should be pulsed low (clock has to be stable) to set the core in correct state. If boot address selection is enabled (*boot_sel* pin pulled high), boot address should be driven to data bus simultaneously with rst_x signal. If boot address selection is disabled, core will boot at address 0x00000000h. Normal operation will start two clock cycles after the rising edge of the rst_x signal. See document COFFEE interface about signal timing at reset.

Defaults after reset and boot procedure

Core will boot in super user and 32 bit modes. Interrupts are disabled. A typical boot procedure would be to execute assembly written boot code which sets all CCB registers to suitable values and switches to user mode by executing *retu* instruction. See instruction specifications. See register section about reset values of configuration registers.

4.5. Configuring the processor

Several features of the core can be configured via the core configuration block (CCB) which is a memory mapped register bank. When writing a new value to a configuration register, the new value will be valid when the instruction accessing CCB is in stage 5 of the pipeline. It follows that, if some configurations affect the execution of some instructions, or some configurations should be valid, when executing certain instructions, one has to make sure that there are enough instructions between the ones accessing CCB and dependent instructions. These can be *nop* instructions or other instructions which do not depend on values of the configuration registers. Table below shows few examples of situations where it is essential to have few instructions between a CCB write and an instruction depending on the configuration made. If you are not sure about the number of '*guard*' instructions, use four.

instruction	Purpose	notes	Dependency
st R1, R0, 0h	Remapping CCB to new address.	Assume that R0 contains the address of the CCB_BASE register and R1 contains a new address for CCB.	The 2nd store instruction needs the value of CCB_BASE in stage 3 of the pipeline. CCB_BASE is valid when the 1st store instruction is in stage 5 of the pipeline => There needs to be one instruction between the stores. In this case it is addi instruction.
addi R0, R1, 1h	It increments the new address of CCB. R0 should point now to CCB_END.	'guard' instruction	
st R2, R0, 0h	It configures the size of configuration block itself (internal + external blocks)	Assuming R2 contains an address to be written to CCB_END.	
st R1, R0, 0h	Set an interrupt vector.	Assume R0 contains address of EXT_INT0_VEC and R1 points to interrupt service routine.	Interrupt vector will be valid when store instruction has proceeded to stage 5 of the pipeline. Interrupts will be enabled when ei instruction reaches stage 2 of the
nop	idle instructions	Could use some other	

nop	(‘guard’ instructions)	‘useful’ instructions	pipeline. Need to fill stages 3 and 4 to be safe.
ei	Enable interrupts		
st R1, R0, 0h	Configure register translation for coprocessor access.	Assume R0 contains address of CREG_INDX_I and R1 valid configuration.	Configuration will be valid when store instruction has proceeded to stage 5 of the pipeline. Configuration is needed when cop instruction reaches stage 2 of the pipeline. Need to fill stages 3 and 4 to be safe.
nop	idle instructions (‘guard’ instructions)	Could use some other ‘useful’ instructions	
nop			
cop sqr(R2, R15)	Transfer an instruction word to coprocessor for execution		

5. Interrupts and exceptions

5.1. Interrupts

COFFEE core currently supports connecting eight external interrupt sources directly. If coprocessors are not connected the four inputs reserved for coprocessor exception signals can be used as interrupt request lines giving possibility to connect twelve sources. An external interrupt handler can be connected to expand the number of sources even further.

If internal interrupt handler is used, the priorities between sources can be set by software with external handler priorities fixed according to table below. Note that priorities for coprocessor exceptions/interrupts are always set by software.

Internal exception handler has synchronization circuitry allowing signals to be directly connected to the core. If an external handler is used, synchronization is bypassed in order to reduce signaling latency. See interface section.

Status signals are provided to give feedback about the status of the latest interrupt request. Interrupt sources can be masked individually, and disabled (or enabled) at once using di (or ei) instructions.

All interrupts are vectored. The address of an interrupt service routine can be the corresponding vector directly (see interrupt registers) or a combination of the vector and an offset given externally.

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 5.1 Interrupt priorities **if external handler is used**, 0 - highest

Interrupt interface modes

Two interfacing modes are supported: *external handler* and *internal handler*. The mode is selected by EXT_HANDLER signal. A summary is given in the table below. Note that an external handler usually allows priorities between sources to be set quite freely. In this case an external handler sees a fixed priority between the lines it is driving. The user may see whatever configuration.

Table 5.2 Interrupt interface modes

Mode/ EXT_HANDLER state	Request signal timing	Interrupt vector calculation	Priorities
internal handler / LOW	asynchronous	BASE address directly [1]	set by software (see configuration registers)
external handler / HIGH	synchronous	BASE(31 .. 12) & OFFSET & "0000" [2]	fixed between lines (usually configurable via external handler)

1 BASE address is set by software. See CCB configuration registers.

2 The 8 bit OFFSET provided by an external handler, & means concatenation. Coprocessor exceptions/interrupts do not use OFFSET.

Signaling an interrupt

An interrupt request is signaled by driving a high pulse on one of the interrupt lines. The timing of the pulse depends on the mode: whether an external handler is used or not. The timing of the coprocessor interrupt/exception lines is fixed. See interface section about the timing details. Each interrupt line has a pulse detection circuitry and an interrupt request gets through when that circuitry sees a pulse, that is, after seeing a falling edge. If an external handler is used, the offset should be driven simultaneously with the request line.

Once detected, a request is saved in a register called INT_PEND, which is visible to the software. After this it has to go through the priority resolving and masking stage. The following conditions have to be true for a pending request to get through:

- Interrupts enabled: IE bit in processor status register (PSR) must be high.
- Interrupt mask register has to have a high bit (‘1’) in the corresponding position.
- No interrupts with higher priority are pending or in service.
- No exceptions on pipeline (see document about exceptions)

Once a request gets through, the processor starts execution of an interrupt service routine as soon as possible: pipeline is executed to a point where it is safe to switch to interrupt service routine. This takes 1 – 3 cycles depending on the contents of the pipeline. When a service routine is started the corresponding bit in the INT_SERV register is set. At the same time, the processor drives a pulse to INT_ACK output in order to signal to an external handler that the latest request got through and is now in service. *This is the earliest point where a new request from the same source can be accepted.*

The latency from asserting an external interrupt signal to the moment when control of the core detects the signal is multiple cycles. The latency also depends on the mode of

operation of the interrupt interface. Latency is calculated from the falling edge of the EXT_INTERRUPT signal. Different cases are shown in the table below.

Table 5.3 Interrupt signals latency

Interface type	**signal synchronization**	**pulse detection**	**priority check and masking**	**total cycles**
asynchronous (EXT_HANDLER low)	2 clock cycles	1 clock cycle or less depending on timing of the EXT_INTERRUPT –signal.	1 clock cycle	4
synchronous (EXT_HANDLER high)	-		1 clock cycle	2

Priority resolving

A priority for a particular source is set by writing a four bit value in a field reserved for that source in the EXT_INT_PRI or COP_INT_PRI –register. Priority can have any value between 0 and 15, zero being the highest priority.

Whether the priority is fixed (external handler used) or set by software, priority resolving works the same way. If multiple interrupts are signaled simultaneously, the one with the highest priority (lowest number) will be served first. Note that for coprocessor exceptions/interrupts the priority 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.

If the same interrupt that is currently in service, is signaled, the interrupt service routine is restarted as soon as it has finished (of course assuming there’s no interrupt requests with higher priority pending). A request with higher priority can interrupt the current

service routine if interrupts have been re-enabled with *ei* instruction (nesting of interrupts).

Switching to an interrupt service routine

The following steps are taken when switching to an interrupt service routine:

- return address is saved to hardware stack (a special logic structure to allow fast switching)
- processor status register (PSR) is saved to hardware stack
- condition register CR0 is saved to hardware stack
- The start address of an interrupt service routine is calculated(see table 2) and placed to program counter
- Signal INT_ACK is pulsed (***except with coprocessor exceptions/interrupts!***)
- The bit corresponding to the interrupt source is set high in INT_SERV –register.
- The bit corresponding to the interrupt source is cleared from INT_PEND –register.
- Further interrupts are disabled by setting IE bit low in PSR
- Processor status: user mode and, instruction decoding are set according to control registers INT_MODE_IL and INT_MODE_UM. (If super user mode is set, register set 2 is selected as default for reading and writing)
- Execution of the interrupt service routine in question is started.

Returning from an interrupt service routine

An interrupt service routine *has to execute a reti* instruction in order to resume program execution where it was interrupted. This causes the following things to happen:

- Processor status is restored from the hardware stack
- CR0 is restored from the hardware stack
- Program counter is restored from the hardware stack
- Signal INT_DONE is pulsed (***except with coprocessor exceptions/interrupts!***)
- The INT_SERV bit is cleared.

- Interrupts are enabled if they were enabled before entering the service routine. (There is a possibility that di instruction is executed just before entering the service routine, but after a request got through in which case the interrupt is served but interrupts will be disabled on return)

Internal interrupt handler control & status registers

Bit positions and interrupt sources are associated as follows:
(INT_MODE_IL, INT_MODE_UM, INT_MASK, INT_SERV, INT_PEND)
Bit 11 – EXT_INT7, Bit 10 – EXT_INT6,..., Bit 4 – EXT_INT0, Bit 3 – COP3_INT,..., Bit 0 – COP0_INT

Table 5.4 Internal interrupt handler registers (in CCB)

Offset	Mnemonic	Width	Description	Notes
02h	COP0_INT_VEC	32	Co-processor 0 interrupt service routine start address	It should be properly aligned.
03h	COP1_INT_VEC	32	Co-processor 1 interrupt service routine start address	
04h	COP2_INT_VEC	32	Co-processor 2 interrupt service routine start address	
05h	COP3_INT_VEC	32	Co-processor 3 interrupt service routine start address	
06h	EXT_INT0_VEC	32	External interrupt 0 service routine base address.	
07h	EXT_INT1_VEC	32	External interrupt 1 service routine base address.	
08h	EXT_INT2_VEC	32	External interrupt 2 service routine base address.	
09h	EXT_INT3_VEC	32	External interrupt 3 service routine base address.	
0ah	EXT_INT4_VEC	32	External interrupt 4 service routine base address.	
0bh	EXT_INT5_VEC	32	External interrupt 5 service routine base address.	
0ch	EXT_INT6_VEC	32	External interrupt 6 service routine base address.	
0dh	EXT_INT7_VEC	32	External interrupt 7 service routine base address.	
0eh	INT_MODE_IL	12	Instruction decoding mode flags for interrupt routines.	See registers – document: PSR
0fh	INT_MODE_UM	12	User mode flags for interrupt routines.	
10h	INT_MASK	12	Register for masking external and cop interrupts individually. A low bit (‘0’) means blocking an interrupt source; a high bit enables an interrupt.	
11h	INT_SERV	12	Interrupt service status bits (active high).	Read only. See chapter Tricks.
12h	INT_PEND	12	Pending interrupt requests (active high).	
13h	EXT_INT_PRI	32	Bits 31 .. 28 : INT 7 priority Bits 27 .. 24 : INT 6 priority ... Bits 7 .. 4 : INT 1 priority Bits 3 .. 0 : INT 0 priority	0 – highest priority 15 – lowest priority Priorities for external interrupts can only be set if internal handler is used.
14h	COP_INT_PRI	16	Bits 15 .. 12 : COP3 priority Bits 11 .. 8 : COP2 priority Bits 7 .. 4 : COP1 priority Bits 3 .. 0 : COP0 priority	

Notes

- *di* instruction itself can be interrupted! It is guaranteed that instructions between **di, ei** pair cannot be interrupted but an interrupt can take place between di and the following instruction.

- **Clearing a pending interrupt without running the service routine; *this thing concern only clearing a pending request from the internal handler register!*** The ability to clear bits in the INT_PEND register directly would lead to situations where an external interrupt handler would not know the real status of the latest interrupt request because INT_ACK -signal would never go high for these 'canceled' interrupts. This kind of inconsistency is not acceptable and that's why INT_PEND is a read only register.

If there is a need to 'cancel' a request it can be done as follows (If internal CCB is mapped to protected memory area, super user mode is needed):

- Interrupts should be disabled during these operations!
- Save the current value in the interrupt vector register of the int source in question.
- Replace the old vector with a new one which points to a dummy routine (remember OFFSET, if external handler is present) which executes reti – instruction only (and maybe some acknowledge instructions for external handler).
- Set the interrupt source to highest priority and make sure that no other source shares the same priority (of course save old values).
- Set mask bit for the interrupt source in question (save old value of INT_MASK)
- enable interrupts
- Check the INT_PEND register and disable interrupts when the bit in question is low.
- Restore vector and priorities.
- Continue normally

Do not do this!

Do not change interrupt priorities while in interrupt service routine if you use nested interrupts (unless you are 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.

5.2. Exceptions

In this document an *exception* means an event which will halt the processing of the current thread immediately and causes the core to switch to an exception handling routine. An exception is considered an error condition and has to be dealt with immediately. Note that very often in literature exception means interrupting the processor in general. See also interrupts section.

Table 5.5 Exception types and codes.

Priority	Code	Name	Description
10	00000000	instruction address violation[3]	While in user mode, instruction is fetched from memory address not allowed for user.
6	00000001	unknown opcode	Version 1.0 of COFFEE RISC does not have any unused opcodes which makes this obsolete.
7	00000010	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 super user only instruction in user mode.
3	00000011	miss aligned jump address [4]	Calculated jump target is not aligned to word (32 bit mode) or halfword(16 bit mode) boundary.
2	00000100	jump address overflow	A PC relative jump below the bottom of the memory or above the top of the memory.
9	00000101	miss aligned instruction address[1]	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 - Exception handler entry address is not aligned to word boundary (this will lock the core by causing an eternal loop!) - System entry address is not aligned to word boundary
8	111xxxxx	trap [2]	processor encountered a trap instruction
5	00000110	arithmetic overflow	The result of a signed arithmetic operation exceeds 2^{31}-1 or falls below -2^{31}.
0	00000111	data address violation	While in user mode, a data address refers to memory address not allowed for user.

1	00001000	data address overflow	Trying to index data below of the bottom or above of the top of the memory
4	00001001	Illegal jump	Trying to jump to protected instruction memory area while in user -mode.
x	00001010 ... 00011111		Reserved for future extensions

Notes

[1] In this case, the address is saved, since it cannot be known which instruction(if any) caused the exception.
[2] 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.
[3] If sequential execution traverses the boundary of the protected instruction memory area, the address of the instruction pointed to is saved.
[4] A jump between memory areas using different encoding will result in unpredictable behavior.

Handling an exception

In case of an exception, core performs following tasks:

- Saves the address of the instruction causing the exception (or just an address, see table on previous page) to CCB register EXCEPTION_PC.
- Saves to CCB register EXCEPTION_PSR processor status flags which were used when the violating instruction was decoded.
- Saves the exception code (see table above) to CCB register EXCEPTION_CS.
- Disables interrupts.
- Switches to 32 bit decoding mode and super user mode with register set 2 as default for reading and writing.
- Starts execution from a handler routine pointed by the CCB register EXCEP_ADDR.

Following things are guaranteed by hardware:

- The violating instruction is not able to modify the state of the processor (registers, status flags, data memory).
- All instructions before the violating one (in the order of execution) are executed.
- None of the instruction following the violating one are executed (pipeline is flushed up to the violating instruction).

- If multiple instructions on pipeline cause an exception simultaneously, the one which is first in the order of execution is taken into account.
- Interrupt requests cannot get through if an exception is signaled.
- An exception handler routine will always see updated values of EXCEPTION_XX registers immediately.

Returning from the exception handler

Depending on the handler, execution can be resumed from a different context or from the same context or it might not be resumed at all. In any case, appropriate flags should be written to SPSR (see registers) and the resume address should be written to PR31 (the link register). Then, executing retu instruction will update the PSR with flags written to SPSR and load the program counter with the value in PR31 causing the processor to start executing instructions from the desired memory location in the desired mode.

Notes

- Remember to initialize EXCEP_ADDR register appropriately in boot code. Incorrect address may cause eternal loop which will lock the processor until it is reset.
- Even though the violating instruction cannot change the contents of the memory, the address it refers to may appear on address bus.
- Interrupts are disabled when entering the handler routine but can be enabled by software (care must be taken).
- If the exception is caused by an interrupt service routine (see interrupts) and the routine is disabled permanently, you should pop the return address of that routine from the hardware stack to ensure correct operation of other interrupt routines. This is explained in the document interrupts.
- Exceptions are an inefficient way to interface super user mode. Use scall instruction instead of trap instruction where appropriate.

5.3. Handling exceptions and interrupts

This document describes what happens on pipeline in case of an exception or interrupt or a combination of these.

Definitions

An interrupt

An external/internal device requests CPU time. This is the normal way to interrupt the processor in a multitasking system. Interrupt request can originate for example from a timer or an external IO –device, coprocessor etc.

An exception

An instruction causes a violation. An exception is considered to be abnormal condition. Software originated exceptions can be synthesized using trap –instruction.

General philosophy

From the hardware point of view, exceptions require immediate attention and actions cannot be delayed even one clock cycle. This is because hardware has to make sure that the instruction causing the exception does not modify the state of the processor: flags, register or memory contents. If it does, it will most probably cause following instructions to fail also.

From software point of view, the processing of an exception in one thread can be delayed if some other thread currently needs CPU time. It is enough to halt the thread where the exception occurred and invoke a handler routine whenever the execution of the violating thread should continue. Information about exception is passed to the handler.

From software point of view, interrupts should be serviced immediately. Especially systems which have strict real time requirements do not allow servicing to be delayed too much. Anyway ‘immediately’ has a slightly different meaning for software than for hardware: Typically switching to an interrupt routine means executing many instructions before the actual processing of the interrupt event (anything up to hundreds of instructions). In hardware, switching takes typically less than five clock cycles!

From hardware point of view, interrupts are not an error condition and as such do not require immediate attention if something with higher priority is processed. In all cases COFFEE core will pass control to an interrupt service routine as soon as possible. In

practice the interrupt response time will be predictable. In fact response will be delayed only in these cases: cache miss causes stall cycles, interrupts are disabled by software, an interrupt with a higher priority is in service or an exception occurs simultaneously with detecting an interrupt request or during a context switch.

Processing of interrupts

Signaling an interrupt

The latency from asserting an external interrupt signal to the moment when control of the core detects the signal is multiple cycles. The latency also depends on the mode of operation of the interrupt interface. Latency is calculated from the falling edge of the EXT_INTERRUPT signal. Latency from signaling to context switch in different cases is shown in the table below. After edge detection stage, the request is saved in PEND register for further processing. If interrupts are disabled, a request will be pending until interrupts are again enabled. As soon as the core acknowledges the pending request it will be visible in the SERV –register after which a new request from the same source can be accepted.

Table 5.6, Interrupt signaling latency

mode	**signal synchronization**	**edge detection**	**priority check and masking**	**total cycles**
asynchronous (EXT_HANDLER low)	2 clock cycles	1 clock cycle or less depending on timing of the EXT_INTERRUPT –signal.	1 clock cycle	4
synchronous (EXT_HANDLER high)	-		1 clock cycle	2

Deciding return address

Table 5.7, deciding the return address

case	explanation	return address/source	notes
0	One of the following instructions in stage 1:	Calculated jump target address if the branch is taken or the address of the	All of the listed instructions cause

	bc, bnc, begt, belt, beq, bgt, blt, bne, jal, jalr, jmp, jmpr, retu or scall.	instruction following branch slot instruction if branch is not taken.	execution to branch somewhere. Slot instruction is executed.
1	swm -instruction in stage 1 or 2	Address of the instruction following the two required nop –instructions.	There has to be two nop instructions after a swm.
2	mulu, muli, muls or mulus in stage 1	Address instruction itself.	One of these and a following mulhi –instruction is atomic. => Cannot be executed separately.
3	**reti** –instruction in stage 1, 2 or 3	Address on top of the hardware stack.	In practise this means that, an interrupt service routine is interrupted by a higher priority request (nested interrupts)
4	All other cases	Address of the instruction being fetched that is the current PC value.	

Notes:

- The return address will be written to PC before saving it to hardware stack, which means that it will be visible to the instruction cache even though the instruction pointed to is not executed. The only exception to this is case 3 in the above table: If the needed return address is already on top of the stack, it is not popped to PC.

- If an exception happens during context switching, it takes priority and the interrupt request is left pending.

Switching to an interrupt routine

Switching to an interrupt service routine takes multiple clock cycles. The number of clock cycles depends on the contents of the pipeline and possible stalls caused by cache memory misses or data dependencies.

The total amount of cycles from the falling edge of the EXT_INTERRUPT/COP_EXC – signal to the moment when the address of the first instruction of an interrupt service routine is on the I_ADDR –bus *is from 3 to n cycles.* N depends on pipeline stalls and contents and the interrupt status of the processor.

Before switching to an interrupt service routine, instructions already on pipeline are executed to ‘safe’ state. See *Table 8, Instructions and their safe states* in document *‘Instruction execution cycle times’*.

Figure below illustrates context switching logic for both exceptions and interrupts.

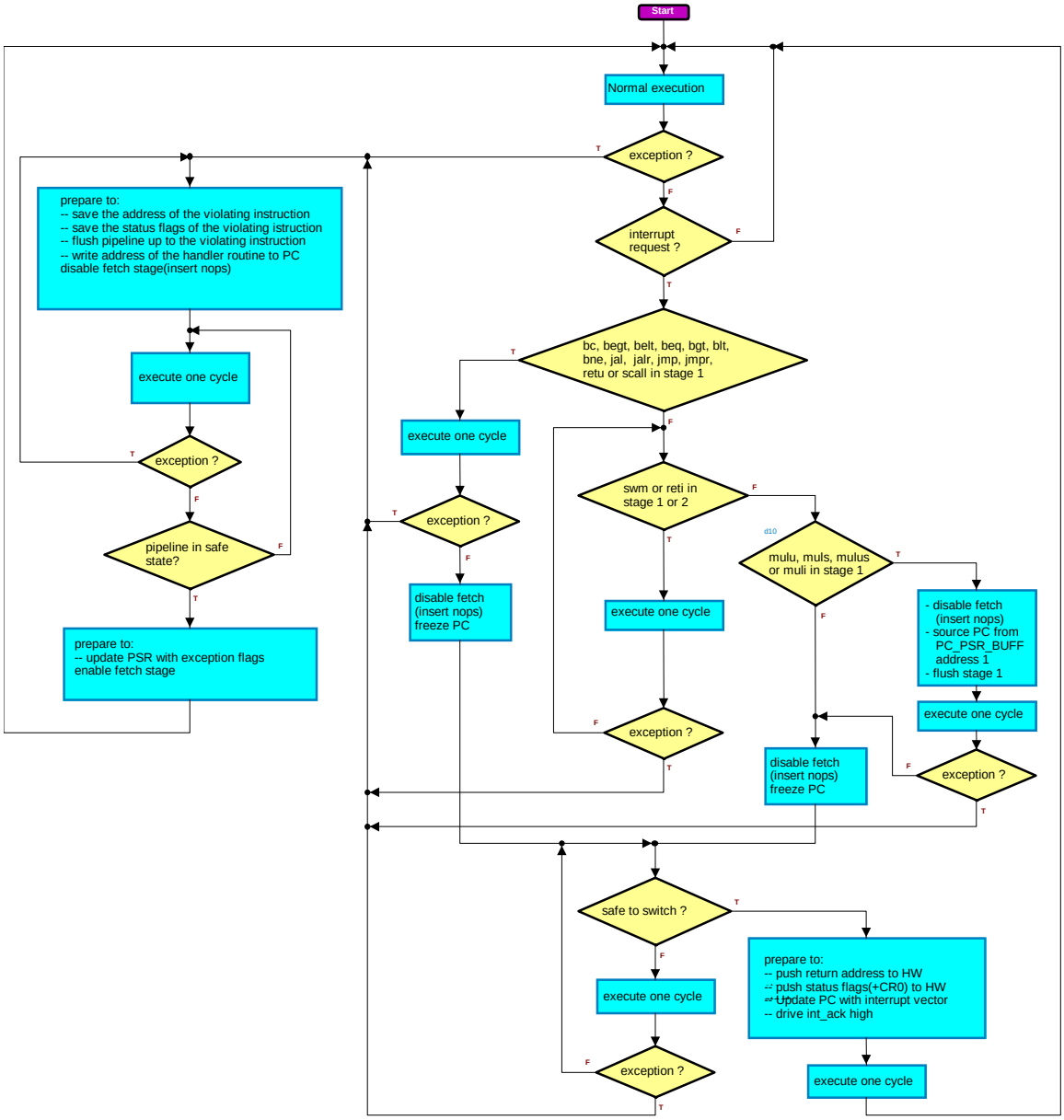

Figure 5.1, Interrupt & exception logic

Returning from an interrupt service routine

Safe return is guaranteed by executing reti instruction in stage 2 instead of stage 1. When reti is in stage 2, it can be seen if preceding instructions will cause exceptions. If not, context can be safely restored. Return address will be on memory bus when reti is in stage 4 of the pipeline.

Processing of exceptions

Table 5.8 Exception types and codes

Priority	Code	Name	Description
10	00000000	instruction address violation [3]	While in user mode, instruction is fetched from memory address not allowed for user.
6	00000001	unknown opcode	Version 1.0 of COFFEE RISC does not have any unused opcodes which makes this obsolete.
7	00000010	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 super user only instruction in user mode.
3	00000011	miss aligned jump address [4]	Calculated jump target is not aligned to word (32 bit mode) or halfword(16 bit mode) boundary.
2	00000100	jump address overflow	A PC relative jump below the bottom of the memory or above the top of the memory.
9	00000101	miss aligned instruction address [1]	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 - Exception handler entry address is not aligned to word boundary (this will lock the core by causing an eternal loop!) - System entry address is not aligned to word boundary
8	111xxxxx	trap [2]	processor encountered a trap instruction
5	00000110	arithmetic overflow	The result of a signed arithmetic operation exceeds 2^{31}-1 or falls below -2^{31}.
0	00000111	data address violation	While in user mode, a data address refers to memory address not allowed for user.
1	00001000	data address overflow	Trying to index data below of the bottom or above of the top of the memory
4	00001001	Illegal jump	Trying to jump to protected instruction memory area while in user -mode.
x	00001010 ... 00011111		Reserved for future extensions

Table 5.9 Exception signaling stages

name	violating instruction in stage
unknown opcode	2

Illegal instruction	2
miss aligned jump address	3
jump address overflow	
Illegal jump	
instruction address violation	1
miss aligned instruction address	
trap	2
arithmetic overflow	3
data address violation	4
data address overflow	4

Priorities

Priority 0 means most urgent and 10 means the lowest priority. Priorities come directly from the order of execution. When two or more instructions cause exception in different parts of the pipeline, the one with the highest priority is taken into account.

Switching to exception handler routine

The offending instruction and all following instructions in the pipeline (instructions which follow the violating one in the order of execution) are flushed. The address of the violating instruction is saved along with status flags (PSR), which were valid when decoding the instruction. Also a cause code is saved. See CCB registers. The remaining instructions on the pipeline (instructions which precede the violating one in order of execution) are executed until the pipeline is in safe state, which means that no more exceptions can take place(and processor state does not change). New instructions are not fetched during this pipeline clean operation. If during pipeline clean another exception occurs, the pipeline is flushed up to that instruction and exception data corresponding to the violating instruction is saved (in EXCEPTION_CS, EXCEPTION_PC and EXCEPTION_PSR). After this the cleaning of pipeline will continue until it's safe to switch to the exception handler routine.

When the pipeline is clean, PSR will be updated with default handler flags shown below and execution from address defined in CCB register EXCEP_ADDR is started.

RESERVED	IE	IL	RSWR	RSRD	UM
xxx	0	1	1	1	0

This kind of operation guarantees that an exception is always caught and instructions which preceded the violating one are executed properly. Instructions which follow the violating one are not executed.

Offending instructions are not able to modify the state of the processor or contents of the memory or registers. Note that Exception data registers inside CCB (EXCEPTION_CS, EXCEPTION_PC and EXCEPTION_PSR) will be overwritten immediately. If an exception happens in an exception handler routine (little hope for the software to recover!) the handler routine is restarted and the link to the original context might be lost depending on the handler routine.

Figure 1 (previous chapter) illustrates the exception logic.

6. Instruction specifications

6.1. General Information

This document describes the machine instructions implemented in COFFEE RISC 1.
The following set of instructions is the minimum set which every assembler should provide. With pseudo instructions the assembly language interface can be extended.

Abbreviations used

- *creg* : condition register index, number in the range 0...7
- *cond* : condition (see table 'Condition codes' at the end)
- *dreg* : destination register index (32 bit mode) , number in the range 0...31
- *sreg*i, *sreg* : source register index (32 bit mode) , number in the range 0...31
- *dr* : destination register index (16 bit mode) , number in the range 0...7
- *sr*i : source register index (16 bit mode) , number in the range 0...7

- *imm, imm1, imm2* : immediate constant, see table ‘Permitted values for immediate constants’
- *cp_sreg* : coprocessor source register , number in the range 0...31
- *cp_dreg* : coprocessor destination register , number in the range 0...31

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.

Syntax definition is an abstraction. The only purpose is to illustrate what an instruction expects as input and produces as output. The **Syntax** of an assembly language program written for COFFEE RISC depends on the assembler and is documented in the respective assembler manual.

If the **Syntax** of an instruction is different in 16 bit mode than in 32 bit mode then both **Syntax**es are presented: First the 32 bit version and then 16 bit version separated with a backslash. If both **Syntax**es 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.

6.2. Instruction definitions

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 generated 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 arithmetics.

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 generated 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 arithmetics. See the permitted values for the immediate in the table 'Permitted values for immediate constants'.

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 'Permitted values for immediate constants'.

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.

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 the table 'Permitted values for immediate constants'.

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 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 the table 'Permitted values for immediate constants'.

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 the table 'Permitted values for immediate constants'.

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 the table 'Permitted values for immediate constants'.

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 instrcution 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 the table 'Permitted values for immediate constants'.

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 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 the table 'Permitted values for immediate constants'.

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 the table 'Permitted values for immediate constants'.

bne

Syntax: bne creg, imm/bne imm

Description: If the flags in the condition register *creg* indicate that the condition ‘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 the table 'Permitted values for immediate constants'.

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 the table 'Permitted values for immediate constants'.

chrs

Syntax: chrs imm

Description: Specifies which register set is used for reading or writing. The source register(s) and the destination register doesn't have to reside in the same set. The register sets to be used are specified by the immediate *imm* according to the following table:

imm	write	read
0 (00b)	set 1 (user set)	set 1 (user set)

1 (01b)	set 1 (user set)	set 2 (super user set)
2 (10b)	Set 2 (super user set)	set 1 (user set)
3 (11b)	Set 2 (super user set)	set 2 (super user set)

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 generated 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.

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 contitionally.

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 allways 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 the table 'Permitted values for immediate constants'.

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 the different order (right to left) of operands in 16 bit version. Assembler should provide consistent notation (this is not an assembler specification).

conh

Syntax: (cond, creg) conh dreg, sreg2, sreg1/conh dr, sr

Description: Concatenates the least signicant 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.

cop

Syntax: cop imm1, imm2 (Coprocessor Operation)

Description: Moves the immediate *imm2* (instruction word of the coprocessor in question) to coprocessor number *imm1*. The immediate *imm1* specifies one of four possible coprocessors with values 0, 1, 2 or 3. The length of the *imm2* is 24 bits.

Notes: Can be used only in 32 bit mode. This instruction cannot be executed conditionally. See coprocessor interface. See core control block (CCB) registers about register index translation.

di

Syntax: di

Description: Disables maskable interrupts.

Notes: Not permitted to be executed conditionally. An exception is generated on an attempt to use this command in user mode. See 'Interrupts and exceptions' for definitions and details.

ei

Syntax: ei

Description: Enables maskable interrupts.

Notes: Not permitted to be executed conditionally. An exception is generated on an attempt to use this command in user mode. See 'Interrupts and exceptions' for definitions and details

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 following table.

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

imm	byte
0	byte0
1	byte1
2	byte2
3	byte3

Notes: See table permitted values for immediates.

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 .. 5

define the length of the field and bits 4 .. 0 define the LSB position. In the 16 bit mode *dr* is the second source and the destination.

Notes: Examples

Suppose that the bitfield shown below should be extracted from register R0 (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

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:

```
slli    R1, R1, 5 /* shift the length to bits 10 .. 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 table permitted values for immediate.

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.

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 the table ‘Permitted values for immediate constants’.

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) which 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 the table ‘Permitted values for immediate constants’.

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) which 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.

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 exb instruction. See the permitted values for the immediate in the table ‘Permitted values for immediate constants’.

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 the table Permitted values for immediate constants.

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 the table ‘Permitted values for immediate constants’.

mov

Syntax: (cond, creg) mov dreg, sreg1

Description: Copies the contents of the source register *sreg1/sr* to the destination register *dreg/dr*.

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_sreg* is an index to the coprocessor register file.

Notes: See ‘Coprocessor Interface’.

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.

Notes: See 'Coprocessor Interface'.

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 bits. See the permitted values for the immediate in the table 'Permitted values for immediate constants'.

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 bits.

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 bits.

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 bits.

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.

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).

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 the table 'Permitted values for immediate constants'.

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. See programming hints.

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: See programming hints. Not allowed to be executed conditionally. Reti instruction has to be followed by three nops!

retu

Syntax: retu

Description: Used for returning or moving from system code/super user 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). It is available only in super user mode.

Notes: See scall. See programming hints. Not allowed to be executed conditionally. The instruction following retu always has to be a nop!

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 super user code the default settings are 32 bit mode, interrupts disabled and super user 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.

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. See programming hints.

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 the table 'Permitted values for immediate constants'.

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* implies 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 the table 'Permitted values for immediate constants'.

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* implies 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 the table 'Permitted values for immediate constants'.

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* implies 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 the table 'Permitted values for immediate constants'.

st

Syntax: (cond, creg) st sreg2, sreg1, imm

Description: Stores the data in the source register *sreg2/sr2* to memory location whos 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 the table 'Permitted values for immediate constants'.

sub

Syntax: (cond, creg) sub dreg, sreg1, sreg2/sub dr, sr

Description: The content 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 generated 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 arithmetics

Flags: Z, C, N

subu

Syntax: (cond, creg) subu dreg, sreg1, sreg2/subu dr, sr

Description: The content 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. See programming hints

swm

Syntax: swm imm

Description: Changes the instruction decoding mode. The value of the immediate *imm* specifies the mode: *imm* = 16 => switch to 16bit 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 the table 'Permitted values for immediate constants'. This instruction has to be followed by two nop -instructions!

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 EXCEPTION_PC register and the exception cause code in exception cause register (EXCEPTION_CS). The exception cause code is composed of the given immediate and fixed offset in order to make the exception code unique. Therefore it is not possible to generate hardware exceptions by issuing trap instruction. The value of the PSR which was valid when decoding trap instruction is saved in register EXCEPTION_PSR.

Notes: See document 'exceptions' about the exception cause code.

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*.

Table 6.1 Permitted values for immediate constants

Instruction	**Permitted values for** *imm*			**Notes**
	16-bit	**32-bit**		
		Conditional	**Unconditional**	
addi	$-2^6 \dots 2^6-1$	$-2^8 \dots 2^8-1$	$-2^{14} \dots 2^{14}-1$	
addiu	$0 \dots 2^7-1$	$0 \dots 2^9-1$	$0 \dots 2^{15}-1$	
andi	$0 \dots 2^7-1$	$0 \dots 2^9-1$	$0 \dots 2^{15}-1$	
bxx [1]	$-2^9 \dots 2^9-1$	-	$-2^{21} \dots 2^{21}-1$	Should be even in 32bit mode
chrs	0...3	-	0..3	
cmpi	$-2^6 \dots 2^6-1$	-	$-2^{16} \dots 2^{16}-1$	
cop	-	-	imm1: 0...3	Only 32 bit mode
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	$-2^9 \dots 2^9-1$	-	$-2^{24} \dots 2^{24}-1$	Should be even in 32bit mode
jmp	$-2^9 \dots 2^9-1$	-	$-2^{24} \dots 2^{24}-1$	Should be even in 32bit mode
ld	-8...7	$-2^8 \dots 2^8-1$	$-2^{14} \dots 2^{14}-1$	
lli	-	-	$0 \dots 2^{16}-1$ (or $-2^{15} \dots 2^{15}-1$)	Only 32 bit mode
lui	-	-	$0 \dots 2^{16}-1$ (or $-2^{15} \dots 2^{15}-1$)	Only 32 bit mode
movfc	0...3	0..3	0..3	
movtc	0...3	0..3	0..3	
muli	$-2^6 \dots 2^6-1$	$-2^8 \dots 2^8-1$	$-2^{14} \dots 2^{14}-1$	
ori	$0 \dots 2^7-1$	$0 \dots 2^9-1$	$0 \dots 2^{15}-1$	
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	$-2^8 \dots 2^8-1$	$-2^{14} \dots 2^{14}-1$	
swm[2] trap	16 or 32 0...31	- -	16 or 32 0...31	

[1] xx is one of the following: *c*, *nc*, *lt*, *ne*, *gt*, *eq*, *egt* or *elt*

2 Future extensions may allow other values.

3 Values up to 63 can fit to imm1, but only the ones specified are used by hardware.

Table xx, Condition codes

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

Notes
Instructions which cannot be put into branch slot are: retu, reti, scall, swm or another jump/branch

6.3. Instruction execution cycle times

General

Address or data available in stage X means that it has been calculated during the previous cycle(s) and can be used as input to stage X. In all cases data will be written to register file during stage 5.

In stage X means that the instruction in question has propagated to stage X even though the instruction might not be ‘active’ anymore, that is, it does not change the state of the registers nor outputs of the core.
For example all jumps basicly evaluate an address in stage 1 which is availabe in stage 2. Some of them save a link address, so they are active until write back stage.

Cycle times below are for the ideal case of zero pipeline stall cycles. Pipeline stalls are mainly caused by cache memory misses and data dependencies. In ideal conditions the throughput of pipeline is one instruction per cycle.

Other instructions on the pipeline can use the data/**Flags** as soon as it’s ready (column 5 in table 2).

Table 6.2 Stage definitions

n	Operations	Notes
0	- instruction address increment - current instruction address check (calculated previously) - Instruction fetch (from the current address).	
1	- 16bit to 32bit instruction extending - immediate operand extending - jump address calculation - decoding for control 1(CCU) - operand forwarding (ALU operands) - register operand fetch & operand selection - Execution conditions check (jumps and others). Includes condition register bank read. - evaluation of new status flags (PSR) - instruction check (unused opcodes, mode dependent instructions)	Execution condition and branch condition is checked (not evaluated) in stage 1!
2	- coprocessor operand selection - forwarding of data latched from memory bus - ALU execution, step 1 - address calculation for data memory access - flag evaluation (Z, N, C)	
3	- coprocessor access - condition register bank write (with scon, read) - ALU execution, step 2 - Data memory addresses checks: user, CCB and overflow. - data forwarding for memory access (st – instruction only)	CR has internal forwarding. Flags calculated in the previous cycle can be seen directly on output of CR if needed. Special output for scon –instruction, all_out, does not have forwarding.
4	- cor control block (CCB)access - data memory access - ALU execution, step 3	
5	- register write back	Note that register file RF has internal forward control which means that data calculated during stage 4 is visible directly to stage 1 if needed

Instruction timing

Table 6.3 Instruction cycle timing

Instruction	ALU cycles	latency from instruction fetch to data available	Address on bus	Address check complete	Data	Condition Flags	PSR Flags
	cycle count		ready/available in stage				
add	1	3	-	-	3	3	-
addi	1	3	-	-	3	3	-
addiu	1	3	-	-	3	3	-
addu	1	3	-	-	3	3	-
and	1	3	-	-	3	-	-
andi	1	3	-	-	3	-	-
bnc	0	-	2	3	-	-	-
bc	0	-	2	3	-	-	-
begt	0	-	2	3	-	-	-
belt	0	-	2	3	-	-	-
beq	0	-	2	3	-	-	-
bgt	0	-	2	3	-	-	-
blt	0	-	2	3	-	-	-
bne	0	-	2	3	-	-	-
chrs	0	-	-	-	-	-	2
cmp	1	-	-	-	-	3	-
cmpi	1	-	-	-	-	3	-
conb	1	3	-	-	3	-	-
conh	1	3	-	-	3	-	-
cop	0	-	3 [6]	-	-	-	-
di	0	-	-	-	-	-	2
ei	0	-	-	-	-	-	2
exb	1	3	-	-	3	-	-
exbf	1	3	-	-	3	-	-
exbfi	1	3	-	-	3	-	-
exh	1	3	-	-	3	-	-
jal	0	3 [5]	2	3	3 [5]	-	-
jalr	0	3 [5]	2	3	3 [5]	-	-
jmp	0	-	2	3	-	-	-
jmpr	0	-	2	3	-	-	-
ld [8]	1	5 [3]	4 [7]	4 [7]	5	-	-
lli	1	3	-	-	3	-	-
lui	1	3	-	-	3	-	-
mov	1 [1]	3	-	-	3	-	-
movfc	0	4 [4]	3 [6]	-	4	-	-

movtc	0	-	3 [6]	-	-	-	-
mulhi	1 [2]	5	-	-	5	-	-
muli	3	5	-	-	5	-	-
muls	3	5	-	-	5	-	-
muls_16	2	4	-	-	4	-	-
mulu	3	5	-	-	5	-	-
mulu_16	2	4	-	-	4	-	-
mulus	3	5	-	-	5	-	-
mulus_16	2	4	-	-	4	-	-
nop	0	-	-	-	-	-	-
not	1	3	-	-	3	-	-
or	1	3	-	-	3	-	-
ori	1	3	-	-	3	-	-
rcon	0	-	-	-	-	3	-
reti	0	-	4??	3	-	-	2
retu	0	-	2	3	-	-	2
scall	0	3 [5]	2	3	3	-	2
scon	0	4	-	-	4	-	-
sext	1	3	-	-	3	-	-
sexti	1	3	-	-	3	-	-
sll	1	3	-	-	3	3	-
slli	1	3	-	-	3	3	-
sra	1	3	-	-	3	-	-
srai	1	3	-	-	3	-	-
srl	1	3	-	-	3	-	-
srli	1	3	-	-	3	-	-
st [8]	1	-	4 [7]	4 [7]	-	-	-
sub	1	3	-	-	3	3	-
subu	1	3	-	-	3	3	-
swm	0	-	-	-	-	-	2
trap	0	-	3	-	-	-	-
xor	1	3	-	-	3	-	-

[1] Data is only routed through ALU
[2] Executed in step 3 of ALU, based on data evaluated on previous cycle.
[3] Data from memory
[4] Data from a coprocessor
[5] Data in this case is the return address (link) to be saved to the link register.
[6] Address in this case is coprocessor index and coprocessor register index => cop register address.
[7] If address check is not passed, memory access will not take place.
[8] If address falls in range of CCB addresses, no memory access is generated.

Program Counter update timing

Program counter can be updated from various sources:

- PC increment (normal sequential execution)
- Jump address calculation unit (PC relative jumps)
- Output port of the register file (jumps to absolute addresses)
- Interrupt control unit (Interrupt vectors)
- CCB special output ports (system calls and exceptions)
- data bus (boot address can be read from the data bus, if enabled)
- hardware stack (returning from an interrupt routine)

The actual timing, that is, the moment when a new address can be seen on the instruction address bus, depends on the source. The following table summarizes the timing

Table 6.5, Instruction address timing

Cause of change in program flow	**Address source**	**Address calculated**	**Address on bus**
pc relative jumps: b*xx*, jmp, jal	Current PC and extended immediate offset from the instruction in stage 1	stage 1	stage 2
absolute jumps: jmpr, jalr, retu, scall	**scall:** a CCB register output **others**: a RF register output	**-**	stage 2
return from an interrupt routine: reti	hardware stack	Saved to HW stack before switching to service routine.	stage 4
sequential increment [1]	Current PC and PSR IL bit	**next address:** stage 0	stage 0
switching to exception handler [2]	a CCB register output	-	x cycles after the exception was signalled.
switching to an interrupt handler [2]	a CCB register output and external offset if used.	-	x cycles after the interrupt was signalled.
reset	data bus if boot_sel – signal is driven high, otherwise address is set internally to zero.	-	See chapter ‘timing specification’ in document COFFEE_interface.

[1] Stages relate to instructions: In stage 0 the program counter points to the instruction being fetched. At the same time, next address is calculated. When an instruction is in stage 1 the program counter points to the next memory location. The memory address pointed to in stage 0 was evaluated on the previous cycle.
[2] See document about interrupts and exceptions

Note that after swm command, program counter is incremented twice with the old increment. Table 4 below shows the correct operation.

Some assumptions made to fill in the table below:

- Assume START is aligned to word boundary and the processor is in 32 bit mode.
- PC increment is calculated using previous mode, that is, the mode which was valid when the instruction currently in decode was fetched from memory.

Table 6.6 Switching mode

instruction in decode		**addr bus**	**processor mode**	
instruction pointed to	**address <=**	**PC**	**previous mode**	**current mode**
add	START	START + 4		32
sub	START + 4	START + 8	32	32
mov	START + 8	START + 12	32	32
swm	START + 12	START + 16	32	32
nop	START + 16	START + 20	32	16
nop	START + 20	START + 22	16	16
add	START + 22	START + 24	16	16
sub	START + 24	START + 26	16	16
mov	START + 26	START + 28	16	16
swm	START + 28	START + 30	16	16

nop	START + 30	START + 32	16	32
nop	START + 32	START + 36	32	32
add	START + 36	START + 42	32	32
sub	START + 42	START + 46	32	32
mov	START + 46	START + 50	32	32
add	START	START + 4		32
sub	START + 4	START + 8	32	32
mov	START + 8	START + 12	32	32
swm	START + 12	START + 16	32	32
nop	START + 16	START + 20	32	16
nop	START + 20	START + 22	16	16
add	START + 22	START + 24	16	16
sub	START + 24	START + 26	16	16
swm	START + 26	START + 28	16	16
nop	START + 28	START + 30	16	32

nop	START + 30	START + 32	*32*	*32*
add	START + 32	START + 36	32	32
sub	START + 36	START + 42	32	32
mov	START + 42	START + 46	32	32

Underlined row shows a case where increment is two even though the processor is in 32 bit mode. In these cases the address is aligned by hardware. This has no impact on programmer if normal alignment rules are followed.

Different cases when switching mode

X refers to an arbitrary word address (address divisible by four).

Case 1, switching from 16bit to 32bit, aligned case

byte address ⇒	**x + 0**	**x + 1**	**x + 2**	**x + 3**
halfword address ⇒	**x + 0**		**x + 2**	
word address ⇒	**x + 0**			
instruction ⇒	swm		nop	
	nop		-	
bits ⇒	31...24	23...16	15...8	7...0

Notes about case 1:

- The last nop instruction above can be replaced with 32 bit version filling also the empty space.

Case 2, switching from 16bit to 32bit, non-aligned case

byte address ⇒	**x + 0**	**x + 1**	**x + 2**	**x + 3**
halfword address ⇒	**x + 0**		**x + 2**	
word address ⇒	**x + 0**			
instruction ⇒	add		swm	
	nop		nop	
bits ⇒	31...24	23...16	15...8	7...0

Case 3, switching from 32bit to 16bit

byte address ⇒	**x + 0**	**x + 1**	**x + 2**	**x + 3**
halfword address ⇒	**x + 0**		**x + 2**	
word address ⇒	**x + 0**			
instruction ⇒	swm			
	nop			
	addi		mulu	
bits ⇒	31...24	23...16	15...8	7...0

Notes about case 3:

- the 32 bit nop can be 'replaced' with two 16 bit nops to get a more general rule: ALWAYS ADD TWO 16 BIT NOPS AFTER SWM –INSTRUCTION INDEPENDENT OF MODE!

Pipeline stalls

Table 6.7 Pipeline stalls resolving

Stall type	**Explanation**	**Resolving**	**Insert nops to stage**	**Disabled stages**	**Enabled stages**	**stall/wait cycles**
icache access wait	Wait cycle counter for icache, dcache or coprocessor has a nonzero value in it.	Wait for the counter in question to reach zero. Note that once started, a counter will not halt before zero.	1	0	1...5	1...15
dcache access wait			-	0...5	-	
cop access wait			-	0...5	-	
icache miss	There is no valid data in the requested address.	Wait for the i_cache_miss /d_cache_miss signal to go low.	1	0	1...5	n
dcache miss			-	0...5	-	n
flag dependency	A branch instruction or an instruction executed conditionally needs flags which are not ready yet.	Wait in stage 1 for the flags to be ready.	2	0...1	2...5	1
ALU data dependency	An instruction needs register operand(s)	Wait in stage 1 until data is ready and can	2	0...1	2...5	1...2

	which is/are not ready	be forwarded.				
jump address dependency	A jump needs register data which is not ready yet	Wait in stage 1 until data is ready and can be forwarded.	2	0...1	2...5	1...3
bus reserved	ld or st – instruction needs data memory bus but it's reserved by an external device.	Wait in stage 3 (ld or st) until signal *bus_req* goes low	-	0...5	-	n
atomic stall	A 32 multiplication instruction in stage 1 and icache access wait or icache miss active. [2]	Wait for the memory access to finish.	2	0...1	2...5	n/1...15
PC not writable stall	A jump – instruction needs to write PC but branch slot instruction is not fetched yet.	Wait for the memory access to finish.	2	0...1	2...5	n/1...15
external stall request	stall –input is driven high	wait for the stall signal to go low	-	0...5	-	n

[1] The minimum access time for data memory, instruction memory and coprocessor access can be defined by software to be 1 to 16 clock cycles (1 start cycle + 0...15 wait cycles). Once an access starts it won't be stopped or restarted but it can be extended if some other stalls are active AFTER the minimum access time set by software. This means that overlapping stalls do not extend access times.

[2] atomic stall has priority over icache miss or icache access wait. A 32 bit multiplication instruction followed by mulhi instruction is an atomic operation, that is, these instructions have to be executed together and cannot be separated. When waiting for the next instruction from memory we cannot know if it is mulhi or not, thereby we must stall stage 1.

Number of wait bubbles caused by dependencies

Table 6.8 Number of bubbles (nops) added in case of data dependencies (Instruction which need register operand(s) except jmpr and jalr) [2]

Position of the instruction [1]	**Number of ALU cycles [1]**		
	1	2	3
2	0 bubbles	1 bubbles	2 bubbles
3	0 bubbles	0 bubbles	1 bubbles
4	0 bubbles	0 bubbles	0 bubbles

[1] The instruction which the other (currently in stage 1) depends on.
[2] 2nd register operand of st –instruction is ignored when checking dependencies.

Table 6.9 Number of bubbles (nops) added in case of data depencencies: jmpr and jalr.

Position of the instruction [1]	**Number of ALU cycles [1]**		
	1	2	3
2	1 bubbles	2 bubbles	3 bubbles
3	0 bubbles	1 bubbles	2 bubbles
4	0 bubbles	0 bubbles	1 bubbles

[1] The instruction which the other (currently in stage 1) depends on.

Condition flags (Z, N, and C) are always available when an instruction updating them is in stage 3. Therefore an instruction updating flags followed by an instruction using them causes one bubble to be added.

An interrupt or an exception causes a hardware assisted context switch to take place. The pipeline is executed to a safe state feeding nop –instructions in and advancing instructions already on pipeline until they are all in ‘safe state’.

- An instruction is in safe state if
- It won’t change PSR
- It won’t change flags in condition register CR0
- It cannot cause any exceptions
- It won’t change the value of PC

Note that in case of an exception, program counter is immediately updated with the address of an exception handler routine, whereas in case of an interrupt PC may still change if there is jump in stage 1 or swm instruction on pipeline.

Table 6.10 Instructions and their safe states

	Modifies/causes a check	Safe in stage
add	Modifies flags in condition register CR0. Overflow checked.	3
addi		
addiu	Modifies flags in condition register CR0	3
addu		
bc	Updates program counter, New address is checked.	3
begt		
belt		
beq		
bgt		
bnc		
blt		
bne		
chrs	Modifies PSR flags. Mode check (chrs not valid in user mode)	2
cmp	Modifies flags in one of the condition registers.	If **Flags** targeted to CR0 => 3 else => 1
cmpi		
cop	Mode check (cop not valid in 16 bit mode)	2
di	Modifies PSR flags Mode check (di and ei not valid in user mode.)	2
ei		
exbfi	Mode check (exbfi not valid in 16 bit mode)	2
jal	Updates program counter, New address is checked.	3
jalr		
jmp		
jmpr		
ld	Calculates a memory address which has to be checked.	4
lli	Mode check (lli and lui not valid in 16 bit mode)	2
lui		
rcon	Updates the whole condition register file.	3
reti	Updates program counter, CR0 and processor status (PSR). Address not checked in the same context.	3*
retu	Updates program counter and processor status (PSR). Address is checked. Mode check (retu not valid in user mode)	3
scall	Updates program counter and processor status	3

	(PSR). Address is checked.	
sll	Modifies flags in condition register CR0.	3
slli		3
st	Calculates a memory address which has to be checked.	4
sub	Modifies flags in condition register CR0. Overflow checked.	3
subu	Modifies flags in condition register CR0.	3
swm	Modifies PSR Flags. Changes PC increment.	3
trap	Updates program counter and processor status (PSR). Address is checked after switching to exception handler. Incorrect address will result in eternal loop!!	2. (trap causes an exception, so it's never 'safe' for interrupts)
all others		1

Under normal circumstances reti instruction modifies PC and PSR in stage 3 but in case of a hardware assisted context switch its only effect is to ensure correct state of the hardware stack. If an interrupt request gets through while reti is on pipeline (nested interrupts only), hardware stack preserves its state. If an exception occurs while reti is on pipeline (illegal user address) return address is popped but not saved anywhere.

- If an instruction further on pipeline is going to write SPSR (writable as register 30 of register set 2) and there's a scall instruction in stage 1, the one(s) further on the pipeline are invalidated! This prevents status corruption and ensures safe return (using retu -instruction).

6.4. ISA Summary

Mnemonic	Description	Operands	Notes
Integer arithmetic			
add	add 32 bit integers	dreg <= reg1, reg2	
addi		dreg <= reg, imm	
addiu			
addu		dreg <= reg1, reg2	
mulhi	multiply 32 bit integers	dreg <= intermediate	Upper 32 bits of a 64 bit result
muli		dreg <= reg, imm	
muls			
mulu			
mulus			
muls_16	multiply 16 bit		

mulu_16	integers	dreg <= reg1, reg2	
mulus_16			
sub	subtract 32 bit integers		
subu			
Byte and bitfield manipulation			
exb	extract byte from word	dreg <= reg, imm	
exbf	extract bitfield from word	dreg <= reg1, reg2	
exbfi		dreg <= reg, imm	32 bit version only Not allowed to be executed conditionally
exh	extract halfword from word		
lli	Load lower/upper halfword with immediate value	dreg <= imm	32 bit version only Not allowed to be executed conditionally
lui		dreg <= reg, imm	
sext	Sign extend an integer	dreg <= reg1, reg2	
sexti		dreg <= reg, imm	
conb	Concatenate bytes/halfwords	dreg <= reg1, reg2	
conh			
Boolean bitwise operations			
and	bitwise and	dreg <= reg1, reg2	
andi		dreg <= reg, imm	
not	bitwise not	dreg <= reg	
or	bitwise or	dreg <= reg1, reg2	
ori		dreg <= reg, imm	
xor	bitwise xor	dreg <= reg1, reg2	
Conditional jumps (branches)			
bc	Branch if condition is true.	pc <= pc, imm	Pre-evaluated **Flags** from one of the eight condition registers are used to evaluate condition.
begt			
belt			
beq			
bgt			
blt			
bnc			
bne			
Other jumps			
jal	jump and save link address	pc <= pc, imm dreg <= pc + increment	Not allowed to be executed conditionally.

jalr		pc <= reg dreg <= pc + increment	
jmp	jump	pc <= pc, imm	Not allowed to be executed conditionally
jmpr		pc <= reg	
Integer comparison			
cmp	Compare and evaluate condition **Flags**.	creg <= reg1, reg2	Not allowed to be executed conditionally.
cmpi		creg <= reg, imm	
Shifts			
sll	logical shift left	dreg <= reg1, reg2	Only left shift produces **Flags**
slli		dreg <= reg, imm	
sra	arithmetic shift right	dreg <= reg1, reg2	
srai		dreg <= reg, imm	
srl	logical shift right	dreg <= reg1, reg2	
srli		dreg <= reg, imm	
Memory load and store & data moving			
ld	load a word from memory	dreg <= mem[reg + imm]	Address does not have to be aligned to word boundary. Usage of bits 0 to 1 depend on implementation.
st	store a word to memory	mem[reg1 + imm] <= reg2	
mov	move a word from register to register.	dreg <= reg	
Coprocessor instructions			
cop	coprosessor instruction	cop <= imm	32 bit version only Not allowed to be executed conditionally
movfc	mov data from coprocessor	dreg <= cop, cpreg	
movtc	mov data to coprocessor	cop <= reg, cpreg	
Mode changing instructions			
chrs	Change register set to operate with	psr <= imm	Not allowed to be executed conditionally. chrs, di, ei and retu available in super user mode only. Version 1.0 supports to
di	disable interrupts	psr <= IE <= '0'	
ei	enable interrupts	psr <= IE <= '1'	
swm	switch between decoding modes:	psr <= imm	

reti	return from an interrupt service routine	pc <= hw_stack_addr psr <= hw_stack_psr	decoding modes: 16 bit ISA and 32 bit ISA.
retu	return to user/SPSR defined mode	pc <= lreg psr <= spsr	These instructions should be used to interface operating system or similar.
scall	system entry	psr <= sys_psr pc <= sys_entry_addr	
Miscellaneous			
rcon	Restore all condition registers from general purpose register.	creg <= reg	Not allowed to be executed conditionally
scon	Move the contents of all condition registers to a general purpose register	dreg <= creg	
trap	software exception	psr <=	Should be used to catch software exceptions.
nop	no operation, idle		