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 1, 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 instuction extending - immediate operand extending - jump address calculation - decoding for control 1(CCU) - operand forwarding (ALU operands) - register operand fetch & operand selection - execution condition 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 address 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 2, 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	-	-
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**				
nop	0	-	-	-	-	-	-
not	1	3	-	-	3	-	-
or	1	3	-	-	3	-	-
ori	1	3	-	-	3	-	-
rcon	0	-	-	-	-	3	-
reti	0	-	2	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 incrementer (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 summarises the timing

Table 3, 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 4, 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
Non aligned case below				
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.

Summa summarum: 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 5, Pipeline stall 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) which is/are not ready	Wait in stage 1 until data is ready and can be forwarded.	2	0...1	2...5	1...2
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 minimun 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 instructios 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, Number of bubbles (nops) added in case of data depencencies: 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 7, 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, 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.

Number of bubbles added when switching context

General

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 8, 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 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 (PSR). Address	3

	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.

Special Notes

Here is a list of things to remember... things that did not belong under any topic.

- 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).
-