INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

T0 Engineering Data

Krste Asanović
James Beck

TR-96-057

December 1996

Abstract

T0 (Torrent-0) is a single-chip fixed-point vector microprocessor designed for multimedia, human-interface, neural network, and other digital signal processing tasks. T0 includes a MIPS-II compatible 32-bit integer RISC core, a 1 Kbyte instruction cache, a high performance fixed-point vector coprocessor, a 128-bit wide external memory interface, and a byte-serial host interface. T0 implements the Torrent ISA described in a separate "Torrent Architecture Manual" technical report. This manual contains detailed information on the T0 vector microprocessor, including information required to build T0 into a system, instruction execution timings, and information on low level T0 software interfaces required for operating system support.

This work was supported by ONR URI Grant N00014-92-J-1617, ARPA contract number N0001493-C0249, NSF Grant No. MIP-9311980, and NSF PYI Award No. MIP-8958568NSF. Additional support was provided by ICSI.

Contents

1 Introduction **5**

2 CPU **7**

2.1 Operating Modes 7

2.2 Emulated instructions 8

3 System Control Coprocessor (CP0) **9**

3.1 Host Communication Registers 9

3.2 Vector Unit Interrupt Registers 10

3.3 Counter/Timer Registers 11

3.4 Exception Processing Registers 12

3.4.1 Status Register 12

3.4.2 Cause Register 13

3.4.3 Exception Program Counter 14

3.4.4 Bad Virtual Address 14

3.5 Processor Revision Identifier 15

4 Vector Unit Coprocessor 2 **16**

4.1 Vector registers 16

4.2 Vector unit control registers 16

4.2.1 VU Implementation and Revision Number (VCR0) 17

4.2.2 Vector Length Register (VCR2) 18

4.2.3 VU Counter (VCR1) 18

4.2.4 VU Condition Register (VCR4) 18

4.2.5 VU Overflow Register (VCR8) 19

4.2.6 VU Saturation Register (VCR12) 19

5 Instruction Encodings **20**

6 Addressing and Memory Protection **24**

7 Reset, Interrupt, and Exception Processing **25**

7.1 Reset 26

7.2 Interrupts 27

7.3 Synchronous Exceptions 28

8 Pipelines **29**

8.1 Instruction Fetch and Decode Pipeline 31

8.2 CPU Execution Pipeline 32

8.3 VU Arithmetic Unit Execution Pipeline 34

8.4 VU Memory Unit Execution Pipeline 36

9 Instruction Cache **37**

9.1 I-cache organization 37

9.2 I-cache miss processing 38

10 Instruction Timings **39**

10.1 Control Hazards 39

10.2 Structural Hazards 39

10.2.1 Memory Pipeline Structural Hazards 40

10.2.2 Scalar Bus Structural Hazards 42

10.2.3 Vector Arithmetic Structural Hazards 42

10.3 Data Hazards 43

10.3.1 CPU Register Data Hazards 44

10.3.2 Vector Length Register Data Hazards 46

10.3.3 Vector Register Data Hazards 46

10.3.4 Vector Flag Register Data Hazards 51
10.4 CP0 Timing and Hazards 53
10.5 Instruction Cache Miss Timings 54

11 Pin Out **56**

12 Clocking **57**

13 SIP **58**

13.1 Signal Pins 58
13.2 SIP Protocol 58
13.3 SIP Shift Registers 60
13.4 SIP instructions 61
13.4.1 BYPASS 61
13.4.2 MEMREAD 61
13.4.3 MEMWRITE 63
13.4.4 ICWRITE 64
13.4.5 TESTIO 65
13.4.6 SIPIO 66
13.4.7 INTWRITE 66
13.4.8 RUNCPU 66
13.5 SIP Single Step 67

14 Reset **68**

15 External Interrupts **68**

16 T0 Hardware Performance Monitor **69**

16.1 Scalar Unit HPM information 69

16.2 Vector Unit HPM information 70

16.3 Further Sources of HPM Information 70

17 Memory Interface **71**

1 Introduction

T0 [1] is a vector microprocessor, the first implementation of the Torrent ISA. The Torrent Architecture Manual describes the Torrent ISA. This document is T0 specific and provides the engineering data required to build a T0 chip into a system, timing information for T0 instruction execution, and information on low level T0 software interfaces required for operating system support.

The overall structure of T0 is shown in Figure 1. The main components are a MIPS-II compatible RISC CPU, an instruction fetch unit with an instruction cache, a system coprocessor (CP0), a vector unit coprocessor (CP2), a 128-bit wide single cycle external memory interface, and a system interface port (SIP). In addition, T0 has two fast external interrupt pins, an internal counter/timer, and facilities for non-intrusive hardware performance monitoring.

T0 is a single chip microprocessor implemented in a $1.0\mu m$ CMOS technology with a maximum clock frequency of 45 MHz. T0 can be run at lower clock rates to accomodate slower memory subsystems. The CPU is a MIPS-II compatible 32-bit integer datapath. The CPU is used for general scalar computation, and to support the vector unit by providing address generation and loop control. The instruction fetch unit manages a 1 KB instruction cache. The cache is direct-mapped with 64 lines each holding 4 instructions. The fast external memory interface together with a prefetching algorithm reduce instruction cache miss penalties. The maximum cache miss penalty is 3 cycles, and the minimum is 0 cycles. The system coprocessor is implemented as coprocessor 0. CP0 provides exception handling, a 32-bit counter/timer, instruction cache management, and SIP I/O registers.

The vector unit (VU) is added to the base MIPS-II architecture as coprocessor 2. The VU is a vector register machine and contains 16 vector registers. Fifteen of these registers, `$vr1`–`$vr15`, are general purpose and hold 32 elements each 32 bits wide. There is also a zero register, `$vr0`, that is hardwired to return the value 0. There are two vector fixed point arithmetic functional units (VP0 and VP1), each with 8 separate datapaths and capable of completing up to 8 32-bit arithmetic or logical operations per cycle. The datapaths in VP0 can perform up to 8 16-bit$\times$16-bit multiplies per cycle. VP1 does not include a multiplier, but otherwise the arithmetic units are identical. The arithmetic functional units can execute "arithmetic pipeline" instructions that chain up to 6 arithmetic and logical operations within a single instruction. There is a single vector memory functional unit (VMP), capable of sustaining up to 8 operand transfers per cycle. The external memory interface supports up to 4 GB of single cycle memory over a 128-bit data bus. Although T0 issues only one instruction per cycle, it overlaps parallel and pipelined execution in multiple functional units to sustain a computational rate of 720 MOP/s [2] concurrently with a memory bandwidth of 360 M operands/s (720 MB/s).

The system interface port (SIP) has a single control signal and an 8b data path in each direction. Functions accessed through SIP include chip testing, interrupt signalling, instruction cache invalidation, instruction single step, and DMA. Peak DMA rates over SIP to and from T0 memory are 30 MB/s and 34 MB/s respectively at 45 MHz.

[1] T0 is an abbreviation of Torrent-0.

[2] Up to 4.3 GOP/s using "arithmetic pipeline" instructions.

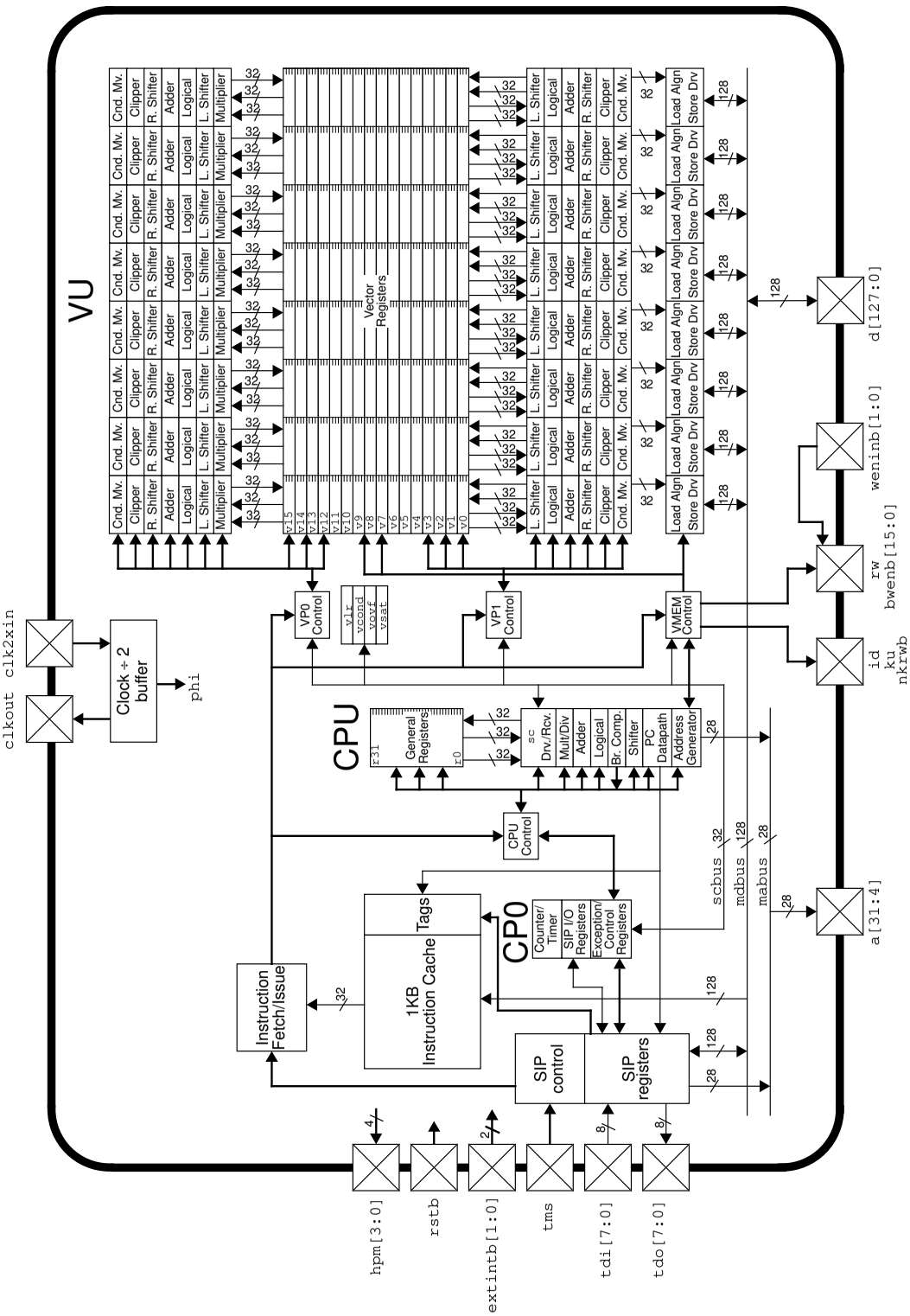

Figure 1: T0 Structure.

2 CPU

The T0 CPU implements the MIPS-II ISA. Figure 2 shows the programmer visible state in the CPU. There are 31 general purpose 32-bit registers `r1`–`r31`. Register `r0` is hardwired to the constant 0. There are three special registers defined in the architecture: two registers `hi` and `lo` are used to hold the results of integer multiplies and divides, and the program counter `pc` holds the address of the instruction to be executed next. These special registers are used or modified implicitly by certain instructions.

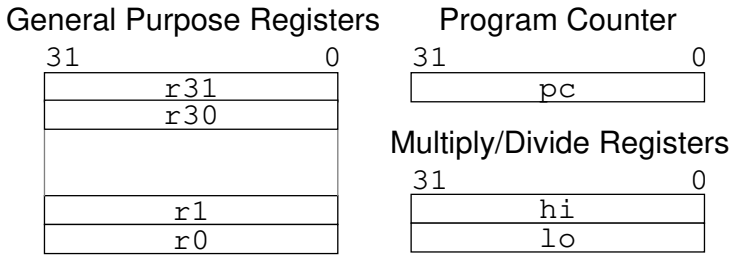

Figure 2: T0 CPU registers.

T0 has a fully pipelined CPU that completes up to one instruction per cycle. T0 has the single MIPS architected branch delay slot. There is a two cycle load-use delay, but both delay slots are fully interlocked.

A hardware multiplier is provided that takes 18 cycles for a 32-bit×32-bit → 64-bit integer multiply. There is a hardware integer divider that takes 33 cycles to perform a 32-bit/32-bit divide returning both a 32-bit integer quotient and a 32-bit remainder. Integer multiplies and divides can proceed in parallel with other instructions provided the `hi` and `lo` registers are not read.

All other CPU instructions apart from branches, loads, multiplies and divides, have single cycle latencies and are fully bypassed so that their results may be used in the following cycle.

In the Torrent architecture, vector memory accesses are unordered with respect to the CPU accesses, and with respect to each other. The MIPS-II SYNC instruction is used to guarantee the order of memory accesses. T0 processes all memory instructions in order, and the SYNC instruction has the effect of waiting for the current memory instruction to complete. This can be used to synchronize T0 memory accesses and host memory accesses over SIP, and also to wait for any pending vector memory address errors.

2.1 Operating Modes

T0 has two operating modes: *user* mode and *kernel* mode. The current operating mode is stored in the KUC bit in the CP0 `status` register. The CPU normally operates in user mode until an exception forces a switch into kernel mode. The CPU will then normally execute an exception

handler in kernel mode before executing a Restore From Exception (RFE) instruction to return to user mode.

2.2 Emulated instructions

Several instructions in the MIPS-II instruction set are not implemented directly by T0. These instructions are trapped and can be emulated in software by the trap handler.

The misaligned load/store instructions, Load Word Left (LWL), Load Word Right (LWR), Store Word Left (SWL), and Store Word Right (SWR), are not implemented. A trap handler can emulate the misaligned access. Compilers for T0 should avoid generating these instructions, and should instead generate code to perform the misaligned access using multiple aligned accesses.

T0 is not designed to operate as part of a shared memory multiprocessor and so the multiprocessor synchronisation instructions, Load Linked (LL) and Store Conditional (SC), are not implemented.

The MIPS-II trap instructions, TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, TNEI, are not implemented. The trap handler can perform the comparison and if the condition is met jump to the appropriate exception routine, otherwise resuming user mode execution after the trap instruction. Alternatively, these instructions may be synthesized by the assembler, or simply avoided by the compiler.

The floating point coprocessor is not present on T0. All MIPS-II coprocessor 1 instructions are trapped and can be emulated. For higher performance, compilers for T0 can directly generate calls to software floating point code libraries rather than emit coprocessor instructions. This will require a modified MIPS calling convention.

3 System Control Coprocessor (CP0)

The system control coprocessor on T0 contains a number of registers used for host communication, the counter/timer, and exception handling. These registers are read and written using the MIPS standard MFC0 and MTC0 instructions respectively. User mode can access the system control coprocessor only if the `cu[0]` bit is set in the `status` register. Kernel mode can always access CP0, regardless of the setting of the `cu[0]` bit. CP0 control registers are listed in Table 1.

Number	Register	Description
0	`fromhost`	SIP input register.
1	`tohost`	SIP output register.
2	`vuepc`	Vector unit exception program counter.
3	`vubadvaddr`	Vector unit bad virtual address.
4–7		*unused.*
8	`badvaddr`	Bad virtual address.
9	`count`	Counter/timer register.
10		*unused.*
11	`compare`	Timer compare register.
12	`status`	Status register.
13	`cause`	Cause of last exception.
14	`epc`	Exception program counter.
15	`prid`	Processor revision/implementation register.
16–31		*unused.*

Table 1: T0 CP0 control registers.

3.1 Host Communication Registers

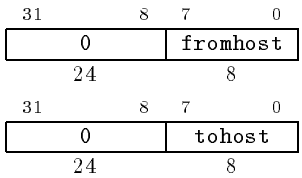

Figure 3: Fromhost and Tohost Register Formats.

There are two registers used for communicating and synchronizing with an external system over SIP. The `fromhost` register is an 8-bit read only register that contains a value written by the host system over SIP. The `tohost` register is an 8-bit read/write register that contains a value that can be read by the host system over SIP. The `tohost` register is cleared after reset to simplify host–T0 synchronization. Their format is shown in Figure 3.

3.2 Vector Unit Interrupt Registers

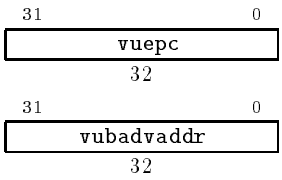

Figure 4: Vector Unit Exception PC and Bad Virtual Address Registers.

The vector memory functional unit can generate an asynchronous interrupt when it encounters an address error on any element of a vector memory instruction. The `ip5` bit of the `cause` register is a sticky bit that is set by any vector address error, and can only be cleared by explicitly writing to the `cause` register. If both the `im5` and `iec` bits of the `status` register are set, an interrupt will be generated whenever `ip5` is set. Refer to Section 7 for further details on interrupt handlng.

The `vupec` register holds the program counter of the last vector memory instruction that had an address error and the `vubadvaddr` register holds the effective virtual address that caused the fault. These registers are updated on any vector address error, even if vector address error interrupts are not enabled. The `vuepc` register always points to the actual instruction that caused the fault, even if the instruction was in a branch delay slot.

Any vector address error stops execution of the current vector memory instruction, and leaves the state of the vector registers and the vector flag registers undefined. Execution cannot be restarted after a vector address error, and so this interrupt is usually considered fatal to the running process.

3.3 Counter/Timer Registers

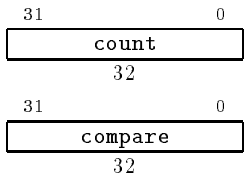

Figure 5: Count and Compare Registers.

T0 includes a counter/timer facility provided by the two coprocessor 0 registers `count` and `compare`. Both registers are 32 bits wide and are both readable and writeable. Their format is shown in Figure 5.

The `count` register contains a value that increments once every clock cycle. The `count` register is normally only written for initialization and test purposes. A timer interrupt is flagged in `ip7` in the `cause` register when the `count` register reaches the same value as the `compare` register. The interrupt will only be taken if both `im7` and `iec` in the `status` register are set. The timer interrupt flag in `ip7` can only be cleared by writing the `compare` register. The `compare` register is usually only read for test purposes.

The `count` register is shadowed read-only in coprocessor 2 control register space as the `vcount` register.

3.4 Exception Processing Registers

A number of CP0 registers are used for exception processing.

3.4.1 Status Register

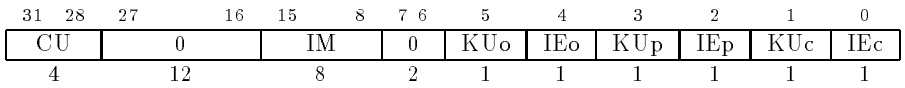

Figure 6: T0 Status Register Format

The `status` register is a 32-bit read/write register formatted as shown in Figure 6. The `status` register keeps track of the processor's current operating state.

The CU field has a single bit for each coprocessor indicating if that coprocessor is usable. Bits 29 and 31, corresponding to coprocessor's 1 and 3, are permanently wired to 0 as these coprocessors are not available in T0. Coprocessor 0 is always accessible in kernel mode regardless of the setting of bit 28 of the `status` register. Both bit 28 and bit 30 may be on simultaneously.

The IM field contains interrupt mask bits. Timer interrupts are disabled by clearing `im7` in bit 15. SIP interrupts are disabled by clearing `im6` in bit 14. Vector address error interrupts are disabled by clearing `im5` held in bit 13. External interrupt 0 is disabled by clearing `im4` in bit 12. External interrupt 1 is disabled by clearing `im3` in bit 11. The other bits within the IM field are not used on T0 and should be written with zeros. Table 7 includes a listing of interrupt bit positions and descriptions.

The KUc/IEc/KUp/IEp/KUo/IEo bits form a three level stack holding the operating mode (kernel=0/user=1) and global interrupt enable (disabled=0/enabled=1) for the current state, and the two states before the two previous exceptions.

When an exception is taken, the stack is shifted left 2 bits and zero is written into KUc and IEc. When a Restore From Exception (RFE) instruction is executed, the stack is shifted right 2 bits, and the values in KUo/IEo are unchanged.

3.4.2 Cause Register

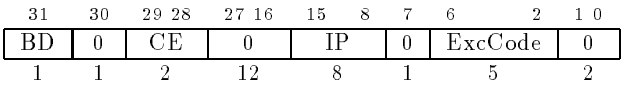

Figure 7: T0 Cause Register Format

The `cause` register is a 32-bit register formatted as shown in Figure 7. The `cause` register contains information about the type of the last exception. Only the `ip5` bit can be written, all other bits are read only.

The ExcCode field contains an exception type code. The values for ExcCode are listed in Table 2. The ExcCode field will typically be masked off and used to index into a table of software exception handlers.

ExcCode	Mnemonic	Description
0	Hint	Host interrupt over SIP.
1	Vint	Vector unit address error interrupt.
2	Tint	Timer interrupt.
4	AdEL	Address or misalignment error on load.
5	AdES	Address or misalignment error on store.
6	AdEF	Address or misalignment error on fetch.
8	Sys	Syscall exception.
9	Bp	Breakpoint exception.
10	RI	Reserved instruction exception.
11	CpU	Coprocessor Unusable.
12	Ov	Arithmetic Overflow.
18	VUE	Vector Unit exception.

Table 2: T0 Exception Types.

If the Branch Delay bit (BD) is set, the instruction that caused the exception was executing in a branch delay slot and `epc` points to the immediately preceding branch instruction. Otherwise, `epc` points to the faulting instruction itself.

If the exception was a coprocessor unusable exception, then the Coprocessor Error field (CE) contains the coprocessor number. This field is undefined for any other exception.

The IP field indicates which interrupts are pending. Field `ip7` in bit 15 flags a timer interrupt. Field `ip6` in bit 14 flags an interrupt from the host over SIP. Flag `ip5` in bit 13 flags a vector unit address error. Flag `ip4` in bit 12 follows the external interrupt 0 pin, and flag `ip3` in bit 11 follows the external interrupt 1 pin. The other IP bits are unused in T0 and should be ignored when read. Table 7 includes a listing of interrupt bit positions and descriptions.

3.4.3 Exception Program Counter

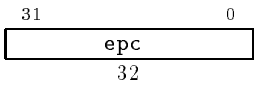

Figure 8: EPC Register.

`Epc` is a 32-bit read only register formatted as shown in Figure 8. When an exception occurs, `epc` is written with the virtual address of the instruction that caused the exception, or if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the branch delay slot.

3.4.4 Bad Virtual Address

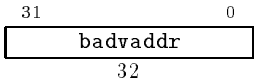

Figure 9: BadVAddr Register.

`Badvaddr` is a 32-bit read only register formatted as shown in Figure 9. When a scalar memory address error generates an AdEL or AdES exception, `badvaddr` is written with the faulting virtual address. The value in `badvaddr` is undefined for other exceptions.

3.5 Processor Revision Identifier

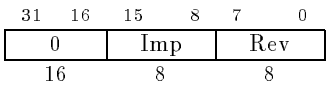

Figure 10: Processor Revision Identifier Register Format.

The `prid` register is a 32-bit read only register that contains the implementation and revision number of the CPU. These values can be used by configuration and diagnostic software.

The `prid` register format is shown in Figure 10. Bits 15–8 define the implementation number, and bits 7–0 define the revision number. Bits 31–16 are reserved and return 0 on T0. The implementation number can be used by user software to detect changes in instruction set or performance. The revision number identifies mask revisions of T0.

Implementation field values are given in Table 3.

Imp. Number	CPU
0	T0
1-255	*reserved*

Table 3: CPU Implementation types.

4 Vector Unit Coprocessor 2

4.1 Vector registers

T0 implements 16 vector registers, `$vr0`–`$vr15`. Vector registers `$vr1`–`$vr15` are general purpose and each contain 32 32b elements. Vector register `$vr0` is hardwired to a vector containing 32 elements with value 0. Reads of `$vr0` return 0, and writes to `$vr0` are ignored. Instructions that attempt to use the unimplemented vector registers, `$vr16`–`$vr31`, cause a reserved instruction exception.

T0 has two vector arithmetic functional units, VP0 and VP1, and a single vector memory functional unit, VMP. Each functional unit can produce up to 8 results per clock cycle.

4.2 Vector unit control registers

The vector unit control registers are listed in Table 4. Any CFC2/CTC2 instruction that attempts to access an unimplemented vector control register will receive an illegal instruction exception.

Number	Register	Description
vcr0	`vrev`	Implementation/revision
vcr1	`vcount`	Counter
vcr2	`vlr`	Vector length
vcr4	`vcond`	Vector condition flags
vcr8	`vovf`	Vector overflow flags
vcr12	`vsat`	Vector saturation flags

Table 4: Vector unit control registers.

4.2.1 VU Implementation and Revision Number (VCR0)

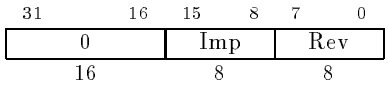

Figure 11: VU Implementation and Revision Register Format.

The `vrev` register is a 32-bit read only register that contains the implementation and revision number of the VU. These values can be used by configuration and diagnostic software.

The `vrev` register format is shown in Figure 11. Bits 15–8 define the implementation number, and bits 7–0 define the revision number. The implementation number can be used by user software to detect changes in instruction set or performance. The revision number identifies mask revisions of T0.

Implementation field values are given in Table 5.

Imp. Number	Vector Unit
0	T0
1-255	*reserved*

Table 5: VU Implementation types.

4.2.2 Vector Length Register (VCR2)

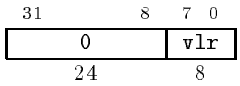

Figure 12: Vector Length Register Format.

The length of a vector operation is specified in an 8-bit vector length register, `vlr`. If a vector instruction is issued when the value in `vlr` is 0, no operations are performed. If a vector instruction is issued when the value in `vlr` is greater than 32, a vector length error exception is raised.

Reads or writes of the vector length register do not affect vector instructions in progress.

4.2.3 VU Counter (VCR1)

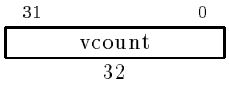

Figure 13: Vector Count Register Format.

The VU count register, `vcount`, is a 32-bit read-only register that holds a cycle counter. It shadows the `count` register in coprocessor 0. The count value is incremented once per clock cycle regardless of host SIP activity, instruction cache misses, or interlocks.

4.2.4 VU Condition Register (VCR4)

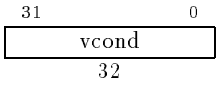

Figure 14: Vector Condition Register Format.

The VU condition register, `vcond`, is a 32-bit read/write register as shown in Figure 14.

The `vcond` register is only altered by vector set less than instructions, vector set equal instructions, and CTC2 writes of `vcond`. After execution of a vector comparison instruction, each bit of `vcond` holds the result of the comparison for each element of the destination vector register. Bit x holds the result of the comparison for element x.

4.2.5 VU Overflow Register (VCR8)

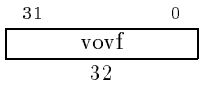

Figure 15: Vector Overflow Register Format.

The VU overflow register, `vovf`, is a 32-bit read/write register as shown in Figure 15. The `vovf` register contains 32 sticky bits holding the overflow status for signed integer adds and subtracts.

The `vovf` register is only altered by vector signed add (ADD.yy) and vector signed subtract (SUB.yy) instructions, and control to coprocessor writes of `vovf`. If any result of a ADD.yy or SUB.yy instruction overflows, the corresponding bit of `vovf` is set. Bit x holds the overflow status of element x. The overflow bits can only be reset by a CTC2 write of `vovf`.

4.2.6 VU Saturation Register (VCR12)

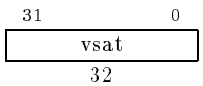

Figure 16: Vector Saturation Register Format.

The VU saturation register, `vsat`, is a 32-bit read/write register as shown in Figure 16. The `vsat` register contains 32 sticky bits holding the saturation status for fixed point adds (FXADD.yy), subtracts (FXSUB.yy), and multiplies (FXMUL.yy).

The `vsat` register is only altered by FXADD.yy, FXSUB.yy, and FXMUL.yy instructions, and CTC2 writes of `vsat`. If any result of a FXADD.yy, FXSUB.yy, or FXMUL.yy instruction saturates, the corresponding bit of `vsat` is set. Bit x holds the saturation status of element x. The saturation bits can only be reset by a CTC2 write of `vsat`.

5 Instruction Encodings

Figures 17, 18, and 19 detail the opcode decoding for T0. A key to the symbols appears below.

* Opcodes marked with an asterisk cause a reserved instruction exception.

ξ Opcodes marked with a xi are illegal but do not cause a reserved instruction exception.

Φ Opcodes marked with a phi cause a coprocessor 1 unusable exception.

δ Opcodes marked with a delta cause a coprocessor 2 unusable exception if the CpU2 bit in the status register is clear, otherwise they cause a reserved instruction exception.

σ Opcodes marked with a sigma cause a reserved instruction exception if *opers* (bit 10) is set.

ρ Opcodes marked with a rho cause a reserved instruction exception if the register number in *rd* doesn't match a coprocessor 2 control register as listed in Table 4.

Θ Opcodes marked with a theta cause a coprocessor 3 unusable exception.

Opcode

31...29 \ 28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	Φ	COP2	Θ	BEQL	BNEL	BLEZL	BGTZL
3	*	*	*	*	*	*	*	*
4	LB	LH	*	LW	LBU	LHU	*	*
5	SB	SH	*	SW	*	*	*	*
6	*	Φ	δ	Θ	*	Φ	δ	Θ
7	*	Φ	δ	Θ	*	Φ	δ	Θ

SPECIAL function

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	MULT	MULTU	DIV	DIVU	*	*	*	*
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

REGIMM rt

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	*	*	*	*	*	*	*	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

Figure 17: T0 CPU Instruction Encodings.

COP0 rs

25...24 \ 23...21	0	1	2	3	4	5	6	7
0	MFC0	ξ	ξ	ξ	MTC0	ξ	ξ	ξ
1	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
2	CO0							
3								

COP0 function

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
1	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
2	RFE	ξ	ξ	ξ	ICINV	ξ	ξ	ξ
3	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
4	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
5	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
6	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ
7	ξ	ξ	ξ	ξ	ξ	ξ	ξ	ξ

Figure 18: T0 Coprocessor 0 Instruction Encodings.

COP2 rs

25...24 \ 23...21	0	1	2	3	4	5	6	7
0	δ	δ	CFC2ρ	δ	δ	δ	CTC2ρ	δ
1	δ	δ	δ	δ	δ	δ	δ	δ
2	IXV	IXS	MEMV	MEMS	IVV	IVS	δ	δ
3	FXAVV	FXAVS	FXSVV	FXSVS	FXMVV	FXMVS	δ	δ

IXV vw

9...8 \ 7...5	0	1	2	3	4	5	6	7
0	δ	δ	δ	δ	δ	δ	δ	δ
1	δ	δ	δ	EXTVσ	δ	δ	δ	δ
2	δ	δ	δ	δ	δ	δ	δ	δ
3	δ	δ	δ	δ	δ	δ	δ	δ

IXS vw

9...8 \ 7...5	0	1	2	3	4	5	6	7
0	δ	δ	δ	δ	δ	δ	δ	δ
1	δ	δ	δ	δ	δ	δ	δ	δ
2	δ	δ	δ	INSSσ	δ	δ	δ	δ
3	δ	δ	δ	EXTSσ	δ	δ	δ	δ

MEMV vw

9...8 \ 7...5	0	1	2	3	4	5	6	7
0	δ	δ	δ	δ	δ	δ	δ	δ
1	δ	δ	δ	δ	δ	δ	δ	δ
2	LBXσ	LHXσ	δ	LWXσ	LBUXσ	LHUXσ	δ	δ
3	SBXσ	SHXσ	δ	SWXσ	δ	δ	δ	δ

MEMS vw

9...8 \ 7...5	0	1	2	3	4	5	6	7
0	LBAIσ	LHAIσ	δ	LWAIσ	LBUAIσ	LHUAIσ	δ	δ
1	SBAIσ	SHAIσ	δ	SWAIσ	δ	δ	δ	δ
2	LBSTσ	LHSTσ	δ	LWSTσ	LBUSTσ	LHUSTσ	δ	δ
3	SBSTσ	SHSTσ	δ	SWSTσ	δ	δ	δ	δ

IVV/IVS rd

15...14 \ 13...11	0	1	2	3	4	5	6	7
0	SUB	SUBU	δ	δ	FLT	FLTU	FEQ	δ
1	SLLV	SRLV	δ	SRAV	SLT	SLTU	SEQ	δ
2	ADDσ	ADDUσ	δ	δ	ANDσ	ORσ	XORσ	NORσ
3	δ	CMVNEZσ	CMVGEZσ	CMVLEZσ	δ	CMVEQZσ	CMVLTZσ	CMVGTZσ

Figure 19: T0 Vector Instruction Encodings.

6 Addressing and Memory Protection

T0 has a full 32-bit virtual address space with a full 32-bit physical address space. Sub-word data addressing is big-endian on T0.

The virtual address space is split into two 2 GB segments, a kernel only segment (*kseg*) from `0x0000_0000` to `0x7fff_ffff`, and a kernel and user segment (*kuseg*) from `0x8000_0000` to `0xffff_ffff`. The segments are shown in Figure 20.

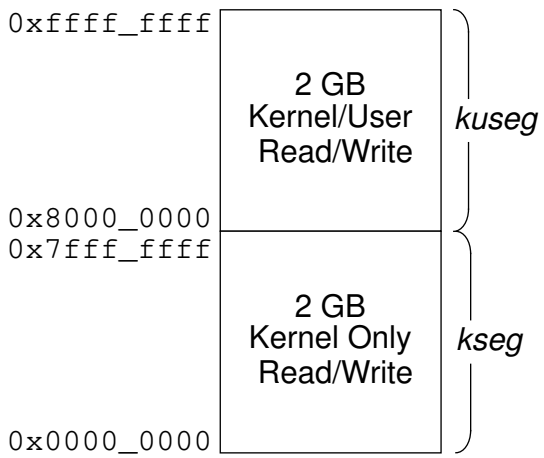

Figure 20: T0 virtual address space.

In kernel mode, the processor can access any address in the entire 4 GB virtual address space. In user mode, instruction fetches or scalar data accesses to the *kseg* segment are illegal and cause a synchronous exception. The AdEF exception is generated for an illegal instruction fetch, and AdEL and AdES exceptions are generated for illegal scalar loads and stores respectively. In user mode, vector data memory accesses to the *kseg* segment cause an asynchronous vector unit address interrupt to be flagged in `ip5` in the `cause` register. For both scalar and vector stores, no data memory will be written at the faulting address. A faulting vector memory operation will cause the state of the vector unit to become undefined.

There is no memory translation hardware on T0. Virtual addresses are directly passed as physical addresses to the external memory system. The external memory system may simply ignore unused high order address bits, in which case each physical memory address will be shadowed multiple times in the virtual address space.

On each memory cycle, T0 outputs information regarding the type (instruction/data, kernel/user, read/write) of each memory access as well as the physical memory address. This information can be used by an external memory protection system to provide finer grain memory protection. For example, user memory access may be restricted to a single shadow of the real physical memory. The external memory protection system can signal faults by interrupting T0.

7 Reset, Interrupt, and Exception Processing

There are three possible sources of disruption to normal program flow: reset, interrupts (asynchronous exceptions), and synchronous exceptions. Reset and interrupts occur asynchronously to the executing program and can be considered to occur *between* instructions. Synchronous exceptions occur *during* execution of a particular instruction. Synchronous exceptions for an instruction are checked at the start of the CPU **Mp** stage.

If more than one of these classes of event occurs on a given cycle, reset has higher priority, and all interrupts have priority over all synchronous exceptions. The tables below show the priorities of different types of interrupt and synchronous exception.

The flow of control is transferred to one of four separate reset, interrupt, and exception vectors, as shown in Table 6. Reset and the external interrupts have separate vectors. All other exceptions share a common vector with different exceptions distinguished by different values in the `exccode` field of the `status` register.

Vector Address	Cause
`0x0000_1000`	Reset
`0x0000_1100`	Exceptions and internal interrupts.
`0x0000_1200`	External interrupt 0.
`0x0000_1300`	External interrupt 1.

Table 6: T0 Reset, Exception, and Interrupt Vectors.

7.1 Reset

When the external reset is deasserted, the PC is reset to `0x0000_1000` with `kuc` set to 0, and `iec` set to 0. The effect is to start execution at the reset vector in kernel mode with interrupts disabled. The `tohost` register is also set to zero to allow synchronization with the host system. The vector unit is idle. All other state is undefined.

A typical reset sequence is shown in Figure 21.

```
reset_vector:
    mtc0 zero, $9    # Initialize counter.
    mtc0 zero, $11   # Clear any timer interrupt in compare.
    mtc0 zero, $13   # Clear any vector interrupt in cause.

    # Initialize status with desired CU, IM, and KU/IE fields.
    li k0, (CU_VAL|IM_VAL|KUIE_VAL)
    mtc0 k0, $12     # Write to status register.

    j kernel_init    # Initialize kernel software.
```

Figure 21: Example reset sequence.

7.2 Interrupts

The five interrupts possible on T0 are listed in Table 7 in order of decreasing priority.

Vector	ExcCode	Mnemonic	IM/IP index	Description
Highest Priority				
0x0000_1100	0	Hint	6	Host SIP interrupt.
0x0000_1100	1	Vint	5	Vector unit interrupt.
0x0000_1100	2	Tint	7	Timer interrupt.
0x0000_1200	*undefined*		4	External interrupt 0.
0x0000_1300	*undefined*		3	External interrupt 1.
Lowest Priority				

Table 7: T0 Interrupts.

All T0 interrupts are level triggered. For each interrupt there is an IP flag in the `cause` register that is set if that interrupt is pending, and an IM flag in the `status` register that enables the interrupt when set. In addition there is a single global interrupt enable bit, `iec`, that disables all interrupts if cleared. A particular interrupt can only occur if both IP and IM for that interrupt are set and `iec` is set, and there are no higher priority interrupts.

The host SIP flag IP6 follows the `intfromhost` bit within the `int` register in the SIP interface. This can be written by the host system over the SIP interface using the SIP INTWRITE instruction. Usually a protocol over the SIPIO registers within the interrupt handler informs the host that it can clear the interrupt flag.

The vector unit interrupt flag IP5 is set by any vector memory instruction address error. The instruction that caused the vector address error is killed, and the vector unit state becomes undefined. The flag bit is sticky and remains set unless explicity cleared by a MTC0 write to the `cause` register.

The timer interrupt flag IP7 is set when the value in the `count` register matches the value in the `compare` register. The flag can only be cleared as a side-effect of a MTC0 write to the `compare` register.

The two external interrupt flags, IP4 and IP3, are inverted clocked copies of the external input pads `extintb[0]` and `extintb[1]`. These provide a fast way to signal interrupts from external hardware.

When an interrupt is taken, the PC is set to the appropriate vector, and the KU/IE stack in the `status` register is pushed two bits to the left, with KUc and IEc both cleared to 0. This starts the interrupt handler running in kernel mode with further interrupts disabled. For internal interrupts the `exccode` field in the `cause` register is set to indicate the type of interrupt. The value in `exccode` is undefined for external interrupts.

The `epc` register is loaded with a restart address. If the instruction that took the interrupt was executing in a branch delay slot, the `bd` bit will be set and `epc` will point to the preceding branch, otherwise `bd` will be clear and `epc` will point to the instruction itself. The `epc` address can be used to restart execution after servicing the interrupt.

7.3 Synchronous Exceptions

Synchronous exceptions are listed in Table 8 in order of decreasing priority.

ExcCode	Mnemonic	Description
Highest Priority		
6	AdEF	Address or misalignment error on fetch.
11	CpU	Coprocessor Unusable.
10	RI	Reserved instruction exception.
8	Sys	Syscall exception.
9	Bp	Breakpoint exception.
12	Ov	Arithmetic Overflow.
18	VUE	Vector Unit exception.
4	AdEL	Address or misalignment error on load.
5	AdES	Address or misalignment error on store.
Lowest Priority		

Table 8: T0 Synchronous Exceptions.

After a synchronous exception, the PC is set to `0x0000_1100`. The stack of kernel/user and interrupt enable bits held in the `status` register is pushed left two bits, and both `kuc` and `iec` are set to 0.

The `epc` register is set to point to the instruction that caused the exception, unless that instruction is in a branch delay slot in which case it points to the preceding branch instruction. The `bd` bit in the `cause` register is set if the exception occured in a branch delay slot. The `exccode` field in the `cause` register is set to indicate the type of exception.

If the exception was a coprocessor unusable exception (CpU), the `ce` field in the `cause` register is set to the coprocessor number that caused the error. This field is undefined for other exceptions.

The overflow exception (Ov) can only occur for scalar ADDI, ADD, and SUB instructions.

The vector unit exception (VUE) only occurs for vector length errors. For VINS.S and VEXT.S instructions, the index register *rd* will contain the faulting length. For VEXT.V instructions, it is necessary to examine both the index register *rd* and the `vlr` register to determine the cause of the fault. For all other instructions, the `vlr` register will contain the faulting length.

If the exception was an address error on a scalar load or store (AdEL/AdES), the `badvaddr` register is set to the faulting address. The value in `badvaddr` is undefined for other exceptions.

8 Pipelines

Each T0 instruction is destined for execution in one of four pipelined execution units: the CPU, the vector memory functional unit (VMP), or one of the two vector arithmetic functional units (VP0 and VP1). The CPU pipeline is also used to process exceptions and interrupts for any vector instruction. Figure 22 illustrates the overall pipeline structure of T0.

All instructions first pass through the instruction fetch (**F**) and instruction decode (**D**) stages, before being dispatched to the appropriate execution unit.

The CPU has an additional 4 pipeline stages: execute (**X**), memory access (**M**), memory data align (**N**), and result write-back (**W**). The CPU can complete one integer instruction per cycle.

The vector memory functional unit has 8 parallel pipelines, segmented into 3 stages: register read (**R**), memory access (**M**), and register write-back (**W**). The vector memory unit can complete up to 8 memory transfers per cycle.

Each of the two vector arithmetic functional units has 8 parallel pipelines, segmented into 4 stages: register read (**R**), execute stage 1 (**X1**), execute stage 2 (**X2**), and register write-back (**W**). Each vector arithmetic unit can produce 8 results per cycle. Within each arithmetic unit pipeline there are 6 cascaded arithmetic and logical operators. These can be combined in the arithmetic pipeline instructions to perform up to 6 cascaded arithmetic and logical operations in one cycle down the pipeline.

Each clock cycle is divided into two phases, p and n (clock low and clock high respectively). Pipe phases are written as **Fp**, **Fn**, **Dp**, **Dn**, etc. The following sections describe the actions performed for each pipe phase in T0.

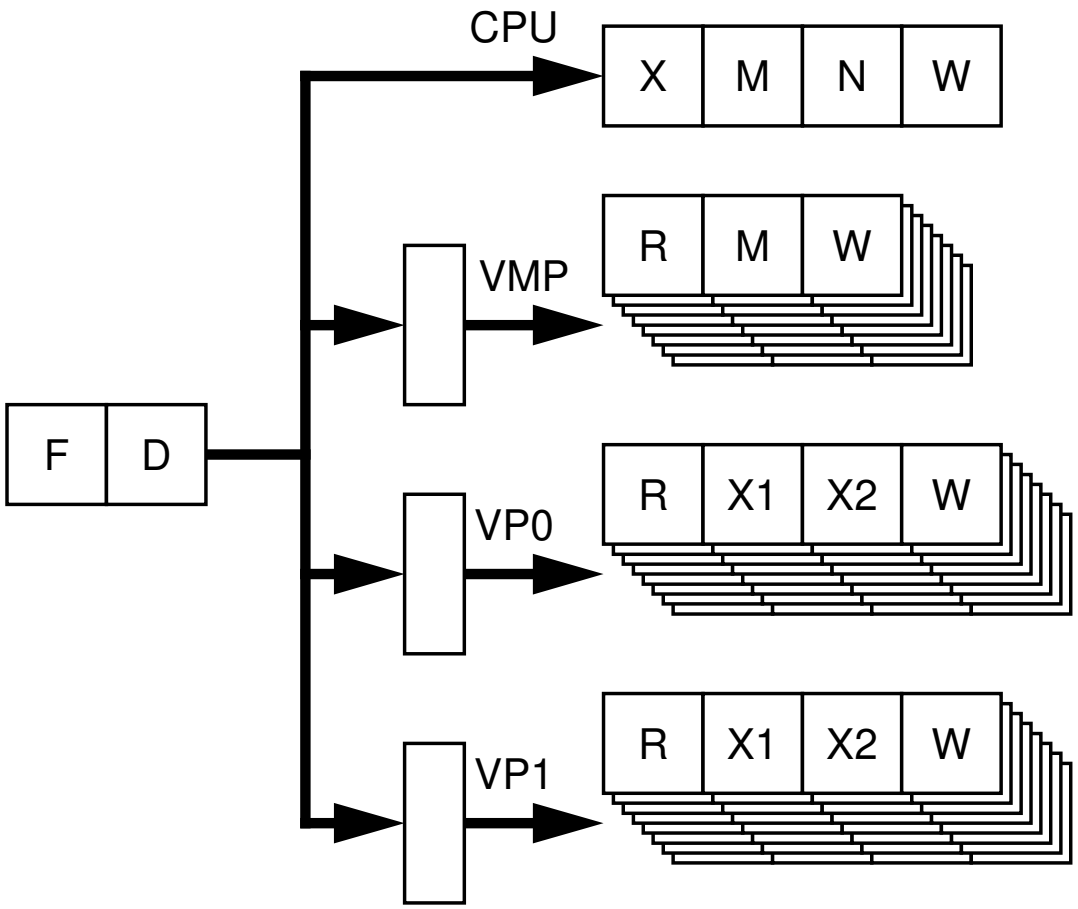

Figure 22: T0 Pipelines.

8.1 Instruction Fetch and Decode Pipeline

Fp

- New program counter is selected and instruction's physical address is fed into the instruction cache decoders.

Fn

- Instruction cache returns indexed instruction and generates hit/miss signal.
- If memory port is free, send out instruction physical address to start prefetch of cache line.

Dp

- Register specifiers sent to CPU register file read port decoders. Note must first decode whether this is a coprocessor operation to mux *rd* instead of *rs* into first read port decoder.
- Check for reserved instructions. Flag reserved instruction exception if necessary.
- Check for SYSCALL and BREAK instructions and flag appropriate exception.
- Perform dependency check on register operands. Calculate bypass control for CPU. Interlock if data hazard on vector operations.
- Check for structural hazard in vector units. Interlock execution if there is no suitable free vector unit.
- Decode CPU instruction. Calculate control for CPU datapath.

Dn

- Register file returns CPU registers. Sign-extend immediate and perform register bypassing.
- CPU ALU and address generator starts execution.
- Dispatch instruction to appropriate execution unit.

8.2 CPU Execution Pipeline

Xp

- Check for structural hazard on memory port. Interlock if this is a load/store and memory port is in use.
- Branch comparator evaluates early. If branch, select program counter in concurrent **Fp** phase.
- Complete execution of CPU ALU operation and address generator.
- If vector-scalar, or scalar-vector, operation, scalar operand sent to vector unit.
- If vector fixed-point operation, configuration register passed to vector control unit.

Xn

- If load/store send physical address to external memory.
- Check ALU result for integer overflow. Flag arithmetic overflow exception if necessary.
- ALU results forwarded to **Dn** stage.
- Check for any enabled interrupts, and check all exceptions for this instruction. Raise exception flags as necessary for this instruction.

Mp

- If any exception flags raised, take exception by killing instructions following in pipeline, jamming PC to exception vector in concurrent **Fp** phase, and updating exception handling state.
- First phase of external memory access.

Mn

- Second phase of external memory access returns value into vector memory unit latches.

Np

- Load data is aligned, and zero/sign-extended in vector memory unit.

Nn

- Load data is passed over `scbus` to scalar unit and forwarded to **Dn** stage for bypassing.
- Register write address passed to register file write port decoder.

Wp

- Write back result to register file.

Wn

- This pipe stage intentionally left blank.

8.3 VU Arithmetic Unit Execution Pipeline

Rp

- Register specifiers passed to vector regfile address decoders.

Rn

- Vector register file reads next vector operands.
- Scalar value received from CPU.
- Configuration register information received from CPU.
- Instruction control lines decoded.

X1p

- Mux vector-vector, vector-scalar, or scalar-vector operands into pipeline.
- Begin multiplier evaluation.
- Begin logic unit evaluation.
- Begin left shifter evaluation.

X1n

- Complete multiplier evaluation.
- Complete logic unit evaluation.
- Complete left shifter evaluation.
- Sign-extend and mux adder inputs.
- Start adder evaluation.

X2p

- Complete adder evaluation.

X2n

- Evaluate conditional move condition.
- Evaluate right shifter and sticky bit logic.
- Mux clipper input.
- Start clipper evaluation.

Wp

- Complete clipper evaluation.
- Write clipper output to vector register file.
- Pass condition/overflow/saturation flags to CPU.

Wn

- Write condition/overflow/saturation flags in CPU.

8.4 VU Memory Unit Execution Pipeline

This section only describes scalar, contiguous and strided vector memory pipeline operations. Indexed operations have a more complicated pipeline scheme.

Rp

- Register specifiers passed to vector regfile address decoders.
- Memory address evaluated in concurrent CPU **Xp** stage.

Rn

- Vector register file reads next vector operands.
- Scalar store or scalar insert value received from CPU.

Mp

- Mux store and vector extract operands out to memory crossbar.

Mn

- Complete drive of store data out to memory.
- Receive load data back from memory.
- Mux load data or vector extract data through crossbar.

Wp

- Sign or zero extend load data.
- Write vector load data to vector register file.

Wn

- Send scalar load data or scalar extract value to CPU.

9 Instruction Cache

T0 has a 1 KB instruction cache (I-cache) that holds 256 instructions. The on-chip I-cache allows instruction fetches to proceed in parallel with data accesses over the external memory port.

9.1 I-cache organization

The I-cache is organized as a direct mapped cache with 64 lines. The cache is indexed with the physical address. Since the physical address index is a simple truncation of the virtual address there is no time penalty for using a physically indexed cache.

Each I-cache line contains a single valid bit, an 18-bit physical tag, and four 32-bit instructions (16 bytes). An I-cache access is shown in Figure 23.

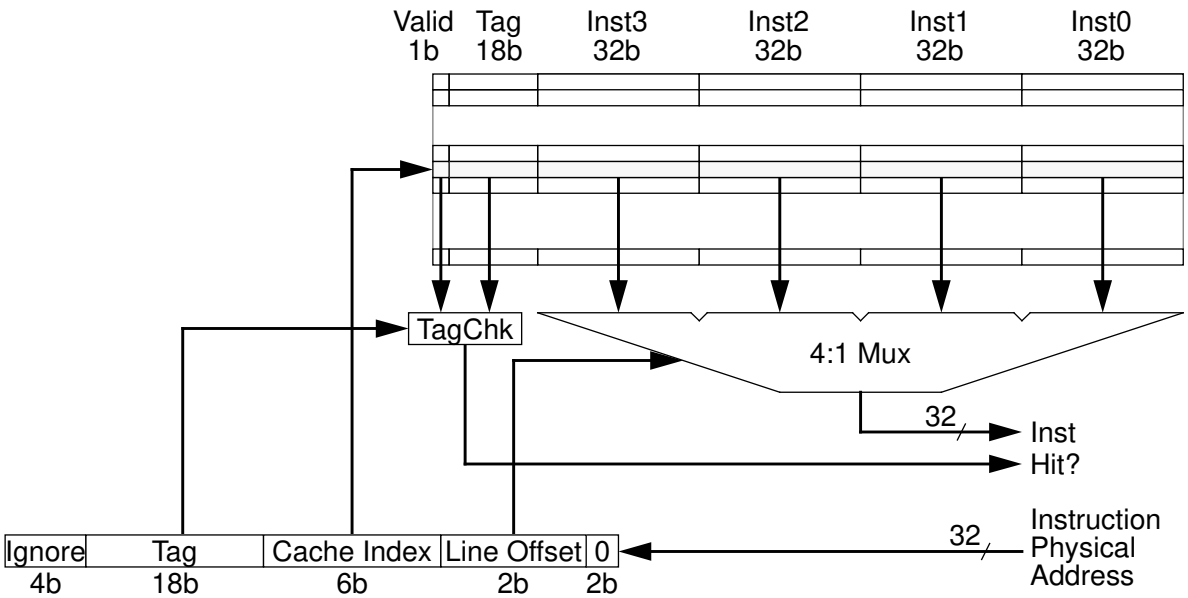

Figure 23: T0 Instruction Cache Read.

Note that the top four bits of the instruction address are ignored for the purposes of cache tag matching. To avoid cache aliasing, the maximum program size must be limited to 256 MB. Alternatively, explicit cache invalidate instructions can be issued before moving between aliased instruction addresses.

9.2 I-cache miss processing

On an I-cache miss a complete 16 byte line is read from external memory. In the worst case, the I-cache has a miss penalty of 3 cycles. When a miss is detected for an instruction during the `F` stage, the cache fetch state machine begins a 3 state miss service loop.

In the first miss cycle, the address of the cache line containing the missing instruction is output on the address bus. During the second miss cycle, the external memory returns the line containing the missing instruction. The third miss cycle is used to write back the new line into the instruction cache, and to forward the fetched instruction to the CPU decode stage.

To help reduce miss penalties, T0 also implements a simple prefetching scheme. Whenever the memory port is not busy with a host SIP access, a CPU load/store, a vector load/store, or a vector extract, the instruction address being fetched internally from the cache is also prefetched from the external memory. The prefetch address is output during the normal F cycle. If there is a miss, the fetch engine can omit the first cycle of the normal miss service loop thus reducing the cache miss penalty to 2 cycles.

To further reduce miss penalties, the fetch stage operates autonomously and can service miss requests while the D stage is interlocked or stalled on the memory pipeline. If instruction i is interlocked in the D stage, the F stage can service a miss on the instruction $i + 1$. In this case the miss penalty may be completely hidden behind the interlock stall. If the D stage is stalled waiting for a vector memory instruction to finish, the concurrent F stage I-cache miss servicing will only steal a single memory cycle from the ongoing vector memory access and can hide the rest of the 3 cycle miss latency behind the memory stall.

10 Instruction Timings

This section gives timing information for T0 instruction execution. T0 is fully interlocked with no architecturally visible hazards, except for the MIPS standard branch delay slot and the MIPS standard integer multiplier/divider hazards. T0 issues at most one instruction per cycle. All of T0 memory is constructed from pipelined SRAM, and hence there is no data cache, and no DRAM paging or refresh penalties. T0 has an instruction cache with miss penalties described below.

The timing information is separated into control hazards, structural hazards, and data hazards.

10.1 Control Hazards

The only control hazard on T0 is the single architected branch delay slot. The instruction following a branch is always executed, except after a not-taken branch likely instruction.

The MIPS-II branch likely instructions ensure that the branch delay slots can always be filled. If no independent instruction can be moved down from the preceding basic block to fill a normal branch delay slot, the instruction at the target of the branch can be moved into the delay slot and the branch changed to a branch likely.

10.2 Structural Hazards

A structural hazard occurs when an instruction cannot issue because one of the resources it requires for execution is in use by a previously issued instruction. There are three sources of such resource conflicts on T0: the memory pipeline, the two vector arithmetic pipelines, and the internal scalar bus (`scbus`). The memory pipeline is also used by the SIP port to read and write T0 memory, and by the instruction fetch unit for instruction cache refills.

Note that there are no structural hazards on the scalar multiplier/divider. There are data hazards that must be obeyed to correctly retrieve results, see Section 10.3.1.

In the following, the times that instructions occupy a resource are given in cycles. If an instruction occupies a resource for a single cycle, then a following instruction that requires the same resource can issue in the next cycle in a fully pipelined manner.

10.2.1 Memory Pipeline Structural Hazards

The memory pipeline handles external SIP memory requests, instruction cache refills, as well as scalar and vector memory operations. SIP memory accesses have highest priority, instruction cache refills have the next highest priority, and scalar and vector memory pipeline operations have the lowest priority. If there is a memory instruction in progress when a SIP access or instruction cache refill occurs, the complete vector unit will stall (the vector arithmetic units must also stall to preserve chaining). The CPU will only stall if the memory instruction in progress was a scalar load/store or a scalar insert/extract. If there are no memory requests on a given cycle, the instruction fetch unit prefetches the next instruction cache line.

SIP memory requests take one cycle in the memory pipeline. Section 13 contains detailed timing information of this access relative to SIP state machine activity.

Instruction cache refills take one cycle in the memory pipeline. This refill cycle is not required if there was an instruction prefetch during the fetch cycle that missed in the instruction cache.

Scalar load/store and scalar insert/extract instructions take one cycle in the memory pipeline.

Scalar `sync` instructions take one cycle in the memory pipeline. These are added to perform memory synchronization, and ordinarily a scheduler should not move other instructions across a `sync`.

Vector extracts run at different rates depending on the alignment of the extract index. If the extract index is a multiple of 8 (0, 8, 16, 24), then values can be read and written within the same datapath over a special bypass path without using the memory crossbar, and so can proceed at the rate of 8 32-bit elements per cycle. Otherwise, the extract must transfer elements using the memory crossbar which is limited to moving at most 4 32-bit elements per cycle. If the extract index is not a multiple of 4 and the vector to be extracted crosses an alignment boundary of 4 elements, then a further misalignment cycle is also required.

Contiguous vector memory instructions transfer up to 16 bytes per cycle between the memory data bus and 8 ports of the vector register file. Byte load/stores move up to 8 bytes per cycle; these are constrained by the number of ports on the vector register file. Halfword load/stores move 16 bytes per cycle, as 8 2-byte values. Word load/stores move 16 bytes per cycle, as 4 4-byte values.

Contiguous vector memory stores occupy the vector memory pipeline for a number of cycles equal to the number of naturally aligned blocks of memory that are written, where a block is 8 bytes for contiguous byte stores and 16 bytes for contiguous halfword and word stores.

Contiguous vector memory loads have a more complicated behaviour. The number of cycles is at least the number of naturally aligned blocks of memory that are read. However, the number of cycles is also bounded by the number cycles taken to write the vector register file. The vector register file can accept up to 8 operands per cycle, but all operands on the same register file row must be available before a register row write takes place. Register rows are 8 elements wide for byte and halfword loads, and 4 elements wide for word loads. For scheduling purposes, a simpler model for the cycle count can be used, just counting the number of memory blocks read. This figure will be at most 1 cycle too small.

The strided vector load/store instructions only load/store one operand per cycle. These take `vlr` cycles to complete.

The indexed vector loads load one operand per cycle but incur a 3 cycle latency to read the first address index. Indexed stores incur the same startup latency, but also require an extra cycle to read a group of 8 indices every 8 operands since there is only one vector register read port in the memory pipeline.

Table 9 summarizes the number of cycles that each memory pipeline operation occupies the pipeline.

Operation	Cycles in VMP
SIP MEMREAD, MEMWRITE, ICWRITE	1
I-cache refill	1
Instruction	**Cycles in VMP**
`lb`, `lbu`, `sb`, `lh`, `lhu`, `sh`, `lw`, `sw`	1
`sync`	1
`vins.s`, `vext.s`	1
`vext.v`, index in *rd* 8-aligned `vext.v`, index in *rd* 4-aligned `vext.v`, index in *rd* not 4-aligned	$\lceil \texttt{vlr}/8 \rceil$ $\lceil \texttt{vlr}/4 \rceil$ $1 + \lceil \texttt{vlr}/4 \rceil$
`lbai.v`, `lbuai.v`, `sbai.v`	$\lfloor \frac{\texttt{rd}+(\texttt{vlr}-1)}{8} \rfloor - \lfloor \frac{\texttt{rd}}{8} \rfloor + 1^{\dagger}$
`lhai.v`, `lhuai.v`, `shai.v`	$\lfloor \frac{\texttt{rd}+2\times(\texttt{vlr}-1)}{16} \rfloor - \lfloor \frac{\texttt{rd}}{16} \rfloor + 1^{\dagger}$
`lwai.v`, `swai.v`	$\lfloor \frac{\texttt{rd}+4\times(\texttt{vlr}-1)}{16} \rfloor - \lfloor \frac{\texttt{rd}}{16} \rfloor + 1^{\dagger}$
`lbst.v`, `lbust.v`, `lhst.v`, `lhust.v`, `lwst.v`, `sbst.v`, `shst.v`, `swst.v`	`vlr`
`lbx.v`, `lbux.v`, `lhx.v`, `lhux.v`, `lwx.v`	$3 + \texttt{vlr}$
`sbx.v`, `shx.v`, `swx.v`	$2 + \lceil \texttt{vlr}/8 \rceil + \texttt{vlr}$

Table 9: Memory pipeline usage for T0 instructions. Vector instruction timing is affected by the vector length `vlr`, and vector contiguous load/store and vector extract timing is also affected by the alignment of the address or extract index in the `rd` register. † The model for contiguous loads has been simplified, and these can take an extra cycle in certain cases. See text for details.

10.2.2 Scalar Bus Structural Hazards

A single on-chip bus carries both coprocessor register values and vector indexed memory operation indices into the CPU. Indexed vector memory instructions occupy the scalar bus for their entire duration. Table 10 summarizes the scalar bus structural hazards.

Instruction	Cycles on `scbus`
`mfc0`, `cfc2`	1
`lbx.v`, `lbux.v`, `lhx.v`, `lhux.v`, `lwx.v`	$3 + \texttt{vlr}$
`sbx.v`, `shx.v`, `swx.v`	$2 + \lceil \texttt{vlr}/8 \rceil + \texttt{vlr}$

Table 10: Scalar bus usage for T0 instructions. Vector instruction timing is affected by the vector length `vlr`.

10.2.3 Vector Arithmetic Structural Hazards

T0 has two vector arithmetic units VP0 and VP1. They are identical except that VP1 does not have a multiplier. All `fxmul` instructions must execute in VP0, but all other instructions can be executed in either pipeline. If both arithmetic units are free and a non-multiply instruction is decoded it will be issued to VP1 rather than VP0.

Each of the two vector arithmetic units completes 8 element operations per cycle. An arithmetic unit is busy for $\lceil \texttt{vlr}/8 \rceil$ clock cycles when processing an arithmetic operation on a vector of length `vlr`.

10.3 Data Hazards

Instructions access memory in order of issue through a single memory pipeline, with memory access occurring in the same pipeline stage for all instructions, and so T0 has no data hazards on memory values. The only data hazards that can occur are on register values. Data hazards are of three types: Read-After-Write, Write-After-Read, and Write-After-Write.

A Read-After-Write (RAW) hazard is a true dependency, whereby an instruction must wait for one of its source registers to be written by a previous instruction. If the new instruction was not delayed, it would get an incorrect earlier value for its source operand.

A Write-After-Read (WAR) hazard is a false dependency, whereby an instruction cannot overwrite one of its destination registers until all previous instructions reading that register have obtained the original value. If the new instruction was not delayed, the old instructions would incorrectly get the new value for the register.

A Write-After-Write (WAW) hazard is a false dependency, whereby an instruction cannot overwrite one of its destination registers until all previous instructions that write the same register have finished. If the new instruction was not delayed, the old instructions would eventually finish and update the register, thus destroying the correct new value.

The following sections detail the data hazard timings for T0, with each section describing one group of registers: the CPU scalar registers including `hi` and `lo`, the vector length register `vlr`, the vector registers `$vr0`–`$vr15`, and the vector flag registers `vcond`, `vovf`, and `vsat`.

Data hazards occur between a pair of instructions, the instruction that first accesses a register and a second instruction that also wants to access the same register. In the tables that summarize data hazards, the first instruction to issue is listed in the rows and the second instruction in the columns. The values in the tables give the minimum number of delay cycles required between the two instructions to avoid the given data hazard on the given register. Except for the WAR hazard on the scalar multiplier/divider registers which must be scheduled, T0 has hardware interlocks that delay issue of a new instruction until all data hazards are resolved. Instruction scheduling is not required for correctness but only to improve performance. The number of delay cycles represents the minimum number of instructions that should be scheduled between dependent instructions to avoid an interlock. For example, scalar loads have two delay cycles. The sequence:

```
lw t1, (a0)
addiu t1, 1         # Two hardware interlock cycles.
```

will incur two hardware interlock cycles. These interlock cycles may be usefully filled with two independent instructions as shown below:

```
lw t1, (a0)
addiu a0, 4         # Schedule other instructions in delay cycles.
slt t2, a0, a1
addiu t1, 1         # No interlock.
```

10.3.1 CPU Register Data Hazards

The result of most CPU instructions are available to the instruction issued in the next cycle. The only exceptions that cause RAW hazards are loads, scalar extracts, coprocessor register reads, and multiplies and divides.

Scalar memory loads (`lb/lbu/lh/lhu/lw`) have a latency of 3 cycles, and so 2 delay cycles. Reads from coprocessor registers (`mfc0/cfc2`) have 2 delay cycles. Scalar extracts from vector registers (`vext.s`) have 2 delay cycles. There are 17 delay cycles between the issue of an integer multiply (`mult/multu`) and the read of the result (`mfhi/mflo`). There are 32 delay cycles between the issue of an integer divide (`div/divu`) and the read of the result (`mfhi/mflo`). There is a single delay cycle between any `mthi/mtlo` instruction and any `mfhi/mflo` instruction, e.g., a `mthi` followed by a `mflo` will experience a single delay cycle even though the two instructions reference different registers.

To simplify the implementation, there is a false interlock on the scalar extract instruction `vext.s`: The scalar *rt* destination register of a `vext.s` instruction is interlocked as though it were a source register even though its value is not read.

All CPU ALU instructions read the register file early in the pipeline and write the register file late in the pipeline, so there are no WAR hazards for the general purpose registers.

The multiplier/divider registers are written early in the pipeline. A `mfhi` or `mflo` instruction cannot be followed by any instruction that changes the `hi` or `lo` registers (`mthi/mtlo/mult/multu/div/divu`). There is no hardware interlock for this WAR hazard, and there must be an intervening instruction for correct execution. Note there only needs to be a single instruction separating the read and write, not two as specified in the MIPS architecture.

There are no WAW hazards for the CPU general purpose registers or the multiplier/divider registers.

Tables 11–13 summarize the data hazards in the CPU registers.

Writer \ Reader	Any GPR Read
`lb/lbu/lh/lhu/lw`	2
`mfc0/cfc2`	2
`vext.s`	2

Table 11: RAW hazards for CPU GPRs. Timings are given as number of delay cycles.

Writer \ Reader	`mfhi/mflo`
`mthi/mtlo`	1
`mult/multu`	17
`div/divu`	32

Table 12: RAW hazards for CPU `hi/lo` registers. Timings are given as number of delay cycles.

Writer \ Reader	`mthi`	`mtlo`	`mult/multu/div/divu`
`mfhi`	1 *mandatory*	0	1 *mandatory*
`mflo`	0	1 *mandatory*	1 *mandatory*

Table 13: WAR hazards for CPU `hi/lo` registers. Timings are given as number of delay cycles. There is no hardware interlock for these hazards and so the delay cycles are labeled *mandatory*. An instruction must be scheduled between the reader and writer for correct execution.

10.3.2 Vector Length Register Data Hazards

The current effective length of the vector registers is specified in the vector length register `vlr`, which is implemented as coprocessor 2 control register 2 and accessed with the MIPS standard `ctc2`/`cfc2` instructions. This vector length is an implicit source operand of all vector instructions (other than scalar insert and extract). There are no RAW hazards on `vlr` for vector instructions; a `ctc2` write of `vlr` will take effect on the next cycle. Scalar unit reads of `vlr` incur the same two cycle delay as any coprocessor register read.

There are no WAR or WAW hazards on `vlr`. Each vector functional unit copies the vector length at instruction issue, so the vector length register can be changed at any time without affecting ongoing vector operations.

10.3.3 Vector Register Data Hazards

The vector register file on T0 provides dedicated read and write ports for each operand of each vector functional unit, except that the vector memory pipeline uses the same read port for store indices and store data during indexed store operations. This removes register port access restrictions and so the only vector register hazards possible are due to RAW, WAR, or WAW dependencies on element values. T0 provides chaining with a fully multi-ported vector register file rather than by bypassing around functional units. This removes the need for a specified chain slot time, and also allows chaining of WAR and WAW hazards.

All vector arithmetic instructions behave identically as regards vector register data hazards, and so in the following they are referred to collectively as VALU instructions. All VALU instructions read vector source registers at the rate of 8 elements per cycle starting at element zero. They also write results to destination vector registers at the rate of 8 elements per cycle starting at element zero. Vector arithmetic instructions have a 3 cycle pipeline, with a 1/2 cycle to read vector registers, 2 cycles of execution, then a 1/2 cycle to write back vector register results. Dependent vector arithmetic instructions can be chained after 2 delay cycles.

Vector memory instructions have a wider range of behaviors depending on instruction class, data size, and address or extract index alignment. Contiguous vector loads and stores of bytes and halfwords can proceed at the rate of 8 elements per cycle, and so can be directly chained with arithmetic operations. Contiguous vector loads of bytes and halfwords have a single delay slot, plus an extra delay slot if the base address was not aligned.

Contiguous vector loads and stores of words can only transfer 4 elements per cycle. This prevents direct chaining with subsequent VALU instructions except towards the end of longer vectors. In this case the dependent VALU instructions can start when it is certain that the contiguous word memory operation can complete its vector register access before the VALU catches up.

Strided and indexed vector load/stores complete at the rate of one element per cycle. This prevents subsequent VALU instructions from chaining except for the last few cycles of long vectors when it is then certain that that strided or indexed operations will complete before the VALU instruction

catches up. Indexed operations have an extra start up penalty due to the time needed to first read index values into the address generator. Indexed stores share a single vector register read port between indices and store data, and so require an extra cycle every 8 cycles to read out the next group of 8 indices.

Scalar and vector extracts can read values from any position in a vector register depending on the value in a scalar index register. The index register value is not known at instruction issue time, and so the instruction dispatch unit must make the conservative assumption that values could be read from anywhere in the source vector register. These extract instructions cannot issue until all pending writes to the source vector register have completed.

Vector extract instructions write values at different rates depending on the extract index (see structural hazards above). When the extract index is 8-aligned, the vector extract instruction produces results at the rate of 8 elements per cycle and can be chained to arithmetic operations. When the extract index is not 8-aligned, values are produced at the rate of 4 elements per cycle and so direct chaining is not possible, except towards the end of long vectors as for contiguous word loads.

Scalar inserts into a vector register can potentially write to any location in its vector destination register depending on the value in a scalar index register. Since the index register value is not known at issue time, the instruction dispatch unit must make a conservative assumption and assume that it could write anywhere in the destination register. To prevent WAR and WAW hazards with previous vector arithmetic instructions, the insert instruction cannot issue until all ongoing arithmetic instructions have finished reading and writing their source and destination vector registers. To simplify the implementation the scalar insert instruction interlock logic does not compare vector register numbers and so will interlock until *all* ongoing vector arithmetic instructions complete, regardless of which vector registers they access.

The vector memory pipeline is one cycle shorter than the vector arithmetic pipelines. To avoid a WAW hazard, a vector memory pipeline operation cannot write the same destination vector register as an immediately preceding vector arithmetic instruction and is delayed for one cycle.

Tables 14–16 summarize the different hazard timings for the vector registers on T0.

Writer \ Reader	VALU (*vt*/*vd*), `s_ai.v` (*vd*), `s_st.v` (*vd*), `l_x.v` (*vt* indices), `s_x.v` (*vt* indices)	`s_x.v` (*vd* data)	`vext.v` (*vd*), `vext.s` (*vd*)
VALU (*vw*)	2	0	$1 + \lceil \texttt{vlr}/8 \rceil$
`lbai.v` (*vd*)	$m8 + 1$	*S.H.*	$m8 + \lceil \texttt{vlr}/8 \rceil$
`lhai.v` (*vd*)	$m16 + 1$	*S.H.*	$m16 + \lceil \texttt{vlr}/8 \rceil$
`lwai.v` (*vd*)	$m16 + \min(\lceil \texttt{vlr}/4 \rceil, 5)$	*S.H.*	$m16 + \lceil \texttt{vlr}/4 \rceil$
`l_st.v` (*vd*)	$\min(\texttt{vlr}, 29)$	*S.H.*	`vlr`
`l_x.v` (*vd*)	$\min(\texttt{vlr} + 3, 32)$	*S.H.*	`vlr` + 3
`vins.s` (*vd*)	1	*S.H.*	1
`vext.v` (*vd*), index in *rd* 8-aligned	1	*S.H.*	$\lceil \texttt{vlr}/8 \rceil$
`vext.v` (*vd*), index in *rd* 4-aligned	$\min(\lceil \texttt{vlr}/4 \rceil, 5)$	*S.H.*	$\lceil \texttt{vlr}/4 \rceil$
`vext.v` (*vd*), index in *rd* not 4-aligned	$1 + \min(\lceil \texttt{vlr}/4 \rceil, 5)$	*S.H.*	$1 + \lceil \texttt{vlr}/4 \rceil$

Table 14: RAW hazards for vector registers. Timings are given as number of delay cycles. The `vlr` value represents the vector length of the first instruction issued, i.e., the writer. Data hazard timing after a vector contiguous load is affected by whether the base address is misaligned: $m8 = 1$ if vector misaligned on an 8 byte boundary and vector crosses an 8 byte boundary, 0 otherwise; $m16 = 1$ if vector misaligned on a 16 byte boundary and vector crosses a 16 byte boundary, 0 otherwise. Entries labeled *S.H.* have data hazards that are always dominated by the structural hazard on the vector memory pipeline.

Writer / Reader	VALU (*vw*)	`l_ai.v` (*vd*), `l_st.v` (*vd*), `l_x.v` (*vd* data), `vext.v` (*vd*)	`vins.s` All registers.
VALU (*vt*/*vd*)	0	0	$\lceil$`vlr`$/8\rceil$
`sbai.v`/`shai.v` (*vd*)	0	*S.H.*	*S.H.*
`swai.v` (*vd*)	$\min(\lceil$`vlr`$/4\rceil, 3)$	*S.H.*	*S.H.*
`s_st.v` (*vd*)	$\min($`vlr`$, 27)$	*S.H.*	*S.H.*
`l_x.v` (*vt* indices)	$\min($`vlr`$+3, 27)$	*S.H.*	*S.H.*
`s_x.v` (*vt* indices)	$\min(2+\lceil$`vlr`$/8\rceil+$`vlr`$, 29)$	*S.H.*	*S.H.*
`s_x.v` (*vd* data)	$\min(2+\lceil$`vlr`$/8\rceil+$`vlr`$, 33)$	*S.H.*	*S.H.*
`vext.s` (*vd*)	0	*S.H.*	*S.H.*
`vext.v` (*vt*), index in *rd* 8-aligned	0	*S.H.*	*S.H.*
`vext.v` (*vt*), index in *rd* 4-aligned	$\min(\lceil$`vlr`$/4\rceil, 3)$	*S.H.*	*S.H.*
`vext.v` (*vt*), index in *rd* not 4-aligned	$1+\min(\lceil$`vlr`$/4\rceil, 3)$	*S.H.*	*S.H.*

Table 15: WAR hazards for vector registers. Timings are given as number of delay cycles. The `vlr` value represents the vector length of the first instruction issued, i.e., the reader. The WAR interlock on `vins.s` does not distinguish between vector registers and so blocks for any ongoing arithmetic operations. The entries labeled *S.H.* have data hazards that are always dominated by the structural hazard on the vector memory pipeline.

Second Writer / First Writer	VALU (*vw*)	`l_ai.v` (*vd*), `l_st.v` (*vd*), `l_x.v` (*vd* data), `vext.v` (*vd*)	`vins.s` All registers.
VALU (*vw*)	0	1	$\lceil$`vlr`$/8\rceil$
`lbai.v`/`lhai.v` (*vd*)	0	0	0
`lwai.v` (*vd*)	$\min(\lceil$`vlr`$/4\rceil, 3)$	*S.H.*	*S.H.*
`l_st.v` (*vd*)	$\min($`vlr`$, 27)$	*S.H.*	*S.H.*
`l_x.v` (*vd* data)	$\min($`vlr`$+3, 30)$	*S.H.*	*S.H.*
`vins.s` (*vd*)	0	*S.H.*	*S.H.*
`vext.v` (*vt*), index in *rd* 8-aligned	0	*S.H.*	*S.H.*
`vext.v` (*vt*), index in *rd* 4-aligned	$\min(\lceil$`vlr`$/4\rceil, 3)$	*S.H.*	*S.H.*
`vext.v` (*vt*), index in *rd* not 4-aligned	$1 + \min(\lceil$`vlr`$/4\rceil, 3)$	*S.H.*	*S.H.*

Table 16: WAW hazards for vector registers. Timings are given as number of delay cycles. The `vlr` value represents the vector length of the first instruction issued. The WAW interlock on `vins.s` does not distinguish between vector registers and so blocks for any ongoing arithmetic operations. The entries labeled *S.H.* have data hazards that are always dominated by the structural hazard on the vector memory pipeline.

10.3.4 Vector Flag Register Data Hazards

There are three vector flag registers implemented as coprocessor 2 control registers: `vcond`, `vovf`, and `vsat`. These flag registers are written by the vector arithmetic pipelines, and read and written by the scalar unit using the MIPS standard `cfc2/ctc2` instructions.

The flag registers have a single bit per vector element, and are written at the rate of 8 bits per cycle by the vector arithmetic units. Flag register writes happen at the end of the vector arithmetic pipeline, in the cycle after the corresponding vector register elements are written. The scalar unit reads flag registers 32 bits at a time, and so scalar flag register reads cannot be chained and must wait for any ongoing vector arithmetic flag write to complete. Similary, the scalar unit writes 32 bits at a time, and to prevent WAW hazards, a scalar flag write must wait for any ongoing vector arithmetic flag write to complete. The `vovf` and `vsat` registers hold sticky bits and new vector arithmetic flag values are OR-ed together with the existing values. The OR occurs in the same cycle as the write and so there are no associated data hazards.

Tables 17–19 summarize the data hazard timings for the vector flag registers. The timings for each of the three flag registers is the same, so only one set of tables is given. The VFLAGW instruction represents any instruction that writes a particular flag register. The only vector arithmetic instructions that write `vcond` are the vector set flag instructions: `flt.__`, `fltu.__`, and `feq.__`. The only vector arithmetic instructions that write `vovf` are the signed integer add and subtract instructions: `add.__` and `sub.__`. The only vector arithmetic instructions to write the `vsat` register are the fixed-point instructions: `fxadd.__`, `fxsub.__`, and `fxmul.__`.

Reader / Writer	`cfc2`
VFLAGW	$\lceil$`vlr`$/8\rceil$
`ctc2`	0

Table 17: RAW hazards for vector flag registers. Timings are given as number of delay cycles.

Writer / Reader	VFLAGW	`ctc2`
`cfc2`	0	0

Table 18: WAR hazards for vector flag registers. Timings are given as number of delay cycles.

Second Writer / First Writer	VFLAGW	`ctc2`
VFLAGW	0	$\lceil$`vlr`$/8\rceil$
`ctc2`	0	0

Table 19: WAW hazards for vector flag registers. Timings are given as number of delay cycles.

10.4 CP0 Timing and Hazards

A `rfe` instruction will take effect on the cycle immediately following the `icinv` instruction.

An `icinv` instruction will cause a cache flush before the instruction fetch of the fourth cycle following.

A value written into a CP0 register by a `mtc0` can be read on the following cycle by a `mfc0`.

A `mtc0` that changes the `cu2`, `cu0`, `im[7:3]`, `kuc`, or `iec` fields in the `status` register takes effect on the second cycle following the instruction. However, an `rfe` instruction may follow directly after a `mtc0` write of the KU/IE stack and will use the new values.

A `mtc0` that changes the `ip5` bit in the `cause` register takes effect on the second cycle following the instruction.

A `mtc0` that writes the `compare` register clears any pending timer interrupt by the second cycle following the instruction.

The counter/timer `count` and `compare` registers are written at the same point in the pipeline. This means that if the `compare` register is written one cycle after the `count` register with a value that is larger by 1 then a timer interrupt will be flagged immediately. Note that intervening SIP activity, interrupts, or cache misses could cause the second register write to be delayed and hence prevent the interrupt occuring for approximately 2^{32} cycles.

10.5 Instruction Cache Miss Timings

T0 has a 1 KB instruction cache (256 instructions) organized as a direct mapped cache holding 64 lines of 4 instructions (16 bytes) each.

Servicing a cache miss takes 3 cycles if the memory port is busy during the fetch stage of the missed instruction. The memory port may be busy due to a SIP memory access, a previously issued vector memory operation, or a scalar memory instruction issued in the last-but-one cycle.

If the memory port is free during the fetch stage, the miss penalty is reduced to 2 cycles.

For example, in the following code sequence the second `lw` uses the memory port and causes the instruction cache miss to take 3 cycles.

```
  .align 4          # Next four instructions on one cache line.
  lw v1, (a0)
  xor a1, a2, a3
  lw t1, 4(a0)      # This instruction uses memory port.
  addiu a1, 1
#-----------------    Cache line boundary.
  bgtz v1, target   # This instruction will have 3 cycle miss penalty.
  sw v1, 4(a0)
```

In the following code, the first `addiu` does not use the memory port so there is only a 2 cycle miss penalty.

```
  .align 4          # Next four instructions on one cache line.
  lw v1, (a0)
  xor a1, a2, a3
  addiu a1, 1       # This instruction does not use memory port.
  lw t1, 4(a0)
#-----------------    Cache line boundary.
  bgtz v1, target   # This will have 2 cycle penalty if not in cache.
  sw v1, 4(a0)
```

Note that branches and jumps never use the memory port, so the target instruction will have a 2 cycle miss penalty if there are no other accesses on the memory port.

```
        jal subroutine    # Does not use memory port.
        addiu a0, t5, t8
        ...
        ...
subroutine:
        lw t0, (a0)       # 2 cycle miss penalty if not in cache.
        ...
```

Instruction cache misses are overlapped with interlocks and stalls. In the following code sequence, the I-cache miss service time of the add is completely hidden by the interlock on the `mfhi` instruction.

```
        mult t0, t1
        b target
        mfhi t0           # Interlocked for 17 cycles.
        ...
target:
        addu t0, t3       # Any cache miss time hidden.
```

The cache miss still needs to steal a single cycle from the memory port for the refill.

11 Pin Out

Table 20 lists the signal names of all active pads for T0.

Name	Direction	Number	Description
System Clock			
clk2xin	I	1	2x clock input.
clkout	O	1	clock output.
Reset			
rstb	I	1	System Reset.
External Interrupts			
extintb[1:0]	I	2	External interrupts.
SIP Port			
tms	I	1	Test mode select.
tdi[7:0]	I	8	Test data in.
tdo[7:0]	O	8	Test data out.
Hardware Performance Monitoring			
hpm[7:0]	O	8	Hardware performance monitor port.
Memory Interface			
a[31:4]	O	28	Address.
nkrwb	O	1	Not killed read/write.
id	O	1	Instruction/data access.
ku	O	1	Kernel/user access.
d[127:0]	I/O	128	Data bus.
weninb[1:0]	I	2	Write enable pulse (1 per 8 bytes).
rw	O	1	Global read/write.
bwenb[15:0]	O	16	Byte write enables.
	Total	208	

Table 20: T0 Signal Pads.

12 Clocking

T0 takes a double frequency input clock, `clk2xin`, that is divided down by 2 internally to ensure a 50% duty cycle. This divided down clock is buffered to form the on-chip clock `phi`. To allow synchronization of external circuitry, `phi` is inverted and sent off chip via the low skew `clkout` pad.

Table 21 gives specifications for the clocking circuitry on T0, and the timing is shown in Figure 24.

`clk2xin`	
Minimum cycle time (ns)	11.2

Table 21: T0 Clock Pad Specifications.

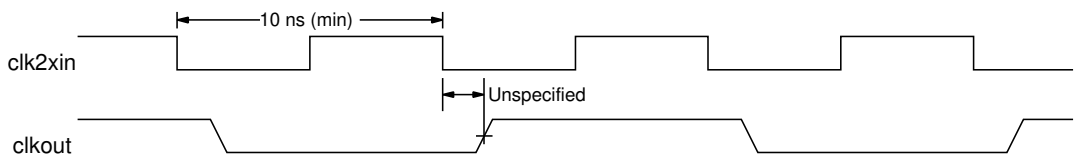

Figure 24: T0 Clock Timing.

13 SIP

The SIP port on T0 is used for chip and board testing, and for data I/O. The SIP port is based on the JTAG standard, with five differences:

- There is no separate test clock (TCLK). T0 SIP uses the internal clock, available on `clkout`.
- No boundary scan instructions are implemented (no EXTEST and no SAMPLE/PRELOAD).
- Data is shifted eight bits per cycle (instead of one bit per cycle) through the scan chain shift registers.
- All data in and data out timing is with respect to the rising edge of `clkout`. (In the JTAG standard, `tdo` is clocked on the falling edge of `clkout`.)
- The port is reset by holding `tms` high for six clock cycles whereupon the TAP state machine enters the *Test-Logic-Reset* state and BYPASS is loaded into the instruction register. This takes one cycle longer than the JTAG standard because T0 SIP has a synchronous reset of the instruction register in the BYPASS state.

13.1 Signal Pins

The T0 SIP signal pins are listed in Table 22. Note the system reset pin, `rstb`, does not affect the SIP port.

Pin name	Direction	Description
`tms`	I	Mode select.
`tdi[7:0]`	I	Data in.
`tdo[7:0]`	O	Data out.

Table 22: T0 SIP signal pins.

13.2 SIP Protocol

The SIP state machine is shown in Figure 25. The transitions are controlled by the value of the `tms` input, indicated by the width of the transition line.

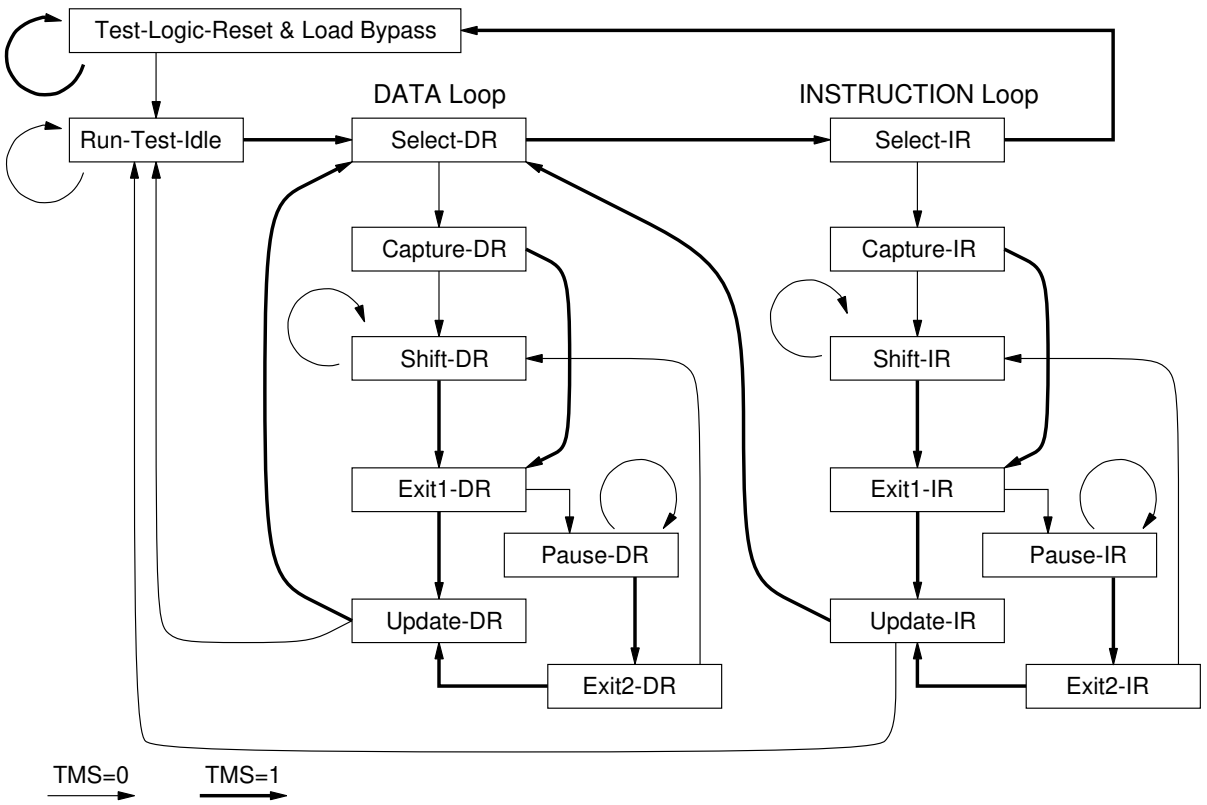

Figure 25: SIP TAP controller states.

The SIP protocol is similar to the JTAG standard, except that eight bits are shifted every cycle in the *Shift-IR* and *Shift-DR* states. Also different is that `tms`, `tdo[7:0]` and `tdi[7:0]` are all specified with respect to the rising edge of `clkout`, as shown in Figure 26.

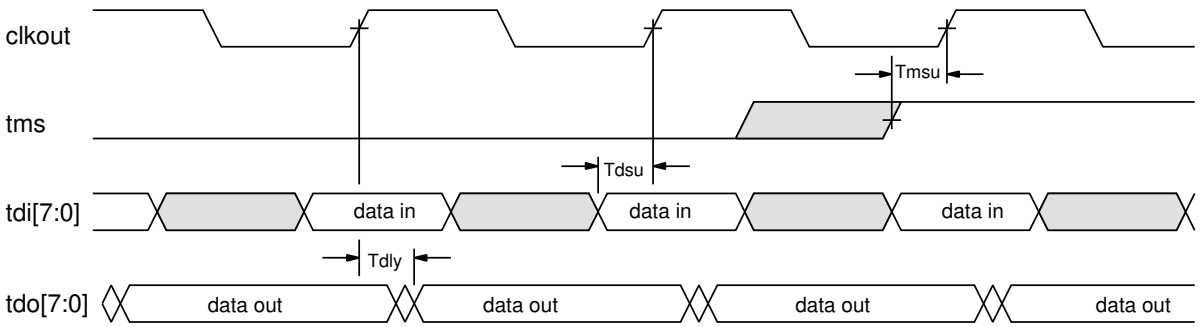

Figure 26: T0 SIP TAP controller timing.

13.3 SIP Shift Registers

SIP shift registers transport data in an 8-bit-serial fashion between `tdi[7:0]` and `tdo[7:0]` and have parallel inputs and outputs. The parallel inputs capture data from within T0, and the parallel outputs are used to write state within T0.

There are only two shift registers in T0, an 8-bit shift register named `regio` and a 160-bit shift register named `memio`, shown in Figure 27. The 8-bit `regio` shift register is used to load the instruction register, and to load various other control registers. The `memio` shift register contains a 32-bit address field and a 128-bit data field and is used to read and write T0 external memory, to write the T0 instruction cache, and to read back the T0 program counter for debugging purposes.

There are two kinds of registers read and written from `regio`, the JTAG instruction register and a number of T0 control registers. The instruction register is loaded from `regio` on the *Scan-IR* loop of the TAP state machine. The other registers are loaded from the `regio` register on the *Scan-DR* loop of the TAP state machine.

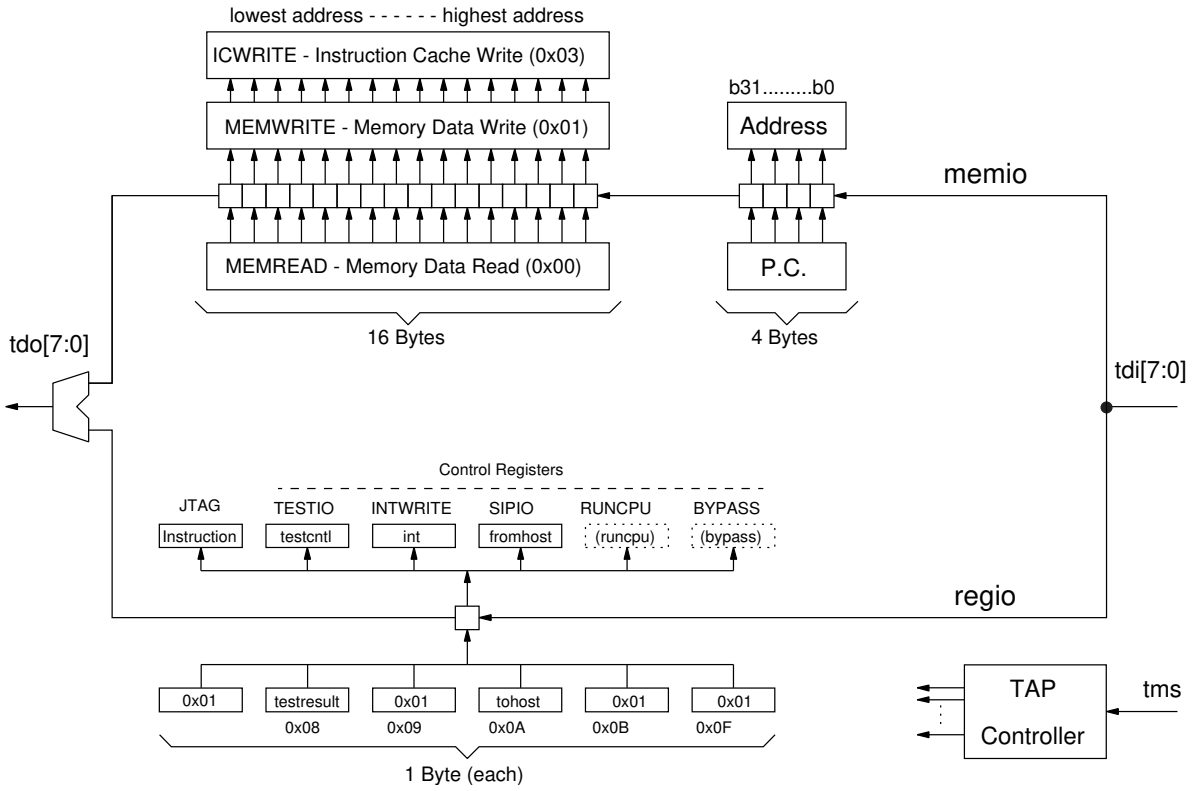

Figure 27: T0 SIP Data Paths.

13.4 SIP instructions

T0 has a 4-bit SIP instruction register, used to specify one of 8 instructions, listed in Table 23 and detailed in the following sections. The upper four bits of the instruction byte are *don't-cares*.

Each instruction specifies which of the two shift registers is used during a *Scan-DR* loop, as well as the operation performed. The low four bits of the value shifted into `regio` appear in the instruction register during the *Update-IR* state. The hex value `0x01` is loaded into `regio` as the TAP controller leaves *Capture-IR*.

Name	Value[3:0]	Shift Register	Description
MEMREAD	`0000`	`memio`	Read memory.
MEMWRITE	`0001`	`memio`	Write memory.
ICWRITE	`0011`	`memio`	Write instruction cache.
TESTIO	`1000`	`regio`	Read/write test registers.
INTWRITE	`1001`	`regio`	Write `int` register.
SIPIO	`1010`	`regio`	Access `fromhost`/`tohost` registers.
RUNCPU	`1011`	`regio`	Run a suspended T0.
BYPASS	`1111`	`regio`	Bypass.

Table 23: T0 SIP instructions.

13.4.1 BYPASS

The BYPASS instruction connects the `regio` shift register between `tdi[7:0]` and `tdo[7:0]`. On entering *Capture-DR*, the hex value `0x01` is loaded into `regio`. No registers are changed on entering *Update-DR*.

13.4.2 MEMREAD

The MEMREAD instruction allows T0 external memory to be read from SIP. The `memio` shift register is connected between `tdi[7:0]` and `tdo[7:0]`. During the *Shift-DR* state, a 32-bit memory read address is shifted into the address field of `memio`. The address field is the first 4 bytes (most significant byte first) of `memio` to be loaded from `tdi[7:0]`. Only bits 31–4 are used to form the external memory address. Entering the *Update-DR* state triggers a memory read cycle from the external memory. The T0 CPU will stall if it attempts a memory access during this SIP memory read cycle. The memory access puts out the read address in the cycle following *Update-DR*, and reads the 16 byte data block from external memory on the next cycle. The timing is shown in Figure 28.

The read block is loaded into the last 128 bits of `memio` on the cycle following the external memory access. At the same time, the current program counter value is loaded into the 32-bit address field

of `memio`. A further data scan is needed to retrieve the read data from `memio`. The first byte to appear from `tdo[7:0]` during *Shift-DR* is the byte with the lowest address. The data is followed by the 32 bits of program counter with the most significant byte shifted out first. The shift out of the program counter value can be omitted if the value is not required.

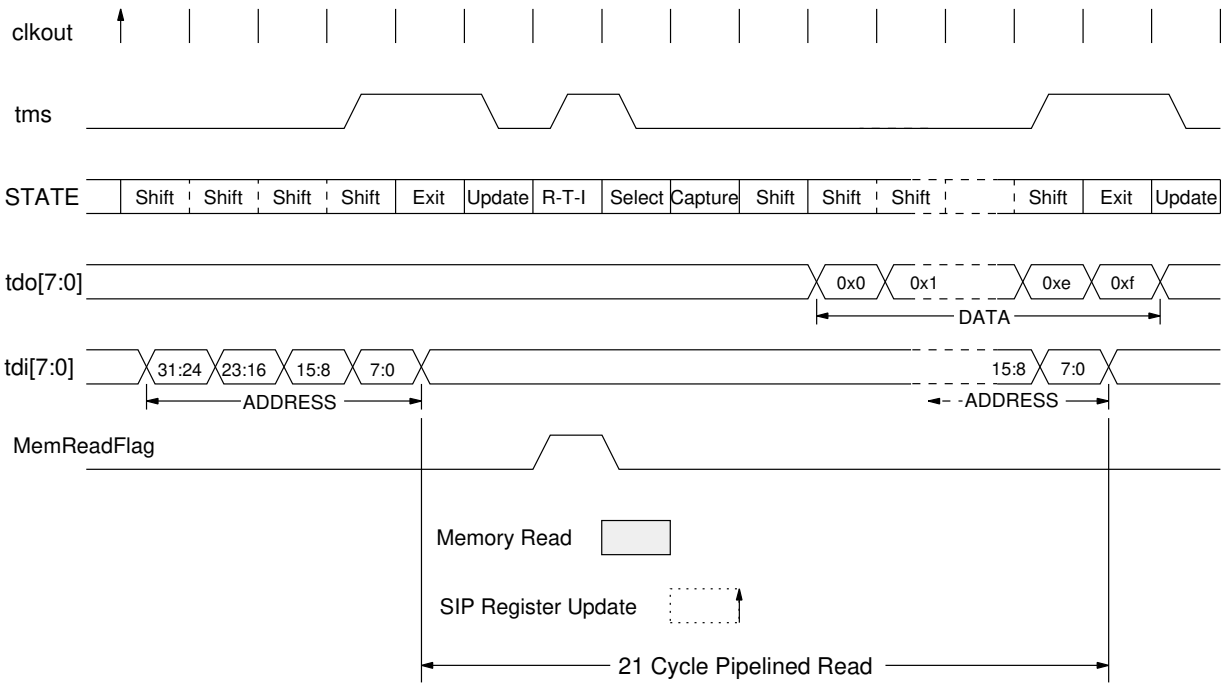

Figure 28: T0 SIP MEMREAD timing.

An isolated memory read will require two scans of `memio`. One to load the 32-bit read address, and a second to retrieve the read data. The second `memio` scan will trigger a second spurious (but harmless) memory read cycle.

Reads of multiple memory blocks can be pipelined, with the next read address shifted into `memio` as the last 32 bits of the previous read cycle's data is shifted out. The fastest block reads are performed in the loop *Select-DR-Scan*, *Capture-DR*, multiple *Shift-DR*s, *Exit1-DR*, *Update-DR*, *Run-Test-Idle*, *Select-DR-Scan*. Note that this gives the required three cycles (in states *Run-Test-Idle*, *Select-DR-Scan* and *Capture-DR*) between *Update-DR* and *Shift-DR* so that the data shifted out is valid. Using this loop, the peak memory read bandwidth is 34 MB/s at 45 MHz. This memory activity can slow the T0 processor down by 4.8% at most.

13.4.3 MEMWRITE

The MEMWRITE instruction allows T0 external memory to be written from the SIP port. The `memio` shift register is connected between `tdi[7:0]` and `tdo[7:0]`. During the *Shift-DR* state, 16 bytes of data (byte at lowest address first) followed by the 32-bit memory write address (most significant byte first) is shifted into `memio`. Entering the *Update-DR* state triggers a memory write cycle from the external memory. The T0 CPU will stall if it attempts a memory access during this SIP memory write cycle. The memory access puts out the write address in the cycle following *Update-DR*, and then writes the 128-bit data value to external memory on the next cycle. The timing of a MEMWRITE access is shown in Figure 29.

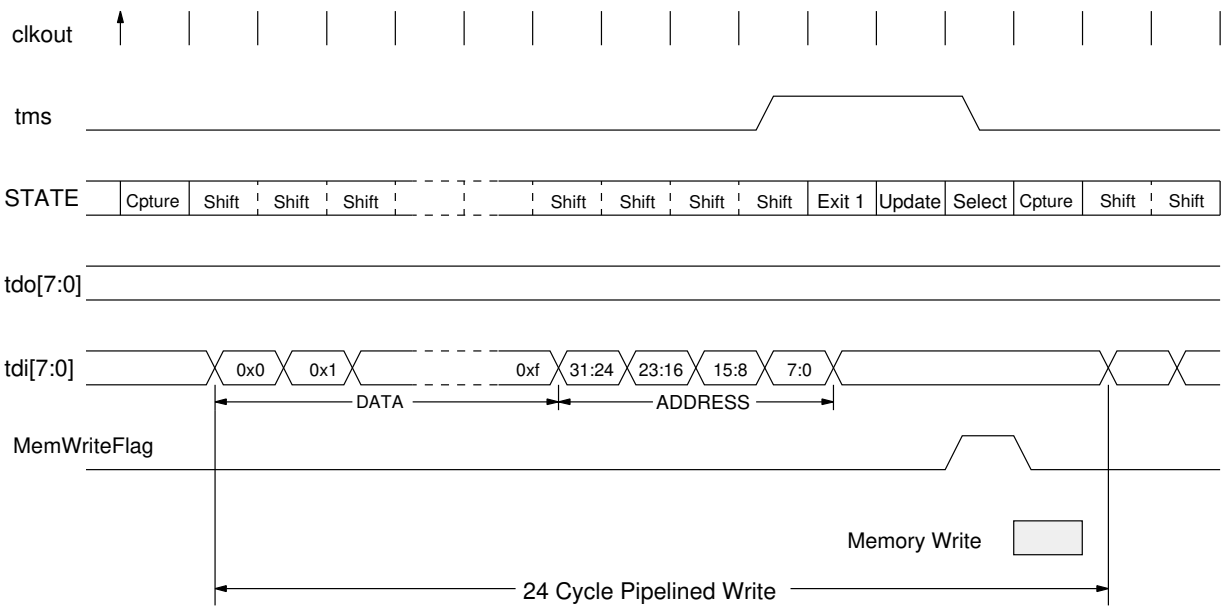

Figure 29: T0 SIP MEMWRITE timing.

The peak memory write speed is given by following the loop *Select-DR-Scan*, *Capture-DR*, multiple *Shift-DR*s, *Exit1-DR*, *Update-DR*, *Select-DR-Scan*. Note that this gives the required two cycles (in states *Select-DR-Scan* and *Capture-DR*) between *Update-DR* and *Shift-DR* so that the data to be written remains valid. Using this loop the peak memory write bandwidth is 30 MB/s at 45 MHz. This memory activity can slow the T0 processor by 4.2% at most.

13.4.4 ICWRITE

The ICWRITE instruction allows the T0 instruction cache to be written from the SIP port. This instruction is used for testing the instruction cache and will destroy the control flow of any executing program. Normally, this instruction (or a sequence of these instructions) will be performed while T0 is held in reset. Note that while T0 is suspended, the next instruction to execute is stalled in the decode stage. Writing the instruction cache while T0 is suspended will not change the instruction that will execute in the first run cycle. In particular, if the first line of the instruction cache is written while T0 is suspended and reset, T0 will execute the old instruction at the reset vector when reset and suspend are deasserted rather than the newly written instruction. To avoid this problem, the instruction cache should be written first with reset asserted but suspend deasserted. Suspend can then be safely asserted before deasserting reset if it is required to single step from the reset vector in a program loaded into the instruction cache.

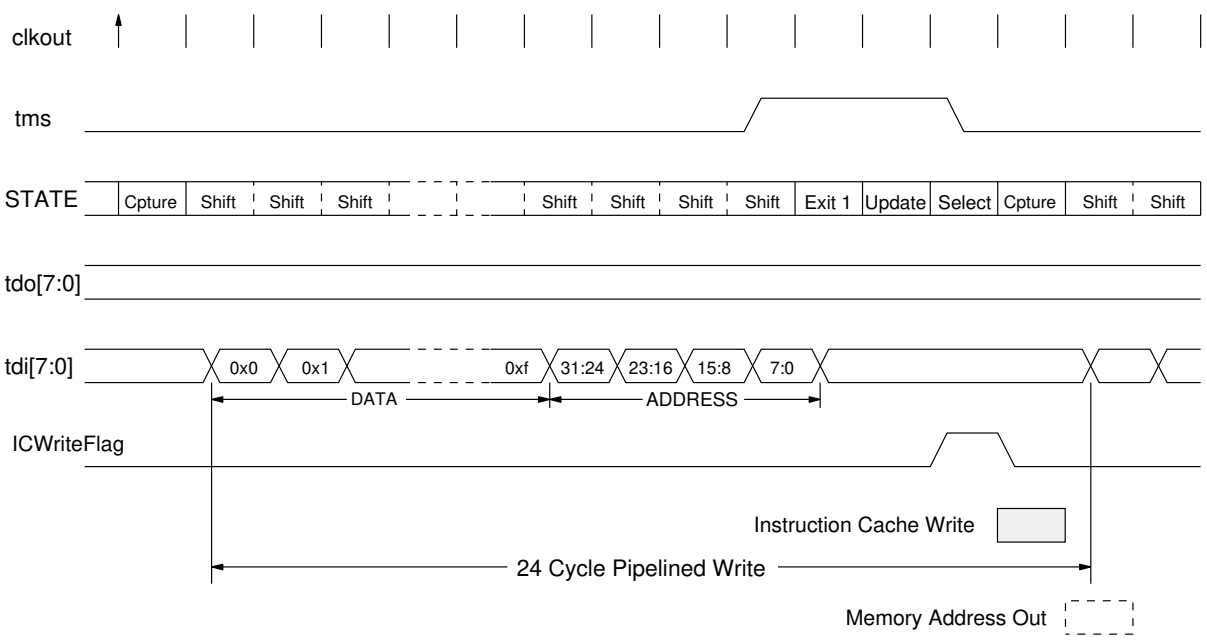

Figure 30: T0 SIP ICWRITE timing.

The `memio` shift register is connected between `tdi[7:0]` and `tdo[7:0]`. During the *Shift-DR* state, 16 bytes of data (byte at lowest address first) followed by the 32-bit cache write address (most significant byte first) is shifted into `memio`. The least significant 4 bits and the most significant 4 bits of the cache write address are ignored. The next least significant 6 bits of the cache write address select the cache line to be written, and the next 18 bits of the address are written to the cache tag field. Entering the *Update-DR* state triggers the cache write cycle. The `memio` address and data values are transferred to the cache on the cycle following *Update-DR*, and written to the cache on the cycle thereafter. The cache write address is output on the T0 address pins on the following cycle.

13.4.5 TESTIO

The TESTIO instruction is used to write the 8-bit `testcntl` register and to read the 8-bit `testresult` register. This instruction connects the `regio` shift register between `tdi[7:0]` and `tdo[7:0]`. On entering *Capture-DR* the value in the `testresult` register is loaded into `regio`. The `testcntl` register is updated with the value of `regio` on entering *Update-DR*.

The format of the `testcntl` register is shown in Figure 31.

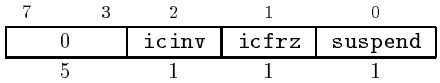

Figure 31: T0 `testcntl` register format.

The `suspend` bit stalls the T0 processor by preventing further instructions from issuing. Instructions that have already been issued complete execution, and any pending instruction cache misses are serviced. T0 requires up to 43 cycles from when `suspend` is asserted in the *Update-DR* state until the processor is guaranteed to be in a quiescent state.

The `icfrz` (instruction cache freeze) bit, is used to lock the current contents of the instruction cache. When this bit is set, the instruction cache always hits regardless of fetch address, and hence acts as an instruction RAM.

The `icinv` (instruction cache invalidate) bit, is used to invalidate the instruction cache. When this bit is set, the instruction cache invalid bits are cleared every cycle and so the instruction cache always misses, causing all instruction fetches to read external memory.

`icfrz` overrides `icinv` if both are asserted.

The format of the `testresult` register is shown in Figure 32.

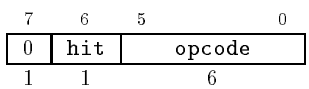

Figure 32: T0 `testresult` register format.

The `opcode[5:0]` and `hit` bits are loaded from the instruction cache after every CPU cycle, except when `suspend` is asserted and the RUNCPU command is not active. This allows the opcode portion of the cache RAM and the output of the tags comparator to be tested.

13.4.6 SIPIO

The SIPIO instruction is used to read and write the 8-bit `tohost` and `fromhost` registers in coprocessor 0. This instruction connects the `regio` shift register between `tdi[7:0]` and `tdo[7:0]`. On entering *Capture-DR*, the value held in the `tohost` register is copied into `regio`. The `fromhost` register is updated with the value of `regio` on entering *Update-DR*.

13.4.7 INTWRITE

The INTWRITE instruction is used to write the 8-bit `int` register. This instruction connects the `regio` shift register between `tdi[7:0]` and `tdo[7:0]`. On entering *Capture-DR* the hex value `0x01` is loaded into `regio`. The `int` register is updated with the value of `regio` on entering *Update-DR*.

The format of the `int` register is shown in Figure 33.

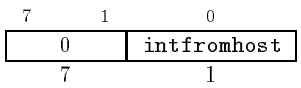

Figure 33: T0 `int` register format.

The `intfromhost` bit is mirrored in the `ip6` bit of the `cause` register and flags a host interrupt to the T0 processor.

13.4.8 RUNCPU

The RUNCPU instruction is only valid when the `suspend` bit is set in the `testcntl` register and is used to temporarily re-enable the issue of further instructions. The RUNCPU instruction connects `regio` shift register between `tdi[7:0]` and `tdo[7:0]`. On entering *Capture-DR*, the hex value `0x01` is loaded into `regio`. No registers are updated with the value of `regio` on entering *Update-DR*.

While RUNCPU is present in the instruction register, for every cycle the TAP state machine is in the *Run-Test-Idle* state the CPU is allowed to issue one more instruction. The issue occurs one cycle after the corresponding *Run-Test-Idle* state. The CPU will not issue an instruction if other interlock conditions would prevent its correct execution.

The RUNCPU instruction also re-enables sampling of the opcode and instruction cache hit signal into the `testresult` register.

13.5 SIP Single Step

T0 allows both the control flow of a program and its effect on memory contents to be examined one instruction at a time over SIP.

While the processor is halted using the `suspend` bit in the `testcntl` register, the RUNCPU instruction can be used to enable instruction issue one cycle at a time. Inbetween each issue cycle, the MEMREAD instruction can be used to read the program counter. If the program counter changes, then the instruction was issued, otherwise some other condition prevented issue. While the CPU is suspended the program counter value that is read is that of the *next* instruction to be executed, not that of the instruction that is interlocked. The MEMREAD and MEMWRITE instructions can be used to examine and modify values in memory inbetween issue cycles.

14 Reset

T0 has a single active low CPU reset signal, `rstb`. This reset is synchronous to the system clock with timings as shown in Table 24.

The reset signal only affects the processor core, SIP is unaffected. This allows the processor to be held in reset mode while program code is downloaded to T0 external memory. Reset must be asserted for at least 6 system clock cycles. While reset the CPU will only execute memory read cycles.

`rstb`	
Setup time	10
Hold time	0

Table 24: T0 Reset Pad Specifications.

When reset is deasserted, the CPU begins execution at the reset vector in kernel mode with interrupts disabled. Refer to Section 7 for further details.

15 External Interrupts

T0 has two synchronous active-low, external interrupt inputs, `extintb[1:0]`. These are sampled with the system clock and must meet the timing specifications shown in Table 25.

`extintb[1:0]`	
Setup time	10
Hold time	0

Table 25: T0 External Interrupt Pad Specifications.

The external interrupts are level triggered and the external interrupt source must wait for a handshake from T0 before deasserting the interrupt signal to ensure the interrupt has been received.

Both external interrupts have lower priority than internal interrupts and `extintb[0]` has higher priority than `extintb[1]`. Each external interrupt has a separate interrupt vector to allow fast custom interrupt handlers. Refer to Section 7 for further details.

16 T0 Hardware Performance Monitor

The T0 HPM facility allows non-intrusive monitoring of T0 activity. The HPM facility provides 8 output pads, `hpm[7:0]`, driven from T0 internal signals. Table 26 lists the information output on the `hpm[7:0]` pads.

Pad	Name	Pipestage
`hpm[0]`	`exception`	**(Xn/Mp)**
`hpm[1]`	`cpumemstall`	**(Xn/Mp)**
`hpm[2]`	`interlock`	**(Dn/Xp)**
`hpm[3]`	`miss`	**(Fn/Dp)**
`hpm[4]`	`vp0busy`	**(X1n/X2p)**
`hpm[5]`	`vp1busy`	**(X1n/X2p)**
`hpm[6]`	`vmpbusy`	**(Mn/Wp)**
`hpm[7]`	`vumemstall`	**(Mn/Wp)**

Table 26: T0 HPM outputs.

16.1 Scalar Unit HPM information

Pads `hpm[3:0]` provide scalar unit information. This information is time aligned and contains pipeline signals for three different instructions in execution. In each case, a set bit indicates that the instruction will be killed in that cycle.

`hpm[0]` reports whether the instruction entering the M stage is taking an exception (including reset).

`hpm[1]` reports whether the instruction entering the M stage took a `cpumemstall` last cycle. `Cpumemstall`s are caused when a scalar memory instruction is stalled because of a host SIP access or an instruction cache refill. A instruction may still take an asynchronous exception (reset or interrupt) even if stalled last cycle.

`hpm[2]` reports whether the instruction entering the X stage was interlocked in the D stage.

`hpm[3]` reports if the instruction entering the D stage was invalid due to a cache miss.

These signals can be used directly to determine what action the processor will take in the next cycle. For example, the `interlock` signal only causes a processor stall if there were no `exception` or `cpumemstall` signalled on the previous instruction, otherwise the interlock is "hidden". Similarly, the `miss` signal only causes a processor stall if there was no `interlock` signalled on the previous instruction, and no `cpumemstall` or `exception` on the instruction before that.

If external hardware delays miss two cycles (call this `miss'`), and interlock one cycle (`interlock'`), then `miss'`, `interlock'`, `cpumemstall`, and `exception` will give the flags for one instruction running down the pipeline. Together with information on previous instructions' signals, this can be used to get per instruction statistics.

16.2 Vector Unit HPM information

Pads `hpm[7:4]` provide vector unit information.

The `vp0busy`, `vp1busy`, and `vmpbusy` signals indicate if the corresponding functional unit was performing useful work that cycle. It does not signal stalled cycles, cycles killed for reset or other exceptions, or instructions that change no state.

The `vumemstall` signal indicates that this cycle experienced a memory stall and so no vector functional unit could make progress. VU memory stalls occur due to host SIP activity or instruction cache refills.

The four bits are aligned to correspond to the same issue cycle, with values output on the **Mn/Wp** edge of the VMP pipeline, and **X1n/X2p** edge of the VP pipelines. For example, `hpm[6:4]` will be zero if `hpm[7]` is set.

16.3 Further Sources of HPM Information

The T0 memory interface `id`, `ku`, and `rw` signals can also be employed to give more information on memory access type. The external `rstb` and `extintb[1:0]` signals can be tapped. External hardware controlling the SIP port can provide further HPM signals corresponding to the SIP activity the processor is currently experiencing. Together with the HPM port it is thus possible to garner the most important dynamic execution statistics in a non-intrusive manner.

17 Memory Interface

T0 has a 128-bit wide pipelined memory interface that supports up to 4 GB of single cycle SRAM. Table 27 lists the signals in the T0 memory interface.

Name	Direction	Number	Description
`a[31:4]`	O	28	Address.
`nkrwb`	O	1	Not killed read/write.
`id`	O	1	Instruction/data access.
`ku`	O	1	Kernel/user access.
`d[127:0]`	I/O	128	Data bus.
`weninb[1:0]`	I	2	Write enable pulse (1 per 8 bytes).
`rw`	O	1	Global read/write.
`bwenb[15:0]`	O	16	Byte write enables.

Table 27: T0 memory interface signals.

Each T0 memory access is pipelined over two cycle, as shown in Figure 34. The address, `a[31:4]`, and access type information, `nkrwb`, `id`, and `ku`, are output on the first cycle. On the second cycle, the write enable signals, `rw` and `bwenb[15:0]`, and the data, `d[127:0]`, are transmitted.

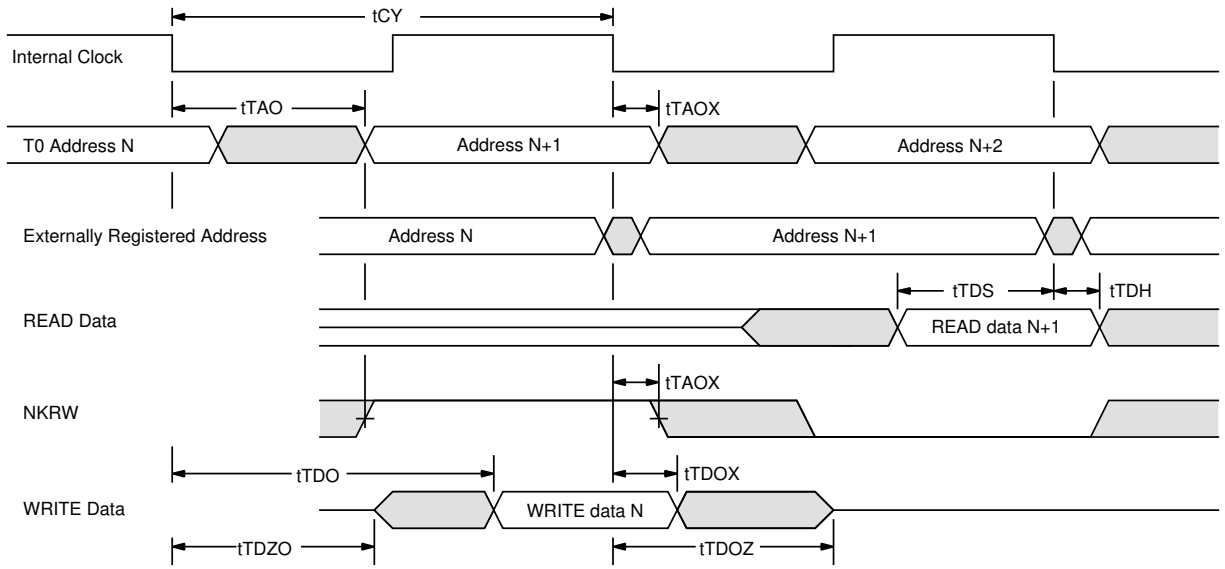

Figure 34: T0 Memory Timing.

The access type information is encoded as shown in Table 28. Note that instruction cache refills occur early in the pipeline before it is known whether the instruction will be executed in kernel or user mode, and so the `ku` value is undefined during instruction cache refills.

nkrwb	id	ku	Access Type
0	0	0	Kernel mode data read
0	0	1	User mode data read
0	1	-	Instruction cache refill
1	0	0	Kernel mode data write
1	0	1	User mode data write
1	1	-	*Impossible*

Table 28: Memory Access Type Encoding.

The `nkrwb` (Not Killed Read Write Bar) signal indicates if the data access may be a write and is produced early in the T0 pipeline. If low, the associated data access in the next cycle will be a read and T0 will have tristated its data output buffers. If high, the associated access may be a write and the external memory should have tristated output buffers. If the write is killed later in the pipeline due to an exception, T0 will tristate its output drivers and the external data bus will be allowed to float for that cycle. T0 will ignore the value on the data bus for that cycle.

The `rw` signal is output on the second cycle of the memory pipeline, and contains a resolved read/write signal that is high for a read or aborted write, and low for a non-aborted write.

The `bwenb[15:0]` signals active low byte write enables for each of the 16 bytes in the data bus. The `weninb[1:0]` input signals shape these byte write enable pulses with `weninb[0]` controlling `bwenb[7:0]` and `weninb[1]` controlling `bwenb[15:8]`. The same pulse will usually be driven to both `weninb[1]` and `weninb[0]`; two copies exist to reduce on chip propagation delays.