SMIPS Processor Specification

6.884 Spring 2005 - Version: 20050215

1 Introduction

SMIPS is the version of the MIPS instruction set architecture (ISA) we'll be using for the processors we implement in 6.884. SMIPS stands for Simple MIPS since it is actually a subset of the full MIPS ISA. The MIPS architecture was one of the first commercial RISC (reduced instruction set computer) processors, and grew out of the earlier MIPS research project at Stanford University. MIPS stood for "Microprocessor without Interlocking Pipeline Stages" and the goal was to simplify the machine pipeline by requiring the compiler to schedule around pipeline hazards including a branch delay slot and a load delay slot. Today, MIPS CPUs are used in a wide range of devices: Casio builds handheld PDAs using MIPS CPUs, Sony uses two MIPS CPUs in the Playstation-2, many Cisco internet routers contain MIPS CPUs, and Silicon Graphics makes Origin supercomputers containing up to 512 MIPS processors sharing a common memory. MIPS implementations probably span the widest range for any commercial ISA, from simple single-issue in-order pipelines to quad-issue out-of-order superscalar processors.

There are several variants of the MIPS ISA. The ISA has evolved from the original 32-bit MIPS-I architecture used in the MIPS R2000 processor which appeared in 1986. The MIPS-II architecture added a few more instructions while retaining a 32-bit address space. The MIPS-II architecture also added hardware interlocks for the load delay slot. In practice, compilers couldn't fill enough of the load delay slots with useful work and the NOPs in the load delay slots wasted instruction cache space. (Removing the branch delay slots might also have been a good idea, but would have required a second set of branch instruction encodings to remain backwards compatible.) The MIPS-III architecture debuted with the MIPS R4000 processor, and this extended the address space to 64 bits while leaving the original 32-bit architecture as a proper subset. The MIPS-IV architecture was developed by Silicon Graphics to add many enhancements for floating-point computations and appeared first in the MIPS R8000 and later in the MIPS R10000. Over the course of time, the MIPS architecture has been widely extended, occasionally in non-compatible ways, by different processor implementors. MIPS Technologies, who now own the architecture, are trying to rationalize the architecture into two broad groupings: MIPS32 is the 32-bit address space version, MIPS64 is the 64-bit address space version. There is also MIPS16, which is a compact encoding of MIPS32 that only uses 16 bits for each instruction. You can find a complete description of the MIPS instruction set at the MIPS Technologies web site [2] or in the book by Kane and Heinrich [3]. The book by Sweetman also explains MIPS programming [4]. Another source of MIPS details and implementation ideas is "Computer Organization and Design: The Hardware/Software Interface" [1].

2 CPU

The SMIPS CPU implements a subset of the 32-bit MIPS-II ISA (all integer operations except trap instructions, misaligned load/stores, and multiprocessor instructions). Figure 1 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.

SMIPS has a single programmer-visible branch delay slot. Loads are fully interlocked, so there is no programmer-visible load delay slot.

Multiply instructions perform 32-bit$\times$32-bit $\rightarrow$ 64-bit signed or unsigned integer multiplies placing the result in the `hi` and `lo` registers. Divide instructions perform a 32-bit/32-bit signed or unsigned 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.

2.1 Operating Modes

The SMIPS CPU has two operating modes: *user* mode and *kernel* mode. The current operating mode is stored in the KUC bit in the system coprocessor (COP0) `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 Unimplemented instructions

Several instructions in the MIPS-II instruction set are not supported by the SMIPS. These instructions are trapped and can be emulated in software by a trap handler.

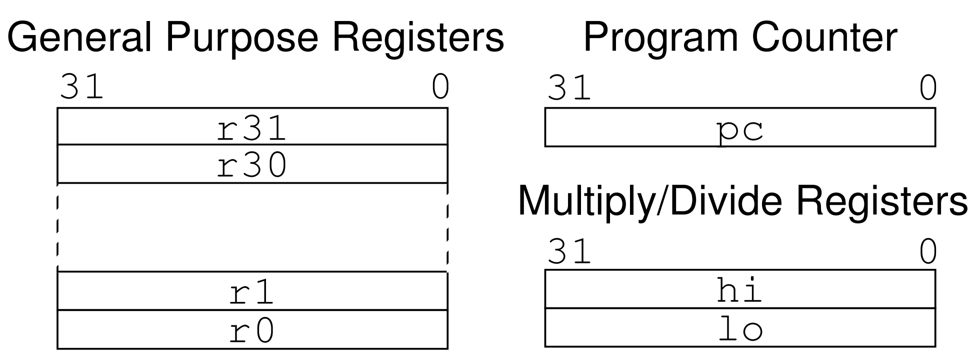

Figure 1: SMIPS CPU registers.

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 SMIPS should avoid generating these instructions, and should instead generate code to perform the misaligned access using multiple aligned accesses.

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

The MIPS-II trap instructions, TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, TNEI, are not implemented. The illegal instruction trap handler can perform the comparison and if the condition is met jump to the appropriate exception routine, otherwise resume 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 (COP1) is not supported. All MIPS-II coprocessor 1 instructions are trapped to allow emulation of floating-point. For higher performance, compilers for SMIPS could directly generate calls to software floating point code libraries rather than emit coprocessor instructions that will cause traps, though this will require modifying the standard MIPS calling convention.

3 System Control Coprocessor (CP0)

The SMIPS system control coprocessor contains a number of registers used for exception handling, communication with a test rig, and the counter/timer. 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
0–7		*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.
15–19		*unused.*
20	`fromhost`	Test input register.
21	`tohost`	Test output register.
22–31		*unused.*

Table 1: CP0 control registers.

3.1 Test Communication Registers

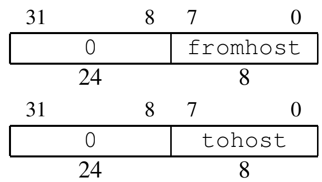

Figure 2: Fromhost and Tohost Register Formats.

There are two registers used for communicating and synchronizing with an external host test system. Typically, these will be accessed over a scan chain. The `fromhost` register is an 8-bit read only register that contains a value written by the host system. The `tohost` register is an 8-bit read/write register that contains a value that can be read back by the host system. The `tohost` register is cleared by reset to simplify synchronization with the host test rig. Their format is shown in Figure 2.

3.2 Counter/Timer Registers

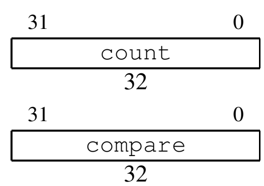

Figure 3: Count and Compare Registers.

SMIPS 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 3.

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.

3.3 Exception Processing Registers

A number of CP0 registers are used for exception processing.

3.3.1 Status Register

31 28	27 16	15 8	7 6	5	4	3	2	1	0
CU	0	IM	0	KUo	IEo	KUp	IEp	KUc	IEc
4	12	8	2	1	1	1	1	1	1

Figure 4: Status Register Format

The `status` register is a 32-bit read/write register formatted as shown in Figure 4. 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 29–31, corresponding to coprocessor's 1, 2, and 3, are permanently wired to 0 as these coprocessors are not available in SMIPS. Coprocessor 0 is always accessible in kernel mode regardless of the setting of bit 28 of the `status` register.

The IM field contains interrupt mask bits. Timer interrupts are disabled by clearing `im7` in bit 15. External interrupts are disabled by clearing `im6` in bit 14. The other bits within the IM field are not used on SMIPS and should be written with zeros. Table 6 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.

3.3.2 Cause Register

31	30	29 28	27 16	15 8	7	6 2	1 0
BD	0	CE	0	IP	0	ExcCode	0
1	1	2	12	8	1	5	2

Figure 5: Cause Register Format

The `cause` register is a 32-bit register formatted as shown in Figure 5. The `cause` register contains information about the type of the last exception and is 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	External 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.

Table 2: 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 external interrupt from the host test setup. The other IP bits are unused in SMIPS and should be ignored when read. Table 6 includes a listing of interrupt bit positions and descriptions.

3.3.3 Exception Program Counter

31 0
epc
32

Figure 6: EPC Register.

`Epc` is a 32-bit read only register formatted as shown in Figure 6. 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.

3.3.4 Bad Virtual Address

31 0
badvaddr
32

Figure 7: BadVAddr Register.

`Badvaddr` is a 32-bit read only register formatted as shown in Figure 7. When a 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.

4 Instruction Encodings

SMIPS uses the standard MIPS instruction set.

4.1 Instruction Formats

There are three basic instruction formats, R-type, I-type, and J-type. These are a fixed 32 bits in length, and must be aligned on a four-byte boundary in memory. An address error exception (AdEF) is generated if the PC is misaligned on an instruction fetch.

R-Type

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

R-type instructions specify two source registers (*rs* and *rt*) and a destination register (*rd*). The 5-bit *shamt* field is used to specify shift immediate amounts and the 6-bit *funct* code is a second opcode field.

I-Type

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16

I-type instructions specify one source register (rs) and a destination register (rt). The second source operand is a sign or zero-extended 16-bit immediate. Logical immediate operations use a zero-extended immediate, while all others use a sign-extended immediate.

J-Type

31 26	25 0
opcode	jump target
6	26

J-type instructions encode a 26-bit jump target address. This value is shifted left two bits to give a byte address then combined with the top four bits of the current program counter.

4.2 Instruction Categories

MIPS instructions can be grouped into several basic categories: loads and stores, computation instructions, branch and jump instructions, and coprocessor instructions.

Load and Store Instructions

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
LB/LH/LW	base	dest	offset
LBU/LHU	base	dest	offset
SB/SH/SW	base	src	offset

Load and store instructions transfer a value between the registers and memory and are encoded with the I-type format. The effective address is obtained by adding register *rs* to the sign-extended immediate. Loads place a value in register *rt*. Stores write the value in register *rt* to memory.

The LW and SW instructions load and store 32-bit register values respectively. The LH instruction loads a 16-bit value from memory and sign extends this to 32-bits before storing into register *rt*. The LHU instruction zero-extends the 16-bit memory value. Similarly LB and LBU load sign and zero-extended 8-bit values into register *rt* respectively. The SH instruction writes the low-order 16 bits of register *rt* to memory, while SB writes the low-order 8 bits.

The effective address must be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit loads/stores and a two-byte boundary for 16-bit loads/store). If not, an address exception (AdEL/AdES) is generated.

The MIPS-I misaligned load/store instructions LWL/LWR/SWL/SWR are not supported in SMIPS. The MIPS-II multiprocessor synchronization instructions (LL,SC,SYNC) are not supported in SMIPS.

Computational Instructions

Computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register *rt* for register-immediate instructions and *rd* for register-register instructions.

There are only eight register-immediate computational instructions.

31–26	25–21	20–16	15–0
opcode	rs	rt	immediate
6	5	5	16
ADDI/ADDIU/SLTI/SLTIU	src	dest	sign-extended immediate
ANDI/ORI/XORI/LUI	src	dest	zero-extended immediate

ADDI and ADDIU add the sign-extended 16-bit immediate to register *rs*. The only difference between ADD and ADDIU is that ADDI generates an arithmetic overflow exception if the signed result would overflow 32 bits. SLTI (set less than immediate) places a 1 in the register *rt* if register *rs* is strictly less than the sign-extended immediate when both are treated as signed 32-bit numbers, else a 0 is written to *rt*. SLTIU is similar but compares the values as unsigned 32-bit numbers. **NOTE: Both ADDIU and SLTIU sign-extend the immediate, even though they operate on unsigned numbers.**

ANDI, ORI, XORI are logical operations that perform bit-wise AND, OR, and XOR on register *rs* and the zero-extended 16-bit immediate and place the result in *rt*.

LUI (load upper immediate) is used to build 32-bit immediates. It shifts the 16-bit immediate into the high-order 16-bits, shifting in 16 zeros in the low order bits, then places the result in register *rt*. The *rs* field must be zero.

NOTE: Shifts by immediate values are encoded in the R-type format using the *shamt* field.

Arithmetic R-type operations are encoded with a zero value (SPECIAL) in the major opcode. All operations read the *rs* and *rt* registers as source operands and write the result into register *rd*. The 6-bit *funct* field selects the operation type from ADD, ADDU, SUB, SUBU, SLT, SLTU, AND, OR, XOR, and NOR.

31–26	25–21	20–16	15–11	10–6	5–0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	src1	src2	dest	0	ADD/ADDU/SUB/SUBU
SPECIAL=0	src1	src2	dest	0	SLT/SLTU
SPECIAL=0	src1	src2	dest	0	AND/OR/XOR/NOR

ADD and SUB perform add and subtract respectively, but signal an arithmetic overflow if the result would overflow the signed 32-bit destination. ADDU and SUBU are identical to ADD/SUB except no trap is created on overflow. SLT and SLTU performed signed and unsigned compares respectively, writing 1 to *rd* if $rs < rt$, 0 otherwise. AND, OR, XOR, and NOR perform bitwise logical operations. **NOTE: NOR *rd, rx, rx* performs a logical inversion (NOT) of register *rx*.**

Shift instructions are also encoded using R-type instructions with the SPECIAL major opcode. The operand that is shifted is always register *rt*. Shifts by constant values (SLL/SRL/SRA) have the shift amount encoded in the *shamt* field. Shifts by variable values (SLLV/SRLV/SRAV) take the shift amount from the bottom five bits of register *rs*. SLL/SLLV are logical left shifts, with zeros shifted into the least significant bits. SRL/SRLV are logical right shifts with zeros shifted into the most significant bits. SRA/SRAV are arithmetic right shifts which shift in copies of the original sign bit into the most significant bits.

31–26	25–21	20–16	15–11	10–6	5–0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	0	src	dest	shift	SLL/SRL/SRA
SPECIAL=0	shift	src	dest	0	SLLV/SRLV/SRAV

Multiply and divide instructions target the `hi` and `lo` registers and are encoded as R-type instructions under the SPECIAL major opcode. Multiply instructions take two 32-bit operands in registers *rs* and *rt* and store their 64-bit product in registers `hi` and `lo`. MULT performs a signed multiplication while MULTU performs an unsigned multiplication. DIV and DIVU perform signed and unsigned divides of register *rs* by register *rt* placing the quotient in `lo` and the remainder in `hi`. Divides by zero do not cause a trap. A software check can be inserted if required.

31–26	25–21	20–16	15–11	10–6	5–0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	src1	src2	0	0	MULT/MULTU/DIV/DIVU
SPECIAL=0	0	0	dest	0	MFHI/MFLO
SPECIAL=0	src	0	0	0	MTHI/MTLO

The values calculated by a multiply or divide instruction are retrieved from the `hi` and `lo` registers using the MFHI (move from `hi`) and MFLO (move from `lo`) instructions, which write register *rd*. MTHI (move to `hi`) and MTLO (move to `lo`) instructions are also provided to allow the multiply registers to be written with the value in register *rs* (these instructions are used to restore user state after a context swap).

Jump and Branch Instructions

Jumps and branches can change the control flow of a program. The MIPS ISA has a single architected branch delay slot, so the instruction following a jump or branch is always executed (except for annulling branch likely instructions when the branch is not taken).

Absolute jumps (J) and jump and link (JAL) instructions use the J-type format. The 26-bit jump target is concatenated to the high order four bits of the program counter of the delay slot, then shifted left two bits to form the jump target address (using Verilog notation, the target address is `{pcds[31:28],target[25:0],2'b0}`, where `pcds` is the PC of the instruction in the delay slot.). JAL stores the address of the instruction following the delay slot (PC+8) into register `r31`.

31–26	25–0
opcode	jump target
6	26
J/JAL	offset

The indirect jump instructions, JR (jump register) and JALR (jump and link register), use the R-type encoding under a SPECIAL major opcode and jump to the address contained in register *rs*. JALR writes the link address into register *rd*. **NOTE: JR is redundant given that JALR with a destination of `r0` has the same effect. This seems to be a bug in the MIPS ISA!**

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	src	0	0	0	JR
SPECIAL=0	src	0	dest	0	JALR

All branch instructions use the I-type encoding. The 16-bit immediate is sign-extended, shifted left two bits, then added to the address of the instruction in the delay slot (PC+4) to give the branch target address.

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
BEQ/BNE	src1	src2	offset
BLEZ/BGTZ	src	0	offset
REGIMM	src	BLTZ/BGEZ	offset
REGIMM	src	BLTZAL/BGEZAL	offset

BEQ and BNE compare two registers and take the branch if they are equal or unequal respectively. BLEZ and BGTZ compare one register against zero, and branch if it is less than or equal to zero, or greater than zero, respectively. BLTZ and BGEZ examine the sign bit of the register *rs* and branch if it is negative or positive respectively. The BLTZAL and BGEZAL variants store a link register value in `r31` *regardless* of whether the branch is taken or not. The link value is the address of the instruction following the delay slot (PC+8).

The MIPS-II extension to the ISA added a set of "branch likely" instructions (BEQL/BNEL/BLEZL/BGTZL/BLTZL/BGEZL/BLTZALL/BGEZALL) that annul the instruction in the branch delay slot if the branch is not taken. These give the compiler more flexibility in filling branch delay slots. If no instruction preceding the branch can be moved in to the delay slot, the instruction at the branch target can always be moved into the delay slot (the branch is changed to a branch likely and the target address is advanced by one instruction to skip over the original target).

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
BEQL/BNEL	src1	src2	offset
BLEZL/BGTZL	src	0	offset
REGIMM	src	BLTZL/BGEZL	offset
REGIMM	src	BLTZALL/BGEZALL	offset

System Coprocessor Instructions

The MTC0 and MFCO instructions access the control registers in coprocessor 0, transferring a value from/to the coprocessor register specified in the *rd* field to/from the CPU register specified in the *rt* field. It is important to note that the coprocessor register is always in the *rd* field and the CPU register is always in the *rt* field regardless of which register is the source and which is the destination.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
COP0	MF	dest	cop0src	0	0
COP0	MT	src	cop0dest	0	0
COP0	CO	0	0	0	RFE
COP0	CO	0	0	0	ICINV

The restore from exception instruction, RFE, pops the top value of the interrupt and kernel/user status register stack, restoring the previous values. This is normally used in the delay slot of a JR instruction that restores the PC.

The ICINV (instruction cache invalidate) instruction flushes the instruction cache to ensure any preceding writes to instruction memory will be observed by the processor fetch unit.

4.3 Encoding Charts

Figures 8 and 9 detail the opcode decoding for SMIPS. 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.

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	δ	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 8: SMIPS 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 3	CO0							

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 9: SMIPS COP0 Instruction Encodings.

4.4 Instruction Tables

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Load and Store Instructions						
100000	base	dest	signed offset			LB rt, offset(rs)
100001	base	dest	signed offset			LH rt, offset(rs)
100011	base	dest	signed offset			LW rt, offset(rs)
100100	base	dest	signed offset			LBU rt, offset(rs)
100101	base	dest	signed offset			LHU rt, offset(rs)
101000	base	dest	signed offset			SB rt, offset(rs)
101001	base	dest	signed offset			SH rt, offset(rs)
101011	base	dest	signed offset			SW rt, offset(rs)
I-Type Computational Instructions						
001000	src	dest	signed immediate			ADDI rt, rs, signed-imm.
001001	src	dest	signed immediate			ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate			SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate			SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate			ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate			ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate			XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate			LUI rt, zero-ext-imm.
R-Type Computational Instructions						
000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rshamt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rshamt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100000	ADD rd, rs, rt
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100010	SUB rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt

Table 3: Instruction listing for SMIPS.

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Multiply/Divide Instructions						
000000	00000	00000	dest	00000	010000	MFHI rd
000000	rs	00000	00000	00000	010001	MTHI rs
000000	00000	00000	dest	00000	010010	MFLO rd
000000	rs	00000	00000	00000	010011	MTLO rs
000000	src1	src2	00000	00000	011000	MULT rs, rt
000000	src1	src2	00000	00000	011001	MULTU rs, rt
000000	src1	src2	00000	00000	011010	DIV rs, rt
000000	src1	src2	00000	00000	011011	DIVU rs, rt
Jump and Branch Instructions						
000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
010100	src1	src2	signed offset			BEQL rs, rt, offset
010101	src1	src2	signed offset			BNEL rs, rt, offset
010110	src	00000	signed offset			BLEZL rs, offset
010111	src	00000	signed offset			BGTZL rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset
000001	src	00010	signed offset			BLTZL rs, offset
000001	src	00011	signed offset			BGEZL rs, offset
000001	src	10000	signed offset			BLTZAL rs, offset
000001	src	10001	signed offset			BGEZAL rs, offset
000001	src	10010	signed offset			BLTZALL rs, offset
000001	src	10011	signed offset			BGEZALL rs, offset
System Coprocessor (COP0) Instructions						
010000	00000	dest	cop0src	00000	000000	MFC0 rt, rd
010000	00100	src	cop0dest	00000	000000	MTC0 rt, rd
010000	10000	00000	00000	00000	010000	RFE
010000	10000	00000	00000	00000	010100	ICINV

Table 4: Instruction listing for SMIPS.

5 Addressing and Memory Protection

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

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 10.

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 loads and stores respectively. For faulting stores, no data memory will be written at the faulting address.

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

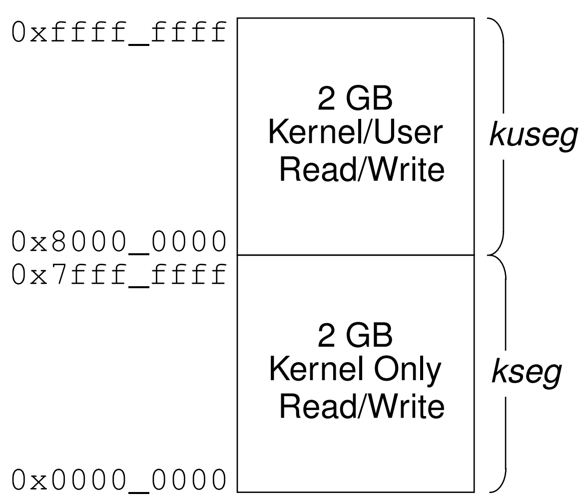

Figure 10: SMIPS virtual address space.

6 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.

If more than one of these classes of event occurs on a given cycle, reset has highest 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 two locations as shown in Table 5. Reset has a separate vector from all other exceptions and interrupts.

Vector Address	Cause
`0x0000_1000`	Reset
`0x0000_1100`	Exceptions and internal interrupts.

Table 5: SMIPS Reset, Exception, and Interrupt Vectors.

6.1 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. All other state is undefined.

A typical reset sequence is shown in Figure 11.

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

    # 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 11: Example reset sequence.

6.2 Interrupts

The two interrupts possible on SMIPS are listed in Table 6 in order of decreasing priority.

Vector	ExcCode	Mnemonic	IM/IP index	Description
Highest Priority				
`0x0000_1100`	0	Hint	6	Tester interrupt.
`0x0000_1100`	2	Tint	7	Timer interrupt.
Lowest Priority				

Table 6: SMIPS Interrupts.

All SMIPS 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 external interrupt flag IP6 can be written by the host test system over a scan interface. Usually a protocol over the host scan interface informs the host that it can clear down the interrupt flag.

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.

When an interrupt is taken, the PC is set to the interrupt 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. The `exccode` field in the `cause` register is set to indicate the type of interrupt.

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.

6.3 Synchronous Exceptions

Synchronous exceptions are listed in Table 7 in order of decreasing priority.

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.

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.
4	AdEL	Address or misalignment error on load.
5	AdES	Address or misalignment error on store.
		Lowest Priority

Table 7: SMIPS Synchronous Exceptions.

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 ADDI, ADD, and SUB instructions.

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

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, second edition, February 1997. ISBN 1558604286.

[2] MIPS Technologies Inc. MIPS32 architecture for programmers, 2002. `http://www.mips.com/publications/processor_architecture.html`

[3] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 2nd edition, September 1991. ISBN 0135904722.

[4] D. Sweetman. *See MIPS Run*. Morgan Kaufmann, April 1999. ISBN 1558604103.