`c.lt.d FRsrc1, FRsrc2` *Compare Less Than Double*
`c.lt.s FRsrc1, FRsrc2` *Compare Less Than Single*
Compare the floating point double in register `FRsrc1` against the one in `FRsrc2` and set the condition flag true if the first is less than the second.

`cvt.d.s FRdest, FRsrc` *Convert Single to Double*
`cvt.d.w FRdest, FRsrc` *Convert Integer to Double*
Convert the single precision floating point number or integer in register `FRsrc` to a double precision number and put it in register `FRdest`.

`cvt.s.d FRdest, FRsrc` *Convert Double to Single*
`cvt.s.w FRdest, FRsrc` *Convert Integer to Single*
Convert the double precision floating point number or integer in register `FRsrc` to a single precision number and put it in register `FRdest`.

`cvt.w.d FRdest, FRsrc` *Convert Double to Integer*
`cvt.w.s FRdest, FRsrc` *Convert Single to Integer*
Convert the double or single precision floating point number in register `FRsrc` to an integer and put it in register `FRdest`.

`div.d FRdest, FRsrc1, FRsrc2` *Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2` *Floating Point Divide Single*
Compute the quotient of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †
Load the floating float double (single) at `address` into register `FRdest`.

`mov.d FRdest, FRsrc` *Move Floating Point Double*
`mov.s FRdest, FRsrc` *Move Floating Point Single*
Move the floating float double (single) from register `FRsrc` to register `FRdest`.

`mul.d FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Single*
Compute the product of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`neg.d FRdest, FRsrc` *Negate Double*
`neg.s FRdest, FRsrc` *Negate Single*
Negate the floating point double (single) in register `FRsrc` and put it in register `FRdest`.

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †
Store the floating float double (single) in register `FRdest` at `address`.

`sub.d FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Double*
`sub.s FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Single*
Compute the difference of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

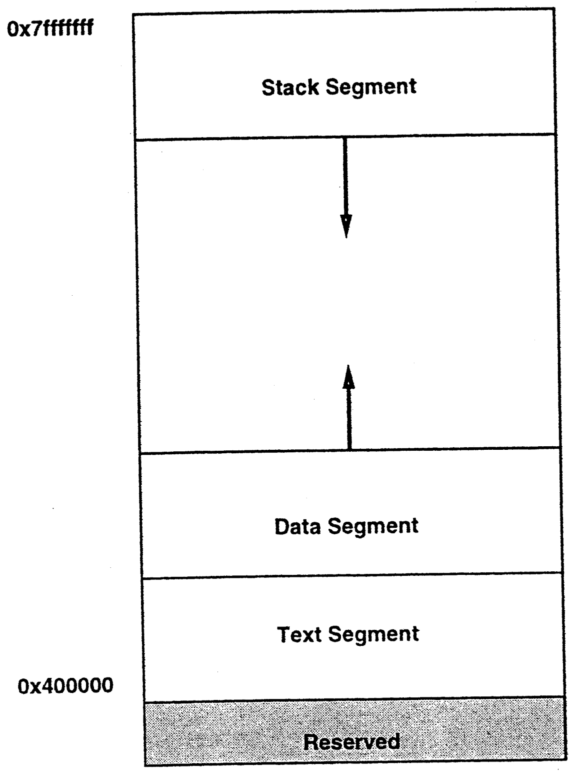

Figure 5: Layout of memory.

2.12 Exception and Trap Instructions

`rfe` *Return From Exception*
Restore the Status register.

`syscall` *System Call*
Register `$v0` contains the number of the system call (see Table 1) provided by SPIM.

`break n` *Break*
Cause exception *n*. Exception 1 is reserved for the debugger.

`nop` *No operation*
Do nothing.

3 Memory Usage

The organization of memory in MIPS systems is conventional. A program's address space is composed of three parts (see Figure 5).

At the bottom of the user address space (0x400000) is the text segment, which holds the instructions for a program.

Above the text segment is the data segment (starting at 0x10000000), which is divided into two parts. The static data portion contains objects whose size and address are known to the

41

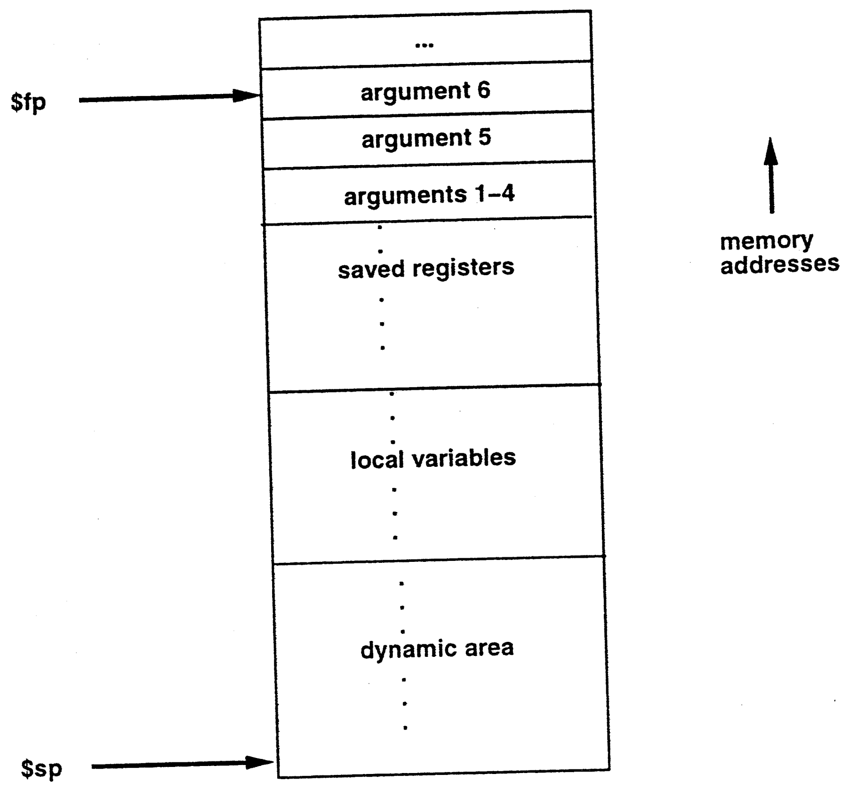

Figure 6: Layout of a stack frame. The frame pointer points just below the last argument passed on the stack. The stack pointer points to the last word in the frame.

compiler and linker. Immediately above these objects is dynamic data. As a program allocates space dynamically (i.e., by `malloc`), the `sbrk` system call moves the top of the data segment up.

The program stack resides at the top of the address space (0x7fffffff). It grows down, towards the data segment.

4 Calling Convention

The calling convention described in this section is the one used by *gcc*, not the native MIPS compiler, which uses a more complex convention that is slightly faster.

Figure 6 shows a diagram of a stack frame. A frame consists of the memory between the frame pointer (`$fp`), which points to the word immediately after the last argument passed on the stack, and the stack pointer (`$sp`), which points to the last word in the frame. As typical of Unix systems, the stack grows down from higher memory addresses, so the frame pointer is above stack pointer.

The following steps are necessary to effect a call:

1. Pass the arguments. By convention, the first four arguments are passed in registers `$a0`–`$a3` (though simpler compilers may choose to ignore this convention and pass all arguments via the stack). The remaining arguments are pushed on the stack.

2. Save the caller-saved registers. This includes registers `$t0`–`$t9`, if they contain live values at the call site.

Y2

3. Execute a `jal` instruction.

Within the called routine, the following steps are necessary:

1. Establish the stack frame by subtracting the frame size from the stack pointer.

2. Save the callee-saved registers in the frame. Register `$fp` is always saved. Register `$ra` needs to be saved if the routine itself makes calls. Any of the registers `$s0`–`$s7` that are used by the callee need to be saved.

3. Establish the frame pointer by adding the stack frame size - 4 to the address in `$sp`.

Finally, to return from a call, a function places the returned value into `$v0` and executes the following steps:

1. Restore any callee-saved registers that were saved upon entry (including the frame pointer `$fp`).

2. Pop the stack frame by adding the frame size to `$sp`.

3. Return by jumping to the address in register `$ra`.

5 Input and Output

In addition to simulating the basic operation of the CPU and operating system, SPIM also simulates a memory-mapped terminal connected to the machine. When a program is "running," SPIM connects its own terminal (or a separate console window in `xspim`) to the processor. The program can read characters that you type while the processor is running. Similarly, if SPIM executes instructions to write characters to the terminal, the characters will appear on SPIM's terminal or console window. One exception to this rule is control-C: it is not passed to the processor, but instead causes SPIM to stop simulating and return to command mode. When the processor stops executing (for example, because you typed control-C or because the machine hit a breakpoint), the terminal is reconnected to SPIM so you can type SPIM commands. To use memory-mapped IO, `spim` or `xspim` must be started with the `-mapped_io` flag.

The terminal device consists of two independent units: a *receiver* and a *transmitter*. The receiver unit reads characters from the keyboard as they are typed. The transmitter unit writes characters to the terminal's display. The two units are completely independent. This means, for example, that characters typed at the keyboard are not automatically "echoed" on the display. Instead, the processor must get an input character from the receiver and re-transmit it to echo it.

The processor accesses the terminal using four memory-mapped device registers, as shown in Figure 7. "Memory-mapped" means that each register appears as a special memory location. The Receiver Control Register is at location 0xffff0000; only two of its bits are actually used. Bit 0 is called "ready": if it is one it means that a character has arrived from the keyboard but has not yet been read from the receiver data register. The ready bit is read-only: attempts to write it are ignored. The ready bit changes automatically from zero to one when a character is typed at the keyboard, and it changes automatically from one to zero when the character is read from the receiver data register.

Bit one of the Receiver Control Register is "interrupt enable". This bit may be both read and written by the processor. The interrupt enable is initially zero. If it is set to one by the

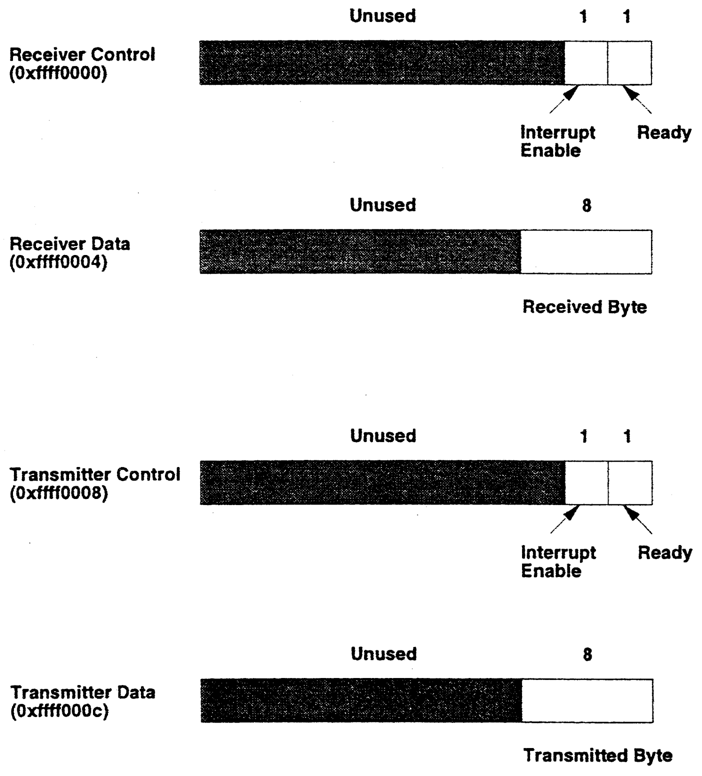

Figure 7: The terminal is controlled by four device registers, each of which appears as a special memory location at the given address. Only a few bits of the registers are actually used: the others always read as zeroes and are ignored on writes.

44

processor, an interrupt is requested by the terminal on level zero whenever the ready bit is one. For the interrupt actually to be received by the processor, interrupts must be enabled in the status register of the system coprocessor (see Section 2).

Other bits of the Receiver Control Register are unused: they always read as zeroes and are ignored in writes.

The second terminal device register is the Receiver Data Register (at address 0xffff0004). The low-order eight bits of this register contain the last character typed on the keyboard, and all the other bits contain zeroes. This register is read-only and only changes value when a new character is typed on the keyboard. Reading the Receiver Data Register causes the ready bit in the Receiver Control Register to be reset to zero.

The third terminal device register is the Transmitter Control Register (at address 0xffff0008). Only the low-order two bits of this register are used, and they behave much like the corresponding bits of the Receiver Control Register. Bit 0 is called "ready" and is read-only. If it is one it means the transmitter is ready to accept a new character for output. If it is zero it means the transmitter is still busy outputting the previous character given to it. Bit one is "interrupt enable"; it is readable and writable. If it is set to one, then an interrupt will be requested on level one whenever the ready bit is one.

The final device register is the Transmitter Data Register (at address 0xffff000c). When it is written, the low-order eight bits are taken as an ASCII character to output to the display. When the Transmitter Data Register is written, the ready bit in the Transmitter Control Register will be reset to zero. The bit will stay zero until enough time has elapsed to transmit the character to the terminal; then the ready bit will be set back to one again. The Transmitter Data Register should only be written when the ready bit of the Transmitter Control Register is one; if the transmitter isn't ready then writes to the Transmitter Data Register are ignored (the write appears to succeed but the character will not be output).

In real computers it takes time to send characters over the serial lines that connect terminals to computers. These time lags are simulated by SPIM. For example, after the transmitter starts transmitting a character, the transmitter's ready bit will become zero for a while. SPIM measures this time in instructions executed, not in real clock time. This means that the transmitter will not become ready again until the processor has executed a certain number of instructions. If you stop the machine and look at the ready bit using SPIM, it will not change. However, if you let the machine run then the bit will eventually change back to one.