Appendix A

MIPS32 Integer Instruction Set Architecture (ISA)

A.1 General features

MIPS32 uses a flat memory model with a 32-bit address. The integer ISA defines one data type, the 32-bit integer. The CPU uses a register-register (load/store) operation model — all ALU operands have at least one source and one destination in the registers.

MIPS32 contains 32 physical integer registers, each 32 bits wide. Integer registers are named \$0, \$1, $\cdots$, \$31 and are addressed as 00000 to 11111 in the register address space. Reading register \$0 always returns zero — Reg[\$0] = 0 (constant). Writes to \$0 are legal but have no effect. All other physical registers are identical (no special purpose registers).

MIPS32 uses three basic addressing modes: register, immediate and memory. The memory addressing mode (displacement) can implement register deferred (by specifying a zero displacement) and absolute (by using \$0 that always returns 0).

A.2 MIPS32 assembly programming

The restricted scope of RISC machine languages can make assembly code and compiler output difficult to read. Assemblers for MIPS32 employ macros that simplify reading and reading assembly code. Macros are converted to MPIS32 machine code by the assembler. For example, the assembly code

```
move $ra, $zero
```

is translated to

```
add $31, $0, $0
```

where `$zero` is a symbol for `$0` and `$ra` is a symbol for `$31`, the register used by the `jal target` instruction to save the return address from a function call.

A.3 Integer registers

Although MIPS32 registers \$1, $\cdots$, \$31 are physically equivalent, programmers use a set of conventions assigning physical registers to symbolic names that represent traditional roles is assembly programming. Compilers and assemblers accept symbolic names as a form of macro and convert them to physical register names.

Symbolic Name	Physical Register	Programming Use
`$zero`	`$0`	Constant 0
`$at`	`$1`	Reserved for assembler
`$v0 -- $v1`	`$2 -- $3`	Value of result
`$a0 -- $a3`	`$4 -- $7`	Arguments passed to function
`$t0 -- $t9`	`$8 -- $15 , $24 -- $25`	Temporary — not saved in function call
`$s0 -- $s7`	`$16 -- $23`	Saved — saved in function call
`$k0 -- $k1`	`$26 -- $27`	Kernel — reserved for system calls
`$gp`	`$28`	Global data pointer
`$sp`	`$29`	Stack pointer
`$fp`	`$30`	Frame pointer
`$ra`	`$31`	Return address

A.4 MIPS instruction set

The table below shows the machine language instructions in the MIPS32 ISA. In the table, the following abbreviations are used:

d	destination register
s	source register
t	source register
imm	16-bit constant (immediate) sign extended to 32 bits
target	offset to target address
off	offset added to register value s to form memory address
u	suffix indictes unsigned operation

MIPS32 Instructions

Instruction	Operands	Description
add	d, s, t	d ← s+t (signed)
addu	d, s, t	d ← s+t (unsigned)
addi	d, s, imm	d ← s+imm (signed)
addiu	d, s, imm	d ← s+imm (unsigned)
and	d, s, t	d ← bitwise AND of s with t
andi	d, s, imm	d ← bitwise AND of s with imm

beq	s, t, target	branch if s == t
bgez	s, target	Branch if the signed integer in register s is ≥ 0
bltz	s, target	Branch if the signed integer in register s is ≤ 0
bne	s, t, target	branch if s != t
div	s, t	lo $\leftarrow$ s div t hi $\leftarrow$ s mod t signed operands
divu	s, t	lo $\leftarrow$ s div t hi $\leftarrow$ s mod t unsigned operands
j	target	after a delay of one machine cycle, PC $\leftarrow$ offset to target address
lb	d, off(s)	d $\leftarrow$ Sign-extended byte from memory address s+off off is 16-bit signed
lbu	d, off(s)	d $\leftarrow$ Zero-extended byte from memory address s+off off is 16-bit signed
lh	d, off(s)	t $\leftarrow$ Sign-extended halfword from memory address s+off off is 16-bit signed
lhu	d, off(s)	t $\leftarrow$ Zero-extended halfword from memory address s+off off is 16-bit signed
lui	d, imm	upper two bytes of $t $\leftarrow$ two byte imm lower two bytes of $t $\leftarrow$ 0x0000
lw	d, off(s)	d $\leftarrow$ Word from memory address s+off off is 16-bit signed
mfhi	d	d $\leftarrow$ hi Move From Hi
mflo	d	d $\leftarrow$ lo Move From Lo
mult	s, t	hi / lo $\leftarrow$ s * t signed operands
multu	s, t	hi / lo $\leftarrow$ s * t unsigned operands
nop	—	no operation
nor	d, s, $0	d $\leftarrow$ bitwise NOT of s
nor	d, s, t	d $\leftarrow$ bitwise NOR of s with t
or	d, s, $0	d $\leftarrow$ s
or	d, s, t	d $\leftarrow$bitwise OR of s with t
ori	d, $0, imm	d $\leftarrow$ zero-extended imm
ori	d, s, imm	d $\leftarrow$ s OR zero-extended imm

sb	d, off(s)	byte at off+s $\leftarrow$ low-order byte from register $d. off is 16-bit signed
sh	d, off(s)	two bytes at off+s $\leftarrow$ two low-order bytes from register $d. off is 16-bit signed
sll	$0, $0, 0	no operation
sll	d, s, shft	d $\leftarrow$ logical left shift of s by shft positions where $0 \leq$ shft < 32
slt	d, s, t	if s $<$ t d $\leftarrow 1$ else d $\leftarrow 0$ signed operands
slti	d, s, imm	if s $<$ imm d $\leftarrow 1$ else d $\leftarrow 0$ signed operands
sltiu	d, s, imm	if s $<$ imm d $\leftarrow 1$ else d $\leftarrow 0$ unsigned operands
sltu	d, s, t	if s $<$ t d $\leftarrow 1$ else d $\leftarrow 0$ unsigned operands
sra	d, s, shft	d $\leftarrow$ arithmetic right shift of s by shft positions where $0 \leq$ shft < 32
srl	d, s, shft	d $\leftarrow$ logical right shift of s by shft positions where $0 \leq$ shft < 32
sub	d, s, t	d $\leftarrow$ s - t with overflow trap
subu	d, s, t	d $\leftarrow$ s - t no overflow trap
sw	d, off(s)	Word at memory address (s+off) $\leftarrow$ $t off is 16-bit signed
syscall	—	call OS service with parameters passed by register
xor	d, s, t	d $\leftarrow$ bitwise exclusive or of s with t
xori	d, s, imm	d $\leftarrow$ bitwise exclusive or of s with imm

A.5 Pseudo-instructions

Pseudo-instructions are instructions macros for use in assembly programs. Macros are converted to MIPS machine instructions by the assembler.

Instruction	Description
blt $8, $9, label	slt $1, $8, $9 bne $1, $0, label
li $8, imm	if imm $>$ 0x0000FFFF lui $1, upper 16 bits of imm ori $8, $1, lower 16 bits of imm else ori $8, $1, lower 16 bits of imm

abs $1, $2	addu $1, $2, $0
	bgez $2, 8 (offset=8 $\rightarrow$ skip 'sub' instruction)
	sub $1, $2, $0

move $1, $2	add $1, $2, $0
la $a0, address	lui $1, upper 16 bits of address (to upper 16 bits of $1). ori $a0, $1, displacement in data section
b target	bgez $0, target
lw $t0, address	lui $1, upper 16 bits of address lw $8 , disp($1) where disp = lower 16 bits of address

A.6 C language calling conventions

Entry into a function in C — including main — requires a unique stack frame for that instance. The structure of the stack frame is:

Stack frame of calling function	
Padding	Insures frame is multiple of 8 bytes
Return address	`$ra` used to return to calling function
Saved `$fp`	Frame pointer of calling function
Local variables	Variables defined in called function
Padding	Insures local variable frame is multiple of 8 bytes
Saved registers	Values of `$s0` $\cdots$ `$s7` used by calling function
Passed arguments	Copied from stack of calling function
Padding	
User stack	For stack implemented in called function

The calling function:

1. Pushes parameters for the function call onto its user stack (at the bottom of its stack frame).
2. Computes the addresses of these parameters into registers `$a0`, `$a1`, `$a2`, `$a3`
3. Performs `jal called_function` which saves the address of the return instruction in `$ra` = `$31`, and jumps to the instruction at label `called_function`.

The called function:

1. Points `$sp` = `$29` to the bottom of the stack frame, to allocate space for saved registers, passed parameters, and local variables.
2. Saves `$ra` = `$31` and `$fp` = `$30` near the top of the stack frame.
3. Sets `$fp` = `$sp` for use in the called function.
4. Writes initialized values of local variables into stack frame using addresses `disp($sp)`.
5. Executes function code.
6. Restores saved values of `$fp` and `$sp`.
7. Returns to calling function by jumping to the address in register `$ra` = `$31`.

A simple function receiving no parameters or saved registers, and defining two local integer variables initialized to 5 and 6, will have the form:

```
addiu $sp, $sp, -16
sw $ra, 12($sp)
sw $fp, 8($sp)
move $fp, $sp
li $v0, 6
sw $v0, 0($fp)
li $v0, 5
sw $v0, 4($fp)
⟨body of code⟩
move $sp, $fp
addiu $sp, $sp, 16
j $ra
```

At the end of program run, the loader places the OS code for program exit (10 in Linux) into register `$v0` = `$2` and executes the `syscall` machine instruction to return allocated program resources to the OS.

A.7 The data segment

Program code is stored and run from a text segment in the memory space allocated to the program. Large sections of data may be stored in a separate data section, and linked to the code with a pointer (address). A section of data may be copied to the stack frame by invoking the `memcpy` macro.

This macro is invoked as:

```
addiu $v0, $fp, displacement (pointer to destination)
la $v1, label (pointer to source data)
li $a0 , number of bytes to copy
move $a2, $a0
move $a1, $v1
move $a0, $v0
jal memcpy
⟨fallthrough instruction⟩
```

The `memcpy` macro copies the data from the source to destination and executes `j $ra` to return to ⟨`fallthrough instruction`⟩.