Introduction

These two sides of A4 should give you everything you need to know to start you off programming in MIPS assembly language. If not, then you've found a bug in the documentation :-)

As it happens, the Altera DE2 development board doesn't actually have a MIPS processor; in fact it doesn't have a processor at all. What it does have is a Cyclone II FPGA, and because an FPGA is programmable hardware, you can run anything you like on it – including a MIPS processor that's been written in Verilog – and that's how there's a MIPS processor running on the board.

The DE2 board is also capable of automatically programming the FPGA with the contents of some EEPROM when the board is turned on, which means the MIPS processor, and its default program (a boot loader) starts running as soon as the board is turned on.

The MIPS processor does only implement a subset of the MIPS instruction set (there's no floating point, and a few other things are missing too), but it does come with support for lots of other things instead: hardware multiply & divide, 16KB of L1 cache, 8MB of SDRAM, keyboard, serial port, VGA, and sound support.

Registers

The MIPS processor has 32 32-bit registers. Their names, numbers, uses, and whether the callee must preserve them across a function call are detailed in the table below:

Name	Number	Use	Callee must preserve
$zero	$0	constant 0	N/A
$at	$1	assembly temporary	no
$v0 - $v1	$2 - $3	function returns	no
$a0 - $a3	$4 - $7	function arguments	no
$t0 - $t7	$8 - $15	temporaries	no
$s0 - $s7	$16 - $23	saved temporaries	**yes**
$t8 - $t9	$24 - $25	temporaries	no
$k0 - $k1	$26 - $27	kernel use	no
$gp	$28	global pointer	**yes**
$sp	$29	stack pointer	**yes**
$fp	$30	frame pointer	**yes**
$ra	$31	return address	N/A

Example

```
        .file 1 "fib.s"
        .previous
        .text                           # The text section - where your code goes
        .align 2                        # align to 4 bytes (since 1 << 2 = 4)
        .ent main                       # the main() function
main:
        addiu $sp,$sp,-8                # $sp = $sp + -8
        sw $16,0($sp)                   # save register $16 to memory address $sp + 0
        sw $31,4($sp)                   # save register $31 to memory address $sp + 4

        li $16,1                        # $16 = 1
main.loop:
        .set noreorder                  # the set directives tell the assembler...
        .set nomacro                    #   not to put a nop in the branch delay slot
        jal serialWriteNumber           # jump and link to serialWriteNumber
        move $4,$16                     # $4 = $16 (instruction in the branch delay slot)

        bne $16,$0,main.loop            # if ($16 != $0) goto main.loop;
        addiu $16,$16,1                 # $16 = $16 + 1 (instruction in the branch delay slot)
        .set macro                      # we turn the default behaviour back on
        .set reorder

        lw $16,0($sp)                   # load register $16 from memory address $sp + 0
        lw $31,4($sp)                   # load register $31 from memory address $sp + 4
        .set noreorder    .set nomacro
        j $31                           # jump to address stored in register $31
        addiu $sp,$sp,8
        .set macro        .set reorder
        .end main
        .align 2                        # align to 4 bytes (since 1 << 2 = 4)
        .ent showoff                    # the showoff() function
showoff:
        la $4,errortext                 # load register $4 with the address of errortext
        jal serialWriteString

showoff.loop:
        b showoff.loop                  # loop forever...
        .end showoff                    #   the assembler will put a nop in the branch delay slot
        .data                           # The data section
        .align 2                        # align to 4 bytes (since 1 << 2 = 4)
errortext: .ascii "foo\015\012\000" # char* errortext = "foo\r\n";
        .size initedglobalchar, 1       # align defaults to 1 byte
initedglobalchar:  .byte 97             # char initedglobalchar = 97;
        .align 1                        # align to 2 bytes (since 1 << 1 = 2)
        .size initedglobalshort, 2
initedglobalshort: .half 4919           # short initedglobalshort = 0x1337;
        .align 2                        # align to 4 bytes (since 1 << 2 = 4)
        .size initedglobalint, 4
initedglobalint:   .word -559038737 # int initedglobalint = 0xdeadbeef;
        .align 2                        # align to 4 bytes (since 1 << 2 = 4)
        .size d, 8
d:   .half 1   .byte 2   .space 1   # struct { short a; char b; int c; } d = {1, 2, 3};
     .word 3                            #   note the space to make the int aligned properly
# The comm stuff
        .comm globalarray,12,4          # int globalarray[3];
                                        # note that the 12 is the length in bytes...
                                        #   and the 4 is the alignment
        .comm globalchar,1,1            # char globalchar;
        .comm globalshort,2,2           # short globalshort;
        .comm globalint,4,4             # int globalint;
```

I-type instructions

instruction	detail	limitations
`addi $rt, $rs, imm` `addiu $rt, $rs, imm`	$\$rt = \$rs + sign\_extend(imm)$ $\$rt = \$rs + sign\_extend(imm)$	$-32768 \leq imm \leq 32767$
`andi $rt, $rs, imm`	$\$rt = \$rs \ \& \ zero\_extend(imm)$	$0 \leq imm \leq 65535$

`beq $rs, $rt, label`
`bgez $rs, label`
`bgtz $rs, label`
`blez $rs, label`
`bltz $rs, label`
`bne $rs, $rt, label`
	if($\$rs == \$rt$) $PC = label$
	if($\$rs \geq 0$) $PC = label$
	if($\$rs > 0$) $PC = label$
	if($\$rs \leq 0$) $PC = label$
	if($\$rs < 0$) $PC = label$
	if($\$rs \neq \$rt$) $PC = label$
		the label must be between -32768 and 32767 inclusive instructions away from the instruction in the branch delay slot

`lb $rt, label`
`lb $rt, label + offset`
`lb $rt, offset($rs)`
	$\$rt = sign\_extend(memory[label])$
	$\$rt = sign\_extend(memory[label + offset])$
	$\$rt = sign\_extend(memory[\$rs + offset])$	

`lbu $rt, label`
`lbu $rt, label + offset`
`lbu $rt, offset($rs)`
	$\$rt = zero\_extend(memory[label])$
	$\$rt = zero\_extend(memory[label + offset])$
	$\$rt = zero\_extend(memory[\$rs + offset])$	

`lh $rt, label`
`lh $rt, label + offset`
`lh $rt, offset($rs)`
	$\$rt = sign\_extend(memory[label])$
	$\$rt = sign\_extend(memory[label + offset])$
	$\$rt = sign\_extend(memory[\$rs + offset])$
		$address \ \% \ 2 = 0$

`lhu $rt, label`
`lhu $rt, label + offset`
`lhu $rt, offset($rs)`
	$\$rt = zero\_extend(memory[label])$
	$\$rt = zero\_extend(memory[label + offset])$
	$\$rt = zero\_extend(memory[\$rs + offset])$
		$address \ \% \ 2 = 0$

`lui $rt, imm`	$\$rt = imm << 16$	$0 \leq imm \leq 65535$

`lw $rt, label`
`lw $rt, label + offset`
`lw $rt, offset($rs)`
	$\$rt = memory[label]$
	$\$rt = memory[label + offset]$
	$\$rt = memory[\$rs + offset]$
		$address \ \% \ 4 = 0$

`ori $rt, $rs, imm`	$\$rt = \$rs \mid zero\_extend(imm)$	$0 \leq imm \leq 65535$

`sb $rt, label`
`sb $rt, label + offset`
`sb $rt, offset($rs)`
	$memory[label] = \$rt$
	$memory[label + offset] = \$rt$
	$memory[\$rs + offset] = \$rt$	

`sh $rt, label`
`sh $rt, label + offset`
`sh $rt, offset($rs)`
	$memory[label] = \$rt$
	$memory[label + offset] = \$rt$
	$memory[\$rs + offset] = \$rt$	

`sll $rd, $rt, sa`	$\$rd = \$rt << sa$	$0 \leq sa \leq 31$
`slt $rt, $rs, imm` `sltu $rt, $rs, imm`	$\$rt = \$rs < sign\_extend(imm) \ ? \ 1 : 0$ $\$rt = \$rs < sign\_extend(imm) \ ? \ 1 : 0$	$-32768 \leq imm \leq 32767$
`sra $rd, $rt, sa` `srl $rd, $rt, sa`	$\$rd = \$rt >> sa$ $\$rd = \$rt >>> sa$	$0 \leq sa \leq 31$

`sw $rt, label`
`sw $rt, label + offset`
`sw $rt, offset($rs)`
	$memory[label] = \$rt$
	$memory[label + offset] = \$rt$
	$memory[\$rs + offset] = \$rt$	

`xori $rt, $rs, imm`	$\$rt = \$rs \ \hat{} \ zero\_extend(imm)$	$0 \leq imm \leq 65535$

J-type instructions

instruction	detail	limitations
`j label`	$PC = label$	
`jal label`	$\$31 = PC + 8; PC = label$	

Pseudo instructions

instruction	detail	limitations
`b label`	$PC = label$	the label must be between -32768 and 32767 inclusive instructions away from the instruction in the branch delay slot
`bge $rs, $rt, label`	if($\$rs \geq \$rt$) $PC = label$	
`bgeu $rs, $rt, label`	if($\$rs \geq \$rt$) $PC = label$	
`bgt $rs, $rt, label`	if($\$rs > \$rt$) $PC = label$	
`bgtu $rs, $rt, label`	if($\$rs > \$rt$) $PC = label$	
`ble $rs, $rt, label`	if($\$rs \leq \$rt$) $PC = label$	
`bleu $rs, $rt, label`	if($\$rs \leq \$rt$) $PC = label$	
`blt $rs, $rt, label`	if($\$rs < \$rt$) $PC = label$	
`bltu $rs, $rt, label`	if($\$rs < \$rt$) $PC = label$	
`la $rt, label`	$\$rt = label$	
`li $rt, imm`	$\$rt = imm$	
`move $rd, $rs`	$\$rd = \$rs$	
`nop`		

R-type instructions

instruction	detail	limitations
`add $rd, $rs, $rt` `addu $rd, $rs, $rt`	$\$rd = \$rs + \$rt$ $\$rd = \$rs + \$rt$	
`and $rd, $rs, $rt`	$\$rd = \$rs \ \& \ \$rt$	
`div $rs, $rt` `divu $rs, $rt`	$HI = \$rt \ \% \ \$rt; LO = \$rs \ / \ \$rt$ $HI = \$rt \ \% \ \$rt; LO = \$rs \ / \ \$rt$	
`j $rs` `jal $rs`	$PC = \$rs$ $\$31 = PC + 8; PC = \$rs$	
`mfhi $rd` `mflo $rd`	$\$rd = HI$ $\$rd = LO$	
`mthi $rs` `mtlo $rs`	$HI = \$rs$ $LO = \$rs$	
`mul $rd, $rs, $rt`	$\$rd = \$rs * \$rt$	
`mult $rs, $rt` `multu $rs, $rt`	$HI = (\$rs * \$rt) << 32; LO = \$rs * \$rt$ $HI = (\$rs * \$rt) << 32; LO = \$rs * \$rt$	
`nor $rd, $rs, $rt`	$\$rd = !(\$rs \mid \$rt)$	
`or $rd, $rs, $rt`	$\$rd = \$rs \mid \$rt$	
`sll $rd, $rt, $rs`	$\$rd = \$rt << \$rs$	
`slt $rd, $rs, $rt` `sltu $rd, $rs, $rt`	$\$rd = \$rs < \$rt \ ? \ 1 : 0$ $\$rd = \$rs < \$rt \ ? \ 1 : 0$	
`sra $rd, $rt, $rs` `srl $rd, $rt, $rs`	$\$rd = \$rt >> \$rs$ $\$rd = \$rt >>> \$rs$	
`sub $rd, $rs, $rt` `subu $rd, $rs, $rt`	$\$rd = \$rs - \$rt$ $\$rd = \$rs - \$rt$	
`xor $rd, $rs, $rt`	$\$rd = \$rs \ \hat{} \ \$rt$	

Comments

All instructions treat the registers as signed integers (stored in two's complement), unless they end in 'u', in which case they treat the registers as unsigned integers.
The instruction following a branch or jump instruction is always executed, before the branch or jump is actually taken.