Function_declare:

function [int1] [int2]	.text .align 2 .global function .type @function function: add sp,sp,-stk sw ra stk-4(sp)
	stk = (int2 / 4 +1) * 16
end function	.size function, .-function
	stk = 0
f_main [0] [17] …… end f_main	.size f, .-f .text .align 2 .global main .type main, @function main: add sp,sp,-80 sw ra,76(sp) …… .size main, .-main
	stk = (17 / 4 + 1) * 16 = 80

Global_var_declare:

global_var = malloc int	.comm global_var,int*4,4
v2 = malloc 800012	.comm v2,3200048,4
v2 = malloc 800012	.comm v2,800012,8
	.global global_var .section .sdata .align 2 .type global_var, @object .size global_var, 4 global_var: .word int
v0 = 0	.global v0 .section .sdata .align 2 .type v0, @object .size v0, 4 v0: .word 0

ops

reg = integer	li reg,integer
reg1 = reg2 op reg3	---------------------------------------
reg1 = reg2 + reg3	add reg1,reg2,reg3
reg1 = reg2 - reg3	sub
reg1 = reg2 * reg3	mul
reg1 = reg2 / reg3	div
%	rem
<	slt
>	sgt
&&	
||	or reg1,reg2,reg3 snez reg1,reg1
!=	xor reg1,reg2,reg3 snez reg1,reg1
==	
reg1 = reg2 op integer	---------------------------------------
+	add reg1,reg2,integer
<	slti reg1,reg2,integer
reg1 = reg2	mv
reg1 [int] = reg2	sw
reg1 = reg2 [int]	lw
if reg1 op reg2 goto Label	---------------------------------------
<	blt reg1,reg2,.label
>	bgt reg1,reg2,.label
!=	bne reg1,reg2,.label
==	beq reg1,reg2,.label
<=	ble reg1,reg2,.label
>=	ble reg2,reg1,.label
goto label	j label
label :	.label:
call function	call finction
store reg int	sw reg,int*4(sp)
load int reg	lw reg,int*4(sp)
load global_var reg	lui reg,%hi(global_var) lw reg,%lo(global_var)(reg)
loadaddr int reg	add reg,sp,int*4
loadaddr global_var reg	lui reg,%hi(global_var) add reg,reg,%lo(global_var)
return	lw ra,stk-4(sp) add sp,sp,stk jr ra

RISC-V Assembly Programmer's Manual

Copyright and License Information

Command-Line Arguments

I think it's probably better to beef up the binutils documentation rather than duplicating it here.

Registers

ISA and ABI register names for X, F, and CSRs.

Addressing

Addressing formats like %pcrel_lo(). We can just link to the RISC-V PS ABI document to describe what the relocations actually do.

Instruction Set

Links to the various RISC-V ISA manuals that are supported.

Instructions

Here we can just link to the RISC-V ISA manual.

Instruction Aliases

ALIAS line from opcodes/riscv-opc.c

Pseudo Ops

Both the RISC-V-specific and GNU .-prefixed options.

The following table lists assembler directives:

Directive	Arguments	Description
.align	integer	align to power of 2 (alias for .p2align)
.file	"filename"	emit filename FILE LOCAL symbol table
.globl	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
.local	symbol_name	emit symbol_name to symbol table (scope LOCAL)
.comm	symbol_name,size,align	emit common object to .bss section
.common	symbol_name,size,align	emit common object to .bss section
.ident	"string"	accepted for source compatibility
.section	[{.text,.data,.rodata,.bss}]	emit section (if not present, default .text) and make current
.size	symbol, symbol	accepted for source compatibility

Directive	Arguments	Description
.text		emit .text section (if not present) and make current
.data		emit .data section (if not present) and make current
.rodata		emit .rodata section (if not present) and make current
.bss		emit .bss section (if not present) and make current
.string	"string"	emit string
.asciz	"string"	emit string (alias for .string)
.equ	name, value	constant definition
.macro	name arg1 [, argn]	begin macro definition \argname to substitute
.endm		end macro definition
.type	symbol, @function	accepted for source compatibility
.option	{rvc,norvc,pic,nopic,push,pop}	RISC-V options
.byte		8-bit comma separated words
.2byte	expression [, expression]*	16-bit comma separated words (unaligned)
.4byte	expression [, expression]*	32-bit comma separated words (unaligned)

Directive	Arguments	Description
.8byte	expression [, expression]*	64-bit comma separated words (unaligned)
.half	expression [, expression]*	16-bit comma separated words (naturally aligned)
.word	expression [, expression]*	32-bit comma separated words (naturally aligned)
.dword	expression [, expression]*	64-bit comma separated words (naturally aligned)
.dtprelword	expression [, expression]*	32-bit thread local word
.dtpreldword	expression [, expression]*	64-bit thread local word
.sleb128	expression	signed little endian base 128, DWARF
.uleb128	expression	unsigned little endian base 128, DWARF
.p2align	p2,[pad_val=0],max	align to power of 2
.balign	b,[pad_val=0]	byte align
.zero	integer	zero bytes

The following table lists assembler relocation expansions:

Assembler Notation	Description	Instruction / Macro
%hi(symbol)	Absolute (HI20)	lui

Assembler Notation	Description	Instruction / Macro
%lo(symbol)	Absolute (LO12)	load, store, add
%pcrel_hi(symbol)	PC-relative (HI20)	auipc
%pcrel_lo(label)	PC-relative (LO12)	load, store, add
%tprel_hi(symbol)	TLS LE "Local Exec"	auipc
%tprel_lo(label)	TLS LE "Local Exec"	load, store, add
%tprel_add(offset)	TLS LE "Local Exec"	add

Labels

Text labels are used as branch, unconditional jump targets and symbol offsets. Text labels are added to the symbol table of the compiled module.

```
loop:
        j loop
```

Numeric labels are used for local references. References to local labels are suffixed with 'f' for a forward reference or 'b' for a backwards reference.

```
1:
        j 1b
```

Absolute addressing

The following example shows how to load an absolute address:

```
.section .text
.globl _start
```

```
_start:
            lui a1,      %hi(msg)      # load msg(hi)
            addi a1, a1,  %lo(msg)      # load msg(lo)
            jalr ra, puts
2:          j2b

.section .rodata
msg:
            .string "Hello World\n"
```
which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:    000005b7          lui      a1,0x0
                           0: R_RISCV_HI20   msg
   4:    00858593          addi     a1,a1,8 # 8 <.L21>
                           4: R_RISCV_LO12_I  msg
```

Relative addressing

The following example shows how to load a PC-relative address:

```
.section .text
.globl _start
_start:
1:          auipc a1,    %pcrel_hi(msg) # load msg(hi)
            addi  a1, a1, %pcrel_lo(1b)  # load msg(lo)
            jalr ra, puts
2:          j2b

.section .rodata
msg:
            .string "Hello World\n"
```
which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:    00000597          auipc    a1,0x0
                           0: R_RISCV_PCREL_HI20      msg
   4:    00858593          addi     a1,a1,8 # 8 <.L21>
                           4: R_RISCV_PCREL_LO12_I     .L11
```

Load Immediate

The following example shows the `li` psuedo instruction which is used to load immediate values:

```
.section .text
.globl _start
_start:

.equ CONSTANT, 0xcafebabe

       li a0, CONSTANT
```

which generates the following assembler output as seen by objdump:

```
0000000000000000 <_start>:
   0:    00032537           lui          a0,0x32
   4:    bfb50513           addi     a0,a0,-1029
   8:    00e51513           slli     a0,a0,0xe
   c:    abe50513           addi     a0,a0,-1346
```

Load Address

The following example shows the `la` psuedo instruction which is used to load symbol addresses:

```
.section .text
.globl _start
_start:


       la a0, msg


.section .rodata
msg:
             .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:    00000517           auipc    a0,0x0
                            0: R_RISCV_PCREL_HI20       msg
   4:    00850513           addi     a0,a0,8 # 8 <_start+0x8>
                            4: R_RISCV_PCREL_LO12_I     .L11
```

Constants

The following example shows loading a constant using the %hi and %lo assembler functions.

```
.equ UART_BASE, 0x40003000

       lui a0,     %hi(UART_BASE)
       addi a0, a0, %lo(UART_BASE)
```

This example uses the `li` pseudo instruction to load a constant and writes a string using polled IO to a UART:

```
.equ UART_BASE, 0x40003000
.equ REG_RBR, 0
.equ REG_TBR, 0
.equ REG_IIR, 2
.equ IIR_TX_RDY, 2
.equ IIR_RX_RDY, 4

.section .text
.globl _start
_start:
1:      auipc a0, %pcrel_hi(msg)    # load msg(hi)
        addi a0, a0, %pcrel_lo(1b)  # load msg(lo)
2:      jal ra, puts
3:      j 3b

puts:
        li a2, UART_BASE
1:      lbu a1, (a0)
        beqz a1, 3f
2:      lbu a3, REG_IIR(a2)
        andi a3, a3, IIR_TX_RDY
        beqz a3, 2b
        sb a1, REG_TBR(a2)
        addi a0, a0, 1
        j 1b
3:      ret

.section .rodata
msg:
            .string "Hello World\n"
```

Floating-point rounding modes

For floating-point instructions with a rounding mode field, the rounding mode can be specified by adding an additional operand. e.g. `fcvt.w.s` with round-to-zero can be written as `fcvt.w.s a0, fa0, rtz`. If unspecified, the default `dyn`rounding mode will be used.

Supported rounding modes are as follows (must be specified in lowercase):

- `rne`: round to nearest, ties to even
- `rtz`: round towards zero
- `rdn`: round down
- `rup`: round up
- `rmm`: round to nearest, ties to max magnitude
- `dyn`: dynamic rounding mode (the rounding mode specified in the `frm` field of the `fcsr` register is used)

Control and Status Registers

The following code sample shows how to enable timer interrupts, set and wait for a timer interrupt to occur:

```
.equ RTC_BASE,      0x40000000
.equ TIMER_BASE,    0x40004000

# setup machine trap vector
1:      auipc   t0, %pcrel_hi(mtvec)        # load mtvec(hi)
        addi    t0, t0, %pcrel_lo(1b)       # load mtvec(lo)
        csrrw   zero, mtvec, t0

# set mstatus.MIE=1 (enable M mode interrupt)
        li      t0, 8
        csrrs   zero, mstatus, t0

# set mie.MTIE=1 (enable M mode timer interrupts)
        li      t0, 128
        csrrs   zero, mie, t0

# read from mtime
        li      a0, RTC_BASE
        ld      a1, 0(a0)

# write to mtimecmp
        li      a0, TIMER_BASE
        li      t0, 1000000000
        add     a1, a1, t0
        sd      a1, 0(a0)

# loop
loop:
        wfi
```

```
       j loop

# break on interrupt
mtvec:
       csrrc  t0, mcause, zero
       bgez t0, fail      # interrupt causes are less than zero
       slli t0, t0, 1     # shift off high bit
       srli t0, t0, 1
       li t1, 7           # check this is an m_timer interrupt
       bne t0, t1, fail
       j pass

pass:
       la a0, pass_msg
       jal puts
       j shutdown

fail:
       la a0, fail_msg
       jal puts
       j shutdown

.section .rodata

pass_msg:
       .string "PASS\n"

fail_msg:
       .string "FAIL\n"
```