RISC-V Assembler & ABI

Basics

Registers

Numeric	ABI name	Meaning	Saver
x0	zero	Hard-wired zero	n/a
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	n/a
x4	tp	Thread pointer	n/a
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments /return value	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

R-type computational instructions

- **sll a0,a1,a2** # x10 = x11 << x12
- **sll x10,a1,a2** # the same
- **sub sp,sp,t0** # x2 = x2 - x5
- **mul a0,s0,a1** # x10 = x8 * a1

  # using "M" standard extension
- **sltu a0,zero,a0** # x10 = (x10 == 0) ? 1 : 0

I-type computational instructions

- **addi sp,sp,-12** `# x2 = x2 - 12`
- **add sp,sp,-12** `# the same`
- **add sp,sp,-0xc** `# hexadecimal immediate`
- **add a0,a0,2048** `# Error: illegal operands`

  `# WHY???`

Loads (I-type)

- **`lw ra, 4(sp)`** `# ra = Mem[sp + 4]`
- **`ld s1, -8(s0)`** `# s1 = Mem[s0 - 8]`

*intop*i xd,xr,imm

means

xd = xr *intop* signext(imm)

ldop xd,imm(xr)

means

xd = *ldop*(xr + signext(imm))

Stores (S-type)

- **`sw ra, 4(sp)`** `# Mem[sp + 4] = ra`
- **`sd s1, -8(s0)`** `# Mem[s0 - 8] = s1`

PC-relative jumps (J-type)

- **`jal ra, 544`**
  ```
  # ra = pc + 4, pc = pc + 544
  # compute addresses by hand
  # never used this way (and hardly works)
  ```

- **`jal ra, target_label`**
  ```
  # let the assembler (and the linker)
  # do the trick
  …
  target_label:
  …
  ```

- **`jal x0, target_label`** `# pc value is lost`
- **`jal target_label`** `# the same (concise)`

Register-relative jumps (I-type)

- **`jalr ra, a0, 0`** `# ra = pc + 4, pc = a0`
- **`jalr x0, a0, 0`** `# forget pc, pc = a0`
- **`jalr a0, 0`** `# the same`
- **`jalr x0, a0`** `# the same`
- **`jalr a0`** `# the same`

Branches (B-type)

- **`beq a0, a1, 544`**

  ```
  # if( a0 == a1 ) goto ( pc + 544 )
  # compute addresses by hand
  # never used this way (and hardly works)
  ```

- **`beq a0, a1, target_label`**

  ```
  # let the assembler (and the linker)
  # do the trick
  ```

LUI instructions (U-type)

- **lui a5,0x04C12**

  ```
  # addi a0,a0,0xDB7: illegal operands
  addi a0,a5,-585  # -585 = signext(0xDB7)
  # Here a0 = 0x04C11DB7 (not 0x04C12DB7!)
  ```

- **lui a5,%hi(0x04C11DB7)**

  ```
  addi a0,a5,%lo(0x04C11DB7)
    # let the assembler do the trick
  ```

LUI and AUIPC instructions (U-type)

- ```
  lui a5,%hi(data_label)
  lw a0,%lo(data_label)(a5)
      # let the assembler (and the linker)
      # calculate absolute addresses for us
  ```

- ```
  auipc ra,%pcrel_hi(far_target_label)
  jalr ra, ra, %pcrel_lo(far_target_label)
      # let the assembler (and the linker)
      # calculate pc-relative addresses for us
  ```

Assembler pseudo-instructions (1)

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi zero,zero,0`	No operation
`mv rd,rs`	`addi rd,rs,0`	Copy register
`not rd,rs`	`xori rd,rs,-1`	Bitwise XOR
`neg rd,rs`	`sub rd,zero,rs`	Negate (2's complement)

`seqz rd,rs`
`snez rd,rs`
`sltz rd,rs`
`sgtz rd,rs`
	`sltiu rd,rs,1`
	`sltu rd,zero,rs`
	`slt rd,rs,zero`
	`slt rd,zero,rs`
		Set if zero
		Set if non-zero
		Set if greater than zero
		Set if less than zero

Assembler pseudo-instructions (2)

Pseudoinstruction	**Base Instruction(s)**	**Meaning**
`beqz rs,off`	`beq rs,zero,off`	Branch if = zero
`bnez rs,off`	`bne rs,zero,off`	Branch if != zero
`blez rs,off`	`bge zero,rs,off`	Branch if <= zero
`bgez rs,off`	`bge rs,zero,off`	Branch if >= zero
`bltz rs,off`	`blt rs,zero,off`	Branch if < zero
`bgtz rs,off`	`blt zero,rs,off`	Branch if > zero
`bgt rs,rt,off`	`blt rt,rs,off`	Branch if >
`ble rs,rt,off`	`bge rt,rs,off`	Branch if <=
`bgtu rs,rt,off`	`bltu rt,rs,off`	Branch if >, unsigned
`bleu rs,rt,off`	`bgeu rt,rs,off`	Branch if <, unsigned

Assembler pseudo-instructions (3)

Pseudoinstruction	**Base Instruction(s)**	**Meaning**
`li rd,imm`	*Myriad sequences*	Load immediate

li usage	**Base Instruction(s)**
`li a0,-2048`	`addi a0,x0,-2048`
`li a0,2048`	`lui a0,0x1` `addi a0,a0,-2048`
`li a0,0xFFFFFFFF (RV32I)`	`addi a0,zero,-1`

`li a0,0xFFFFFFFF (RV64I)`	`addi a0,zero,1`
	`slli a0,a0,32`
	`addi a0,a0,-1`

Assembler pseudo-instructions (4)

Pseudoinstruction	**Base Instruction(s)**	**Meaning**
`la rd,symbol`	`auipc rd,%pcrel_hi(symbol)` `addi rd,rd,%pcrel_lo(symbol)`	Load address
`lw rd,symbol` `ld rd,symbol`	`auipc rd,%pcrel_hi(symbol)` `l… rd,%pcrel_lo(symbol)(rd)`	Load global
`sw rd,symbol,rt` `sd rd,symbol,rt`	`auipc rt,%pcrel_hi(symbol)` `s… rd,%pcrel_lo(symbol)(rt)`	Store global

Note: PC-relative *data* addressing (PIC)

Assembler pseudo-instructions (5)

Pseudoinstruction	**Base Instruction(s)**	**Meaning**
`j off` `jr rs`	`jal zero,off` `jalr x0,rs,0`	Jump Jump register
`jal off` `jalr rs`	`jal ra,off` `jalr ra,rs,0`	Jump and link Jump and link register
`ret`	`jalr x0,x1,0`	Return from subroutine
`call off`	`auipc t1,%pcrel_hi(off)` `jalr ra,t1,%pcrel_lo(off)`	Call far-away subroutine
`tail off`	`auipc t1,%pcrel_hi(off)` `jalr x0,t1,%pcrel_lo(off)`	Tail call far-away subroutine

Assembler directives

- begin with a period (‘.’)
- tell the assembler how to translate a program
  - does not produce machine instructions
- some examples:
  - `.equ` – defines a constant
  - `.globl` – indicates that some symbol is a global one
  - `.text` – succeeding lines contain instructions
  - `.data` – succeeding lines contain data
  - `.word` – defines array of 32-bit words
  - `.align` – pad the location counter to a particular storage boundary
  - …

ABI: Stack

- Stack grows downwards
- Stack pointer shall be aligned to a 128-bit boundary upon procedure entry
- Procedures must not rely upon the persistence of stack-allocated data whose addresses lie below the stack pointer (sp)

ABI: Integer Calling Convention (1)

- Scalars are passed in a single argument register (a0-a7), or on the stack by value if none is available.
- Scalars that are 2×XLEN bits wide are passed in a pair of argument registers, or on the stack by value if none are available.
  - If exactly one register is available, the low-order XLEN bits are passed in the register and the high-order XLEN bits are passed on the stack
- Scalars wider than 2×XLEN are passed by reference and are replaced in the argument list with the address
- Arguments passed by reference may be modified by the callee.

ABI: Integer Calling Convention (2)

- After an argument has been passed on the stack, all future arguments will also be passed on the stack
- The stack pointer sp points to the first argument not passed in a register

ABI: Integer Calling Convention (3)

- Values are returned in the same manner as a first named argument of the same type would be passed.
  - If such an argument would have been passed by reference, the caller allocates memory for the return value, and passes the address as an implicit first parameter

Example 1: hanoi (1)

```
// hanoi.c

// Перемещение верхнего диска с колышка from на колышек to
extern void move( unsigned from, unsigned to );

// Рекурсивное решение задачи о ханойских башнях:
// переложить n дисков с колышка from на колышек to
static unsigned hanoi_worker(
    unsigned n, unsigned from, unsigned to );

// Решение задачи о ханойских башнях
unsigned hanoi( unsigned n ) {
    const unsigned from = 0;
    const unsigned to = 2;
    return hanoi_worker( n, from, to );
}
```

Example 1: hanoi (2)

```
…
// Рекурсивное решение задачи о ханойских башнях:
// переложить n дисков с колышка from на колышек to
static unsigned hanoi_worker( unsigned n,
                              unsigned from, unsigned to ) {
    if( n == 1 ) {
        move( from, to );
        return 1;
    }

    const unsigned via = ( 0 + 1 + 2 ) - ( from + to );

    unsigned step_counter = 0;
    step_counter += hanoi_worker( n - 1, from, via );
    step_counter += hanoi_worker( 1, from, to );
    step_counter += hanoi_worker( n - 1, via, to );

    return step_counter;
}
```

Example 1: hanoi (3)

```
    .file  "hanoi.c"
    .option nopic
    .align 2
    .text                    # code section
    .type  hanoi_worker, @function
                             # "hanoi_worker" is a function
hanoi_worker:
        add     sp,sp,-8     # allocate stack frame space (faked)
        sw      ra,7(sp)     # save return address (faked)
        sw      s0,6(sp)     # save s0 (faked)
        sw      s1,5(sp)     # save s1 (faked)
        sw      s2,4(sp)     # save s2 (faked)
        sw      s3,3(sp)     # save s3 (faked)
        sw      s4,2(sp)     # save s4 (faked)
        mv      s0,a0        # s0 = a0 (= n)
        mv      s3,a1        # s3 = a1 (= from)
        mv      s2,a2        # s2 = a2 (= to)
        li      a5,1         # a5 = 1
        bne     a0,a5,.L2    # if( a0 != 1 ) goto .L2
        …
```

Example 1: hanoi (4)

```
      …
      # bne  a0,a5,.L2  # if( a0 != 1 ) goto .L2

      mv    a1,a2       # a1 = a2 (= to)
      mv    a0,s3       # a0 = s3 (= from)
      call  move        # move( from, to )
      mv    a0,s0       # a0 = s0 (= n) (WHY?)

.L1:
      lw    ra,7(sp)    # restore return address (faked)
      lw    s0,6(sp)    # restore s0 (faked)
      lw    s1,5(sp)    # restore s1 (faked)
      lw    s2,4(sp)    # restore s2 (faked)
      lw    s3,3(sp)    # restore s3 (faked)
      lw    s4,2(sp)    # restore s4 (faked)
      add   sp,sp,8     # deallocate stack frame space (faked)
      jr    ra          # return a0
```

Example 1: hanoi (5)

```
        …
.L2:    li      s1,3            # s1 = 3
        sub     s1,s1,a2        # s1 = s1 - a2 = 3 - to
        sub     s1,s1,a1        # s1 = s1 - a1 = 3 - to - from (via)
        add     s0,a0,-1        # s0 = a0 + -1 = n - 1
        mv      a2,s1           # a2 = s1 = 3 - to - from
        mv      a0,s0           # a0 = s0 = n - 1
        call    hanoi_worker    # a0 = hanoi_worker( n-1, from, via )
        mv      s4,a0           # s4 = a0 (step_counter)
        mv      a2,s2           # a2 = s2 (= to)
        mv      a1,s3           # a1 = s3 (= from)
        li      a0,1            # a0 = 1
        call    hanoi_worker    # a0 = hanoi_worker( 1, from, to )
        add     s4,s4,a0        # s4 = s4 + a0 (step_counter)
        mv      a2,s2           # a2 = s2 (= to)
        mv      a1,s1           # a1 = s1 (= via)
        mv      a0,s0           # a0 = s0 (= n - 1)
        call    hanoi_worker    # a0 = hanoi_worker( n - 1, via, to )
        add     a0,s4,a0        # a0 = s4 + a0 (step_counter)
        j       .L1             # goto .L1
```

Example 1: hanoi (6)

```
	.align 2
	.globl hanoi
	.type  hanoi, @function
hanoi:
	add    sp,sp,-4        # allocate stack frame space (faked)
	sw     ra,3(sp)        # save return address
	li     a2,2            # a2 = 2 (to)
	li     a1,0            # a1 = 0 (from)
	call   hanoi_worker    # a0 = hanoi_worker( n, to, from )
	lw     ra,3(sp)        # restore return address (faked)
	add    sp,sp,4         # deallocate stack frame space (faked)
	jr     ra              # return a0
	.size  hanoi, .-hanoi
	.ident "GCC: (GNU) 7.1.1 20170509"
```

Example 2: mul2 (1)

```
// mul2.c

int64_t mul2i( int32_t a, int32_t b ) {
    const int64_t a64 = a;
    const int64_t b64 = b;
    return ( a64 * b64 );
}

uint64_t mul2u( uint32_t a, uint32_t b ) {
    const uint64_t a64 = a;
    const uint64_t b64 = b;
    return ( a64 * b64 );
}
```

Example 2: mul2 (2)

```
# -include stdint.h (compiler option)
# -mabi=ilp32 (compiler option)
# -march=rv32im (compiler option)
      …
mul2i:
      mv      a5,a1       # = b
      mulh    a1,a0,a1    # a1 = a * b (high, signed)
      mul     a0,a0,a5    # a0 = a * b (low)
      ret                  # return a1:a0
      …
mul2u:
      mv      a5,a1       # = b
      mulhu   a1,a0,a1    # a1 = a * b (high, unsigned)
      mul     a0,a0,a5    # a0 = a * b (low)
      ret                  # return a1:a0
      …
```

Example 2: mul2 (3)

```
# -include stdint.h (compiler option)
# -mabi=ilp32 (compiler option)
# -march=rv32i (compiler option)
      …
mul2i:
      add    sp,sp,-4   # allocate stack frame space (faked)
      mv     a2,a1      # a2 = b
      sra    a3,a1,31   # a3 = b >> 31 (arith.) ~ a3:a2=signext(b)
      sra    a1,a0,31   # a1 = a >> 31 (arith.) ~ a1:a0=signext(a)
      sw     ra,3(sp)   # save return address (faked)
      call   __muldi3   # a1:a0 = a3:a2 * a1:a0 (low)
      lw     ra,3(sp)   # restore return address (faked)
      add    sp,sp,4    # deallocate stack frame space (faked)
      jr     ra         # return a1:a0
      …
```

Example 2: mul2 (4)

```
# -include stdint.h (compiler option)
# -mabi=ilp32 (compiler option)
# -march=rv32i (compiler option)
      …
mul2i:
      add     sp,sp,-4   # allocate stack frame space (faked)
      mv      a2,a1      # a2 = b
      li      a3,0       # a3 = 0 ~ a3:a2=zeroext(b)
      li      a1,0       # a1 = 0 ~ a1:a0=zeroext(a)
      sw      ra,3(sp)   # save return address (faked)
      call    __muldi3   # a1:a0 = a3:a2 * a1:a0 (low)
      lw      ra,3(sp)   # restore return address (faked)
      add     sp,sp,4    # deallocate stack frame space (faked)
      jr      ra         # return a1:a0
      …
```

Example 2: mul2 (5)

```
# -include stdint.h (compiler option)
# -mabi=lp64 (compiler option)
# -march=rv64im (compiler option)
      …
mul2i:
      mul    a0,a0,a1  # a0 = a * b (low)
      ret              # return a0
      …
```

Example 3: too_many_args (1)

```
// too_many_args.c

extern int too_many_args( int r0, int r1, int r2, int r3,
                          int r4, int r5, int r6, int r7,
                          int s0, int s1, int s2 );

int too_many_args_caller( int a ) {
    return too_many_args(
               a << 1, a << 2, a << 3, a << 4,
               a << 5, a << 6, a << 7, a << 8,
               a >> 1, a >> 2, a >> 3 );
}
```

Example 3: too_many_args (2)

```
    …
too_many_args_caller:
    add     sp,sp,-8          # allocate stack frame space
    sra     a3,a0,3           # a3 = a >> 3
    sra     a4,a0,2           # a4 = a >> 2
    sra     a5,a0,1           # a5 = a >> 1
    sw      a3,2(sp)          #\  push arguments on stack (faked):
    sw      a4,1(sp)          # }     [a >> 1][a >> 2][a >> 3]...
    sw      a5,0(sp)          #/      ^ sp-point-here
    sll     a7,a0,8           #\
    sll     a6,a0,7           # }
    sll     a5,a0,6           # }
    sll     a4,a0,5           # } a0 = a << 1, ..., a7 = a << 8
    sll     a3,a0,4           # }
    sll     a2,a0,3           # }
    sll     a1,a0,2           # }
    sll     a0,a0,1           #/
    sw      ra,7(sp)          # save return address (faked)
    call    too_many_args     # a0 = too_many_args(...)
    lw      ra,7(sp)          # restore return address (faked)
    add     sp,sp,8           # deallocate stack frame space (faked)
    jr      ra                # return a0
    …
```

Example 4: tail_call (1)

```
// tail_call.c

extern int too_many_args( int r0, int r1, int r2, int r3,
                          int r4, int r5, int r6, int r7,
                          int s0, int s1, int s2 );

int too_many_args_caller2( int r0, int r1, int r2, int r3,
                           int r4, int r5, int r6, int r7,
                           int s0, int s1, int s2 ) {
    return too_many_args( r0, r1, r2, r3,
                          r4, r5, r6, r7,
                          s0, s1, s2 );
}
```

Example 4: tail_call (2)

```
      # -O1 (compiler option)
      …
too_many_args_caller2:
      add     sp,sp,-8          # allocate stack frame space (faked)
      sw      ra,7(sp)          # save ra (faked)
      lw      t1,10(sp)         # our s2 (faked)
      sw      t1,2(sp)          # callee's s2 (faked)
      lw      t1,9(sp)          # our s1 (faked)
      sw      t1,1(sp)          # callee's s1 (faked)
      lw      t1,8(sp)          # our s0 (faked)
      sw      t1,0(sp)          # callee's s0
      call    too_many_args     # a0 = too_many_args(…)
      lw      ra,7(sp)          # restore sa (faked)
      add     sp,sp,8           # deallocate stack frame space (faked)
      jr      ra                # return a0
```

Example 4: tail_call (3)

```
        # -O2 (compiler option)
        …
too_many_args_caller2:
        tail   too_many_args  # return too_mant_args(…)
```

Example 5: tail_call2 (1)

```
// tail_call2.c

extern int some_fun( int a );

int too_many_args( int r0, int r1, int r2, int r3,
                   int r4, int r5, int r6, int r7,
                   int s0, int s1, int s2 ) {
    const int a = r0 + r1 + r2 + r3 +
                  r4 + r5 + r6 + r7 +
                  s0 + s1 + s2;
    return some_fun( a );
}
```

Example 5: tail_call2 (2)

```
      # -O2 (compiler option)
      …
too_many_args:
      add    a1,a0,a1   # r0 + r1
      add    a1,a1,a2   # r0 + r1 + r2
      add    a1,a1,a3   # r0 + … + r3
      add    a1,a1,a4   # r0 + … + r4
      lw     a0,0(sp)   # s0 (faked)
      add    a1,a1,a5   # r0 + … + r5
      add    a1,a1,a6   # r0 + … + r6
      add    a1,a1,a7   # r0 + … + r7
      add    a1,a1,a0   # r0 + … + r7 + s0
      lw     a0,1(sp)   # s1 (faked)
      add    a1,a1,a0   # r0 + … + r7 + s0 + s1
      lw     a0,2(sp)   # s2 (faked)
      add    a0,a1,a0   # r0 + … + r7 + s0 + s1 + s2 (a)
      tail   some_fun   # return some_fun( a )
```

Example 6: byref (1)

```
// byref_caller.c

extern int byref_callee( int arr[ 3 ] );

static int global_arr[ 3 ] = { 10, 20, 30 };

int byref_caller( int a ) {
    return byref_callee( global_arr );
}
```

Example 6: byref (2)

```
      …
      .text  # code
      …
byref_caller:
      lui    a0,%hi(global_arr)
      addi   a0,a0,%lo(global_arr)
      tail   byref_callee

      …
      .data  # data
      …
global_arr:
      .word  10
      .word  20
      .word  30
      …
```

Example 6: byref (3)

```
// byref_callee.c

int byref_callee( int arr[ 3 ] ) {
    return ( arr[ 0 ] + arr[ 1 ] + arr[ 2 ] );
}



# byref_callee.s
      …
byref_callee:
      lw     a5,0(a0)   # arr[ 0 ]
      lw     a4,1(a0)   # arr[ 1 ] (faked)
      lw     a0,2(a0)   # arr[ 2 ] (faked)
      add    a5,a5,a4   # arr[ 0 ] + arr[ 1 ]
      add    a0,a5,a0   # a0 = arr[ 0 ] + arr[ 1 ] + arr[ 2 ]
      ret                # return a0
      …
```