chapter2 汇编语言

汇编语言的典型操作

赋值

取大立即数

使用 `lui` 与 `addi` 两步操作，因为任何一个立即数都没有 32 位

例如，赋值 $4000000 = 976 \times 16^3 + 2304$

```
lui s3, 976
addi s3, s3, 2304
```

最少指令判断条件

1. Reduce an index-out-of-bounds check `If (x20>=x11 & x20<0)  goto IndexOutofBounds`

   ```
   bgeu x20, x11, IndexOutofBounds
   ```

C语言到汇编语言

简单操作

C code

```
f = (g + h) – (i + j);
```

RISC-V code

```
add   x5, x20, x21    // register x5 contains g + h
add   x6, x22, x23    // register x6 contains i + j
sub   x19, x5, x6     // f gets x5 – x6, which is (g + h) – (i + j)
```

取地址对应数据

C code:

```
g = h + A[8] ;          // A is an array of 100 doublewords
```

(Assume: g ---- x20 h ---- x21
base address of A ---- **x22**)

RISC-V code:

```
ld    x9, 64(x22)       // temporary reg x9 gets A[8]
add   x20, x21, x9      // g = h + A[8]
```

C code:

```
A[12] = h + A[8] ;    // A is an array of 100 words
```

(Assume: h ---- x21 base address of A ---- x22)

RISC-V code:

```
ld    x9 , 64(x22)    // temporary reg x9 gets A[8]
add   x9, x21, x9     // temporary reg x9 gets h + A[8]
sd    x9, 96(x22)     // stores  h + A[8]  back into A[12]
```

C code:

```
g = h + A[i] ;          // A is an array of 100 doublewords
```

(Assume: g, h, ***i*** -- x1, x2, ***x4*** base address of A -- x3)

RISC-V code:

```
add   x5, x4, x4      #  temp reg x5 = 2 * i
add   x5, x5, x5      #  temp reg x5 = 4 * i
add   x5, x5, x5      #  temp reg x5 = 8 * i
add   x5, x5, x3      #  x5 = address of A[i] (8 * i + x3)
ld    $x6, 0(x5)      #  temp reg x6 = A[i]
add   x1, x2, x6      #  g = h + A[i]
```

判断

(Assume: f ~ j ---- x19 ~ x23)

F=

C code:

```
if ( i == j )  f = g + h ; else f = g – h;
```

RISC-V assembly code:

```
      bne    x22, x23, ELSE   // go to ELSE if  i != j
      add    x19, x20, x21    // f = g + h  ( skipped if  i  not equals  j )
      beq    x0, x0,  EXIT
ELSE: sub    x19, x20, x21    // f = g - h  ( skipped if  i  equals  j )
EXIT:
```

case

Compiling a switch using ***jump address table***

(Assume: f ~ k ---- x20 ~ x25 x5 contains 4)

C code:

```
switch ( k ) {
        case 0 :  f = i + j ; break ;  /* k = 0 */
        case 1 :  f = g + h ; break ;  /* k = 1 */
        case 2 :  f = g - h ; break ;  /* k = 2 */
        case 3 :  f = i - j ; break ;  /* k = 3 */
}
```

```
blt   x25, x0, Exit      // test if k < 0
bge   x25, x5, Exit      // if k >= 4, go to Exit
slli  x7, x25, 3         // temp reg x7 = 8 * k
add   x7, x7, x6         // x7 = address of JumpTable[k]
ld    x7, 0(x7)          // x7 gets JumpTable[k]
jalr  x1, 0(x7)          // jump entrance
```

Exit:

jump address table

L0:address

L1:address

L2: address

L3:address

```
L0:  add   x20, x23, x24    // k = 0 so f gets i + j
     jalr  x0, 0(x1)        // end of this case so go to Exit
L1:  add   x20, x21, x22    // k = 1 so f gets g + h
     jalr  x0, 0(x1)        // end of this case so go to Exit
L2:  sub   x20, x21, x22    // k = 2 so f gets g - h
     jalr  x0, 0(x1)        // end of this case so go to Exit
L3:  sub   x20, x23, x24    // k = 3 so f gets i - j
     jalr  x0, 0(x1)        //end of switch statement
```

数组与字符串

Compiling a switch using ***jump address table***

(Assume: f ~ k ---- x20 ~ x25 x5 contains 4)

C code:

```
switch ( k ) {
        case 0 :  f = i + j ; break ;  /* k = 0 */
        case 1 :  f = g + h ; break ;  /* k = 1 */
        case 2 :  f = g - h ; break ;  /* k = 2 */
        case 3 :  f = i - j ; break ;  /* k = 3 */
}
```

(Assume: i -- x19, x's base --x10, y's base ----x11)

- C code: Null-terminated string Y→X

```
void strcpy ( char x[ ], char y[ ] )
{
    size_t i ;
    i = 0 ;
    while ( ( x[ i ] = y[ i ] ) != '\ 0' )      /* copy and test byte */
        i += 1 ;
}
```

- RISC-V assembly code:

```
strcpy: addi sp, sp, -8          // adjust stack for 1 doubleword
        sd   x19, 0(sp)          // save x19
        add  x19, x0, x0         // i = 0
L1:     add  x5, x19, x11        // address of y[ i ] in x5
        lbu  x6, 0(x5)           // x6 = y [ i ]
        add  x7, x19, x10        // address of x[ i ] in x7
        sb   x6, 0(x7)           // x[ i ] = y[ i ]

        beq  x6, x0, L2          // if y[ i ] == 0, go to L2
        addi x19, x19, 1         // i = i + 1
        jal  x0, L1              // go to L1
L2:     ld   x19, 0(sp)          // restore x19
        addi sp, sp, 8           // pop 1 doubleword off stack
        jalr x0, 0(x1)           // return
```

循环

(Assume: i and k---- x22 and x24 base of save ---- x25)

C code:

```
while ( save[i] == k )
        i += 1 ;
```

RISC-V assembly code:

```
Loop:   slli   x10, x22, 3      // Temp reg x10 = i * 8
        add    x10, x10, x25    // x10 = address of save[i]
        ld     x9, 0(x10)       // x9 gets save[i]
        bne    x9, x24, Exit    // go to Exit if save[i] != k
        addi   x22, x22, 1      // i += 1
        beq    x0, x0, Loop     // go to Loop
Exit:
```

用栈存储

C code:

```
long long int leaf_example (
   long long int g, long long int h,
   long long int i, long long int j) {
       long long int f;
       f = (g + h) - (i + j);
       return f;
   }
```

- Arguments g, …, j in x10, …, x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack

```
leaf_example:
  addi sp,sp,-24          Save x5, x6, x20 on stack
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp)
  add  x5,x10,x11         x5 = g + h
  add  x6,x12,x1          x6 = i + j
  sub  x20,x5,x6          f = x5 – x6
  addi x10,x20,0          copy f to return register
  ld   x20,0(sp)          Resore x5, x6, x20 from stack
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)           Return to caller
```

函数调用

Example Compiling a recursive procedure that computes n!, suppose argument n is in x10, and results in x10

```
long long  fact ( long long  n )
{
     if ( n  <  1 )  return  ( 1 ) ;
         else  return  ( n  *  fact ( n  -  1 ) ) ;
}
```

RISC-V assembly code

```
fact:   addi   sp, sp, 16          // adjust stack for 2 items
        sd     x1, 8(sp)           // save the return address
        sd     x10, 0(sp)          // save the argument  n
        addi   x5, x10, -1         // x5 = n  -  1
        bge    x5, x0, L1          // if  n  >=  1, go to L1(else)
        addi   x10, x0, 1          // return 1 if n <1
        addi   sp, sp, 16          // Recover sp (Why not recover x1and x10 ?)
        jalr   x0, 0(x1)           // return to caller
```

```
L1:  addi   x10, x10, -1        // n >= 1: argument gets ( n - 1 )
     jal    x1, fact            // call fact with ( n - 1 )
     add    x6, x10, x0
     ld     x10, 0(sp)          // restore argument n
     ld     x1, 8(sp)           // restore the return address
     addi   sp, sp, 16          // adjust stack pointer to pop 2 items
     mul    x10, x10, x6        // return n*fact ( n - 1 )
     jalr   x0,  0(x1)          // return to the caller
```