Facultad de Informática
Universidad Complutense de Madrid

Fundamentos de Computadores II - Práctica 3
Programación con subrutinas

Daniel Báscones (danibasc@ucm.es)

15 de marzo de 2023

1. Objetivos

Tras haber aprendido a gestionar las instrucciones, registros y memoria en RISC-V, es importante conocer cómo organizar el código en funciones para ofrecer mayor reusabilidad y comodidad al programador. En esta práctica veremos:

- Las funciones, la pila de llamadas y los marcos de activación.
- Los convenios de paso de argumentos a funciones, así como de recogida de resultados.

2. Subrutinas y llamadas

Se conoce como subrutina a un fragmento de código que podemos invocar desde cualquier punto del programa (incluyendo otra o la propia subrutina), retomándose la ejecución del programa, en la instrucción siguiente a la invocación, cuando la subrutina haya terminado. Las subrutinas se usan para simplificar el código y poder reusarlo, y son el mecanismo con el que los lenguajes de alto nivel implementan procedimientos, funciones y métodos.

Una subrutina puede aceptar parámetros (argumentos) que modifiquen su comportamiento, exactamente igual que una función de alto nivel. Por convenio en RISC-V, estos se colocan en los registros de argumento `a0`-`a7`. El valor de retorno de una subrutina se debe colocar en `a0`, permitiéndose además devolver un segundo valor en `a1`.

Las subrutinas son parte del código (`.text`). Para ubicarlas, se les coloca una etiqueta con la que poder posteriormente invocarlas. A continuación se muestra un ejemplo:

Programa principal	Subrutina 1	Subrutina 2

```
main:                   recorre:                mira:
    [...]                   [...]                   [...]
    call recorre            call mira               ret
    [...]                   [...]
                            ret
```

Como se puede observar, para el control de llamadas a funciones en RISC-V existen dos pseudoinstrucciones clave:

- `call etiqueta`: Esta instrucción calcula la dirección de destino de la etiqueta, y salta a la misma para iniciar la función correspondiente. A fin de que la función llamada pueda devolver el control, también coloca la dirección de retorno en el registro `ra`.
- `ret`: Esta instrucción devuelve el control a la dirección situada en `ra`, complementándose con la anterior para finalizar la llamada.

Por tanto, siempre que se llame a una subrutina se utilizará la instrucción `call`, y para regresar, `ret`.

Ahora bien, hay un detalle con el que tener especial cuidado. Para realizar operaciones, toda subrutina debe modificar ciertos registros. Al llamar a una subrutina, puedo tener valores críticos en ciertos registros, que necesito a la vuelta. Veamos un ejemplo:

```
a: .word 3
main:
    la t0, a          //cargo en t0 la dirección de a
    lw a0, 0(t0)      //cargo en a0 el valor de a (3)
    call cuadrado     //llamo a la función para calcular a^2
    sw a0, 0(t0)      //guardo el resultado en t0
```

El código anterior parece inofensivo pero, ¿qué pasa si la subrutina modifica el estado del registro `t0`? Cuando guardemos el resultado, lo pondremos en una dirección de memoria que no corresponde. Podemos pensar en una serie de soluciones:

- La más evidente parece utilizar en la subrutina `cuadrado` registros diferentes a `t0`. Pero, ¿Y para programas muy grandes? Solo tenemos 32 registros, se acabarían rápido.
- Otra potencial solución parece ser cargar de nuevo la dirección de `a` tras la llamada. En este caso funciona pero, ¿Y si fuera un valor calculado anteriormente, que ya no puedo recalcular?.

Se nos pueden ocurrir muchas otras soluciones, aunque quizá no libres de problemas. Para solventar este problema, y evitar que los valores se pierdan y se sobreescriban, contamos con el siguiente convenio:

Los registros se dividen, a rasgos generales, en *temporales* y *salvados*. Cualquier subrutina **debe** preservar, a su retorno, **el estado de todos los registros salvados**, mientras que es **libre de modificar los temporales**. Los registros salvados son, para RISC-V, `s0-s11`, mientras que los temporales son `a0-a7`,`t0-t6`.

Teniendo esto en cuenta, basta con que utilicemos `s0` en lugar de `t0` para que el problema quede solucionado. La subrutina `cuadrado` deberá encargarse de que `s0` no se pierda.

3. La pila

Todo esto suena muy bien pero, ¿Dónde guardo `s0`?. Veamos un ejemplo:

Programa principal	Subrutina 1	Subrutina 2

```
main:
    //modifico s0
    call recorre
    //uso s0
    modificado
```

```
recorre:
    //modifico s0
    call mira
    //uso s0
    modificado
    ret
```

```
mira:
    //uso s0
    do mira
    ret
```

En este caso, ¡la subrutina `recorre` estaría borrando el dato que tiene guardado en `s0` el programa `main`! ¡y `mira` también borra el dato de `recorre`! ¿Vaya jaleo, no?

La solución (ya definitiva) a todo esto está en seguir la siguiente regla:

Si al diseñar una subrutina, uso cualquier registro salvado, debo guardarlo en la pila al comenzar, y recuperarlo de la pila al terminar.

Así, nuestro código anterior se convierte en:

Programa principal	Subrutina 1	Subrutina 2

```
main:
    //guardo en s0
    call recorre
    //uso s0
```

```
recorre:
    //apilo s0
    //guardo en s0
    call mira
    //uso s0
    //desapilo s0
    ret
```

```
mira:
    //apilo s0
    //uso s0
    do mira
    //desapilo s0
    ret
```

Para trabajar con la pila disponemos de un registro especial, el *stack pointer* o puntero de pila. Éste nos indica en todo momento la última posición ocupada en memoria por la pila. Debemos tener en cuenta que la pila crece hacia direcciones descendentes. Es decir, que si la última posición ocupada de memoria es la `0x20000`, la siguiente palabra (4 bytes) libre está en la dirección `0x1fffc`.

Con esto visto, siempre que queramos guardar un registro en memoria, deberemos encadenar dos instrucciones:

```
addi sp, sp, -4
sw rd, 0(sp)
```

En primer lugar, guardamos el registro destino que queramos en la pila. Posteriormente actualizamos el puntero de pila para apuntar de nuevo a la última posición ocupada. Si por ejemplo guardamos varios registros, podemos realizar la actualización una única vez:

```
addi sp, sp, -12
sw rd1, 8(sp)
sw rd2, 4(sp)
sw rd3, 0(sp)
```

Para desapilar los registros, bastará con hacer las operaciones en el orden inverso:

```
lw rd1, 8(sp)
lw rd2, 4(sp)
lw rd3, 0(sp)
addi sp, sp, 12
```

Veamos todo esto en un diagrama práctico:

Programa principal	Estructura de memoria y pila

```
main:
    //guardo en s0
    call recorre
    //uso s0

recorre:
    //apilo s0
    //guardo en s0
    call mira
    //uso s0
    //desapilo s0
    ret

mira:
    //apilo s0
    //uso s0
    do mira     <-- estamos aquí
    //desapilo s0
    ret
```

```
.

Dirección |             Contenido             |
----------+-----------------------------------+---------
0x00000   |              código               |
  ....    |              código               |  <- pc
0x0014c   |          fin del código           |
0x00150   |               libre               |
  ....    |               libre               |
0x09ffc   |               libre               |
0x10000   |               datos               |
  ....    |               datos               |
0x10100   |         fin de los datos          |
  ....    |               libre               |
0x1fff8   |               libre               |
0x1fffc   | s0 de recorre apilado por mira    |  <- sp
0x20000   | s0 de main apilado por recorre    |
```

4. Prólogo y epílogo

A la hora de diseñar una subrutina, es conveniente seguir una estructura predefinida que nos permita contemplar todos los detalles que hemos comentado en la sección anterior.

- **Prólogo:** La subrutina deberá, en primer lugar, guardar todos los registros salvados que vaya a utilizar, a fin de poder recuperarlos antes de devolver el control. Además, tendrá que actualizar el puntero de pila (sp) para reflejar esta nueva ocupación:

```
// prólogo de la subrutina
subrut:
// 1) Actualizar el puntero de pila
    addi sp, sp, -52
// 2) Guardar todos los registros salvados
// (sólo los que se modifiquen)
    sw ra, 48(sp) //ra solo si se llama a otra subrutina
    sw s0, 44(sp)
    ...
    sw s11, 0(sp)
// La funcionalidad puede comenzar
    ...
```

- **Epílogo:** Tras ejecutar el código correspondiente, es imprescindible dejar el estado del procesador tal como se encontraba cuando nos llamaron. Es decir, hay que restaurar todos los registros salvados, y devolver el puntero de pila (sp) al estado inicial.

```
// acaba la funcionalidad
    ...
// epílogo de la subrutina
// 1) recuperar los registros
    lw ra, 48(sp) //ra solo si se llama a otra subrutina
    lw s0, 44(sp)
    ...
    lw s11, 0(sp)
// 2) recuperar el puntero de pila
    addi sp, sp, 52
// 3) podemos devolver el control
    ret
```

Adicionalmente, en caso de necesitar guardar valores intermedios de la subrutina en memoria (por ejemplo variables locales que no nos caben en registros), simplemente tendríamos que guardarlos en la pila (actualizando sp), y posteriormente desapilarlos.

5. Ejemplo del uso de la pila

En la siguiente tabla se presenta un ejemplo del uso de la pila, partiendo de un código de alto nivel con varias funciones que se llaman entre sí:

```c
/////////////// CÓDIGO DE ALTO NIVEL ////////////
.

#define N 4


int V[N] = {4, -2, 3, 7};
int W[N] = {2, 5, -3, 0};


int res;


void main() {
    res = maxDist(V, W, N);
}









int maxDist(int X[], int Y[], int n) {
    int max = 0;
    for (int i = 0; i < n, i++) {
        int dist = subabs(X[i], Y[i]);
        if (dist > max)
            max = dist;
    }
    return max;
}









int subabs(int x, int y) {
    int res = x - y;
    if (res < 0)
        res = -res;
    return res;
}
```

```asm
///////////////// ENSAMBLADOR ////////////////
.extern _stack
.global main
.equ N 4

.data
V:   .word 4, -2, 3, 7
W:   .word 2, 5, -3, 0

.bss
res: .space 4

.text
main:
    la sp, _stack   //inicializa sp
    la a0, V        //1er arg: dir de V
    la a1, W        //2o  arg: dir de W
    li a2, N        //3er arg: N
    call maxDist    //llamada a maxDist
    la t0, res
    sw a0, 0(t0)    //guardado de res
fin:
    j fin

maxDist:
    addi sp, sp,-24 /////
    sw ra, 20(sp)   //
    sw s0, 16(sp)   //
    sw s1, 12(sp)   // PRÓLOGO
    sw s2, 8(sp)    //
    sw s3, 4(sp)    //
    sw s4, 0(sp)    /////
    li s0, 0        //s0 guarda max
    li s1, 0        //s1 guarda i
    mv s2, a0       //s2 guarda X
    mv s3, a1       //s3 guarda Y
    mv s4, a2       //s4 guarda n
bucle:
    bge s1, s4, return_md
    lw a0, 0(s2)    //1er arg: X[i]
    lw a1, 0(s3)    //2o  arg: Y[i]
    call subabs     //llamada a subabs
    ble a0, s0, nextiter
    mv s0, a0
nextiter:
    addi s1, s1, 1  //actualizo iterador
    addi s2, s2, 4  //X++
    addi s3, s3, 4  //Y++
    j bucle         //repito bucle
return_md:
    mv a0, s0       //coloco valor de retorno
    lw ra, 20(sp)   /////
    lw s0, 16(sp)   //
    lw s1, 12(sp)   //
    lw s2, 8(sp)    // EPÍLOGO
    lw s3, 4(sp)    //
    lw s4, 0(sp)    //
    addi sp, sp, 24 /////
    ret             //devuelvo control

subabs:
    sub a0, a0, a1  //calculo distancia
    bge a0, zero, return_sa
    neg a0, a0      //cambio a positivo
return_sa
    ret             //devuelvo valor en a0
```

Realizamos una serie de observaciones importantes acerca de este código, que son de gran relevancia a la hora de diseñar en ensamblador:

- Inicializamos el puntero de pila `sp` a un valor `_stack` que viene dado externamente por `.extern _stack`. Este valor estará configurado en el `ld_script`, y dará una dirección de memoria libre donde colocar la pila. Desde el ensamblador simplemente la usamos.
- Cuando una función tiene argumentos, se colocan en orden en los registros de argumento `a0`-`a7`, para posteriormente llamarla.
- Cuando una función devuelve un valor, éste volverá en el registro de argumento `a0`. Debemos recogerlo de ahí y guardarlo.
- Si en una subrutina modifico cualquier registro salvado `s0`-`s11`, debo guardarlo en el prólogo y restaurarlo en el epílogo. Si además la subrutina llama a otra, también deberá guardar el registro `ra`.
- Al llamar a una subrutina, ésta puede cambiar todos los registros temporales `t0`-`t6` y de argumento `a0`-`a7`. Debemos mover esos valores a registros salvados `s0`-`s11` si pretendemos utilizarlos a la vuelta de la subrutina.

6. Desarrollo de la práctica

En la ISA base de RISC-V, tenemos un conjunto de instrucciones muy versátil y potente, aunque algo restringido al disponer solo de unas decenas de instrucciones. Una instrucción muy interesante de la que no disponemos por defecto (solo si utilizamos la extensión **RV32M**) es la de multiplicación. Esto es debido a que el procesador, en su versión más sencilla, incluye solo un sumador en la ALU para acelerar el ciclo de reloj. Si queremos multiplicar y no disponemos de la instrucción `mul`, no nos queda otra que emular la multiplicación mediate un algoritmo software.

1. Codificar en ensamblador una subrutina que realice la multiplicación de dos números enteros (no está permitido usar `mul`). Para ello, se puede tomar el siguiente algoritmo como referencia, que para multiplicar a por b, suma b veces el número a.

```
int mul(int a, int b) {
    int res = 0;
    while (b > 0) {
        res += a;
        b--;
    }
    return res;
}
```

2. Codificar en ensamblador una subrutina que calcule el producto escalar (*dot product*) de dos vectores. Se proporciona el siguiente código en alto nivel como referencia:

```
int dotprod(int V[], int W[], int n) {
    int acc = 0;
    for (int i = 0; i < n; i++) {
        acc += mul(V[i], W[i]);
    }
    return acc;
}
```

3. Codificar en ensamblador el siguiente programa, que aprovecha las anteriores funciones para determinar si un vector tiene mayor norma (longitud) que otro:

```
#define N 4
int A[N] = {3, 5, 1, 9};
int B[N] = {1, 6, 2, 3};

int res;

void main() {
    int normA = dotprod(A, A, N);
    int normB = dotprod(B, B, N);
    if (normA > normB)
        res = 0xa;
    else
        res = 0xb;
}
```

   Podremos ver si el resultado es correcto, inspeccionando en memoria el valor de `res`.