

Practica 5 - WinMIPS64 - Pila

22 de octubre de 2024

Subrutinas

- El soporte de la arquitectura para la invocación de subrutinas es diferente que el soporte que existe en otras arquitecturas (Pila, CALL, RET)
- La instrucción encargada de realizar un llamado a una subrutina es *jal* (jump and link):
 - La instrucción guarda en el registro *\$ra* la dirección de retorno de la subrutina
 - Cambia el PC por la dirección de la subrutina
- Para retornar, basta retornar a la posición apuntada por el registro *\$ra*
 - Esto podemos hacerlo con la instrucción *jr \$ra*

Subrutinas

Acción	VonSim	WinMIPS64
Llamada a subrutina	<code>CALL subrutina</code> Se apila la dirección de retorno Se actualiza IP con el valor de la subrutina	<code>jal subrutina</code> Se guarda en el registro <i>\$ra</i> la dirección de retorno Se actualiza PC con el valor de la subrutina
Retorno de la subrutina	<code>RET</code> Se desapila la dirección de retorno y se actualiza IP con ese valor	<code>jr \$ra</code> Se actualiza PC con el valor del registro <i>\$ra</i> , que tiene la dirección de retorno de la subrutina

- ¿Qué pasa si una subrutina invoca a otra subrutina?

Normalización de registros

En lugar de usar r0-r31 es más conveniente darle nombre más significativos a los registros:

Registro	Nombre	Uso	Preservado
r0	\$zero, \$0	Siempre tiene el valor 0 y no se puede cambiar.	No
r1	\$at	Assembler Temporary – Reservado para ser usado por el ensamblador.	No
r2 - r3	\$v0 - \$v1	Valores de retorno de la subrutina llamada.	No
r4 - r7	\$a0 - \$a3	Argumentos pasados a la subrutina llamada.	No
r8 - r15	\$t0 - \$t7	Registros temporarios. No son conservados en el llamado a subrutinas.	No
r16 - r23	\$s0 - \$s7	Registros salvados durante el llamado a subrutinas.	Si
r24 - r25	\$t8 - \$t9	Registros temporarios. No son conservados en el llamado a subrutinas.	No
r26 - r27	\$k0 - \$k1	Para uso del kernel del sistema operativo.	No
r28	\$gp	Global Pointer – Puntero a la zona de la memoria estática del programa.	Si
r29	\$sp	Stack Pointer – Puntero al tope de la pila.	Si
r30	\$fp	Frame Pointer – Puntero al marco actual de la pila.	Si
r31	\$ra	Return Address – Dirección de retorno en un llamado a una subrutina.	Si

- Preservado implica que los registros deben devolverse *inalterados* desde una subrutina.
- \$s0-\$s7 representan las variables locales de la subrutina / programa.
- \$t0-\$t9 son registros para almacenar resultados auxiliares.
- Este uso de registros es el recomendado y esta normalizado
- Si una subrutina altera el valor de algún registro que debe ser preservado, debe conservar el valor original para poder restaurarlo.

Ejemplo

```
1      .data
2      base: .word 16
3      exponente: .word 4
4      result: .word 0
5
6      .text
7      ld $a0, base($0)
8      ld $a1, exponente($0)
9      jal a_la_potencia
10     sd $v0, result($0)
11     halt
12
13 a_la_potencia: daddi $v0, $0, 1
14 lazo:      beqz $a1, terminar
15           daddi $a1, $a1, -1
16           dmul $v0, $v0, $a0
17           j lazo
18 terminar:  jr $ra
```

Introducción

En esta arquitectura no tenemos soporte nativo para el manejo de pilas.

Veamos en que consiste el manejo de pila del VonSim:

- Tiene un registro SP, que se usa solo para el manejo de la pila y que inicialmente vale 08000h.
- Tenemos una instrucción PUSH que nos permite *apilar* valores de 16 bits en la misma.
- Tenemos una instrucción POP que nos permite *desapilar* valores de 16 bits de la misma.
- La pila también se utiliza para el manejo de subrutinas, pero ya tenemos un reemplazo para eso.

Pila

- VonSim posee un registro específico para la Pila
- WinMIPS64 posee 32 registros de propósito general, podemos elegir uno y usarlo exclusivamente para la pila.
- La convención de nombres de registros ya vistos ya contempla al registro *r29* como el registro *\$sp*.
- El registro SP está inicializado en 08000h, debemos inicializar el registro de pila a un valor adecuado.
- Podemos inicializar el registro *\$sp* en 0400h, que es la posición de memoria de datos más alta que tenemos en WinMIPS64.

```
1 .code
2 daddi $sp, $0, 0x400
3 ...
```

Apilar

La instrucción PUSH del VonSim es la encargada de apilar un valor en la pila. Veamos en que consiste apilar un valor:

- $(SP) \leftarrow (SP) - 2$; *Decremento Puntero de pila*
- $[SP + 1 : SP] \leftarrow (fuente)$; *Guardo valor en la pila*

Para imitar esta operación deberíamos:

- Decrementar en 8 el puntero de pila. ¿Por qué 8?
- Guardar en la posición de memoria del puntero de pila el valor.

Apilar - Detalle

Apilando el valor del registro \$t1

```
1      ...  
2      daddi $sp, $sp, -8  
3      sd $t1, 0($sp)      ; Guardo en la pila  
4      ...
```

Desapilar

La instrucción POP del VonSim es la encargada de desapilar un valor de la pila. Veamos en que consiste:

- $(fuente) \leftarrow [SP + 1 : SP]$; *Guardo valor de la pila*
- $(SP) \leftarrow (SP) + 2$; *Incremento el Puntero de pila*

Para imitar esta operación deberíamos:

- Leer el dato de la posición de memoria del puntero de pila y guardarlo en un registro.
- Incrementar en 8 el puntero de pila.

Desapilar - Detalle

Desapilando en el registro \$s1

```
1      ...  
2      ld $s1, 0($sp)           ; Leo desde la pila  
3      daddi $sp, $sp, +8  
4      ...
```

Multiples valores

Veamos que pasa si se apilan varios valores seguidos:

```
1  ...  
2  daddi $sp, $sp, -8  
3  sd $s1, 0($sp)      ; Apilo s1  
4  
5  daddi $sp, $sp, -8  
6  sd $s2, 0($sp)      ; Apilo s2  
7  
8  daddi $sp, $sp, -8  
9  sd $s3, 0($sp)      ; Apilo s3  
10  
11 daddi $sp, $sp, -8  
12 sd $s4, 0($sp)      ; Apilo s4  
13  
14 ...
```

Podríamos agrupar los *daddi* y ver si podemos tener menos instrucciones.

Multiples valores

```
...  
daddi $sp, $sp, -32 ; Reservo 32 bytes  
sd $s1, 24($sp) ; Apilo s1  
sd $s2, 16($sp) ; Apilo s2  
sd $s3, 8($sp) ; Apilo s3  
sd $s4, 0($sp) ; Apilo s4  
...
```

- Ajustamos `$sp` al inicio, le restamos 8 por cada registro a apilar.
- Luego guardamos cada registro, desplazandolo de a 8 posiciones.

Multiples valores - Continuación

Y para desapilar, lo hacemos de manera similar

```
1      ...  
2  
3      ld $s1, 24($sp) ; Restauero s1  
4      ld $s2, 16($sp) ; Restauero s2  
5      ld $s3, 8($sp)  ; Restauero s3  
6      ld $s4, 0($sp)  ; Restauero s4  
7      daddi $sp, $sp, +32  
8      ...
```

Pilas y Subrutina

Toda subrutina se dividirá siempre en tres partes

- *Prólogo*: se resguarda en la pila todos los registros que deban ser preservados
- *Cuerpo* de la subrutina: con el código propio de la misma
- *Epílogo*: se restauran los registros preservados en el prólogo

```
1  
2 subrutina:  
3     daddi $sp, $sp, -16  
4     sd $ra, 8($sp) ; Apilo ra  
5     sd $s0, 0($sp) ; Apilo s0  
6  
7     ...  
8  
9     ld $ra, 8($sp) ; Restauero ra  
10    ld $s0, 0($sp) ; Restauero s0  
11    daddi $sp, $sp, +16  
12    jr $ra ; $ra tiene valor de retorno
```

- Si no se modifican los registros `$s0` a `$s7`, no es necesario guardarlos
- Una subrutina sencilla podria tener un prólogo y epílogo vacío

Anidamiento de Subrutinas

- Si una subrutina llama a otra subrutina, va a alterar el valor del registro $\$ra$. Por lo tanto debemos preservar el valor que recibimos de $\$ra$.
- Si no llama a otra subrutina, no es necesario que guarde el valor contenido en $\$ra$.
- Esto es valido tanto para subrutinas que se llamen a si mismas (recursivas) como para subrutinas que llamen a otras.

Ejemplo

```
1  .data
2  valor: .word 10
3  result: .word 0
4  .text
5      daddi $sp, $0, 0x400 ; Inicializa $sp
6      ld $a0, valor($0)
7      jal factorial
8      sd $v0, result($0)
9      halt
10 factorial: daddi $sp, $sp, -16
11            sd $ra, 0($sp)
12            sd $s0, 8($sp)
13            beqz $a0, fin_rec
14            dadd $s0, $0, $a0
15            daddi $a0, $a0, -1
16            jal factorial
17            dmul $v0, $v0, $s0
18            j fin
19 fin_rec:   daddi $v0, $0, 1
20 fin:      ld $s0, 8($sp)
21          ld $ra, 0($sp)
22          daddi $sp, $sp, 16
23          jr $ra
```