Herramientas ARM

Jose Manuel Lopez Lemus

Universidad San Carlos de Guatemala

Facultad de ingeniería.

Ingeniería en ciencias y sistemas

Indice

Competencia

Analiza e implementa herramientas y entornos de desarrollo específicos para arquitecturas ARM, reconociendo su importancia en el diseño y programación de sistemas embebidos.

Usando el ensamblador de ARM (GDB)

Objetivos

Comprender el funcionamiento del ensamblador ARM y su sintaxis básica para desarrollar programas en lenguaje de bajo nivel.

Aplicar el uso del depurador GDB para analizar, ejecutar paso a paso y depurar programas escritos en lenguaje ensamblador para arquitecturas ARM.

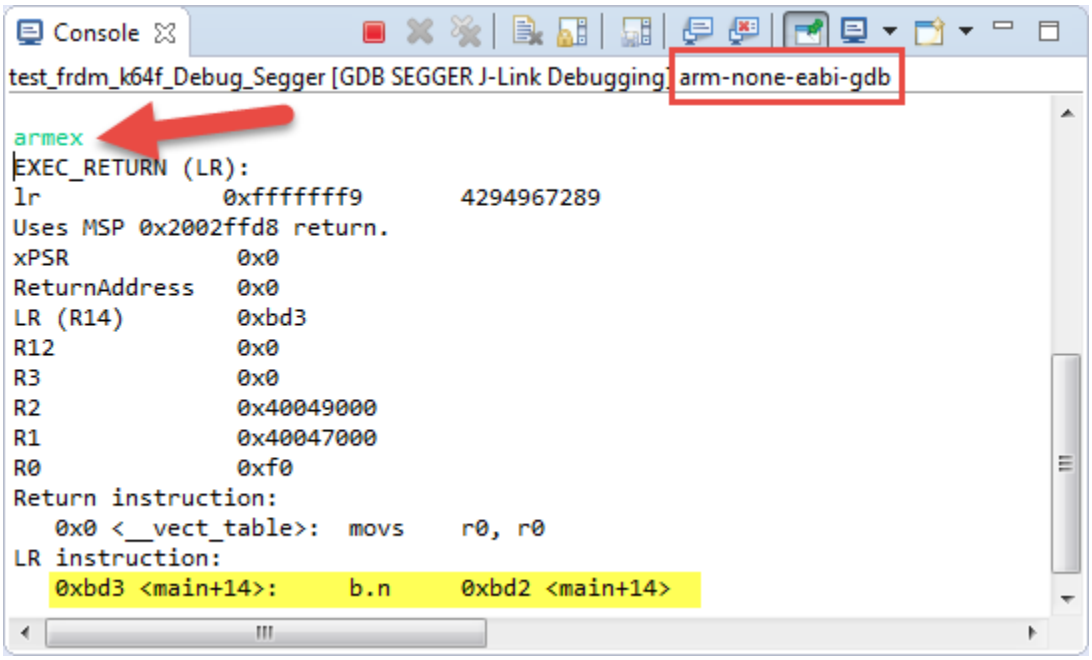
Introducción

El lenguaje ensamblador de ARM es fundamental en el desarrollo de sistemas embebidos y aplicaciones que requieren control detallado del hardware. ARM, por su arquitectura RISC (Reduced Instruction Set Computer), permite una programación eficiente y optimizada a nivel de instrucciones. Una herramienta clave para trabajar con programas ensamblador es GDB (GNU Debugger), un depurador potente que permite al programador examinar lo que ocurre dentro de un programa mientras se ejecuta o después de que ha fallado. Con GDB, se pueden observar los registros, controlar el flujo de ejecución, establecer puntos de ruptura y verificar resultados, todo lo cual es esencial para comprender cómo interactúa el código ensamblador con el procesador ARM. Este subtema busca introducir al estudiante en el uso básico de estas herramientas para fortalecer su dominio en programación de bajo nivel.

Contenido

El lenguaje ensamblador ARM permite programar directamente sobre el hardware, utilizando instrucciones específicas de la arquitectura RISC. Su uso es común en dispositivos embebidos, microcontroladores y sistemas de bajo consumo. Para compilar y depurar estos programas en sistemas Linux, se emplean tres herramientas clave: as (ensamblador), ld (enlazador) y gdb (depurador).

Un programa típico en ensamblador ARM inicia con la sección .text, que contiene el código, y puede usar etiquetas como _start para definir el punto de entrada. El código se escribe utilizando instrucciones como MOV, ADD, SUB, entre otras.



Herramientas básicas:

as: Ensambla el archivo .s y genera un archivo objeto .o.

ld: Enlaza el archivo objeto y produce un ejecutable.

gdb: Permite ejecutar y depurar el programa paso a paso, observar los registros y controlar la ejecución.

Comandos útiles en GDB:

- start: Inicia el programa desde el punto de entrada.
- break _start: Establece un punto de ruptura en una etiqueta.
- next o n: Ejecuta la siguiente instrucción.
- info registers: Muestra el estado de los registros.
- x/4xw $sp: Muestra contenido en memoria desde el stack pointer.

El uso conjunto de estas herramientas permite al estudiante comprender cómo se ejecutan las instrucciones ensamblador, observar cómo cambian los registros, y detectar errores lógicos o de ejecución. Esta práctica es fundamental para dominar el control a nivel de hardware y entender mejor cómo trabaja un procesador ARM.

Pongamos en Práctica la Teoría

Caso de Estudio 1: Suma de dos números y visualización de registros

Se desea escribir un programa en ensamblador ARM que sume dos números (5 y 3) y almacene el resultado en un registro. Luego, se usará GDB para ejecutar el programa paso a paso y observar cómo cambia el contenido de los registros.

Código base (suma.s):

```
.global _start

_start:

    MOV R0, #5      @ R0 = 5

    MOV R1, #3      @ R1 = 3

    ADD R2, R0, R1  @ R2 = R0 + R1
```

Pasos de análisis en GDB:

1. Ensamblar y enlazar el código.
2. Iniciar GDB con gdb suma.
3. Colocar un breakpoint en _start y usar next para ejecutar línea por línea.
4. Usar info registers para verificar que R2 contiene el valor 8.

Aprendizaje clave:
Este caso permite comprender la manipulación directa de registros y el impacto inmediato de las instrucciones MOV y ADD.

Caso de Estudio 2: Manejo de bucle simple con decremento

Se requiere crear un bucle en ensamblador ARM que cuente de 3 a 0, disminuyendo el valor en cada iteración. Usando GDB, se observará el flujo del bucle y los saltos condicionales.

Código base (bucle.s):

```
.global _start

_start:

    MOV R0, #3      @ Inicializar contador

loop:

    SUBS R0, R0, #1 @ Decrementar R0

    BPL loop        @ Saltar a 'loop' si R0 >= 0 (flag positivo)
```

Pasos de análisis en GDB:

1. Ensamblar, enlazar e iniciar GDB con gdb bucle.
2. Establecer un breakpoint en loop y observar el valor de R0 en cada ciclo.
3. Confirmar el uso del flag negativo para la condición de salto.

Aprendizaje clave:
Este caso permite entender el uso de instrucciones condicionales (SUBS, BPL) y cómo los flags del procesador influyen en el flujo del programa.

Conclusion

El uso del ensamblador ARM junto con GDB permite un entendimiento profundo del funcionamiento interno de un programa, ya que proporciona control total sobre los registros, la memoria y el flujo de ejecución. Esto es clave para el desarrollo en sistemas embebidos y aplicaciones de bajo nivel donde la eficiencia y el control son prioritarios.

Creación de Funciones

Objetivos

- Comprender la estructura y reglas para definir y llamar funciones en ensamblador ARM.
- Aplicar el uso correcto del stack y los registros al crear funciones reutilizables.
- Desarrollar la capacidad de modularizar programas ensamblador mediante el uso de subrutinas.

Introducción

En programación ensamblador, especialmente en arquitecturas como ARM, la creación de funciones o subrutinas permite dividir el código en bloques lógicos reutilizables. Las funciones son esenciales para evitar la repetición de código, mejorar la legibilidad y facilitar la depuración. A diferencia de los lenguajes de alto nivel, en ARM se debe gestionar de forma manual el paso de parámetros, el almacenamiento del estado de los registros y el retorno de resultados, todo a través del uso adecuado del stack (pila) y convenciones de llamada como la AAPCS (ARM Architecture Procedure Call Standard). Comprender este proceso es fundamental para escribir código más eficiente, ordenado y escalable.

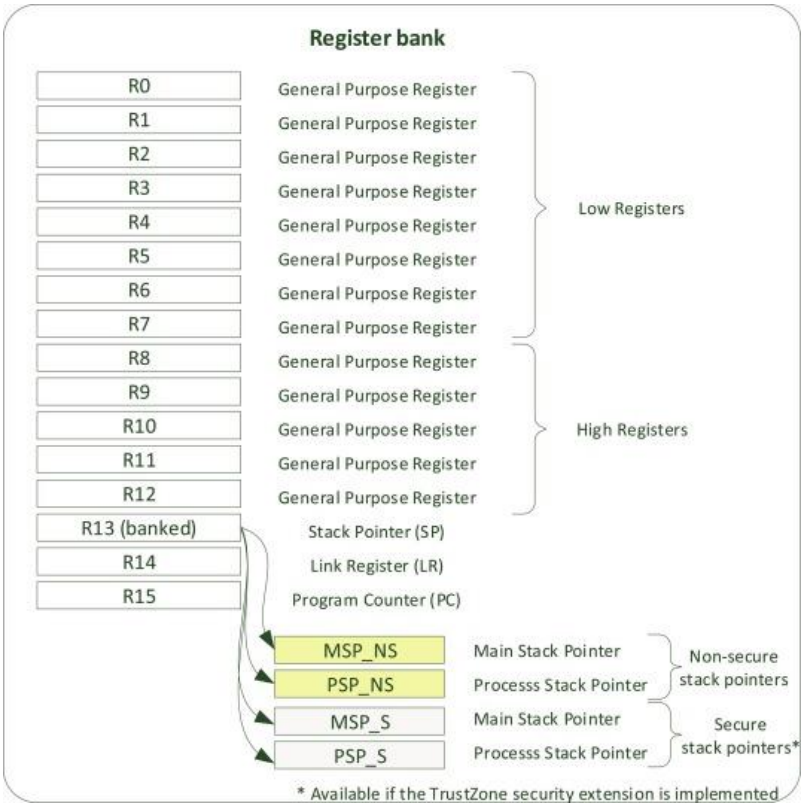
Contenido

En ensamblador ARM, las funciones (o subrutinas) permiten reutilizar bloques de código para realizar tareas específicas. A diferencia de los lenguajes de alto nivel, en ARM el programador debe encargarse explícitamente del paso de parámetros, la preservación del estado de los registros y la recuperación del control al final de la función.

Estructura básica de una función en ARM:

1. **Entrada a la función:**
    - Guardar los registros que se utilizarán (por convención).
    - Usar el stack (PUSH o STMFD) para preservar el estado.
2. **Cuerpo de la función:**
    - Instrucciones que realizan la tarea deseada.
    - Se pueden usar registros temporales (R0–R3 para argumentos y retorno).

3. **Retorno:**
    - Restaurar registros (POP o LDMFD).
    - Usar la instrucción BX LR para volver al llamador.



Convenciones importantes:

- **R0–R3**: Usados para pasar argumentos y devolver resultados.
- **R4–R11**: Deben conservarse si se modifican dentro de la función.
- **LR (Link Register)**: Guarda la dirección de retorno; debe protegerse si la función llama a otras funciones.
- **SP (Stack Pointer)**: Apunta al tope de la pila y debe mantenerse consistente.

El uso correcto de funciones en ensamblador permite una mejor organización del código, facilita la depuración y promueve buenas prácticas como el uso de convenciones de llamada y la protección del estado del programa.

Pongamos en Práctica la Teoría

Caso de Estudio 1: Función para multiplicar dos números

Se desea crear una función que reciba dos números en los registros R0 y R1, los multiplique y devuelva el resultado en R0. La función será llamada desde el punto de entrada _start y se observará el valor resultante en GDB.

Código base:

```
.global _start

_start:
    MOV R0, #7     @ Primer número
    MOV R1, #5     @ Segundo número
    BL multiplicar @ Llamar función
    B fin

multiplicar:
    PUSH {LR}
    MUL R0, R0, R1 @ R0 = R0 * R1
    POP {LR}
    BX LR

fin:
    B fin
```

Objetivo de análisis:
Validar que R0 contiene el resultado (35) tras el retorno de la función, y observar la preservación del registro LR.

Caso de Estudio 2: Función con preservación de múltiples registros

Se construye una función que realiza una operación más compleja y necesita utilizar registros adicionales (R4 y R5). Se evalúa cómo deben guardarse y restaurarse correctamente.

Código base:

```
.global _start

_start:
    MOV R0, #10
    MOV R1, #4
    BL operacion
    B fin

operacion:
    PUSH {R4, R5, LR}
    MOV R4, R0       @ Copiar R0 en R4
    MOV R5, R1       @ Copiar R1 en R5
    ADD R0, R4, R5   @ R0 = R4 + R5 = 14
    SUB R0, R0, #2   @ R0 = 14 - 2 = 12
    POP {R4, R5, LR}
    BX LR

fin:
    B fin
```

Objetivo de análisis:
Observar cómo se manejan múltiples registros temporales dentro de una función y cómo deben ser salvados/restaurados para evitar interferencia con el llamador.

Conclusión General

El dominio de las herramientas ARM, como el lenguaje ensamblador y el depurador GDB, permite al estudiante desarrollar una comprensión profunda del funcionamiento interno de los sistemas embebidos y arquitecturas RISC. Aprender a escribir funciones, manejar registros, y depurar código paso a paso no solo fortalece la lógica de programación, sino que también fomenta buenas prácticas como el modularidad y el uso eficiente de recursos. Estas habilidades son fundamentales para diseñar software de bajo nivel optimizado y confiable, lo que resulta especialmente valioso en campos como la robótica, la electrónica y el desarrollo de firmware.