SISTEMAS INFORMÁTICOS

CURSO 2011-2012

SIMULADOR ARM EN EL ÁMBITO DOCENTE

Realizado por:

José Enrique Celador Hernández

Ignacio Lucas Martín

Carla Torres Bustillos

Dirigido por:

Carlos García Sánchez

Dpto. Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

Agradecimientos

En este apartado, queremos expresar nuestros agradecimientos a nuestras familias, que siempre nos han brindado su apoyo y ánimo a lo largo de la carrera y muy especialmente en este proyecto.

También, queremos agradecer a todas aquellas personas, compañeros, amigos y profesores, que nos han ayudado a lo largo de los años a madurar personal e intelectualmente desde el mundo de la informática. Un camino que ha estado lleno de retos, superaciones y esfuerzos que encuentra su punto culminante con este proyecto.

Finalmente, dar las gracias a todas aquellas personas anónimas que nos han ayudado en la realización de este proyecto. Destacar la colaboración recibida en los foros de la comunidad de la aplicación jEdit, de Debian-ARM y el blog de Francesco Balducci.

Índice General

Páginas

Índice de ilustraciones

RESUMEN: "Simulador ARM en el ámbito docente"

El proyecto desarrollado en el ámbito de la asignatura de Proyectos Informáticos surge ante la limitación en el uso de las instalaciones del entorno de aprendizaje ARM-Embest.

Una de las dificultades en el aprendizaje del alumnado de las asignaturas del área de Estructura de Computadores surge por la falta de disponibilidad del entorno ARM-Embest. Por un lado, el coste de las licencias se traduce en un número limitado de laboratorios disponibles en la facultad, dificultando la dotación de nuevas instalaciones en la coyuntura económica actual. Por otro lado, el coste del entorno hace imposible su adquisición por parte de los alumnos viéndose obligados a hacer uso de las instalaciones en una franja horaria muy restrictiva ante la demanda de los horarios actualmente.

Bajo esta premisa, este proyecto trata de explorar otras opciones que cubran la oferta docente con un coste muy inferior. En este proyecto se investigan opciones basadas en software libre que emulen el comportamiento del entorno ARM-Embest actual. El trabajo que presentamos en esta memoria se basa en la integración de herramientas de emulación/compilación de sistemas ARM, añadiendo un interfaz de comunicación para que cualquier usuario pueda hacer uso de un entorno de aprendizaje alternativo.

Por último, en la memoria también se incluyen una serie de prácticas a modo de ejemplos con su código fuente, que hacen uso del sistema emulado ARM y sus periféricos de forma análoga a como están propuestas en la actualidad.

PALABRAS CLAVE

- ARM
- Compilación cruzada
- Toolchain
- QEMU
- Debian ARM
- Estructura de Computadores (EC)
- Integración de herramientas de software libre

ABSTRACT: "ARM simulator in the academic environment"

The project, developed in the field of Computing Projects subject, arises from the limitations in the use of the facilities of the training environment ARM-Embest.

One of the difficulties students have whilst studying subjects relating to Computer Structure is due to the lack of availability of the ARM-Embest environment. On one hand, the cost of licenses results in a limited number of laboratories in the faculty, hindering the provision of new facilities in the current economic climate. On the other hand, the cost of the environment is prohibitive for students who have no other choice than to make use of the facilities within a very restrictive time period, given the current demand on the infrastructure.

Given this premise, this project attempts to explore other options to attend the academic requirement, at a much lower cost. This project will investigate options based on free software that emulates the behaviour of the current ARM-Embest environment. The work presented in this paper is based on the integration of ARM system emulation, compiling tools, adding a communication interface such that any users may make use of an alternative educational environment.

Finally, the paper also includes a series of practical exercises as examples with source code, that make use of the emulated ARM system and its peripherals in the same way in which they are currently proposed.

KEYWORDS

- ARM
- Cross-compiling
- Toolchain
- QEMU
- Debian ARM
- Computer Organization Designer
- Integration of free software tools

CAPÍTULO 1
MOTIVACIÓN DEL PROYECTO

Capítulo 1 – Motivación del proyecto

En este capítulo, se explica brevemente, las razones que nos han motivado a realizar este proyecto así como los objetivos que se persiguen en él. Finalmente, concluimos el capítulo, explicando la estructura de este documento.

1.1 Justificación y motivación

Hoy en día, existen entornos de desarrollo, como Embest IDE, que junto con una placa y emuladores como JTAG, hacen posible llevar a cabo el uso de procesadores ARM en el ámbito docente. Las universidades cuentan, en sus instalaciones, con entornos de desarrollo para el uso del alumno.

La demanda de uso de dichas instalaciones supone una limitación de acceso a dichas instalaciones. Este hecho viene motivado por el coste de las licencias (alrededor de unos mil euros por puesto), lo que se traduce en un gran desembolso económico por parte de la universidad que ante la coyuntura actual dificulta la dotación de nuevos puestos. El alto precio de la licencia hace inviable la compra por parte del alumno, viéndose obligado a desarrollar su trabajo en los escasos turnos libres disponibles.

Otra dificultad añadida, es el coste que supone mantener y cuidar las placas de desarrollo. Además, con el tiempo, el material se va degenerando con el uso y quedándose paulatinamente obsoleto respecto a las nuevas tecnologías que van surgiendo en el mercado.

Es por esto que hemos pensando que sería una buena idea desarrollar un software libre y gratuito, fácil de manejar con el que el alumno o cualquier usuario, pueda trabajar y desarrollar sus tareas cómodamente, desde casa o desde cualquier otro lugar, eliminando las limitaciones que se encuentran hoy en día en la facultad.

1.2 Objetivos

El objetivo principal de este proyecto es desarrollar una aplicación o herramienta informática libre y gratuita, capaz de simular un procesador ARM, independientemente de la plataforma en la que se encuentre instalada dicha aplicación. Dicha aplicación, se manejará a través de una interfaz gráfica.

A su vez, se pretende diseñar dicha herramienta lo más modular posible. Así, si un usuario quiere hacer uso del entorno de aprendizaje utilizando herramientas externas diferentes a las que hemos empleado en este proyecto, únicamente ha de configurar el entorno sin la necesidad de recodificar el entorno de nuevo.

Para llevar a cabo este propósito, hemos fraccionado el desarrollo de este proyecto en dos partes:

1. **Confección de un procesador ARM**

   El objetivo es crear o instalar un procesador ARM en la plataforma Linux. De esta manera, el usuario podrá ejecutar binarios en lenguaje máquina de ARM, desde una arquitectura distinta. Para ello, se hace uso de software externo de código abierto y gratuito.

2. **Implementación de una interfaz gráfica de usuario**

   En esta parte, se pretende desarrollar una interfaz gráfica, intuitiva y fácil de manejar que permita al usuario realizar su trabajo cómodamente.
   Su implementación se realizará en lenguaje Java usando como entorno de programación NetBeans IDE, de código abierto y multiplataforma.

1.3 Estructura del documento

Empezamos haciendo un repaso general del procesador ARM: su evolución a lo largo de la historia, las distintas familias ARM que existen hoy en día y sus usos en la actualidad, su arquitectura, las tecnologías ARM más importantes, una explicación de por qué usar ARM en la docencia y, finalmente, el entorno de desarrollo Embest (Capítulo 2).

En el siguiente capítulo (Capítulo 3) se explica cómo se han llevado a cabo las dos partes en las que se había dividido el desarrollo del proyecto. Por una parte, se explica cómo se ha construido la herramienta de simulación ARM (obtención del procesador ARM) y por otra, la herramienta de integración (interfaz gráfica de usuario).

En el siguiente capítulo (Capítulo 4) se comentan los resultados obtenidos tras hacer pruebas del simulador proponiendo una serie de prácticas. Esta colección de práctica están inspiradas en las prácticas propuestas, en la actualidad, en la asignatura de Estructura de Computadores (EC) del grado de Informática de la Universidad Complutense de Madrid.

En el capítulo 5, concluimos con las principales aportaciones de este proyecto, posible trabajo futuro y algunas de las dificultades encontradas.

Finalmente, se adjunta un CD que incluye:

- Una guía o manual de usuario de la aplicación
- El fichero ejecutable Java *SimuladorARM*.jar correspondiente a la herramienta de integración.
- La imagen de disco de Debian-ARM
- La herramienta Croostool-ng 1.15.0
- Una carpeta con las prácticas utilizadas para realizar las pruebas de la aplicación
- Una carpeta con el código fuente de la herramienta de integración
- Manuales de la placa Versatilepb y de los periféricos soportados por la misma

CAPÍTULO 2
INTRODUCCIÓN A ARM Y ENTORNO DE DESARROLLO EMBEST

Capítulo 2 – Introducción a ARM y entorno de desarrollo Embest

En este capítulo hacemos un repaso general del procesador ARM: su evolución a lo largo de la historia, las distintas familias ARM que existen hoy en día, su arquitectura, etc. Así como la importancia de usar ARM en la docencia. Terminamos el capítulo explicando sucintamente el entorno de desarrollo Embest y sus características.

2.1 Introducción a ARM

La arquitectura ARM (Advanced RISC Machine) fue originalmente desarrollada por Acorn Computer Limited en Cambridge (Inglaterra), entre 1983 y 1985. Fue el primer microprocesador RISC desarrollado para uso comercial. La arquitectura ARM es el repertorio de instrucciones de 32 bits más extendido y usado en numerosas aplicaciones, especialmente en el diseño de arquitecturas system-on-chip formando el núcleo del sistema.

El concepto novedoso fue diseñar el núcleo de la familia de microcontroladores sin fabricarlos, cediendo la producción a empresas de semiconductores como NXP, Atmel, Texas Instruments, Freescale, Analog Devices, etc. Cada fabricante respetaría el núcleo original y el repertorio de instrucciones, pero pudiendo incorporarle sus propios periféricos (conversores A/D y D/A, UART,…), lo cual supuso un estímulo para el crecimiento de soluciones hardware, pero produjo incompatibilidades entre componentes similares de diversas marcas.

Las características del ARM hacen que su arquitectura sea la más importante en el sector de los sistemas integrados. En primer lugar, los núcleos de ARM son muy simples en comparación con la mayoría de los procesadores de propósito general, lo que significa que se pueden fabricar usando un número relativamente pequeño de transistores, dejando espacio en el chip para aplicaciones específicas. Es decir, que un chip de ARM puede contener varios controladores de periféricos, un procesador de señal digital y memoria en el chip, junto con su núcleo ARM.

En segundo lugar, los dispositivos ARM están destinados a minimizar la energía de consumo, requisito fundamental para los sistemas móviles empotrados. En tercer lugar, la arquitectura ARM es altamente modular, siendo el único componente obligatorio de un procesador ARM el pipeline; todos los demás componentes, incluida la memoria caché, MMU, etc., son opcionales.

Por último, aunque el ARM tenga un procesador pequeño y de bajo consumo, proporciona un alto rendimiento para aplicaciones empotradas.

Teniendo en cuenta lo descrito anteriormente, conviene destacar que la arquitectura ARM incorporó algunas de las características de la arquitectura RISC como:

- *Arquitectura load-store*: las instrucciones que acceden a memoria son distintas de las instrucciones que procesan los datos.
- *Formato fijo de palabra de instrucción*: Todas de 32 bits
- *Máquina de tres direcciones:* (1er operando, 2º operando y resultado) en la que en la instrucción se trabajará con registros punteros a operandos pero no directamente con memoria.

También incorporó algunos conceptos novedosos como:

- *Ortogonalidad*: Las instrucciones siguen un molde repetitivo. No existen registros especiales y esas operaciones pueden realizarse con todos los registros.

- *Implementación de varios modos de trabajo o jerarquías*: Aparece por primera vez en un microcontrolador el modo supervisor y el modo usuario. El primero, con plenos poderes, supervisará la actuación de los programas de usuario, impidiendo que el sistema colapse y que solo pudieran hacerlo aplicaciones de usuario que podrían ser recuperadas en el modo supervisor.

- *Excepciones*: Son los casos particulares de control de flujo (saltos) en los que un efecto posiblemente no deseado de la ejecución produce un error o fallo, y pasa a ejecutar un tramo de programa que ejecute esta situación excepcional. Las interrupciones son un caso particular de las excepciones.

- *Thumb*: Aparece la posibilidad de operar con palabras de 16 bits, aunque las instrucciones son menos versatiles que las de 32 bits.

- *Bajo consumo*: Se bajó la alimentación a 1,8 V para el núcleo y un buen uso de la frecuencia de trabajo.

- *Interrupciones*: Se dispondrá de dos interrupciones vectorizadas IRQ y FIQ, teniendo esta última una mayor prioridad. Se tomarán como casos particulares de las excepciones.

- *Eficiencia en la generación de código en C*: Esta arquitectura fue concebida para trabajar eficientemente con este lenguaje.

- *Herramientas de depuración de bajo coste e incorporadas*: Cada microcontrolador dispone de una interfaz JTAG, que permite utilizar herramientas de depuración en tiempo casi-real de muy bajo coste.

2.2 Historia de la Arquitectura ARM

2.2.1 Acorn

El primer circuito integrado ARM fue desarrollado para las computadoras Acorn. Acorn era una de las principales marcas de ordenadores personales británicos. El éxito inicial de Acorn se originó cuando la BBC (*British Broadcasting Corporation*) creo un nuevo ordenador para el hogar conocido como el microcomputador BBC, llegando a ocupar la primera posición en ventas de ordenadores personales, en Gran Bretaña, en 1982.

El micro BBC fue la base para el procesador 6502 de 8 bits de Rockwell, el mismo chip que impulsó el Apple II. Los modelos iniciales de Acorn ofrecían gráficos a color y un acceso a memoria de 32 KBytes.

2.2.2 El primer ARM

Los trabajos de desarrollo del chip ARM comenzaron a hacerse en 1983 para terminar en abril de 1985, creando un dispositivo que reunía las características del procesador 6502 pero en un entorno RISC de 32 bits, implementando un dispositivo pequeño, fácil de diseñar, probar y fabricar con un coste mínimo.

El primer procesador comercial RISC en el mundo y el primer procesador ARM fue el ARM1. Fabricado por primera vez en abril de 1985 con tecnología VLSI. Sin embargo, este procesador se utilizó como prueba y no llegó a comercializarse.

2.2.3 ARM2

El ARM2 fue la primera versión utilizada comercialmente y se lanzó en 1986. Disponía de un bus de datos de 32 bits, un espacio de direcciones de 26 bits (64 Mb) y dieciséis registros de 32 bits. Uno de estos registros se utilizaba como contador de programa, aprovechándose sus 4 bits superiores y los 2 inferiores para guardar el estado del procesador.

El ARM2 es posiblemente el procesador de 32 bits más simple del mundo, ya que posee sólo 30.000 transistores. Su simplicidad se debe a la ausencia de micro código y caché. Por lo tanto, tiene un consumo de energía bastante bajo, ofreciendo también un rendimiento superior al 286.

2.2.4 ARM3

El chip ARM3, a diferencia de sus predecesores, contiene una memoria caché de 4 KBytes que le permite mejorar su rendimiento. Se le añade una nueva instrucción, SWP que permite el acceso a memoria de forma atómica con una única instrucción.

2.2.5 ARM4 y ARM5

Estos nunca fueron realizados. A finales de los 80 Apple Computer y VLSI Technology comenzaron a trabajar con Acorn en nuevas versiones del núcleo ARM. El trabajo fue tan importante que Acorn creo una nueva compañía llamada Advanced RISC Machines Ltd. que pasó a ser denominada ARM Ltd., donde los siguientes procesadores saltaron las versiones 4 y 5.

2.2.6 ARM6

La unión de Apple y ARM daría como fruto el ARM6, con capacidad de direccionamiento de 32 bits, el procesador tenía 31 registros con seis nuevos modos de procesador. Apple usó el ARM 610 (basado en ARM6) como base de la PDA Apple Newton. En 1994, Acorn empleó el ARM 610 como CPU principal de su computador RiscPC.

2.3 Familias de la Arquitectura ARM

2.3.1 Familia ARM7

Introducida en 1994, la familia ARM7 es la más usada en los sistemas embebidos de 32 bits. Formada por un microprocesador RISC de 32 bits, alcanza los 130MIPs, e incorpora el conjunto de instrucciones de 16 bits de la tecnología Thumb.

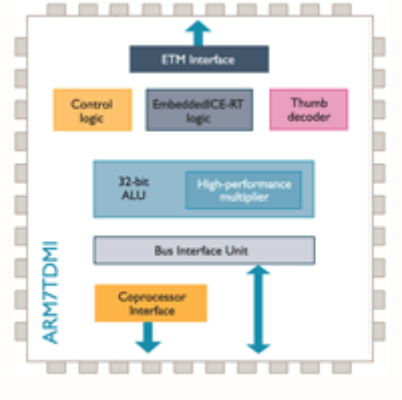


FIGURA 1: ARM7TDMI

La familia ARM7 está compuesta por los siguientes modelos: ARM7TDMI, ARM7TDMI-S, ARM7EJ-S y el ARM720T, cada uno de los cuales tiene sus características específicas.

Aplicaciones

Las más conocidas son:

- Dispositivos de audio: MP3, WMA.
- Teléfonos móviles, especialmente los teléfonos Nokia
- Routers
- iPod de Apple
- Nintendo DS y Game Boy Advanced

Características

- Arquitectura RISC de 32 bits
- Rendimiento cercano a los 130 MIPs con un procesador típico de 0.13µm
- Diseño reducido y muy bajo consumo de potencia
- Programación de alto nivel, comparable con los microcontroladores de 16 bits

- Compatible con varios sistemas operativos como Windows CE, Palm OS, Symbian OS y Linux
- Amplio número de herramientas de desarrollo
- Simulación para los principales entornos de desarrollo EDA (*Electronic Design Automation*)
- Código compatible con los procesadores ARM9, ARM9E, ARM10

2.3.2 Familia ARM9

Dentro de esta familia podemos encontrar dos subcategorías, las basadas en ARM9 y las basadas en ARM9E, que incluyen los procesadores ARM926EJ-S, ARM946EJ-S, ARM966E-S, ARM996HS, ARM920T y ARM922T.

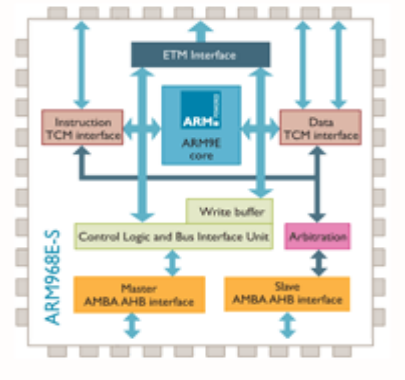


FIGURA 2: ARM968E-S

La familia de procesadores ARM9 toma como base el procesador ARM9TDMI e incorpora los 16 bits del conjunto de instrucciones Thumb, que mejora el rendimiento del código en un 35%. La familia ARM9 abandona la arquitectura von Neumann instalando la arquitectura Harvard con instrucciones separadas y buses para datos y cachés, mejorando significativamente la velocidad. Por otro lado, los procesadores de la familia ARM9E permiten que un solo procesador trabaje con microcontroladores, DSP y aplicaciones Java.

Algunas de las principales diferencias frente a los núcleos ARM7 es que aumentan el número de transistores, lo que facilita menor riesgo de sobrecalentamiento, mejora en las frecuencias de reloj, mejora la velocidad porque los accesos a memoria son más rápidos (loads y stores) debido a la arquitectura Harvard y a los nuevos estados del pipeline.

Aplicaciones

- Productos de consumo digital como: consolas de videojuegos, MP3, video MPEG-4, numerosos dispositivos móviles (smartphones), domótica, etc.
- En la fotografía por ejemplo: cámaras de fotos digitales, cámaras de video digital e impresoras.
- Otras aplicaciones como Routers, PDA's, y ciertas aplicaciones en el campo de la robótica, Bluetooh, Wireless LAN 802.11, GPS, controladores USB, controladores de Bluetooh, scanners médicos, controladores de Disco Duro, etc.

Características

- Procesador RISC de 32 bits que poseen el conjunto de instrucciones ARM y Thumb
- Realiza cinco estados de pipeline a 1.1MIPS/MHz
- Bus de interfaz de 32 bits
- Unidad de memoria magnética (MMU), soporte para Windows CE, Symbian OS, Linux, Palm OS
- Instrucciones integradas y caché de datos
- Excelente soporte de depuración para diseños SoC
- Ocho buffers de entrada de escritura

2.3.3 Familia ARM11

La familia ARM11 comprende un gran abanico dentro de los microprocesadores de alto rendimiento introduciendo los núcleos ARMv6. Esta familia está constituida principalmente por cuatro series de procesadores: ARM1136J(F)-S, ARM1156T2(F)-S, ARM1176JZ(F)-S y el microprocesador ARM11 MPCore. Estos incluyen SIMD (Single Instruction, Multiple Data), soporte para multiprocesador y una nueva arquitectura caché.

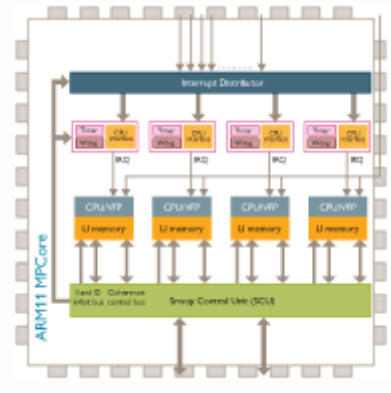


FIGURA 3: ARM11MIPCore

Las principales características que diferencian al ARM11 respecto a la familia ARM9 es que añade instrucciones de soporte multimedia (SIMD), acelera la respuesta IRQ e incorpora todas las características del ARM926EJ-S.

Aplicaciones

Son numerosas las aplicaciones en el mercado, destacando su uso en los smartphones: HTC, Nokia, LG, iPhone, Samsung, Sony Ericsson, Huawei, etc. Además, del kindle de Amazon y el iPod entre otras aplicaciones.

Características

- Arquitectura con conjunto de instrucciones ARMv6
- Extensiones ARM DSP
- Instrucciones SIMD (una instrucción para múltiples datos)
- Núcleo con tecnología Thumb-2 que mejora el rendimiento reduciendo los requerimientos de memoria.
- Bajo consumo de potencia: 0.6 mW/MHz (0.13µm, 1.2V) incluidos los controladores de caché
- Modos de bajo consumo para ahorro de energía
- Gestión inteligente de la energía (IEM)
- Alto rendimiento del procesador: 8 fases de pipeline
- Alto rendimiento del diseño del sistema de memoria
- Sistema de memoria de 64 bits de alto rendimiento que facilita acceder de forma más rápida a los datos para procesos multimedia y aplicaciones de redes.
- La interfaz del vector de interrupciones y el modo de latencia de interrupción baja que aceleran la respuesta de la interrupción y el rendimiento en tiempo real.
- Coprocesador en punto flotante para aplicaciones de control industrial y acelerador de gráficos 3D.

2.4 Arquitectura ARM

La arquitectura ARM es a *Reduced Instruction Set Computer* (RISC) que incorpora características típicas de la arquitectura RISC como:

- Banco de registros uniforme
- Arquitectura load/store, donde los datos son procesados solamente en los registros y no directamente en memoria.
- Modos de direccionamiento simple
- Instrucciones de ancho fijo de 32 bits que facilita la decodificación y el uso del pipeline
- Normalmente, ejecución en un único ciclo

A estas características la arquitectura ARM añade las siguientes peculiaridades:

- Control de la Unidad Aritmético Lógica (ALU) y el registro de desplazamiento (shifter) para maximizar el uso de ambas.
- Auto-incremento y auto-decremento de los modos de direccionamiento para optimizar lo bucles.
- Load y Store Múltiple que maximiza el paso de datos
- Ejecución condicional en la mayoría de las instrucciones para optimizar la ejecución del código.

En la figura se muestra un diseño genérico de la estructura interna de un procesador ARM. El sistema-on-chip esta constituido por el núcleo ARM, los periféricos y el controlador de interrupciones. Cuando se realiza una interrupción el controlador de interrupciones se encarga de diferenciar de qué tipo es y procede a enviar la señal IRQ o FIQ al núcleo ARM.

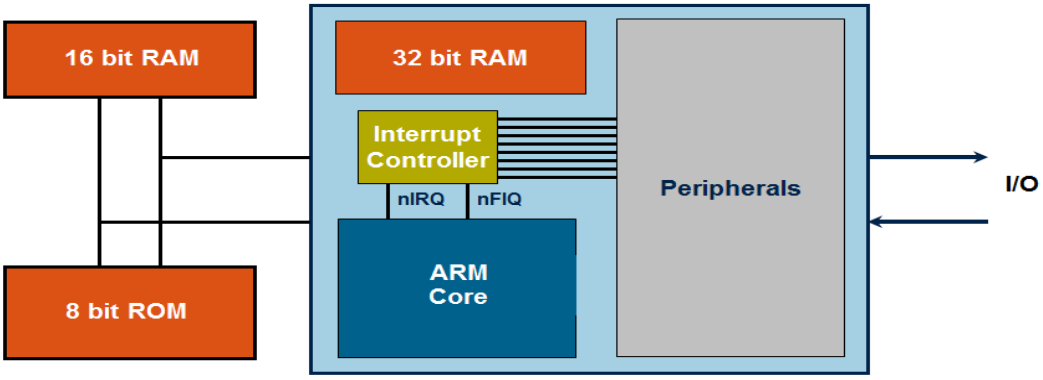


FIGURA 4: Estructura interna ARM

La organización interna de un ARM está constituida como se muestra a continuación:

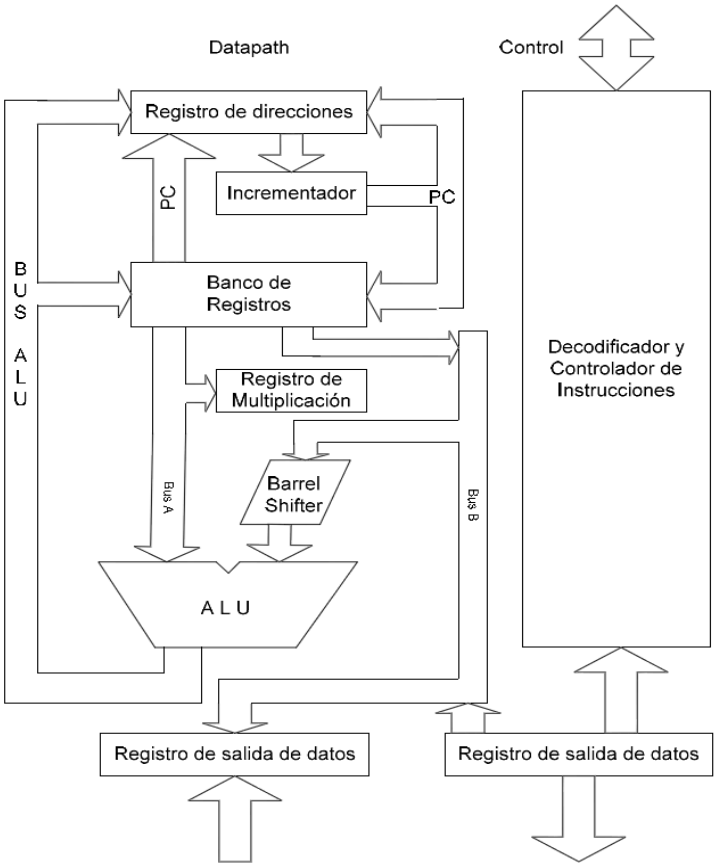


FIGURA 5: Organización interna ARM

En la figura se observa a nivel de bloques la arquitectura de un microcontrolador de ARM, dividido en dos bloques principales el encargado de los datos (datapath) y el decodificador y controlador de instrucciones, destacando los siguientes módulos:

- Posee un banco de registros
- La unidad aritmético lógica recibe uno de los operandos directamente del banco de registros mientras que el otro operando pasa por un registro de desplazamiento (barrel shifter).
- Dos puertos de lectura que están conectados al bus de datos A o al B
- Un puerto de escritura
- Puertos adicionales de lectura o escritura para el contador de programa (PC) r15
- Registro de desplazamiento que permite rotar o mover el segundo operando
- Consta de registros de direcciones, también llamadas direcciones de PC (program counter)

Los tipos de datos soportados por ARM son:

- *Byte*: 8 bits
- *Halfword*: 16 bits (dos bytes)
- *Word*: 32 bits (cuatro bytes)

La mayoría de los procesadores ARM implementan dos tipos de conjunto de instrucciones:

- 32 bits ARM Instruction Set
- 16 bits Thumb Instruction Set

2.4.1 Modos de ejecución

El ARM tiene siete modos básicos de ejecución:

- *User (usr):* Modo de ejecución normal, sin privilegios
- *FIQ (fiq):* Interrupciones rápidas y prioritarias
- *IRQ (irq):* Interrupciones de propósito general
- *Supervisor (svc):* Modo protegido para el sistema operativo, produce un reset cuando una instrucción de interrupción de software es ejecutada.
- *Abort (abt):* Usado para implementar memoria virtual y/o protección de memoria, cuando se ha producido una violación de acceso a memoria.
- *Undefined (und):* Soporte para instrucciones indefinidas

- *System (sys):* Ejecución de tareas privilegiadas del sistema operativo

El cambio de modo puede ser producido por software, a causa de una interrupción externa o por el procesamiento de una excepción. La mayoría de las aplicaciones se ejecutan en modo User. Todos los modos menos el modo User son modos privilegiados. Tienen acceso total a todos los recursos del sistema.

2.4.2 Registros

ARM tiene 31 registros de propósito general de 32 bits, de los cuales 16 de ellos son visibles en todo momento. Los otros son usados para incrementar la velocidad del procesamiento de las excepciones. Las instrucciones ARM pueden direccionar cualquiera de los 16 registros visibles.
El banco principal de 16 registros es usado por el código sin privilegios, son los registros del modo usuario (*User*). El modo *User* es distinto del resto de modos y no tiene privilegios, lo que significa:

- El modo *User* sólo puede pasar a otro modo de ejecución generando una excepción. La instrucción SWI aporta esta funcionalidad.
- Los sistemas de memoria y coprocesadores pueden restringir el acceso al modo *User*.

Tres de los 16 registros visibles tienen una funcionalidad específica:

- *Puntero de pila (Stack Pointer o SP):* El software suele utilizar R13 como puntero de pila
- *Registro de Enlace (Link Register o LR):* El registro 14 es el registro de enlace. Este registro contiene la dirección de la siguiente instrucción a ejecutar tras la ejecución de una instrucción de salto con enlace. También se emplea para almacenar la dirección de retorno al cambiar de modo de ejecución. El resto del tiempo, LR (R14) puede utilizarse como un registro de propósito general.
- *Contador de Programa (Program Counter o PC):* El registro R15 es el contador de programa

Los restantes trece registros no tienen un propósito especial y pueden ser empleados libremente por las aplicaciones.

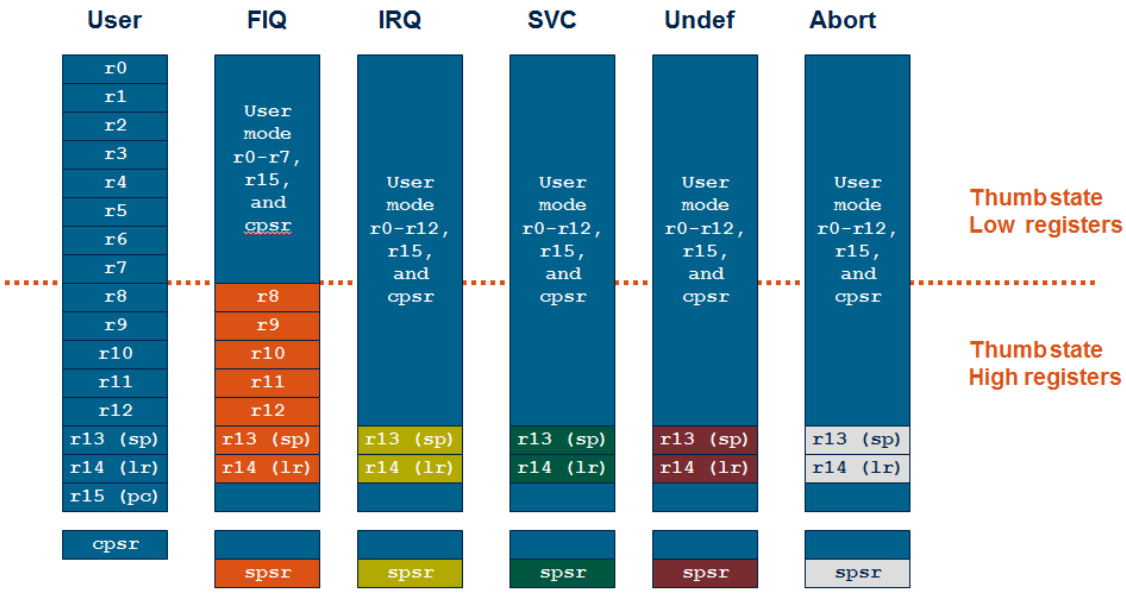


FIGURA 6: Registros ARM

En total el procesador ARM dispone de 37 registros, todos de 32 bits:

- 31 de propósito general, incluyendo el contador de programa
- 6 registros de estado

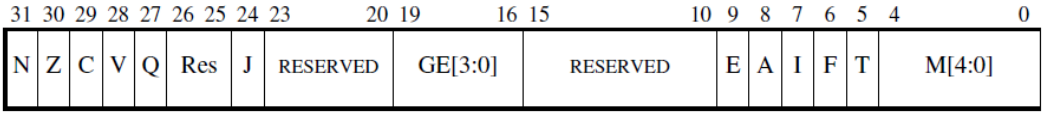
Los registros están en bancos parcialmente solapados, siendo el modo de ejecución actual quien controla qué banco está actualmente visible. En todo momento, los quince registros de propósito general (R0-R15), uno o dos registros de estado, son siempre visibles.

El registro de estado del programa

La arquitectura ARM consta de un registro de estado del programa actual CPSR (Current Program Status Register), más cinco registros de estado del programa guardado SPSR (Saved Program Status Registers) para ser usados con excepciones.
Los CPSR realizan:

- Guardan la información de la última operación realizada par la ALU
- Controla la habilitación y suspensión de las interrupciones
- Pone los modos de ejecución del procesador
- Contiene los bits de condición

El formato del CPSR y del SPSR se muestra en la figura:



Los bits de condición:

- Los bits N, Z, C y V (Negativo, cero, acarreo y desbordamiento) son los bits de condición o flags. Se utilizan para saber si una instrucción condicional debe ser ejecutada.
- El bit Q en las variantes E de ARMv5 y superiores se usa para indicar que se ha producido un desbordamiento y/o saturación en ciertas instrucciones DSP.
- Los bits GE[3:0] en ARMv6 y superiores se usan para comprobar la relaciones de igualdad y de orden en instrucciones SIMD.
- El bit E en arquitecturas ARMv6 y posteriores controla el endianess del manejo de datos

Los bits de inhabilitación de interrupciones:

- El bit A deshabilita los Data Aborts imprecisos cuando vale 1 (sólo en arquitecturas ARMv6 y superiores).
- El bit I deshabilita las interrupciones IRQ cuando vale 1, mientras que el bit F hace lo propio con las interrupciones FIQ.
- Los bits de modo M [4:0] indican el modo de ejecución del procesador

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARMv4 and above)

FIGURA 7: Modos de ejecución

El bit T se utiliza sólo en las arquitecturas Xt:

- T = 0, procesador en estado ARM
- T = 1, procesador en estado Thumb

El bit J: se utiliza en arquitecturas 5TE/J y superiores:

- J = 1, procesador en estado Jazelle.

El resto de bits están reservados para futuras expansiones.

2.4.3 Excepciones

Son generadas por fuentes internas o externas para desviar el procesador para gestionar un evento. El estado interno del procesador un momento antes de gestionar la excepción debe ser guardado de tal forma que el programa original pueda ser continuado una vez que la excepción haya sido completada.

El procesador ARM soporta siete tipos de excepciones, recogidos en la siguiente tabla donde se muestran éstos y el modo de ejecución empleado para procesar cada uno de ellos. Cuando se produce una excepción, se fuerza la ejecución de código desde unas posiciones de memoria fijas correspondientes al tipo de excepción, denominadas vectores de excepción.

Exception type	Mode	VE[a]	Normal address	High vector address
Reset	Supervisor		0x00000000	0xFFFF0000
Undefined instructions	Undefined		0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor		0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort		0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0	0x00000018	0xFFFF0018
		1	IMPLEMENTATION DEFINED	
FIQ (fast interrupt)	FIQ	0	0x0000001C	0xFFFF001C
		1	IMPLEMENTATION DEFINED	

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.

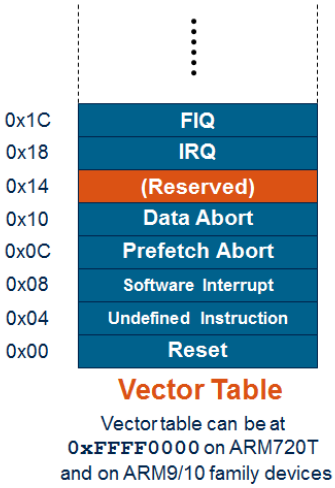


FIGURA 8: Tipos de excepciones

2.4.4 Pipeline

Una de las propiedades importantes de la arquitectura ARM mencionada anteriormente es su simplicidad. Por ello, el pipeline es la parte fundamental en esta arquitectura debido a que incrementa la velocidad de ejecución de las instrucciones, es decir, ejecuta más instrucciones en un ciclo de reloj.

Ahora pasaremos a describir la evolución del pipeline de la arquitectura ARM desde el procesador ARM1 hasta el ARM11.

Pipeline de 3 etapas

La figura muestra un pipeline de 3 etapas ARM que permaneció esencialmente sin cambios desde el procesador ARM7TDMI. Se trata del clásico fetch-decoder-execute, que en ausencia de riesgos del pipeline y de acceso a memoria completa una instrucción por ciclo. La primera etapa del pipeline lee una instrucción de la memoria, incrementa el valor del registro de dirección y almacena el valor de la siguiente instrucción que va a ser buscada (etapa fetch).

La etapa siguiente decodifica la instrucción y prepara las señales de control requeridas para su ejecución. La tercera etapa hace todo el trabajo: lee los operandos del banco de registros, realiza las operaciones de la ALU, lee o escribe en memoria, y si es necesario sobrescribe los valores de los registros.

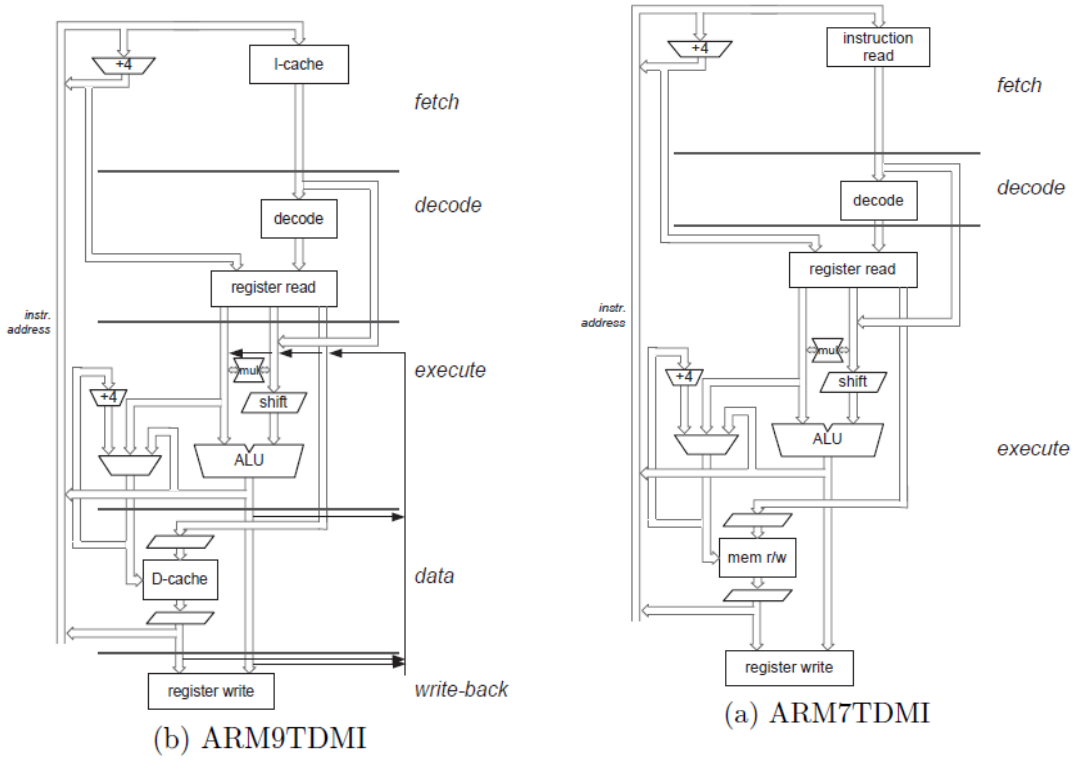


FIGURA 9: Pipeline de 3 etapas ARM

Pipeline de 5 etapas

La estructura del pipeline de 3 etapas dispone de un solo puerto de memoria, lo que significa que cada instrucción de datos atasca el pipeline, porque la instrucción siguiente no puede ser ejecutada mientras la memoria se está leyendo o escribiendo. Una forma de evitar este problema, que se usó en el ARM9TDMI y en las micro arquitecturas posteriores, es separar las instrucciones y los datos de caché. Esto permite modificar el pipeline para evitar paradas con las instrucciones de datos.

En primer lugar, el ARM9TDMI movió el registro de lectura a la etapa de decodificación, ya que la etapa de decodificación de instrucciones era más corta que la de ejecución. En segundo lugar, la etapa de ejecución se dividió en tres etapas. La primera realiza los cálculos aritméticos, la segunda los accesos a memoria (esta etapa permanece inactiva cuando se ejecutan instrucciones de procesamiento de datos) y la tercera escribe los resultados en el banco de registros.

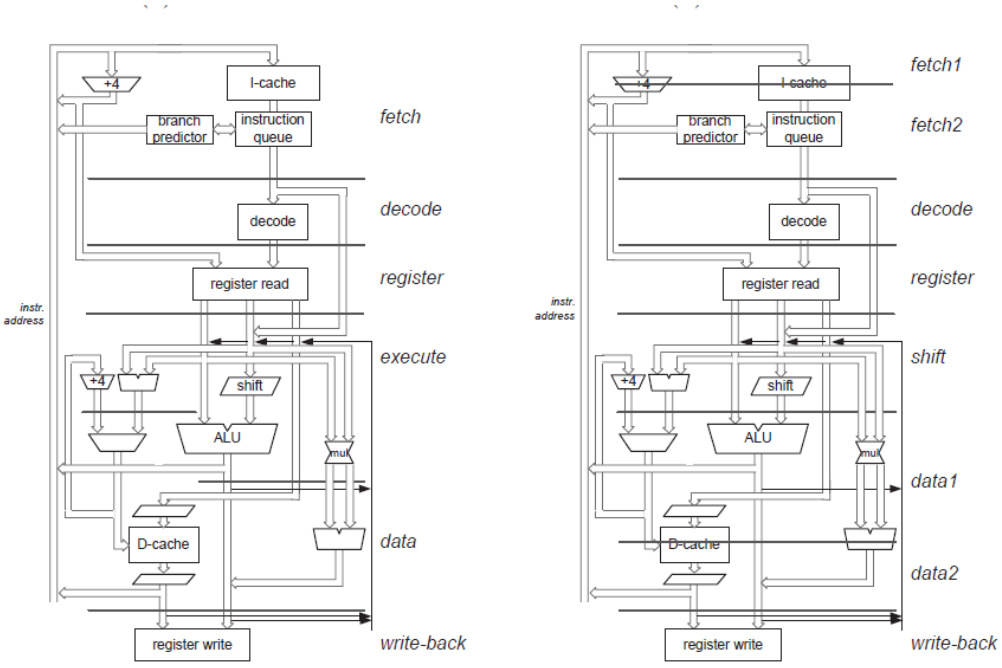
Esto proporciona un pipeline más equilibrado y veloz, pero con una nueva complicación, la necesidad de anticipar los datos para evitar las dependencias de datos entre las distintas etapas sin que se pare el pipeline.

Pipeline de 6 etapas

En el núcleo ARM10 se mejoró el pipeline. Los diseñadores se dieron cuenta de que el rendimiento dependía del ancho de banda para acceder a la memoria. Por lo tanto, hicieron que tanto las instrucciones como los buses de datos trabajaran a 64 bits. La etapa *fetch* permitió buscar dos instrucciones por ciclo. En la fase de ejecución, el bus de 64 bits mejoró el rendimiento de las instrucciones de datos múltiple pudiendo acceder a dos registros simultáneamente.

ARM10 introdujo un sumador para las instrucciones de multiplicar y sumar liberando de esta carga a la ALU. La etapa de acceso a memoria fue la más larga del pipeline, lo que dificultaba el crecimiento de la frecuencia de reloj, por lo que se añade otro sumador para el cálculo de la dirección, para completarlo en menos de un ciclo, dejando en un ciclo y medio, los accesos a memoria.

La última mejora del pipeline en el procesador ARM10 fue la separación de la instrucción de decodificación a un estado separado.



(c) ARM10TDMI (d) ARM11

FIGURA 10: Pipeline de 6 etapas ARM

Pipeline de 8 etapas

El núcleo ARM11 introdujo dos cambios principales en la arquitectura del pipeline. En primer lugar, la operación de desplazamiento se separó en una nueva etapa. En segundo lugar, tanto las instrucciones como los accesos a memoria se distribuyen en dos etapas del pipeline.
Hay que tener en cuenta que en la fase de ejecución con el pipeline de 8 etapas, divide esta en tres pipelines que pueden funcionar concurrentemente en algunas situaciones y ejecutar instrucciones fuera de orden. Sin embargo, las etapas de lectura y decodificación se ejecutan en orden.

2.5 Tecnologías ARM

2.5.1 Thumb

El conjunto de instrucciones Thumb fue introducido en la cuarta versión de la arquitectura ARM con el fin de lograr una mayor consistencia en el código en las aplicaciones embebidas. Thumb cuenta con un subconjunto de las funciones más utilizadas del ARM de 32 bits que se han comprimido en códigos de operación de 16 bits. A la hora de la ejecución, estas instrucciones de 16 bits son descomprimidas a instrucciones de 32 bits en tiempo real. Para hacer posible esta tecnología se utiliza un decodificador de instrucciones Thumb.
El primer procesador con decodificador Thumb fue el ARM7TDMI. Todos los ARM9 y las familias siguientes, han incluido un decodificador de instrucciones Thumb.

2.5.2 Thumb-2

Tecnología sucesora de Thumb que extiende su conjunto de instrucciones de 16 bits con instrucciones adicionales de 32 bits logrando un mayor repertorio de instrucciones. Thumb-2 optimiza el uso de la memoria (hasta un 31% menos), lo cual también reduce el precio del sistema. Por otro lado, aumenta el rendimiento hasta un 38%, disminuyendo el consumo de energía (batería en dispositivos portátiles). Además, todos los dispositivos con Thumb-2 son compatibles con Thumb.

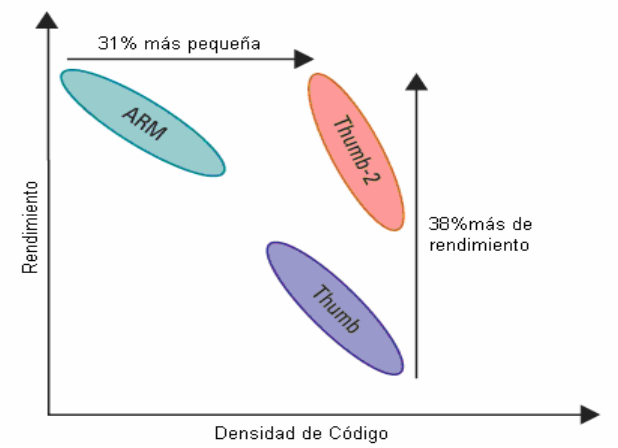


FIGURA 11: Rendimiento Tecnologías ARM

2.5.3 Thumb-2EE

Conocida también como Jazelle RCT, aparece en los procesadores de Cortex-A8 en 2005. Esta tecnología proporciona una pequeña extensión a la tecnología Thumb-2, haciendo que el conjunto de instrucciones se ajuste al código generado durante el tiempo de ejecución.

Las nuevas características incluyen comprobación automática de punteros nulos en cada instrucción load/store, una instrucción para comprobar los límites de los arrays, acceso a los registros r8-r15, e instrucciones especiales para invocar un handler, utilizado normalmente para implementar una característica de un lenguaje de alto nivel, como reservar memoria para un nuevo objeto.

2.5.4 Advanced SIMD (NEON)

La tecnología NEON es una combinación de un conjunto de instrucciones SIMD (Single Instruction Multiple Data) de 64 y 128 bits que mejoran las aplicaciones multimedia y el procesamiento de la señal para la codificación o descodificación de video, gráficos 3D, audio comprimido, procesamiento de imagen, telefonía y sonido. Se pueden ejecutar hasta 16 operaciones SIMD al mismo tiempo.

NEON está compuesto por un completo repertorio de instrucciones, bancos de registros separados y hardware de ejecución independiente. Tiene control en el acceso de datos, soporte para datos de tipo entero (8, 16, 32 y 64 bits) o punto flotante de precisión simple y un firme acoplamiento con el núcleo ARM.

2.5.5 TrustZone

La extensión de seguridad TrustZone permite la creación de una base informática de confianza dentro un sistema-on-chip. Tal base puede incluir un conjunto de aplicaciones de confianza que se ejecutan en un modo especial privilegiado fuera de las operaciones del sistema principal, dando acceso a memoria segura dentro del chip o fuera de él y a los periféricos. En esta zona de seguridad se realizan tareas como la comprobación de la integridad del software, las actualizaciones de seguridad del software, gestión de claves criptográficas, etc.

2.5.6 Instrucciones DSP

Son las instrucciones que se añaden a la arquitectura ARM para el tratamiento digital de las señales DSP (Digital Signal Processing). Los procesadores que disponen de estas instrucciones llevan una E en el nombre. Las nuevas instrucciones son comunes en arquitecturas DSP, como suma y resta saturada o multiplicación y suma con signo.

2.5.7 Jazelle

Jazelle es una implementación hardware de la máquina virtual de Java que permite ejecutar bytecode de Java directamente sobre la arquitectura ARM como una tercera etapa de ejecución junto a los modos existentes ARM y Thumb.
La implementación hardware evita la sobrecarga que supondría la emulación de la máquina virtual o JIT.

2.6 ¿Por qué usar ARM en la docencia?

Hoy en día, los procesadores ARM protagonizan la tendencia actual de diseño de microprocesadores en sistema empotrados. ARM es una filosofía de diseño de procesadores basada en repertorios de pocas instrucciones muy básicas que llevan a rutas de datos con un hardware muy sencillo, de bajo coste y consumo, con multitud de posibilidades para su optimización.

Los diseños ARM están hoy en el núcleo de todos los smartphones y multitud de aplicaciones empotradas. Por esta razón, estudiar la arquitectura ARM en la universidad es prioritario, como evolución del diseño RISC. Las posibles razones para estudiar los procesadores ARM en el ámbito docente son:

- La sencillez de su diseño y que éste sea público, lo que la hace especialmente útil desde el punto de vista didáctico.
- Facilidad para demostrar la relación entre los lenguajes de programación de alto y bajo nivel.
- La importancia de los procesadores ARM en el mercado actual.
- Además conviene subrayar que debido a la sencillez de los repertorios de instrucciones de tipo RISC que lleva el ARM, todas las arquitecturas diseñadas con esta filosofía son muy similares unas a otras, por lo que una vez entendida una de ellas, la comprensión del resto es casi inmediata.

En este enclave, se comprende que el estudio de ARM tenga gran importancia en el estudio universitario en las materias de ingeniería, pues nos permite comprender el lenguaje ensamblador, sus instrucciones, direccionamientos y el trabajo con los registros y la memoria. Y en un segundo lugar, como trabajar con un lenguaje de alto nivel, normalmente C, para la comprensión de la entrada/salida de datos, las interrupciones, las excepciones, las llamadas al sistema y la comunicación con otros dispositivos, que nos proporciona un conocimiento adecuado del funcionamiento de un procesador ARM.

2.7 Entorno de Desarrollo Embest

El Entorno de Desarrollo Integrado Embest IDE es una aplicación con interfaz gráfica de usuario que permite desarrollar y depurar software sobre sistemas embebidos. Para realizar las tareas de compilación esta aplicación utiliza otros programas: el ensamblador (as), el compilador (gcc) y el enlazador (ld) de GNU para ARM. Por lo que se utilizan las reglas y sintaxis propias de estas herramientas.

Embest IDE facilita la elaboración y construcción de proyectos para la arquitectura ARM, establece y controla las comunicaciones entre el *Host* y el *Target*, lanza y depura aplicaciones, etc. Embest IDE actualmente soporta todos los procesadores basados en ARM7 y ARM9. Además, la aplicación puede ser actualizada para los nuevos procesadores ARM.
La interfaz de Embest IDE es la siguiente:

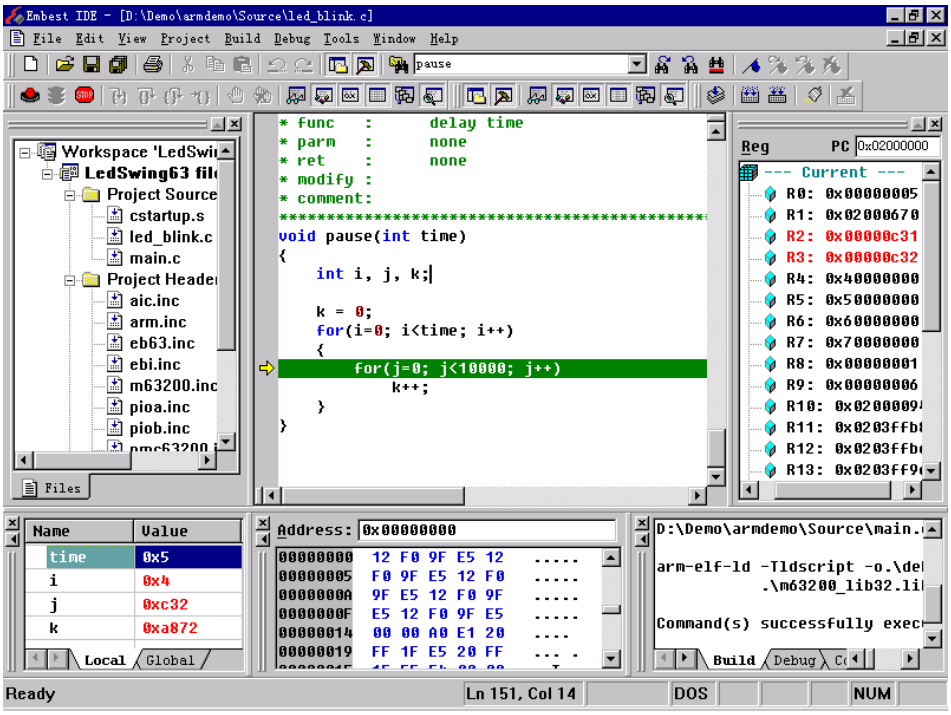


FIGURA 12: Interfaz Embest IDE

Las principales características son:

- Un entorno de desarrollo integrado
- Facilidad para manejar proyectos
- Integración de un editor de código fuente
- Compilador, ensamblador y enlazador GNU
- Librerías GNU para ANSI C
- Soporte para herramientas específicas de ARM
- Depuración de código
- Simulador del conjunto de instrucciones ARM
- Ejemplos en código fuente para procesadores ARM como Atmel/Samsung/Cirrus logic/OKI/ etc.
- Documentación online

Una vez explicada las líneas generales del entorno de desarrollo de Embest IDE, vamos a ver la relación que este tiene con el maletín utilizado en las prácticas de la asignatura de Estructura de Computadores (EC) de Grado en Informática.

Embest IDE necesita de un emulador JTAG también llamado JTAG debbugger. Este emulador permite comunicar a Embest IDE con el núcleo ARM a través de su interfaz. Se utiliza el emulador porque identifica muchos de los errores sin necesidad de usar la memoria de la placa, ni ninguno de los puertos de sus periféricos facilitando la depuración de los programas que serán ejecutados en la placa.

En el caso de nuestro laboratorio, utilizamos el emulador JTAG UNetICE creado por Embest. UNetICE conecta el puerto USB o el puerto Ethernet a la interfaz JTAG de la placa de desarrollo ARM y permite la programación y depuración de la Flash. De UNetICE destacamos las siguientes características:

- Descargar programas a la placa de desarrollo
- Examinar la memoria y los registros
- Permite utilizar múltiples instrucciones
- Ejecución de programas en tiempo real
- Programación de la Flash on-chip
- Da soporte a arquitecturas ARM7 y ARM9
- Soporta interrupciones de software y hardware
- Posee soporte de Internet para desarrollo y depuración remota

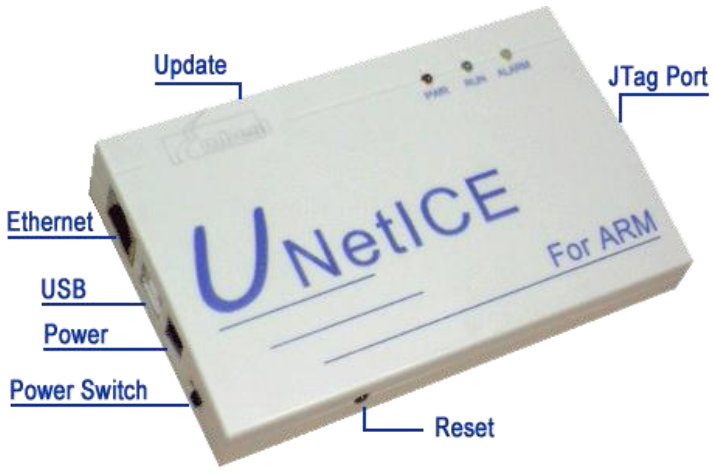


FIGURA 13: Emulador JTAG UNetICE

Una vez terminado un programa, el usuario necesita descargar el código binario en la memoria flash para hacerlo funcionar en tiempo real. Embest proporciona un Programador Flash (Flash Programmer) que permite al usuario directamente escribir la flash de la placa de desarrollo. El Flash Programmer necesita trabajar conjuntamente con el emulador JTAG. En la figura se muestra su interfaz:

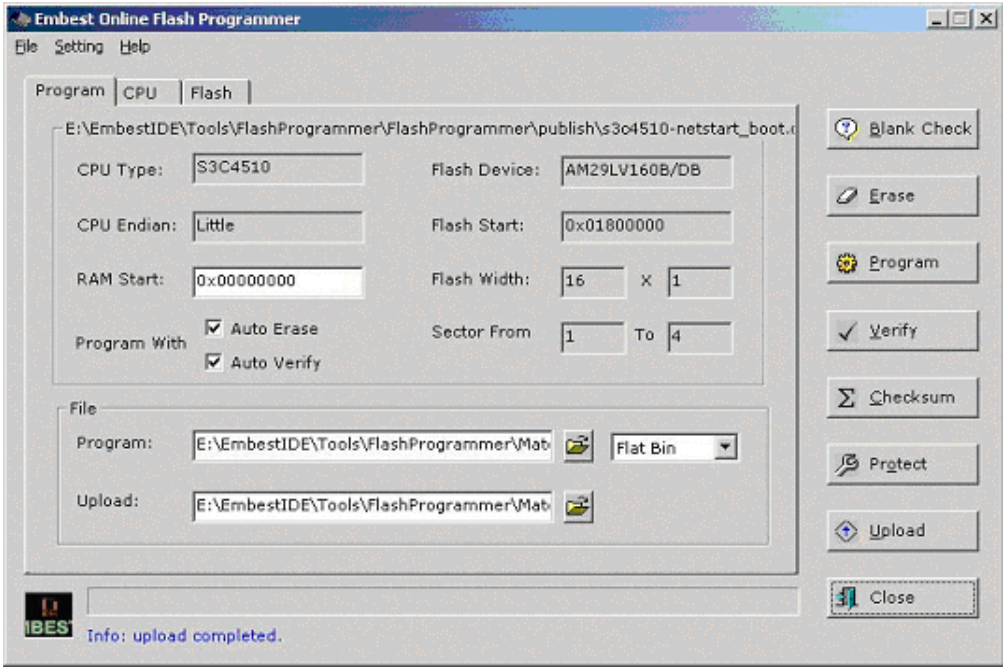


FIGURA 14: Interfaz Memoria Flash

Las características del Flash Programmer de Embest IDE son:

- Soporte para los microprocesadores ARM7 y ARM9: ATMEL AT91, INTEL 28 Series, SST 29/39/49 series.
- Comprobación si el espacio de memoria flash esta vacío, borrado de memoria: programación de memoria, verificar, proteger y actualizar el archivo.
- Especificación de los sectores de memoria sin modificar otros sectores
- Ancho de lectura/escritura de 8, 16 y 32 bits
- Soporte de 1 a 4 chips flash

Finalmente, tenemos la placa de desarrollo S3CEV40 en nuestros laboratorios con el chip de Samsung S3C44B0X, que integra un microcontrolador ARM7TDMI junto con una serie de dispositivos.

Las características principales de la placa de desarrollo S3CEV40 son:

- Suministro de energía de 5V
- 1M x 16 bits Flash
- 4 x 1M x 16 bits SDRAM
- 4Kbit IIC bus serial EEPROM
- 2 puertos de serie: uno con interfaz simple, y otro con interfaz completa que permite conectarse con el RS232 MODEM.
- Botón Reset
- Dos botones de interrupción, dos LEDs
- Interfaz de Disco Duro IDE
- Pantallas LCD y TSP táctiles
- JTAG de 20 pines
- Conector USB
- Teclado de 4 x 4
- Ethernet de 10 Mb/s
- LED de 8 segmentos
- Entrada de micrófono
- Salida IIS de voz que puede ser conectada a dos altavoces
- 16M x 8 bits de Disco Duro
- 320 x 240 panel LCD con pantalla táctil

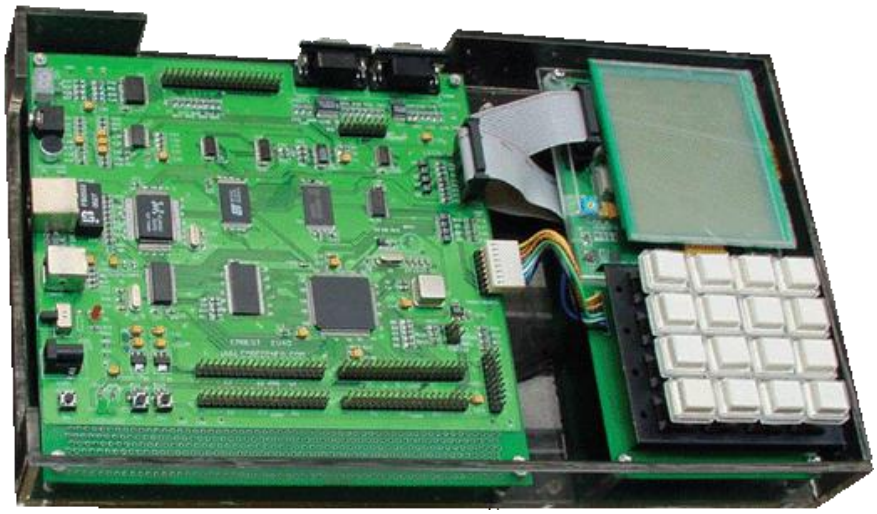


FIGURA 15: Placa de desarrollo S3CEV40

En líneas generales, este el entorno de desarrollo que se utiliza en los laboratorios de la Facultad de Informática, que permite el desarrollo de programas embebidos para un arquitectura ARM.

CAPÍTULO 3
HERRAMIENTA DE SIMULACIÓN ARM Y ENTORNO DE DESARROLLO

Capítulo 3 - Herramienta de Simulación ARM y Entorno de Desarrollo

En este capítulo, se explica la composición, las herramientas necesarias y el proceso de construcción de la herramienta de simulación. A continuación, se explica el diseño y el proceso de elaboración de la herramienta de integración (interfaz gráfica de usuario).

3.1 Descripción de la herramienta

En este apartado, mostramos a modo de resumen, de manera gráfica, los módulos externos de los que está formada nuestra herramienta. En los siguientes apartados, explicaremos con detalle cada uno de ellos.

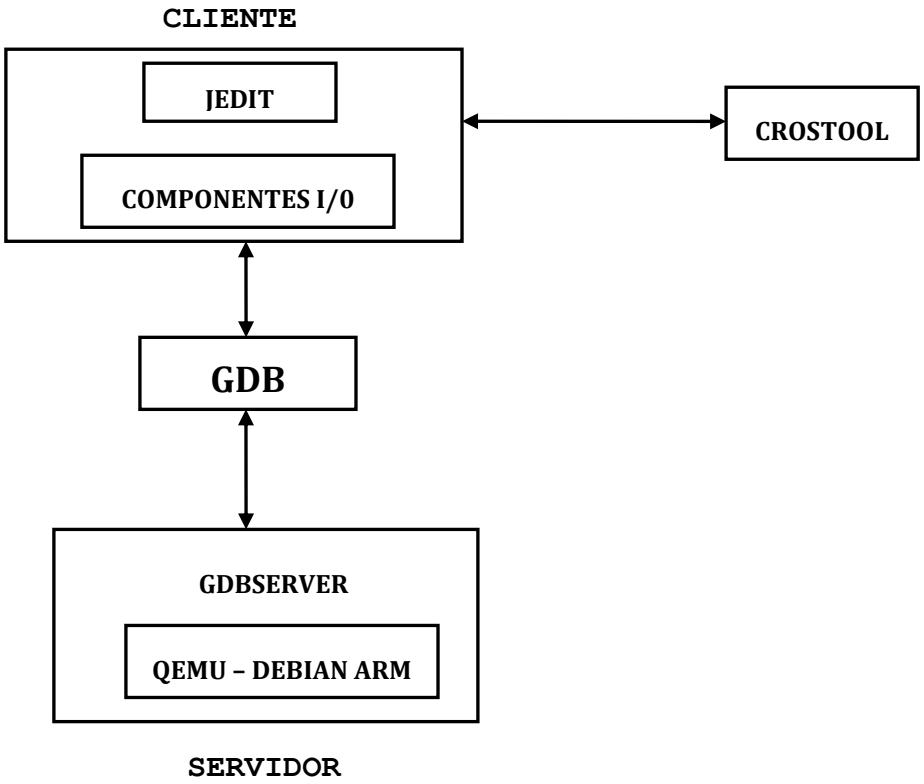


FIGURA 16: Resumen Componentes Externos

Como se puede observar en la *Figura 16*, la herramienta esta formada por cuatro módulos:

i. Cliente (interfaz gráfica)
ii. Croostool (compilador cruzado)
iii. Depurador GDB
iv. Servidor (QEMU – Debian ARM y GdbServer)

Por una parte, la herramienta cuenta con un módulo denominado *Cliente*, formado principalmente por la interfaz gráfica de usuario. Dicha interfaz, lleva integrada un editor de texto (en nuestro caso jEdit), una ventana donde se muestran los componentes de entrada/salida utilizados en la placa S2CEV40, paneles para visualizar el estado de los registros, memoria, etc. Este módulo, interacciona directamente con el compilador cruzado o Croostool instalado en el sistema.

Por otra parte, para ejecutar/depurar código, la aplicación emplea la herramienta GNU-GDB usando como entorno emulador el módulo denominado Servidor. Este último está formado por las herramientas QEMU-Debian ARM y GdbServer.

3.2 Herramientas utilizadas para el desarrollo del proyecto

- Sistema operativo Linux
- Compilador cruzado CROSSTOOL-NG versión 1.15.0
- Emulador QEMU versión 0.14 o superior
- Debian ARM
- NetBeans IDE

3.3 Compilación cruzada

Por lo general, los programadores, estamos acostumbrados a compilar para la arquitectura sobre la que se ejecutará el fichero binario. Es decir, programamos en la misma máquina en la que se ejecutará la aplicación. Sin embargo, esto no siempre es posible.

La compilación cruzada o cross-compiling consiste en generar binarios para una arquitectura distinta a la que estamos utilizando para compilar. Por ejemplo, generar un ejecutable para un procesador ARM desde una arquitectura IA-32 o AMD64.

Para realizar este tipo de compilación es necesario contar con una serie de programas y librerías que establezcan un ambiente propicio para llevar a cabo esta tarea. Este ambiente se denomina entorno de compilación cruzada.
Los componentes necesarios para implementar el entorno de compilación cruzada consisten básicamente en tres elementos:

- Compilador C : compilador de C básico
- Librería C: implementa las llamadas al sistema mediante APIs.
- Binutils: conjunto de programas para compilación, enlazado, ensamblado y depuración de código.

A este conjunto de componentes se le denomina toolchain o cadena de herramientas. Se denomina cadena porque se suelen aplicar estas herramientas en ese orden, como una cadena.
En el mundo Linux, y Open Source en general, la toolchain más utilizada es la GNU. Además, esta toolchain es muy completa: make, gcc, binutils, bison, m4, gdb, autotools, etc.

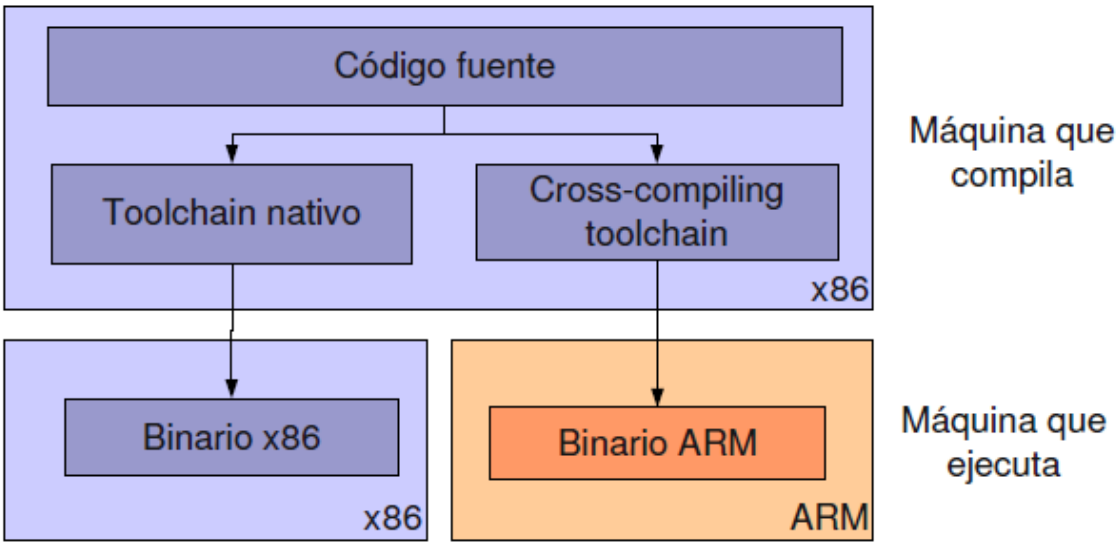


FIGURA 17: Compilación cruzada

3.3.1 Componentes de una compilación cruzada

Los componentes presentes en una compilación cruzada son:

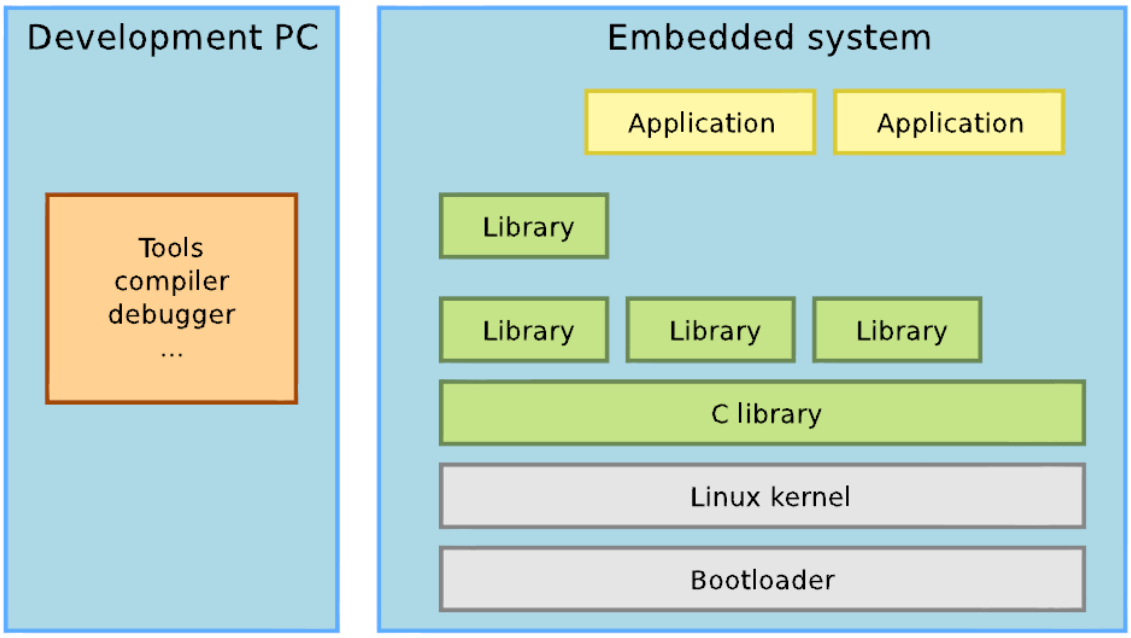


FIGURA 18: Componentes Compilación cruzada

3.3.1.1 Binutils

Es un conjunto de componentes para generar y manipular archivos binarios para una arquitectura dada. Los más destacados:

- *as*, ensamblador, que genera código binario desde código ensamblador.
- *ld*, linkar.
- *ar, ranlib,* para generar archivos .a, útiles para las librerías.
- *objdump, readelf, size, nm, strings*, para analizar archivos binarios.

3.3.1.2 Kernel Headers

La librería de C y los programas compilados necesitan interactuar con el kernel, ya sea por llamadas al sistema y sus números, las estructuras de datos, la definición de constantes, etc. La compilación de los archivos binarios necesita interactuar con el kernel y para evitar problemas se utilizan las cabeceras del kernel.

3.3.1.3 GCC

GNU C Compiler, es el compilador libre más conocido y utilizado. Puede compilar C, C++, Ada, Fortran, Java, Objective-C, Objective-C++ y generar código para un gran número de diferentes arquitecturas como ARM, AVR, Blackfin, MIPS, PowerPC, i386, x86, etc.

3.3.1.4 Librería C

La librería C es un componente esencial en un sistema Linux. Sirve de interfaz entre las aplicaciones y el kernel. Provee una API estándar para el desarrollo de aplicaciones. Tenemos varias librerías para el desarrollo del compilador cruzado: glibc, uClibc, eglibc, dietlibc, newlib, etc.
La elección de una librería u otra se hará a la hora de crear el compilador cruzado, pues el gcc es compilado en función de dicha librería C.
En el proyecto, después de probar con uClibc y glibc, hemos optado por glibc por encontrarse en todos los sistemas GNU, por su portabilidad y por estar actualizada y mantenida constantemente.

3.3.2 Construcción de un compilador cruzado para ARM

La construcción del compilador cruzado va a suponer uno de los ejes vertebradores del proyecto. Sabiendo que la placa que se utiliza en el laboratorio de estructuras de computadores tiene una arquitectura ARM, concretamente ARM7TDMI, lo primero es elaborar un toolchain que permita generar un archivo que se pueda descargar en dicha placa, además de poder simularlo en un entorno equivalente al proporcionado por la placa Embest y sus componentes.
Este trabajo ha sido arduo y complicado, pues nunca nos habíamos enfrentado a la creación de un compilador cruzado que genera los archivos binarios para ARM, desde una arquitectura x86 que funciona como compilador nativo y crea los binarios para dicha arquitectura.

En un principio, investigando como realizar un toolchain para ARM, vimos que era un trabajo muy complejo si lo queríamos realizar manualmente y que generalmente producía bastantes errores en su creación, por lo que optamos por buscar nuevos métodos, encontrando toolchains precompilados (útiles pero poco flexibles a la hora de reconfigurarlos y adaptarlos) o herramientas para crear el compilador (automatizables y más fáciles de adaptar indicando los parámetros adecuados). Como decisión de grupo observamos varias herramientas:

- Crosstool-ng: Activamente mantenida, soporta uClibc, glibc, eglibc, hard y soft float, además de diversas arquitecturas. Tiene un menuconfig, para especificar claramente los parámetros que ejecutarán los scripts y los diversos parches.
- Buildroot: Basado en Makefile, sólo uClibc, mantenido por la comunidad.
- OpenEmbedded: Sistema de compilación completo y complejo.

Finalmente, elegimos crear el toolchain con Crosstool-ng, pues permite automatizar el proceso, para generar los archivos ejecutables para una arquitectura ARM.
Conviene distinguir que tenemos tres tipos de máquinas a la hora de crear un toolchain:

- Build machine, donde se construye el toolchain.
- Host machine, donde el toolchain va a ser ejecutado.
- Target machine, donde los archivos binarios creados por el toolchain van a ser ejecutados.

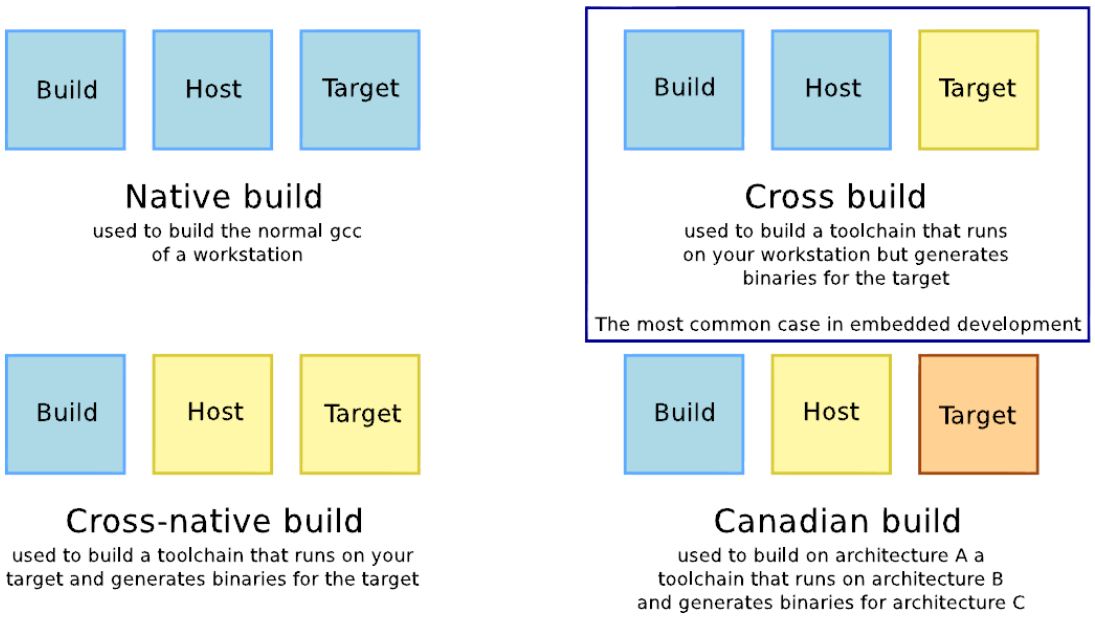


FIGURA 19: Machines

3.3.3 Construcción de nuestro compilador cruzado para ARM

Una vez analizado brevemente los aspectos esenciales existentes en un compilador cruzado, vamos a proceder a la construcción del mismo. Para ello necesitamos 2 GB libres de espacio en el disco duro.

1) Crear el directorio:

   mkdir /home/<usuario>/toolchain

2) Instalar los paquetes necesarios, a través de la terminal:

   sudo apt-get install autoconf automake libtool libexpat1-dev libncurses5-dev bison flex patch curl cvs texinfo build-essential subversion gawk python-dev gperf lzma ncurses-dev

   sudo apt-get install gcj-jdk

   sudo apt-get clean

3) Descargar la herramienta Crosstool:

   - Bajar la última versión Crosstool-ng de HUhttp://crosstool-ng.org/UH, que en nuestro caso es 1.15.0.
   - Descomprimir la herramienta en el directorio /home/<usuario>/ y entrar dentro del directorio descomprimido:
     - ✓ ***tar xvjf crosstool-ng-1.15.0.tar.bz2***
     - ✓ ***cd crosstool-ng-1.15.0***
     - ✓ **./configure --prefix=/home/<usuario>/crosstool/**
     - ✓ **make**
     - ✓ **make install**

El programa nos da el siguiente aviso:

> For auto-completion, do not forget to install 'ct-ng.comp' into your bash completion directory (usually /etc/bash_completion.d).

4) Copiamos ct-ng.comp en el directorio /etc/bash_completion.d/ en modo root:

   sudo cp ct-ng.comp /etc/bash_completion.d/

   Podemos ver la ayuda con el siguiente comando:

   ./ct-ng help

Para facilitar el funcionamiento vamos a exportar los ejecutables a la variable de entorno, pues la instalación del crosstool-ng no se ha realizado en rutas del sistema.

export PATH="${PATH}:/home/<usuario>/crosstool/bin/"

3.3.4 Configuración del Toolchain

Para facilitar el trabajo y después de haber probado numerosas opciones de compilación cruzada, vamos a indicar la forma más rápida y sencilla para su creación.
En la página Web de crosstool-ng encontramos diversos ejemplos que ya han sido probados y ejecutados, y que se pueden consultar con el siguiente comando:

ct-ng list-samples

Entre todos ellos vamos a elegir arm-unknown-linux-gnueabi, sobre el que haremos los cambios oportunos para ajustarlo a nuestras necesidades:

ct-ng arm-unknown-linux-gnueabi

Para redefinir la configuración, utilizamos menuconfig, y así ajustamos aquellos parámetros oportunos para generar adecuadamente nuestro compilador cruzado:

ct-ng menuconfig

Cambiar en:

- TARGET OPTIONS:
  - Architecture level: armv4t
  - Emit assembly for CPU: arm7tdmi
- TOOLCHAIN OPTIONS:
  - Tuple's Alias: arm-linux
- DEBUG OPTIONS:
  - Seleccionar gdb, strace, ltrace.

Eliminar las otras opciones.

Una vez configurado, creamos el directorio /home/<usuario>/src/ para que pueda guardar los paquetes de instalación que necesita.

Después, construimos el compilador cruzado con:

ct-ng build

Algunos problemas que nos pueden surgir, se resuelven copiando el archivo o paquete que nos piden al dar el error en la carpeta src creada previamente.
Finalmente, para usar libremente nuestro compilador cruzado, exportamos los ejecutables a la variable de entorno:

export PATH="${PATH}:/home/<usuario>/x-tools/arm-unknown-linux-gnueabi/bin/"

3.4 Emulador de procesador ARM QEMU

3.4.1 ¿Qué es QEMU?

QEMU es un emulador de procesadores basado en la traducción dinámica de binarios (conversión del código binario de la arquitectura fuente en código entendible por la arquitectura huésped). QEMU tiene capacidad de virtualización dentro de un sistema operativo. Esta máquina virtual puede ejecutarse en cualquier tipo de microprocesador o arquitectura.
Antes de continuar aclararemos el concepto de virtualización y emulación.

Virtualización

La virtualización consistiría sencillamente en ejecutar una máquina dentro de otra. Formalmente, la virtualización sería la creación, a través de software, de una versión virtual de algún recurso tecnológico, como puede ser una plataforma de hardware, un sistema operativo, un dispositivo de almacenamiento u otros recursos de red.

Emulación

La emulación permite ejecutar programas en una plataforma diferente para la cual fueron escritos originalmente. Nosotros utilizaremos QEMU como emulador del hardware ARM, en el cual instalaremos el sistema operativo Debian para ARM para poder trabajar sobre él por línea de comandos.

FIGURA 20: QEMU

Las características principales de QEMU son:

- Soporta emulación de IA-32, AMD64, MIPS, SPARC, ARM, PowerPC y ETRAX CRIS
- Soporte para otras arquitecturas en host y sistemas emulados
- Implementa el formato de imagen de disco Copy-On-Write. Se puede declarar una unidad virtual multi-gigabyte, la imagen de disco ocupará solamente el espacio actualmente utilizado.
- Implementa superposición de imágenes
- Soporte para ejecutar binarios de Linux en otras arquitecturas
- Es posible salvar y restaurar el es de la máquina
- Emulación de tarjetas de red virtuales
- El sistema operativo huésped no necesita ser modificado o parcheado
- Las utilidades de línea de comandos permiten un control total de QEMU
- Control remoto de la máquina emulada a través del servido VNC integrado

Gracias a la utilización de QEMU hemos podido emular adecuadamente un sistema con arquitectura ARM con el trabajar con nuestro compilador cruzado, y así poder testear los programas creados en esta arquitectura. Ahora pasamos a describir todo el proceso de instalación de Debian en una arquitectura ARM emulada en QEMU.

3.4.2 Instalación de Debian ARM en QEMU

Lo primero que necesitamos es instalar QEMU en nuestro sistema operativo Linux, que será el Host, para emular un sistema ARM que hará las veces de Guest.
Para ello, utilizamos la siguiente instrucción:

sudo apt-get install qemu qemu-kvm-extras

Creamos el directorio QEMU en nuestro *home* para guardar todos los archivos que necesitaremos para preparar el entorno de emulación:

mkdir QEMU

Guardamos los dos archivos que aparecen en el enlace dentro del directorio:

HUhttp://ftp.de.debian.org/debian/dists/squeeze/main/installer-armel/current/images/versatile/netboot/U

- ***initrd.gz***
- ***vmlinuz-2.6.32-5-versatile***

Una vez guardados estos dos archivos procedemos a crear un disco duro virtual que será el lugar de alojamiento de nuestro Debian ARM, eligiendo el tamaño según nuestras necesidades:

- ***cd QEMU***
- ***qemu-img create -f qcow hda.img 2G***

Iniciamos la instalación de Debian:

- ***qemu-system-arm -m 256 -M versatilepb -kernel vmlinuz-2.6.32-5-versatile -initrd initrd.gz -hda hda.img -append "root=/dev/ram"***

Esperamos a que se inicie el sistema y seguimos las indicaciones de pantalla como una instalación normal.

Seleccionamos idioma y componentes como el teclado en español (Spanish).

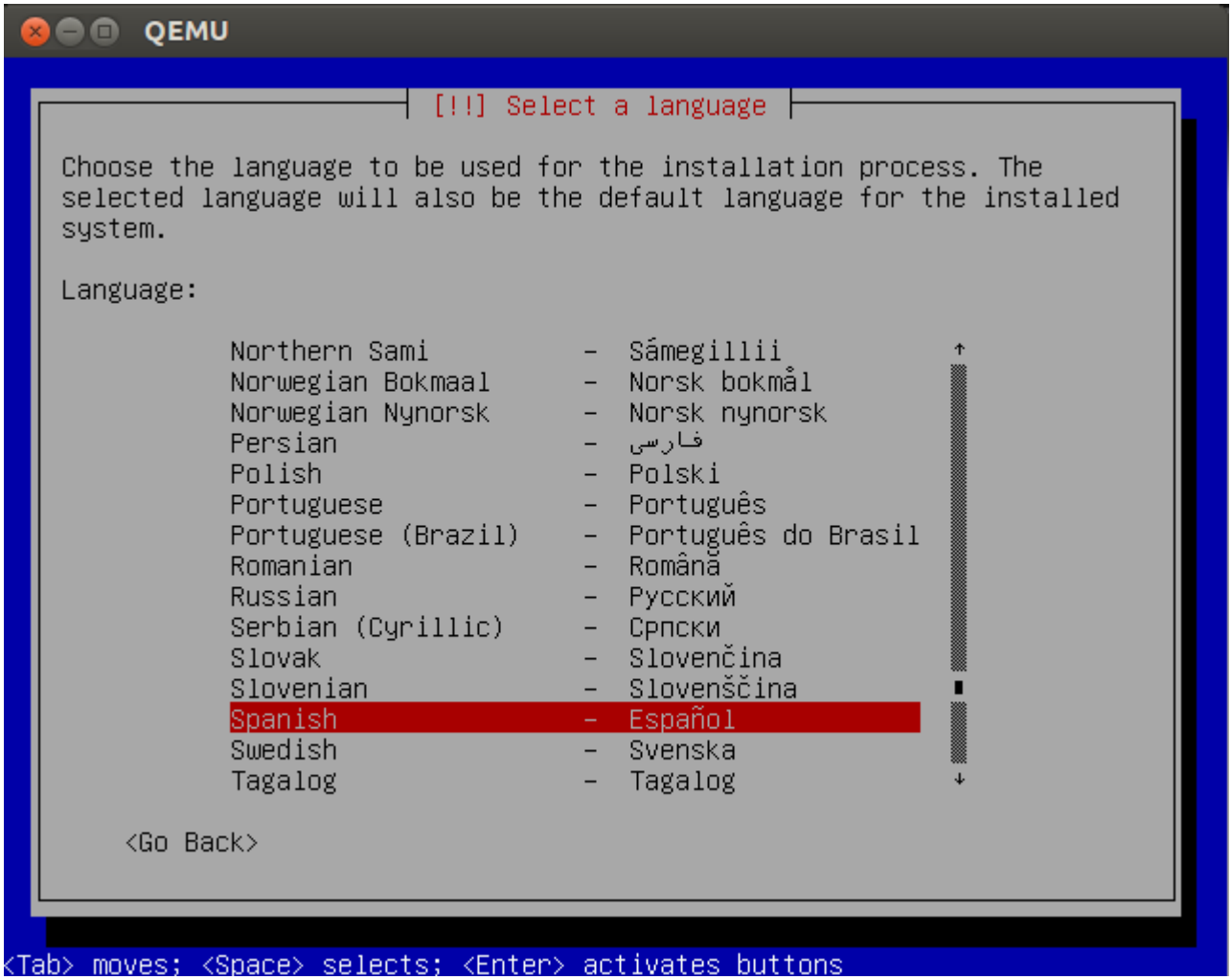


FIGURA 21: Instalación Debian ARM - 1

Utilizaremos la siguiente dirección para obtener nuestra copia de Debian:

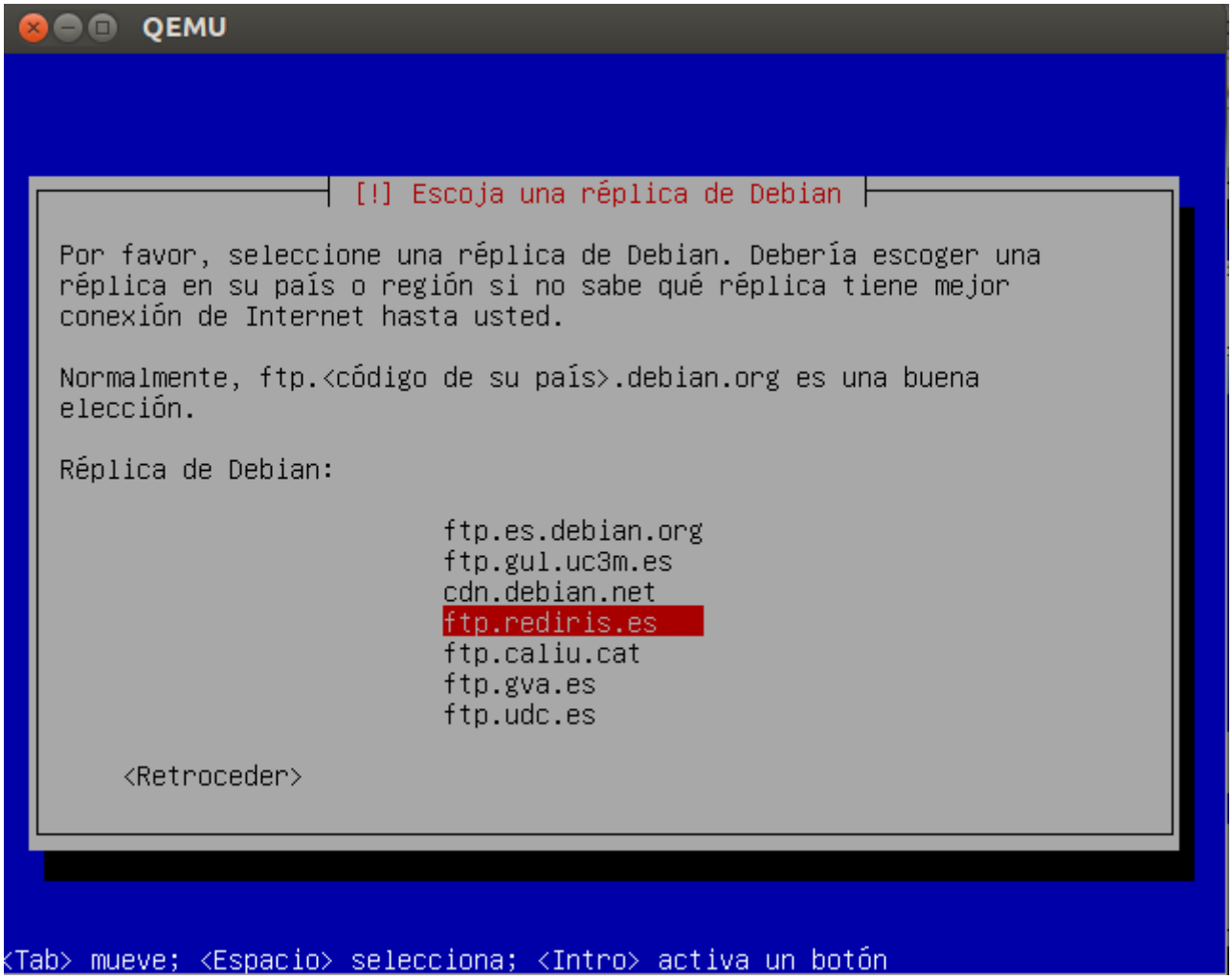


FIGURA 22: Instalación Debian ARM - 2

Instalación del sistema base para el sistema Debian:

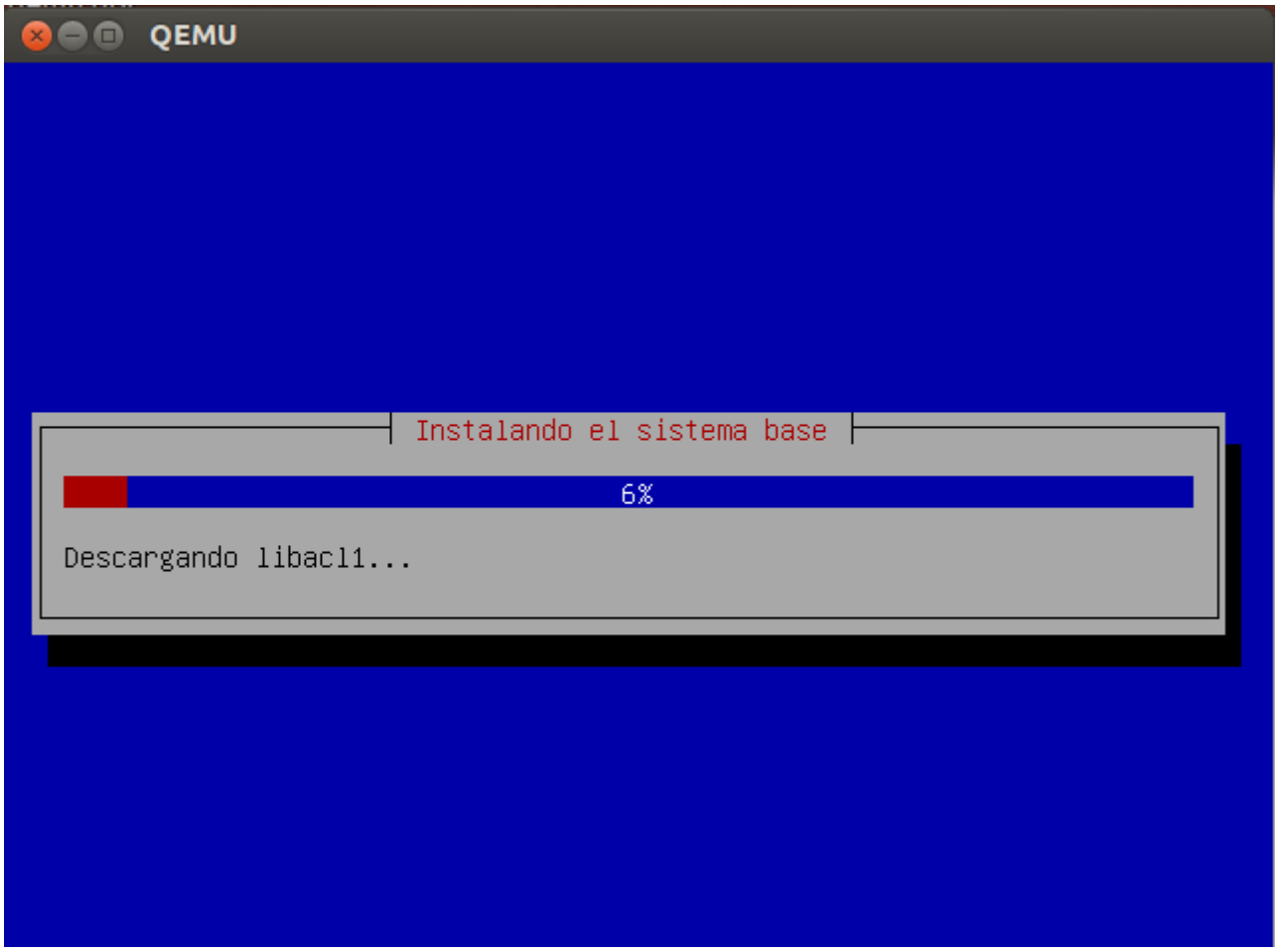


FIGURA 23: Instalación Debian ARM - 3

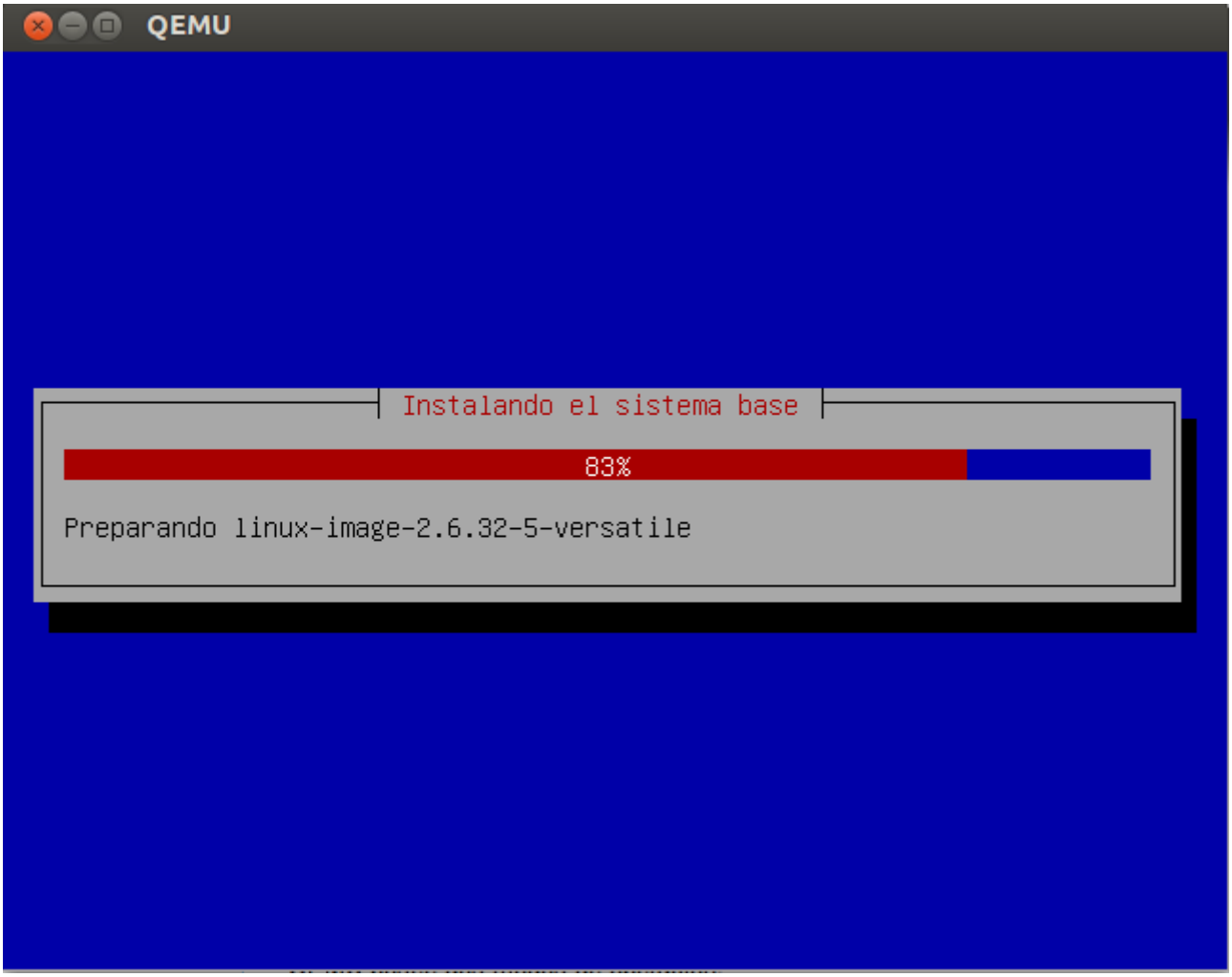


FIGURA 24: Instalación Debian ARM - 4

Podemos añadir varios programas durante la instalación. Añadiremos ssh, pues lo utilizaremos para transmitir archivos entre el Host y el Guest.

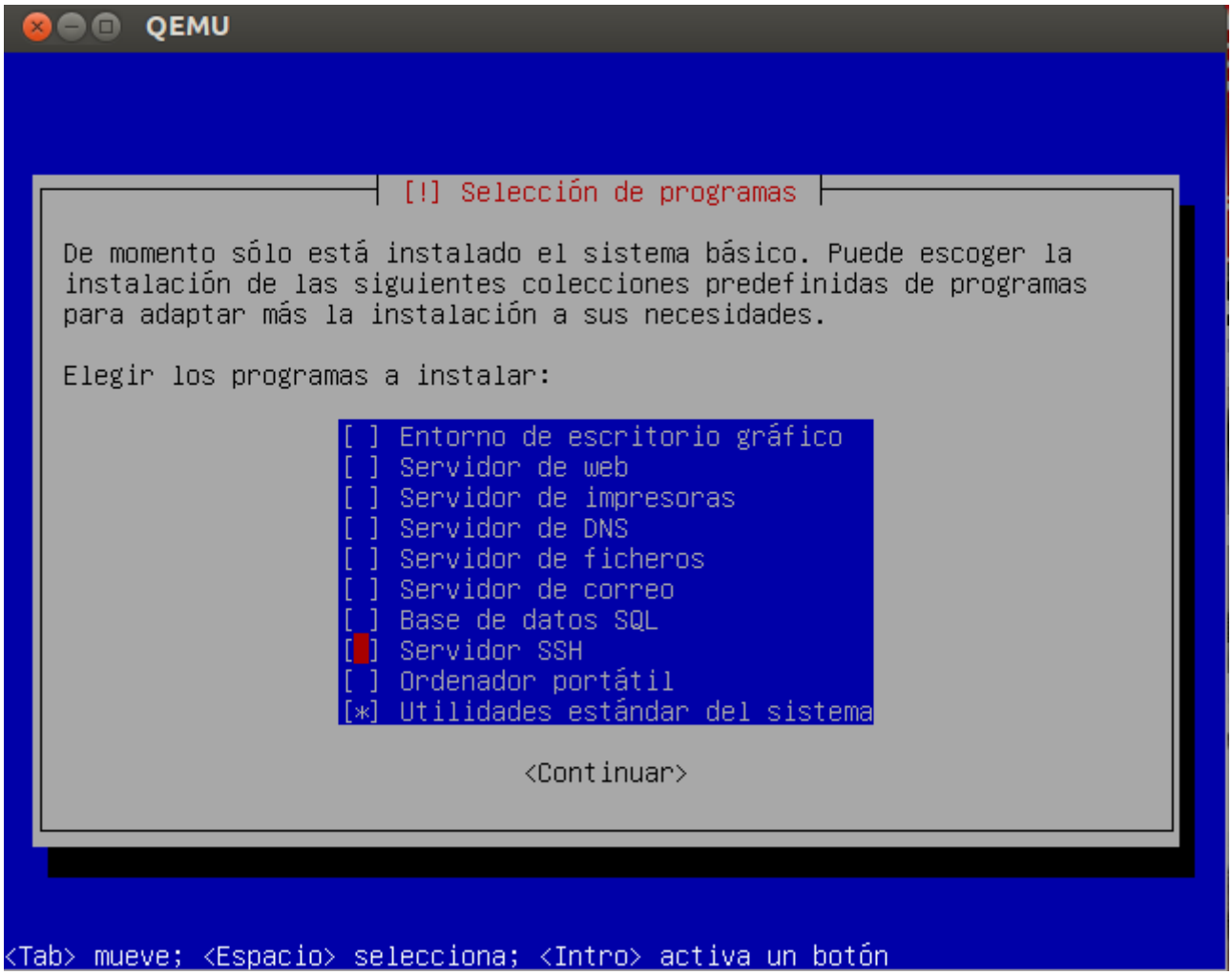


FIGURA 25: Instalación Debian ARM - 5

No necesitamos establecer el cargador de arranque, pues nosotros mismos se lo pasaremos cuando arranquemos QEMU.

FIGURA 26: Instalación Debian ARM - 6

Y una vez terminada la instalación, dejamos que el sistema se reinicie automáticamente para que termine, y luego cerramos QEMU, porque vuelve a comenzar el proceso de instalación.

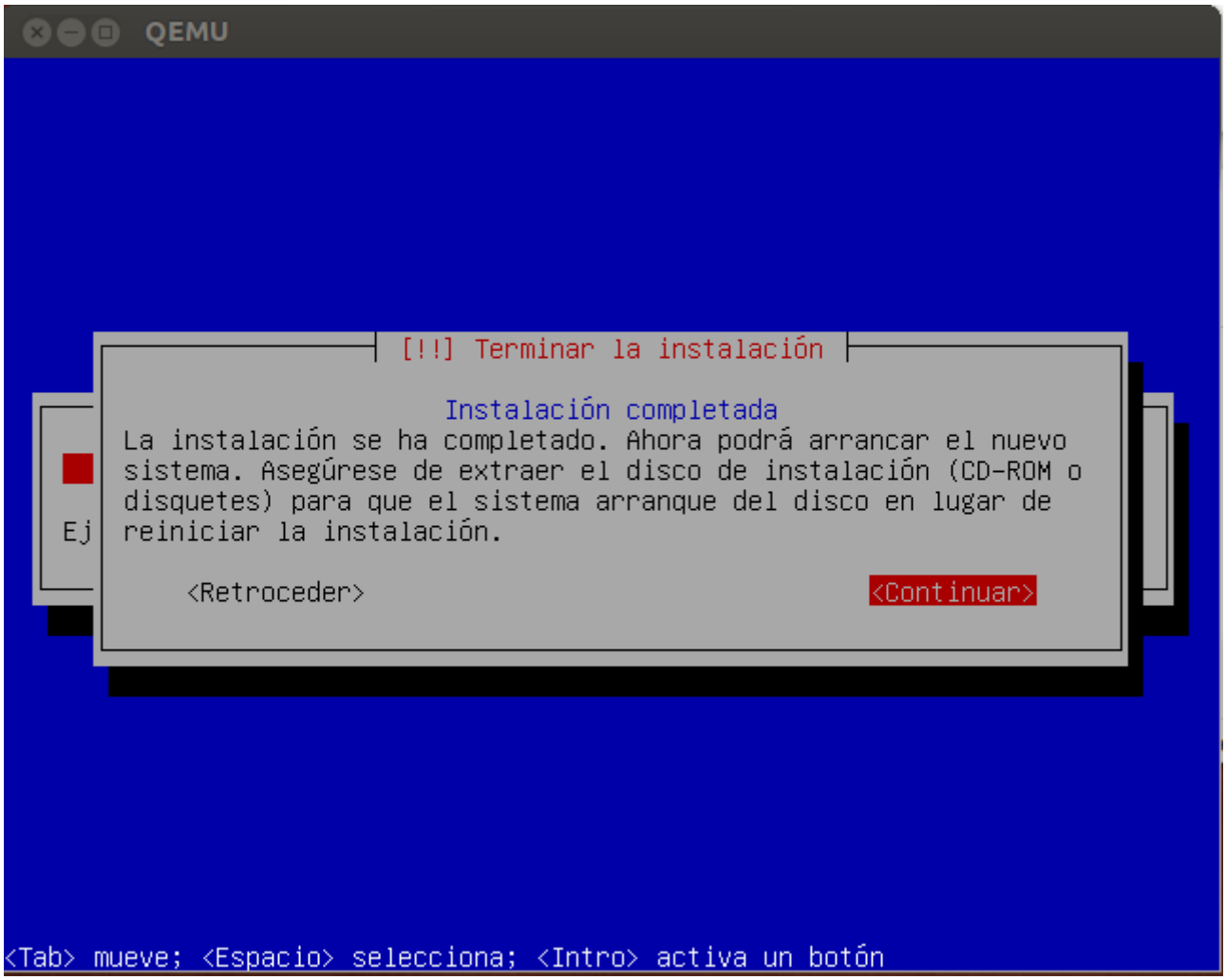


FIGURA 27: Instalación Debian ARM - 7

Descargamos el kernel y la imagen de inicio del siguiente enlace en nuestro directorio QEMU:

HU**http://people.debian.org/~aurel32/qemu/armel/**U

Los archivos son:

- **initrd.img-2.6.32-5-versatile**
- **vmlinuz-2.6.32-5-versatile**

Ejecutamos la instrucción siguiente para arrancar QEMU con un sistema Debian en arquitectura ARM, sobre el que trabajaremos para ejecutar nuestros programas para ARM.

qemu-system-arm -M versatilepb -kernel vmlinuz-2.6.32-5-versatile -initrd initrd.img-2.6.32-5-versatile -hda hda.img -append "root=/dev/sda1"

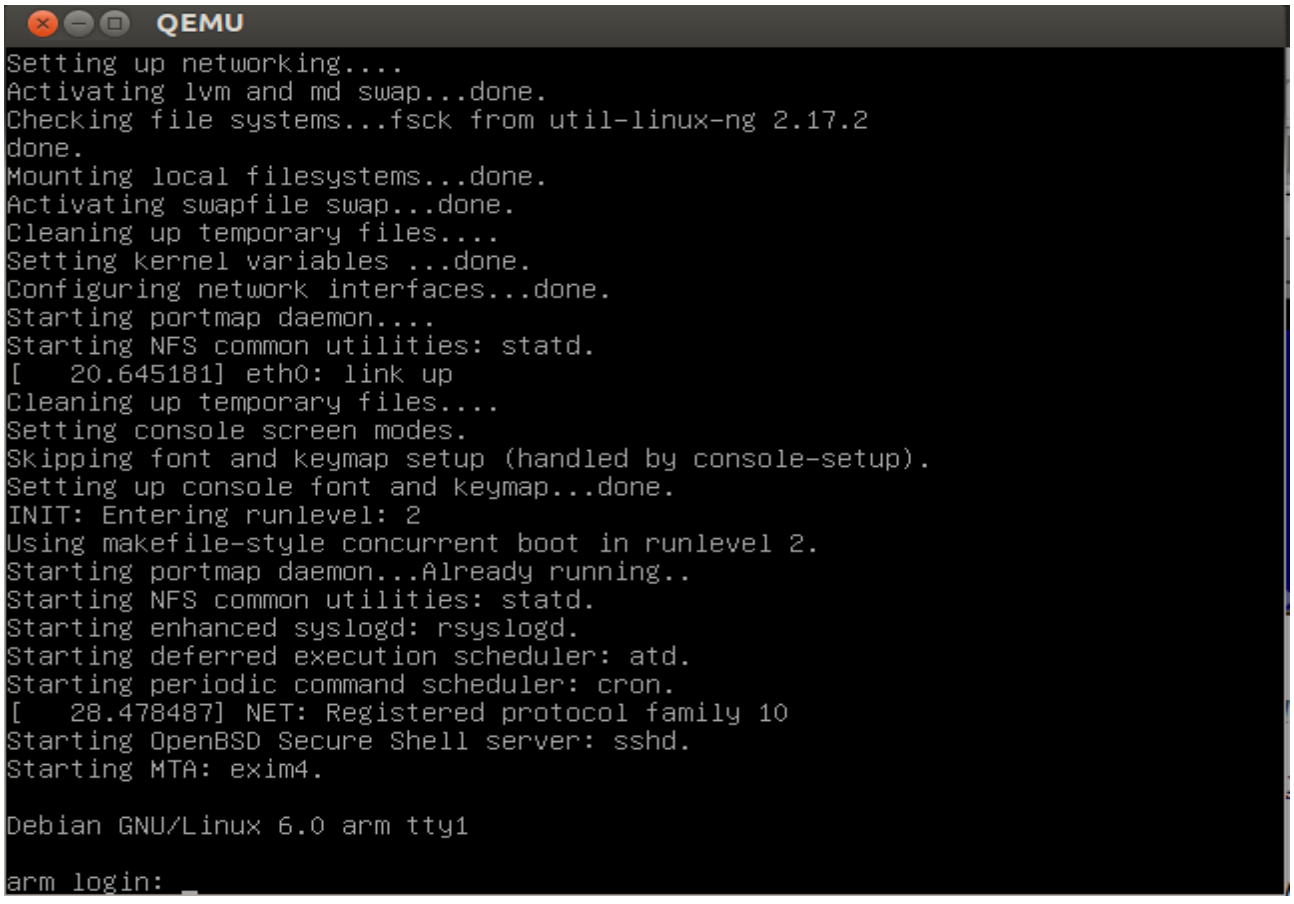


FIGURA 28: Instalación Debian ARM – 8

3.4.3 Descripción de la placa y ARM emulados en QEMU

QEMU emula la placa Versatilepb (Versatile Platform Baseboard). Esta placa ha sido desarrollada para la familia ARM9, más concretamente lleva el procesador ARM926EJ-S (descrito en el capítulo 2).

Sin embargo, el mapa de memoria de esta placa, no simula fehacientemente una placa Versatile Realview Platform Baseboard. En la siguiente Figura se puede observar el mapa de memoria que simula dicha placa:

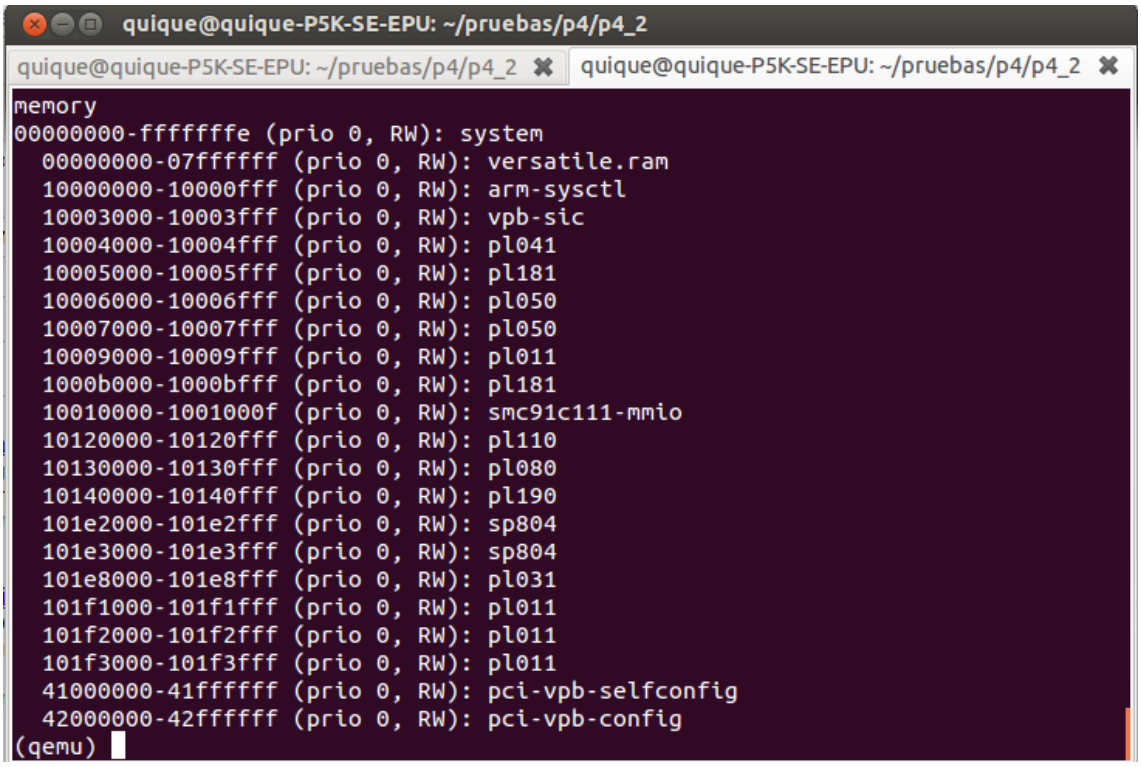


FIGURA 29: Mapa memoria placa QEMU

En esta Figura observamos que se pueden utilizar las siguientes localizaciones de memoria, periféricos y controladores:

- Los registros del sistema
- El controlador de interrupciones secundario
- La interfaz de audio
- Dos interfaces de tarjeta multimedia
- Dos interfaces para el teclado y el ratón
- UART0, UART1, UART2 y UART3
- Interfaz Ethernet
- El controlador de color LCD
- El controlador DMA
- El vector de interrupciones primario
- Dos interfaces para utilizar cuatro TIMERs
- Interfaz para el reloj en tiempo real

3.5 Herramienta de integración

En los apartados anteriores, hemos explicado con detalle cómo obtener y construir los elementos necesarios para crear el procesador ARM. Ahora, es turno de enlazar todos aquellos elementos, reunirlos en uno solo e integrarlo en una interfaz gráfica de usuario para un manejo cómodo.

3.5.1 Diseño de la interfaz

En este punto del proyecto, la finalidad es implementar una interfaz gráfica de usuario intuitiva y sencilla donde el usuario pueda crear código, manipularlo, simularlo e interpretar los resultados.
Para conseguir todo esto, hemos distinguido tres componentes principales en el diseño de la misma:

i. Editor de código
ii. Menús
iii. Paneles de información al usuario

3.5.1.1 Editor de código

FIGURA 30: Editor de Código

Para editar código, la interfaz dispone de un panel, en el que cada vez que el usuario crea código nuevo o abre uno ya existente del sistema de archivos, aparece una pestaña, como la que se observa en la *FIGURA 29*.

Para desarrollar esta parte, hemos reutilizado el área de texto de un editor de texto externo, en concreto, del editor de texto jEdit.
Haremos una breve pausa en la explicación de la herramienta de integración para pasar a hablar brevemente de jEdit.

3.5.1.1.1 jEdit

jEdit es un editor de texto libre distribuido bajo los términos de la Licencia pública general de GNU. Su autor principal es Slava Pestov y ha estado implementándose desde 1998.

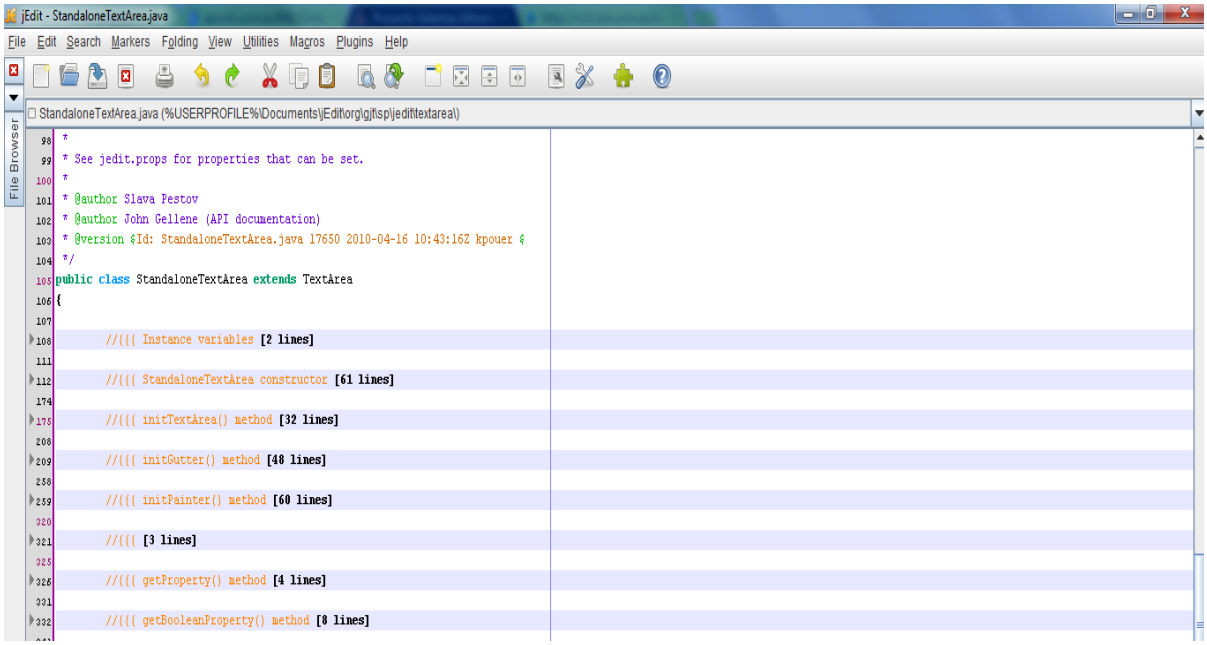


FIGURA 31: Interfaz jEdit

Características principales:

- Incluye soporte multiplataforma
- Se ejecuta en Windows, Mac OS X, Unix y otros sistemas operativos que dispongan de la máquina virtual de Java 1.6 o superior.
- Se puede personalizar y extender con macros escritas en BeanShell, Jhyton, JavaScript y otros lenguajes script.
- Dispone de docenas de plugins para diferentes áreas de aplicaciones pudiéndose así convertir en un editor avanzado de XML/HTML, o en un entorno de desarrollo integrado (IDE), con compilador, completado de código, ayuda contextual y herramientas de depuración, diferenciación visual y de un lenguaje de programación específico. Los plugins pueden ser descargados e instalados desde jEdit a través de la opción “plugin manager”.
- Soporta UTF-8 y otros formatos de codificación del texto
- Soporta el resaltado de sintaxis coloreado para más de 200 formatos de fichero. También se puede incluir nuevos formatos de forma manual utilizando ficheros XML.
- Sin límite de deshacer/rehacer
- Copiar y pegar con un número ilimitado de portapapeles (conocidos como “registros”)
- Recuerda automáticamente el texto previamente borrado
- Opciones para la manipulación de palabras enteras, líneas y párrafos
- “Marcadores” para recordar las posiciones de texto en los archivos
- Selección múltiple para la manipulación de varios trozos de texto a la vez
- Ayuda en línea.

¿Por qué jEdit?

Puesto que la implementación de la herramienta de integración era en lenguaje Java, creímos conveniente elegir el editor de texto jEdit pues es un software de código abierto e implementado en este mismo lenguaje, lo que nos permitiría poder manipular su código y adaptarlo al nuestro si fuese necesario.

Otras de las razones que nos motivó a elegir jEdit es su soporte de resaltado de sintaxis coloreado para más de 200 formatos de fichero. En concreto, resaltado de sintaxis para los lenguajes C y Ensamblador que son los que nos interesa.

Integración del área de texto de jEdit en la interfaz

El editor de textos jEdit tiene entre sus clases, la clase *StandaloneTextArea*, hecha precisamente, para incrustar su área de texto en aplicaciones externas. Esta clase cuenta con un método que crea un área de texto con unas propiedades definidas, previamente.

Para integrar su área de texto, en primer lugar, es necesario importar la clase *StandaloneTextArea* a nuestra aplicación, mediante un archivo *jar*. Con importar el paquete correspondiente a esta clase sería suficiente, sin embargo, importamos el *jar* completo del editor (*jedit.jar*). De esta manera podremos utilizar otras utilidades que ofrece jEdit, en nuestra interfaz.
Con esto, nuestra aplicación reconoce todos los paquetes y clases de las que está formado el código fuente de jEdit.

A continuación, invocamos al método *createTextArea* de la clase antes mencionada y con esto ya tenemos el área de texto.

```
TextArea Text = StandaloneTextArea.createTextArea ()
```

Por otra parte, queremos también aprovechar otra característica de este editor de texto: el resaltado de sintaxis de lenguajes de programación. Como se mencionó antes, en concreto, estamos interesados en los lenguajes de programación C y ensamblador.
Cuando el usuario escriba código queremos que aparezcan resaltadas las palabras reservadas del lenguaje de programación en concreto que esté utilizando (C o ensamblador).

Para obtener esto, jEdit utiliza las clases *Mode* y *ModeProvider*.

Con la clase Mode, definimos un lenguaje de programación para el que queremos tener resaltado de sintaxis.
Con la clase ModeProvider creamos una instancia del nuevo modo definido.
Y finalmente, indicamos al área de texto creada anteriormente, que su modo de edición sea este nuevo modo definido. Por ejemplo, si queremos crear un archivo en lenguaje C, hacemos:

```
Mode mode = new Mode (“c”)
mode.setProperty ("file", "modes/c.xml")
ModeProvider.instance.addMode (mode);
Text.getBuffer ().setMode (mode);
```

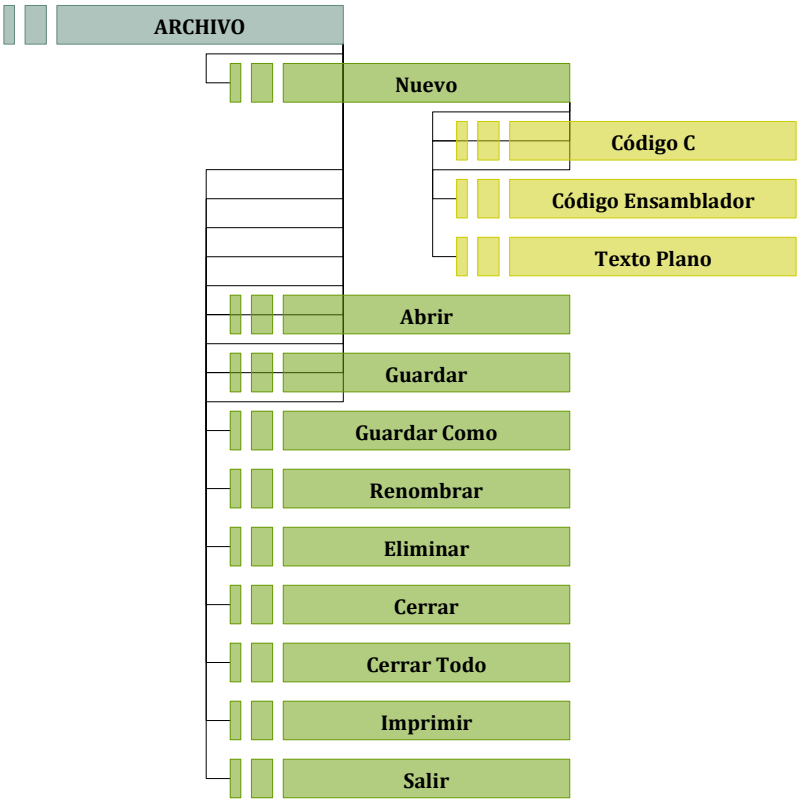
En el código fuente de jEdit hallamos una carpeta llamada *modes* donde se encuentran archivos xml correspondientes a cada uno de los lenguajes de programación que soporta jEdit.

3.5.1.2 Menús

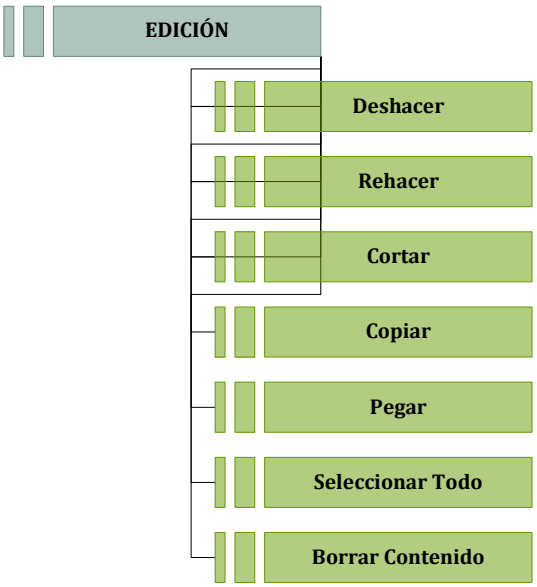
FIGURA 32: Menús

Para la manipulación del código, disponemos de menús con diferentes opciones. También, hemos añadido una barra de herramientas con las opciones de menú más frecuentemente utilizadas.
Dichas opciones, haciendo especial hincapié en algunas de ellas, se resumen a continuación.

Menú Archivo



Menú Edición



Para implementar las opciones que recoge el menú Edición, nos hemos valido también, de algunas de las opciones de jEdit.
jEdit, dispone del archivo *actions.xml* donde se encuentran ya implementadas estas opciones. Una vez que tenemos creada un área de texto, ya podemos acceder a ellas mediante simples instrucciones java.

Algunos ejemplos son:

- **Edición → Deshacer:**

```
TextArea Text = StandaloneTextArea.createTextArea ();
JEditBuffer buffer = Text.getBuffer ();
buffer.undo (Text);
```

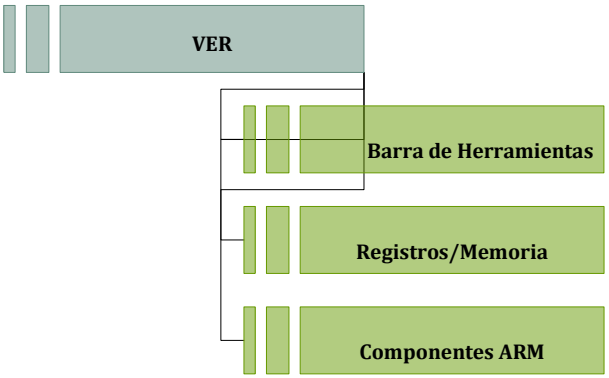
- **Edición → Seleccionar Todo**

```
TextArea Text = StandaloneTextArea.createTextArea ();
Text.selectAll ();
```

- **Edición → Cortar**

TextArea Text = StandaloneTextArea.createTextArea ();
Registers.cut (Text,'$');

Menú Ver



- **Registros/Memoria:** muestra el contenido de los registros y la memoria del procesador

  Para implementar esta opción hemos usado los comandos *info reg* (registros) y *x/nfu dirección_memoria* (memoria) que posee el depurador Gdbserver de nuestro compilador cruzado.

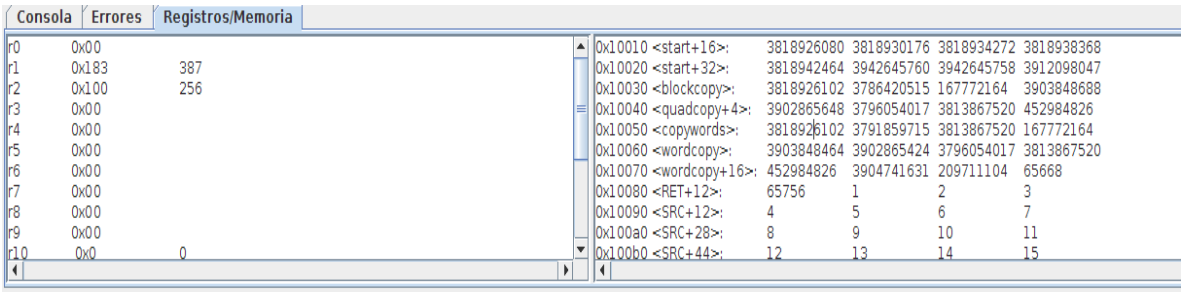


FIGURA 33: Registros/Memoria

- **Componentes ARM:** visualiza, de manera gráfica, los componentes que posee el maletín de los laboratorios. Estos componentes están ligados a las direcciones de memoria, correspondientes, de nuestro procesador ARM.

  - Dos leds y dos pulsadores
  - Un display 8 segmentos
  - Un teclado y una pantalla

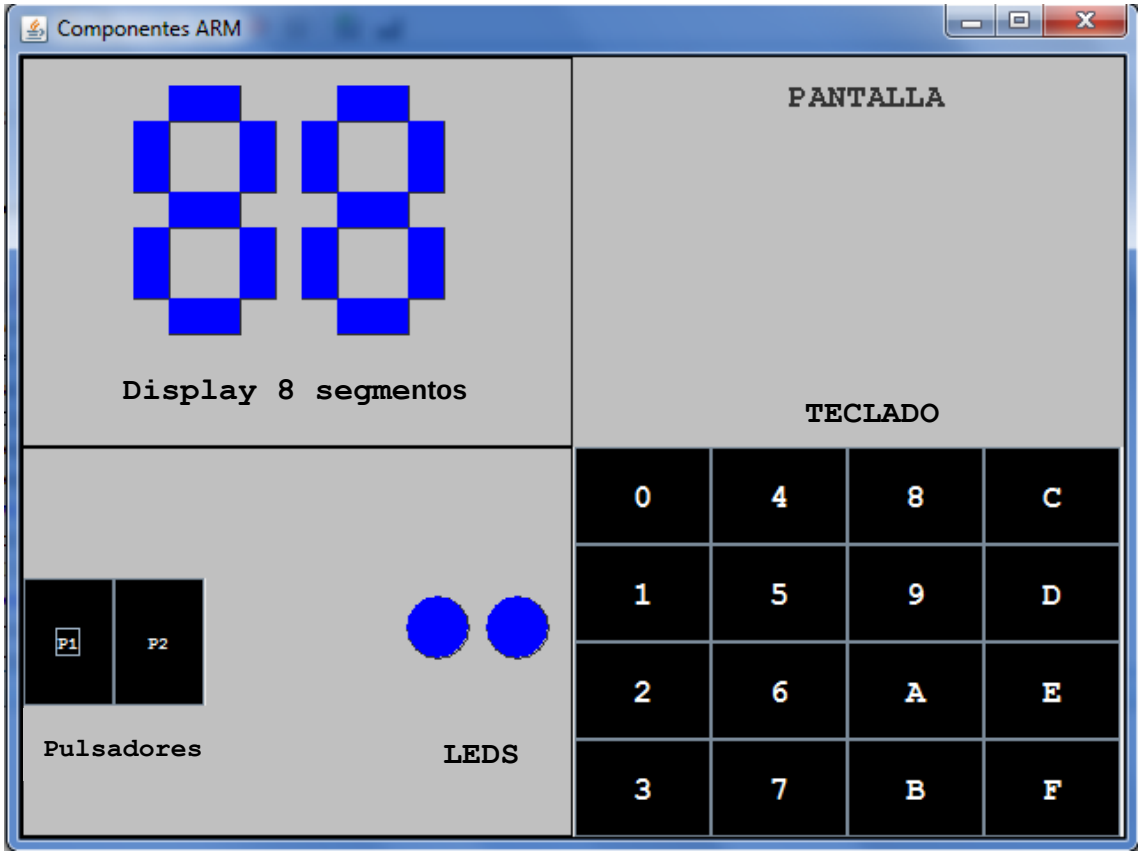
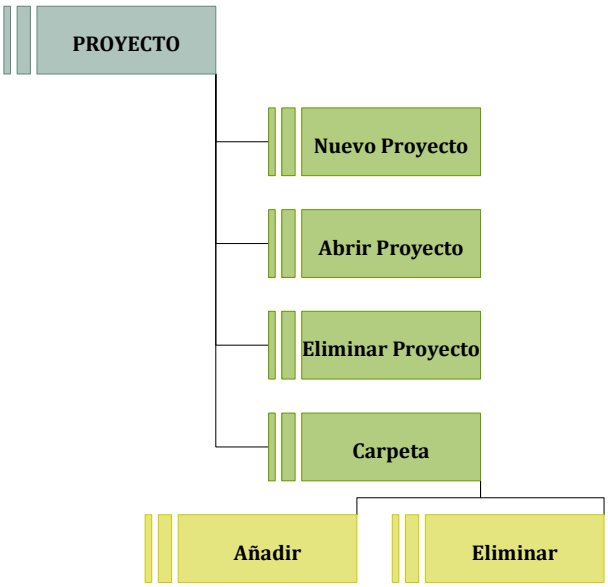


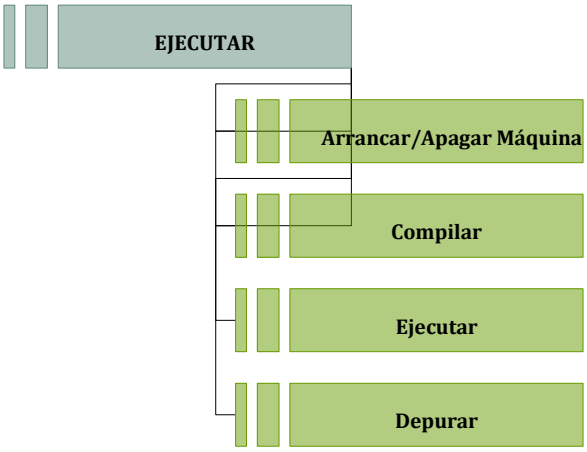
FIGURA 34: Componentes ARM

Menú Proyecto



Por defecto, el espacio de trabajo se ha establecido en el directorio de usuario, en una carpeta llamada *Proyectos* (*/home/<usuario>/Proyectos*). El usuario deberá ubicar aquí dichos proyectos para posteriormente ser simulados.

Menú Ejecutar



- **Arrancar/Apagar máquina:** mediante un comando enciende/apaga la máquina servidor QEMU.

- **Compilar:** detecta posibles errores y/o advertencias en el código implementado. Para implementar esta opción usamos el compilador GCC de en nuestro compilador cruzado.

- **Ejecutar y depurar:** ejecuta y depura, respectivamente, el código implementado. Para implementar estas opciones, usamos el depurador Gdbserver de nuestro compilador cruzado.

Para realizar estas operaciones, es necesaria, una interacción entre la interfaz y nuestro compilador cruzado.

Interacción entre la interfaz gráfica y el compilador cruzado

Para poder arrancar un programa externo (compilador GCC y depurador Gdbserver) desde Java, contamos con la clase *RunTime*. Esta clase permite ejecutar un proceso o thread paralelo a la ejecución normal de la aplicación.

Process p = Runtime.getRuntime ().exec (<ruta-ejecutable>);

Para leer lo que devuelve el ejecutable externo, debemos usar el objeto **Process** que nos devuelve **Runtime.exec()**. Este objeto **Process** representa, de alguna forma, al ejecutable externo mientras está corriendo. Con él podemos obtener la salida del programa, su salida de error e incluso enviarle datos como si se estuvieran tecleando.
Si queremos leer su salida, usamos el método **getInputStream()** de la clase **InputStream.** Puesto que un **InputStream** es algo incómodo para leer cadenas de texto, a continuación, usamos la clase ***BufferedReader*** que por medio del método ***readLine()*** que nos devuelve directamente una línea de texto en forma de ***String***.

Ejemplo de compilación:

Si queremos obtener el fichero objeto de un archivo ensamblador, usando el compilador GCC de nuestro compilador cruzado, hacemos lo siguiente:

```
Comando = "<ruta-crosstool-as> -g <ruta-archivo-origen.s> -o
          <ruta-archivo-destino.o> "

Process p = Runtime.getRuntime().exec(Comando)
```

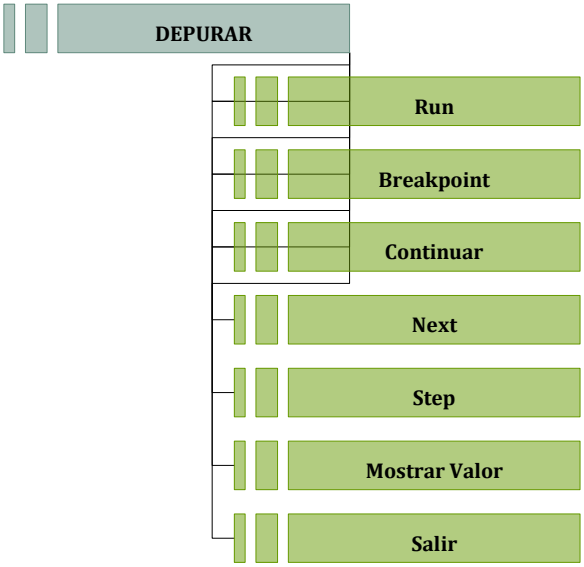
Ejemplo de depuración:

Si queremos depurar un archivo binario usando el depurador Gdbserver de nuestro compilador cruzado, hacemos lo siguiente:

```
Process p;

//Ejecutar con GDB en modo depuración

p = Runtime.getRuntime().exec("<ruta-croostool-gdb>  --interpreter=console")

//Conectar remotamente GDB con QEMU

stdin.println ("target remote: 1234")

//Enviar la ruta del proyecto a depurar

stdin.println ("file <ruta-archivo-origen.elf>")
```

Tras esto, ya podemos depurar, paso a paso, mediante las opciones de depuración, del menú *Depurar* de la interfaz.

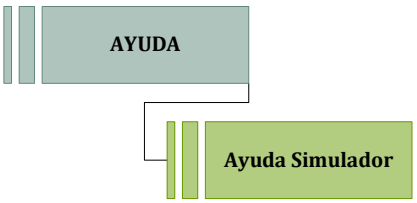
Menú Depurar



Para implementar las opciones de depuración, utilizamos los comandos de depuración del depurador GDB de nuestro compilador cruzado, que son los siguientes:

- Run → r (Ctrl + F5)
- Añadir Breakpoint → b línea
- Borrar Breakpoint → b línea
- Continue → c (Mayúsculas + F6)
- Next → n (F7)
- Step → s (F8)
- Mostrar valor → print valor
- Detener → quit (Mayúsculas + F5)

Menú Ayuda



Al cliquear en esta opción, automáticamente, la aplicación abre una nueva pestaña en nuestro navegador Web con un manual de ayuda.

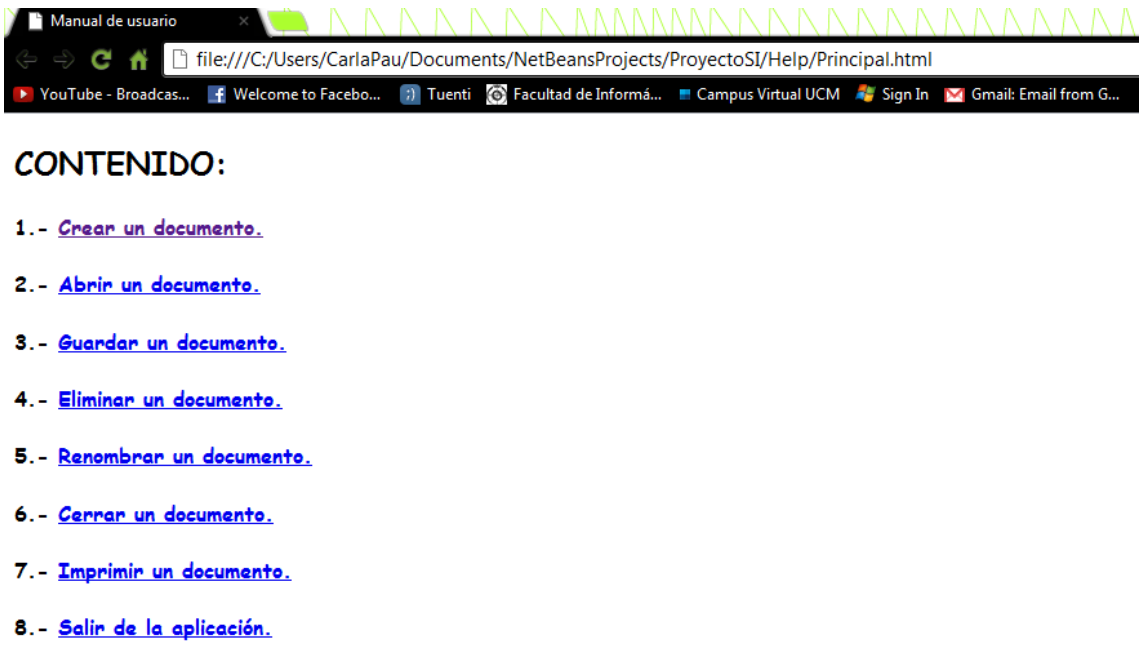


FIGURA 35: Ayuda usuario

3.5.1.3 Paneles de información al usuario

Para que el usuario esté informado en todo momento de los documentos y proyectos creados, la interfaz cuenta con un panel titulado *Workspace*. En este panel se muestra, en forma de árbol, el contenido del directorio de trabajo (carpeta *Proyectos*) que tiene por defecto la aplicación.

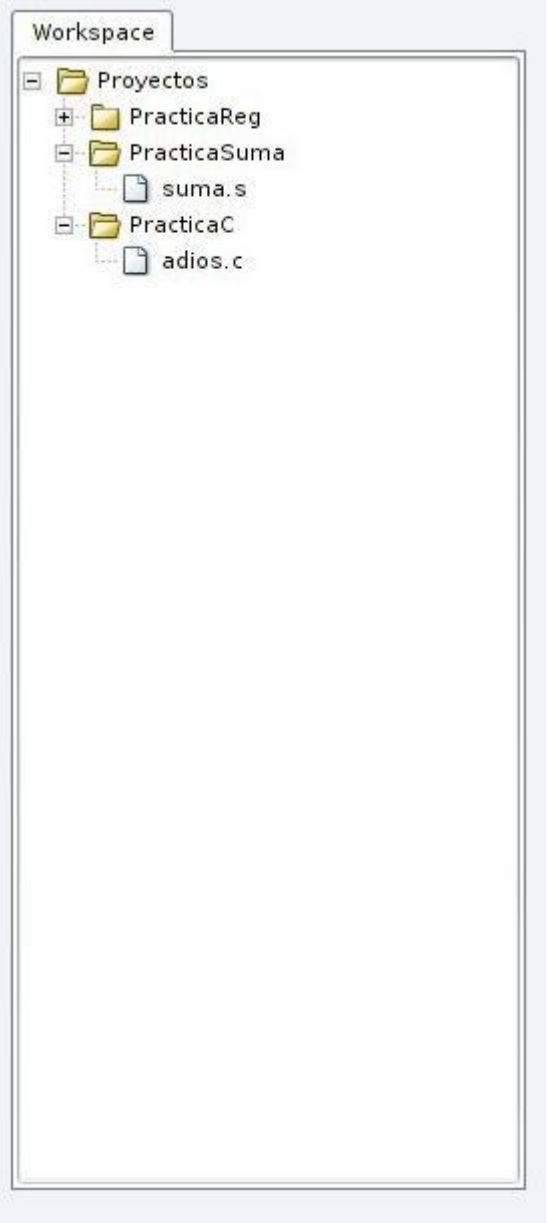


FIGURA 36: Workspace

Para que el usuario pueda comprobar los resultados tras la ejecución de código, la interfaz cuenta con un panel formado por tres pestañas: Salida, Errores y Registros/Memoria.

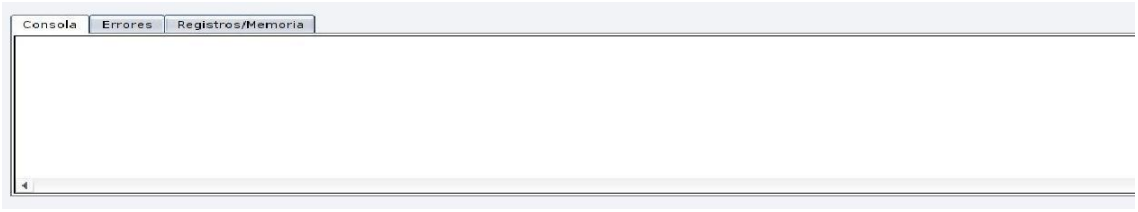


FIGURA 37: Panel de Salidas

En la primera pestaña se mostrarán los resultados tras la ejecución de un proyecto. En la segunda, se mostrarán errores y advertencias que pueden aparecer en la fase de compilación y ejecución del proyecto. Y en la tercera, se mostrará el contenido de los registros y direcciones de memoria del procesador ARM.
Tras explicar con detalle el diseño de la interfaz gráfica veamos el resultado final:

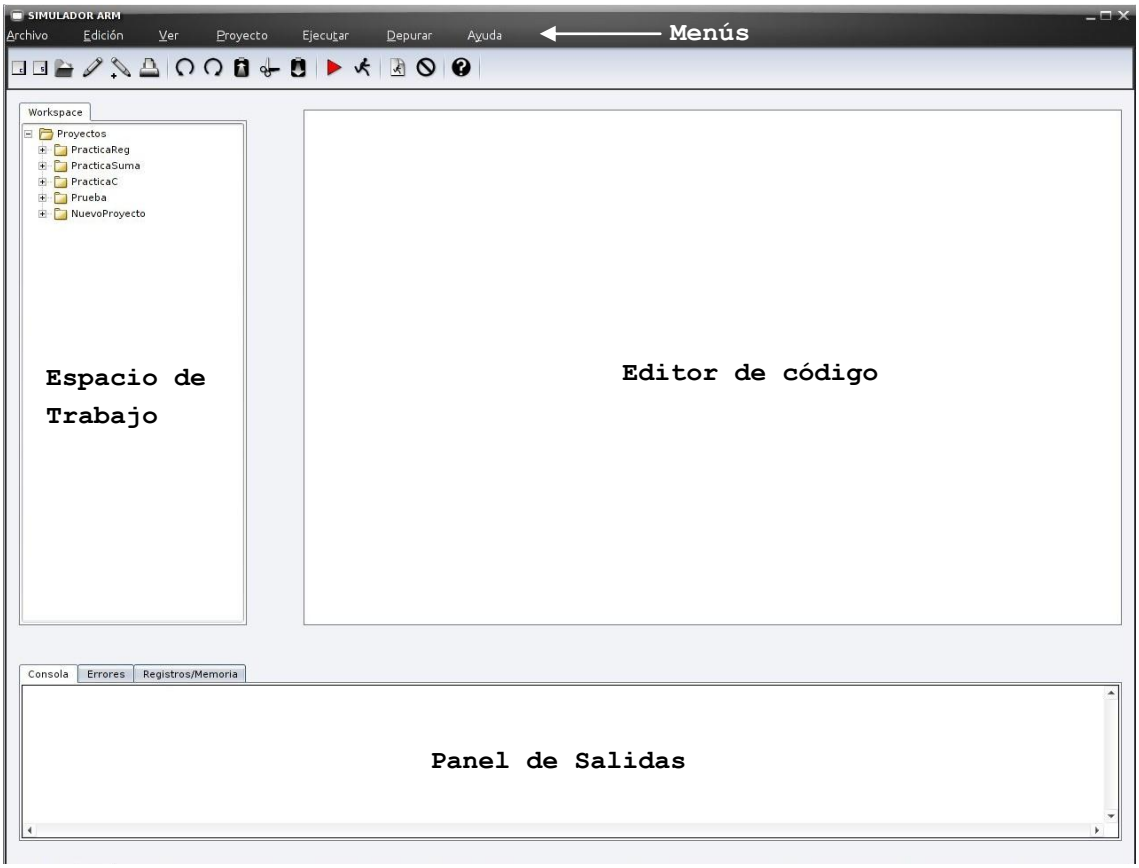


FIGURA 38: Interfaz gráfica de usuario

CAPÍTULO 4
COMPATIBILIDAD PRÁCTICAS ASIGNATURA EC

Capítulo 4 – Compatibilidad Prácticas Asignatura EC

En este capítulo, se explica la comprobación de la herramienta desarrollada en el proyecto, usando como base la simulación de las prácticas de la asignatura Laboratorio de Estructura de Computadores.

Para su correcto funcionamiento, se ha tenido que adaptar parte del código de las prácticas. Dichas adaptaciones (marcadas en negrita) se explican, detalladamente, en cada uno de los ejemplos que han sido probados, los cuales se explican a continuación. Previamente, se describe el objetivo de cada práctica.

4.1 Programación en ensamblador

Objetivo: familiarizarse con la arquitectura ARM y su programación en lenguaje ensamblador.

Ejemplo: se pide crear un programa que realice el producto escalar de dos vectores almacenados en memoria a partir de las direcciones V1 y V2. Cada vector tendrá N componentes de tamaño byte sin signo, donde N será una constante mayor que cero almacenada en memoria. El resultado se almacenará en memoria.

Desarrollo:

1. Creamos un nuevo fichero y lo guardamos con el nombre *ld_script.ld*. Este será el fichero de entrada al enlazador, que determinará las secciones del ejecutable y su ubicación. Este fichero es fundamental para que las prácticas se realicen correctamente y cuya explicación vale para el resto de las prácticas.

   Se ha modificado el inicio de las secciones, pues, al emular nuestra herramienta con QEMU se lee, por defecto, el archivo binario a partir de la dirección *0x00010000*. Este debe ser, explícitamente, indicado si no queremos tener problemas de acceso a memoria, ya que el enlazador asume que el programa es ejecutado a partir de la dirección 0x00000000. Seguidamente, indicamos al enlazador, mediante la instrucción *startup,* que cargue nuestro fichero ensamblador. Y a continuación, declaramos las secciones de memoria que se utilizarán en la práctica.

Otro de los cambios esenciales en este fichero, ha sido situar el puntero de pila (*sp_top*) 4Kbyte por encima. De esta manera, el enlazador mueve el puntero de pila a esa posición evitando que se corrompan las secciones que utilizamos de memoria, puesto que el puntero de pila crece hacia abajo.

```
SECTIONS
{
. = 0x10000;
startup : { Practica1.o(.text)}
.text : { *(.text) }
_bdata = .;
.data : { *(.data) }
_edata = .;
.rodata : { *(.rodata) }
_bbss = .;
.bss : { *(.bss) }
_ebss = .;
. = . + 0x1000;
sp_top = .;              @ Reserva de memoria para el StackPointer
}
```

2. Creamos un fichero con el nombre *Practica1.s* que contiene el código que realiza el producto escalar de dos vectores, tal y como pide el enunciado.

```
.global start
.equ N, 3                       @ Tamaño de ambos vectores
.data
 v1:    .byte 3, 2, 4           @ Vector 1
 v2:    .byte 1, 2, 1           @ Vector 2
.bss

Res:    .space  4                       @ reservamos memoria (4 bytes) resultado
.text
start:  ldr r0, =v1             @ puntero a v1: r0 = M[v1]
        ldr r1, =v2             @ puntero a v2: r1 = M[v2]
        ldr r2, =Res            @ puntero a Res: r2 = M[Res]
        mov r3, #0              @ acumulador
        mov r4, #N              @ contador
```

```
bucle:  ldrb r5, [r0], #1       @ post-indexado: r5 <- [r0]; r0++
        ldrb r6, [r1], #1       @ post-indexado: r6 <- [r1]; r1++
        mla r3, r5, r6, r3      @ acum <- v1(i)*v2(i) + acum
        subs r4, r4, #1         @ cont--
        bne bucle

fin:    str r3, [r2]            @ almacenar el resultado en la memoria
```

4.2 Paso de C a ensamblador

Objetivo: profundizar en los conocimientos sobre la programación en ensamblador sobre ARM aprendiendo a combinar código escrito en un lenguaje de alto nivel como C con código escrito directamente en lenguaje ensamblador.

Ejemplo: crear un programa en C que sume dos números y guarde el resultado en una variable local res. Posteriormente, guardar el contenido de res en una variable global RES.

Desarrollo:

1. Creamos un fichero y lo guardamos con el nombre *ld_script.ld*. Este será es similar al del ejemplo anterior.

2. Creamos un nuevo fichero con el nombre *init.s*. Este fichero será el encargado de inicializar la arquitectura para la ejecución del programa escrito en lenguaje C y de invocar la función de entrada al programa (Main). Conviene destacar que en esta práctica hemos utilizado la variable start como global para que nos sirva de punto de entrada en el fichero enlazador con la función entry(start) al inicio del fichero, indicando al enlazador que ese es su punto de entrada. Es fundamental inicializar el puntero de pila, en la posición adecuada como se indica en el fichero ld_script.ld, ya que en esta práctica tenemos llamadas a rutinas o funciones en C, que necesitan un marco de pila para almacenar la información necesaria en su ejecución. Sin esto, nuestro programa daría una excepción.

```
.global start
.text
.start:
        LDR sp, =stack_top      @ Almacenar valor de StackPointer asignado en
ld_script.ld

.extern Main
        ldr r0,=Main
        mov lr,pc
        mov pc,r0
End:
        B End
        .end
```

3. Creamos un nuevo fichero fuente con el nombre *Main.c* que contiene el código C que suma dos números.

```
extern char _bdata[];
extern char _edata[];
extern char _bbss[];
extern char _ebss[];


char * inidata = (char *) _bdata;
char * enddata = (char *) _edata;
char * inibss = (char *) _bbss;
char * endbss = (char *) _ebss;

int Res;

int sum2(int a1, int a2)
{
        int sum_local;
        sum_local = a1 + a2;
        return sum_local;
 }
```

```
 Main(void)
{
 int res;
res = sum2(1,2);
Res = res;
 }
```

El valor de la variable *Res* es 0x0000000B.

4.3 Excepciones y modos

Objetivo: analizar las excepciones y modos de ejecución de un procesador ARM.

Ejemplo: crear un programa, en código ensamblador, que inicialice correctamente el estado del proceso para todos los modos de ejecución y configure la tabla de direcciones de las rutinas de tratamiento de excepciones. Al finalizar, ejecutará la función Main que generará tres excepciones (Undef, Dabort y SWI) y terminará.

Desarrollo:

1. Creamos un fichero y lo guardamos con el nombre *ld_script.ld*. Las novedades de este fichero, respecto de las prácticas anteriores, es que ahora necesitamos reservar un espacio para el marco de activación del puntero de pila para cada una de las excepciones. Por otra parte, ahora, el punto de entrada del programa lo va a determinar el vector de interrupciones (situado en la dirección de memoria 0x0). Pero QEMU sitúa el código binario a partir de la dirección 0x10000. Por esta razón, este debe ser copiado antes de ser utilizado. Esto se hace con las variables globales *vectors_start* y *vectors_end*, que señalan el principio y el final del vector, y con la función *copy_vectors*, implementada en C, que permite copiar el vector en la dirección correcta.

```
ENTRY(vectors_start)
SECTIONS{
. = 0x10000;
 .startup . : { Practica3.o (.text)}
.text : { *(.text) }
 .data : { *(.data) }
.bss : { *(.bss) }
. = ALIGN(8);
 . = . + 0x1000; /* 4kB of stack memory */
 stack_top = .;
 . = . + 0x1000; /* 4kB of user_stack memory */
user_stack_top = .;
 . = . + 0x1000; /* 4kB of svc_stack memory */
 svc_stack_top = .;
 . = . + 0x1000; /* 4kB of undef_stack memory */
 undef_stack_top = .;
 . = . + 0x1000; /* 4kB of abort_stack memory */
 abort_stack_top = .;
 . = . + 0x1000; /* 4kB of irq_stack memory */
irq_stack_top = .;
. = . + 0x1000; /* 4kB of fiq_stack memory */
 fiq_stack_top = .;}
```

2. Creamos un fichero nuevo con el nombre *init.s*. Este fichero contiene el código que debe inicializar la arquitectura para luego poder ejecutar nuestro programa, junto con el vector descrito previamente.
   El trabajo principal de inicialización lo realizan dos subrutinas:

   - InitStacks: Se encarga de inicializar las pilas de los distintos modos de ejecución, excepto el de usuario-system. Para ello, va cambiando de modo de ejecución, manteniendo enmascaradas las interrupciones, y en cada modo escribe en el registro de pila la dirección de comienzo correspondiente al modo, que está dada por el símbolo **<Modo>Stack_top**. Debemos recalcar que, a la salida de esta rutina el modo debe ser privilegiado. Es más, como la subrutina se invocó desde SVC es preciso regresar a este modo para poder realizar correctamente el retorno de subrutina.

- InitISR: Se encarga de rellenar correctamente la Tabla de Direcciones de ISRs. En concreto, para cada excepción deberá cargar en un registro la dirección de comienzo de la rutina que debe tratar la excepción, dada por el símbolo ISR_<Excepción>. Posteriormente cargará en un segundo registro la dirección de la rutina, dada por el símbolo Handle<Excepción>. Finalmente escribirá el contenido del primer registro (dirección de la ISR) en la dirección dada por el segundo registro (entrada correspondiente en la tabla de ISRs).

Tras ejecutar estas rutinas, el fichero init.s pasará la ejecución a modo usuario, inicializará su pila, e invocará la función Main. El código generado es el siguiente:

```
 .text
.global _start
.global DoUndef
.global DoSWI
…
.global vectors_start
.global vectors_end

vectors_start:
 LDR PC, reset_handler_addr
LDR PC, undef_handler_addr
 LDR PC, swi_handler_addr
…
B .
undef_handler_addr: .word HandleUndef
swi_handler_addr: .word HandleSWI
…
vectors_end:
@Símbolos para facilitar la codificación de los modos de ejecución
.equ USERMODE, 0x10
.equ FIQMODE, 0x11
….
.arm
_start:
```

```
@El procesador está en modo supervisor
bl InitStacks

@ Seguimos en modo supervisor, configuramos las direcciones de las rutinas de tratamiento
bl InitISR

@Pasamos a modo usuario, inicializamos su pila
mrs r0, cpsr
bic r0, r0,#MODEMASK
orr r1, r0,#USERMODE
msr cpsr_cxsf, r1 /* UserMode */
ldr sp, =user_stack_top

@ Saltamos a Main
.extern Main
mov lr, pc
ldr pc, =Main
b .

InitStacks:

mrs r0, cpsr
bic r0, r0, #MODEMASK
orr r1, r0, #UNDEFMODE
orr r0, r1, #NOINT
msr cpsr_cxsf, r0
ldr sp, =undef_stack_top

…
mov pc, lr        @ El lr=r14 propio del modo supervisor, que no se ha visto modificado

InitISR:

ldr r0, =ISR_Undef
ldr r1, =HandleUndef
str r0, [r1]

ldr r0, =ISR_SWI
ldr r1, =HandleSWI
```

```
str r0, [r1]
…
mov pc, lr        @ No hemos cambiado de modo SVC

@ Rutinas de generación de excepciones
DoSWI:
        swi
        mov pc, lr
DoUndef:
        .word 0xE6000010
        mov pc, lr
…
screen: .space 1024

/* Función write_C en ensamblador*/
write_ASM:
        …
.end
```

3. Creamos un nuevo fichero con el nombre Main.c, que contiene las diferentes rutinas en C, que copiarán en la variable screen el nombre de la excepción en función de la excepción ocurrida.

```
extern char screen[];
char *Screen = (char*) screen;

void ISR_SWI(void) __attribute__ ((interrupt ("SWI")));
void ISR_Undef(void) __attribute__ ((interrupt ("UNDEF")));
…
void ISR_Dabort(void) __attribute__ ((interrupt ("ABORT")));

void Main(void){ DoUndef(); DoSWI(); DoDabort(); }

void ISR_Undef(void) {write_ASM("Undef ",Screen); }
…
void ISR_SWI(void) { write_ASM("SWI ",Screen); }
…
```

4.4 E/S Básica

Objetivo: estudiar el sistema de E/S de la placa emulada Versatilepb, haciendo hincapié en su gestión mediante el mecanismo de interrupciones.
Como ejemplo práctico utilizaremos los dispositivos más sencillos de los que disponemos: unos LEDs, un display de ocho segmentos y unos temporizadores.

Ejemplo: en este ejemplo se realiza un programa que ilumina los LEDs de manera alterna a intervalos regulados mediante interrupciones del temporizador.

Desarrollo:

1. Creamos un fichero y lo guardamos con el nombre *ld_script.ld* (similar a los ejemplos anteriores).

2. Creamos un nuevo fichero con el nombre *init.s*. El fichero presenta la misma estructura que en el ejemplo anterior, con una inicialización específica para permitir las interrupciones IRQ.

```
 .text
.global _start
.global vectors_start
 .global vectors_end

vectors_start:
LDR PC, reset_handler_addr
LDR PC, irq_handler_addr
B .
…

reset_handler_addr: .word ISR_Reset
…
irq_handler_addr: .word ISR_IRQ

vectors_end:
```

```
.equ USERMODE, 0x10
.equ FIQMODE, 0x11
.equ IRQMODE, 0x12
…
ISR_Reset:
@set Supervisor stack
LDR sp, =stack_top

@copy vector table to address
BL copy_vectors

@get Program Status Register
MRS r0, cpsr

@go in IRQ mode
BIC r1, r0, #0x1F
ORR r1, r1, #0x12
MSR cpsr, r1

@set IRQ stack
LDR sp, =irq_stack_top

@enable IRQs
BIC r0, r0, #0x80

@go back in Supervisor mode
MSR cpsr, r0

@jump to Main
BL Main
B .
.end
```

3. Creamos un nuevo fichero con el nombre *leds.c.* En este fichero configuramos el comportamiento de los leds. Lo fundamental en este apartado es indicar cual es la dirección de memoria que utilizamos para consultar si los LEDs se iluminan o no. En nuestro caso, hemos optado por utilizar la posición de memoria 0x800000, tomando los dos pines iniciales como los controladores para el led0 (bit 0) y el led1 (bit 1). Si el bit 0 o el bit 1 en dicha posición de memoria tienen un 1, entonces se iluminará el led correspondiente. En caso contrario, estará apagado.

```
#define LED_DIR (*((volatile uint32_t *)(0x00800000)))

int led_state;                                  //estado del LED

@declaración de funciones
void leds_on();                         //todos los LEDs encendidos
void leds_off();                        //todos los LEDs apagados
void led1_on();                         //LED 1 encendido
void led1_off();                                //LED 1 apagado
…
void leds_switch();                     //invierte el valor de los LEDs

@código de las funciones
void leds_on() { LED_DIR = 0x3; }

void leds_off() { LED_DIR = 0x0; }

void led1_on() { led_state = led_state | 0x1;
                 LED_DIR = led_state; }

void led1_off() { led_state = led_state & 0x2;
                 LED_DIR = led_state; }

void led2_on() { led_state = led_state | 0x2;
                 LED_DIR = led_state; }

void led2_off() {       led_state = led_state & 0x1;
                 LED_DIR = led_state; }

void leds_switch () { led_state ^= 0x3;         //cambia el estado de los leds
                      LED_DIR = led_state; }
```

4. Creamos un nuevo fichero con el nombre *Main.c* al que añadimos las funciones para el manejo del temporizador. En este apartado hay que destacar que tenemos que definir las posiciones características del vector de interrupciones cuya memoria base se define con la variable VIC_BASE_ADDR tal y como se describe en el manual de referencia de la placa Versatilepb (1*). Por otra parte, también hay que definir las direcciones de memoria asociadas al temporizador de la placa, y modificar sus registros en función del manual específico para dicho elemento constitutivo de la placa.

```
#define VIC_BASE_ADDR 0x10140000
#define VIC_INTENABLE (*((volatile uint32_t *)(VIC_BASE_ADDR + 0x010)))
#define TIMER0 ((volatile unsigned int*)0x101E2000)
#define TIMER_VALUE 0x1 /* 0x04 bytes */
#define TIMER_CONTROL 0x2 /* 0x08 bytes */
#define TIMER_INTCLR 0x3 /* 0x0C bytes */
#define TIMER_MIS 0x5 /* 0x14 bytes */
#define TIMER_EN 0x80
#define TIMER_PERIODIC 0x40
#define TIMER_INTEN 0x20
#define TIMER_32BIT 0x02
#define TIMER_ONESHOT 0x01
#define PIC_TIMER01 0x10

extern void leds_off();
…
void ISR_IRQ(void) __attribute__ ((interrupt ("IRQ")));
void ISR_FIQ(void) __attribute__ ((interrupt ("FIQ")));
…
void copy_vectors(void) {

 extern uint32_t vectors_start;
 extern uint32_t vectors_end;
 uint32_t *vectors_src = &vectors_start;
 uint32_t *vectors_dst = (uint32_t *)0;
```

(1*) RealView Platform BaseBoard for ARM936EJ-S – User guide

```
…
void ISR_FIQ(void){while(1);}
…
void ISR_IRQ(void){
        leds_switch();
}
void main(void) {

VIC_INTENABLE = PIC_TIMER01;        @TIMER01
*TIMER0 = 1000000;
*(TIMER0 + TIMER_CONTROL) = TIMER_EN | TIMER_PERIODIC | TIMER_32BIT |
TIMER_INTEN;

leds_off();
led1_on();
while(1);

}
```

4.5 Puerto Serie

Objetivo: conocer un dispositivo de comunicaciones serie estándar, la UART. Este dispositivo nos permitirá comunicar la placa con el PC por medio de un protocolo serie asíncrono.

Emularemos el comportamiento típico de una entrada/salida estándar de dos formas: sin utilizar interrupciones y utilizando interrupciones.

Ejemplo 1 (sin interrupciones): enviar caracteres como letras minúsculas y convertirlos en letras mayúsculas. El resultado se visualizará por la salida emulada en QEMU.

Desarrollo:

1. Creamos un fichero con el nombre *ld_script.ld*. Hay que colocar la sección de código a partir de la posición 0x10000 y poner como punto de entrada la rutina _Reset.

```
ENTRY(_Reset)
SECTIONS
{
 . = 0x10000;
.startup . : { vectors.o (.text)}
 .text : { *(.text) }
.data : { *(.data) }
.bss : { *(.bss) }
. = ALIGN(8);
 . = . + 0x1000; /* 4kB of stack memory */
 stack_top = .;
}
```

2. Creamos un fichero nuevo con el nombre *init.s* (similar al de ejemplos anteriores).

```
.text
.global _Reset

_Reset:
B ISR_Reset
B .
…
ISR_Reset:
/* set Supervisor stack */
 LDR sp, =stack_top
 BL main
 B .
.end
```

3. Creamos un fichero nuevo con el nombre *Main.c* donde manipularemos la UART. Observando el manual de referencia (2*), encontramos la descripción concreta de los registros que vamos a utilizar en nuestro código.
   Para realizar este apartado, emulamos el mapa de memoria de un puerto serie PL011 con el registro pl011_T y creamos un puntero para la UART0 en la dirección de la placa Versatilepb (0x101f1000).

```
        typedef volatile struct {
        uint32_t DR;
        uint32_t RSR_ECR;
        uint8_t reserved1[0x10];
        const uint32_t FR;
        uint8_t reserved2[0x4];
        uint32_t LPR;
         uint32_t IBRD;
         uint32_t FBRD;
         uint32_t LCR_H;
         uint32_t CR;
        uint32_t IFLS;
         uint32_t IMSC;
        const uint32_t RIS;
        const uint32_t MIS;
        uint32_t ICR;
         uint32_t DMACR;
        } pl011_T;

enum {
        RXFE = 0x10,
        TXFF = 0x20,
};

pl011_T * const UART0 = (pl011_T *)0x101f1000;

static inline char upperchar(char c) {

        if((c >= 'a') && (c <= 'z')) {
                return c - 'a' + 'A';
         }
        else {
                 return c;
        }
}
```

(2*) ARM PrimeCell UART (PL011) Technical Reference Manual

```
static void uart_echo(pl011_T *uart) {

        if ((uart->FR & RXFE) == 0) {
                while(uart->FR & TXFF);
                uart->DR = upperchar(uart->DR);
         }
}
void main() {
         for(;;) {
                uart_echo(UART0);
         }
}
```

Ejemplo 2 (con interrupciones): tras introducir un caracter por teclado (entrada estándar) devolver por la salida emulada en QEMU, el siguiente caracter correspondiente (según el código ASCII). Es decir, si tecleamos la letra s (caracter número 73), devolver el caracter t (caracter número 74).

1. Creamos un fichero nuevo y lo guardamos con el nombre *ld_script.ld* (similar a de los ejemplos anteriores).

2. Creamos un fichero nuevo con el nombre *init.s* (similar al del ejemplo anterior).

3. Creamos un fichero nuevo con el nombre *Main.c* donde se manipulará la UART tal y como se describió en este apartado del ejemplo anterior.

```
#define UART0_BASE_ADDR 0x101f1000
#define UART0_DR (*((volatile uint32_t *)(UART0_BASE_ADDR + 0x000)))
#define UART0_IMSC (*((volatile uint32_t *)(UART0_BASE_ADDR + 0x038)))
#define VIC_BASE_ADDR 0x10140000
#define VIC_INTENABLE (*((volatile uint32_t *)(VIC_BASE_ADDR + 0x010)))

void __attribute__((interrupt)) ISR_IRQ() {

     UART0_DR = UART0_DR;        /* echo the received character + 1 */
}
```

```
/* all other handlers are infinite loops */
void __attribute__((interrupt)) ISR_Undef(void) { for(;;); }
void __attribute__((interrupt)) ISR_SWI(void) { for(;;); }
void __attribute__((interrupt)) ISR_Pabort(void) { for(;;); }
void __attribute__((interrupt)) ISR_Dabort(void) { for(;;); }
void __attribute__((interrupt)) ISR_FIQ(void) { UART0_DR = UART0_DR + 1; }

void copy_vectors(void) {

      extern uint32_t vectors_start;
      extern uint32_t vectors_end;
      uint32_t *vectors_src = &vectors_start;
      uint32_t *vectors_dst = (uint32_t *)0;

     while(vectors_src < &vectors_end){

               *vectors_dst++ = *vectors_src++;
     }
}

void main(void) {

     VIC_INTENABLE = 1<<12;          /* enable UART0 IRQ */
     UART0_IMSC = 1<<4; /* enable RXIM interrupt */
     While(1);

  }
```

CAPÍTULO 5
APORTACIONES Y DIFICULTADES

Capítulo 5 – Aportaciones y dificultades

En este capítulo, se exponen las principales aportaciones de este proyecto, posible trabajo futuro y finalmente algunas de las dificultades encontradas.

5.1 Aportaciones y conclusiones

En este apartado final exponemos las aportaciones realizadas, las conclusiones y unas posibles líneas de trabajo futuro.

Teniendo en cuenta las limitaciones actuales de los laboratorios: tema de licencias y su alto coste, dificultad de encontrar puestos libres en el laboratorio y la imposibilidad de usar los mismos en un horario flexible, consideramos que la principal aportación de este proyecto es la confección de un simulador ARM para el ámbito docente útil, libre, gratuito y que no depende de la plataforma en la que esté instalado.

La estructura modular de la herramienta es otro de los objetivos conseguidos. Todas las herramientas integradas en la aplicación son gratuitas y, además, de software libre. El compilador cruzado, puede sustituirse por cualquier otro que realice las mismas funcionalidades, como por ejemplo: *Buildroot*, *PTXdist* u *OpenEmbedded*. En cuanto al emulador, se pueden utilizar otras aplicaciones de virtualización como *SkyEye*.

Otra aportación que cabe destacar es la depuración/ejecución de programas de forma remota. Teniendo instalada la máquina emulada en un servidor, esta característica permitiría llevar a cabo la depuración/ejecución a través de dicho servidor (que puede dar servicio desde la facultad), sin necesidad de tener la máquina instalada en el sistema operativo.

En cuanto a la interfaz gráfica de usuario, hemos conseguido diseñarla de tal manera que sea sencilla, intuitiva y fácil de utilizar. Y cuenta con todas las funcionalidades básicas y necesarias para la simulación de código en un ARM.

La limitación del tiempo nos ha impedido realizar más aportaciones a este proyecto, como por ejemplo: emular la placa ARM, con puertos *GPIO*, con la capacidad de simular interrupciones a través de pulsadores y teclado y la utilización de la herramienta vía Web. Así pues, dejamos estas aportaciones como posibles líneas futuras de trabajo.

En conclusión, podemos decir que hemos alcanzado los objetivos propuestos al inicio del proyecto. Hemos obtenido un simulador ARM libre y gratuito para la docencia que integra modularmente herramientas de software libre, facilitando al alumno su utilización fuera de los laboratorios y evitando, por tanto, el alto coste de pagar una licencia de uso del entorno *Embest*.

5.2 Dificultades encontradas

A lo largo del desarrollo del proyecto nos hemos topado con muchas dificultades, algunas no fáciles de resolver. A continuación, comentamos los principales problemas encontrados tanto en la fase de desarrollo de la aplicación como en la fase de pruebas de la herramienta.

5.2.1 En la fase de desarrollo de la herramienta

La primera gran dificultad con la que nos encontramos fue aprender como desarrollar un compilador cruzado, que permitiera transformar el código generado en nuestro PC (Host) a un lenguaje que pudiera entender un procesador ARM (Guest). Tras investigar y probar diversas formas de hacerlo, vimos que crearlo manualmente era muy complejo, por lo que optamos por utilizar alguna herramienta que facilitara su construcción (Croostool-ng en este caso) adaptándolo a nuestras necesidades.

Otra dificultad hallada, ha sido intentar emular un sistema ARM similar al de la placa utilizada en los laboratorios. Probamos varios emuladores, tales como *SkyEye, ARMulator*, y finalmente nos quedamos con *QEMU* por su robustez.
Tras encontrarnos con muchos fallos, vimos que la mejor manera de utilizar y probar archivos compilados con el compilador cruzado, era instalando un sistema operativo Debian para ARM emulado con QEMU, pues sin ello no podíamos depurar con *GDB* y/o *Gdbserver*. A medida que íbamos comprobando estos archivos nos encontramos con excepciones y errores inesperados, provocados porque el compilador cruzado no enlazaba los ficheros con las librerías compartidas, adecuadamente.
Esto conseguimos solventarlo modificando los archivos de inicialización, enlazando y generando ficheros binarios que posteriormente invocaríamos con *QEMU* (como quedan especificados en el capítulo 4).

En cuanto al desarrollo de la interfaz gráfica, encontrar el modo de incorporar el editor de texto en nuestra aplicación, tampoco fue una tarea fácil ni intuitiva de realizar.
Entender un código Java tan avanzado fue una ardua tarea que también tiempo. No por falta de documentación o código comentado, sino por el nivel de conocimiento del lenguaje de programación que poseemos.

Investigar el código fuente no fue suficiente para hallar la solución, así que pensando en nuevas formas de encontrar la solución se nos ocurrió la idea de consultar foros de programación especializados en lenguaje Java. Obtuvimos varias respuestas pero ninguna acorde con lo que necesitábamos.
Finalmente, creímos que lo mejor era contactar directamente con los desarrolladores de *jEdit* (3*), pues quien mejor que ellos para saber si era posible cumplir o no con nuestro objetivo.

Tras intercambiar preguntas y dudas, dimos con la solución anteriormente expuesta. Aunque, no fue posible reutilizar todo, muchas otras opciones tuvimos que desarrollarlas por nuestra cuenta.
Otra solución plausible que nos planteamos en su momento, fue reutilizar el editor de texto entero y añadir mediante plugins las opciones de compilación y ejecución. Pero, finalmente, decidimos quedarnos con la idea inicial, incorporar el editor de texto en nuestra aplicación.

5.2.2 En la fase de pruebas de la herramienta

El principal problema que nos hemos encontrado en la fase de experimentación de la herramienta ha sido reproducir las prácticas simuladas en los laboratorios en la asignatura de EC.
El procesador ARM y la placa emulados no son exactamente los mismos que se utilizan en esta asignatura. La placa Samsung *SC3EV40* junto con el procesador *ARM7TDMI* utilizan regiones de memoria y estructuras distintas. Por este motivo, no hemos podido utilizar las mismas funciones, direcciones y estructuras propias de las prácticas.

Los ficheros de cabecera tan útiles en las prácticas de EC que facilitan la utilización de todas las prestaciones de la placa *SC3EV40*, nos han resultado poco útiles. Hemos tenido que redefinirlos para poder adaptarlos a nuestro emulador y a las prácticas, siguiendo como guía los diferentes manuales de usuario propios de cada periférico o controlador específico.

(3*)http://community.jedit.org/?q=forum

Siguiendo en la misma línea, a lo largo de las pruebas nos hemos encontrado con que la placa Versatilepb emulada en *QEMU* no da soporte a todos los periféricos y controladores, tal y como indica su manual de usuario. No pudiendo, por tanto, simular puertos *GPIO* que hagan las veces de los puertos *GPIO* propios del maletín del laboratorio. Esta dificultad nos ha impedido desarrollar plenamente la herramienta para emular los pulsadores y el teclado 4x4. Es por ello que las pruebas de la herramienta se han centrado en el resto de prácticas, aquellas que no hacen uso de esos componentes.

Esta limitación, se podría solucionar haciendo que el pulsador o la tecla llame a una rutina o través de otro periférico de entrada/salida como la *UART*. Pero, en este caso no estaríamos realizando una interrupción real a través de un puerto *GPIO*.

Otra dificultad que se nos ha presentado es tener que definir explícitamente el fichero de enlace (*.ld*) y adecuar el fichero de inicialización en ensamblador (*.s*). Al utilizar la herramienta de compilación cruzada, los ficheros objetos generados no son capaces de enlazarse correctamente con las librerías compartidas produciendo errores.

A la hora de utilizar *GDB* y el *Gdbserver*, nos encontramos con la dificultad del procesamiento de la interfaz del *TIMER*. Este componente puede producir interrupciones que no necesitan parar el procesador. Para poder captar la interrupción, hemos tenido que utilizar la instrucción *watch* de *GDB*.
De esta manera, se para la ejecución del programa en modo *run* o *continue* y se consulta la dirección de memoria. Así, si queremos simular la iluminación de los *leds* o el *display 7 segmentos* usando como interrupción un temporizador, esta dirección nos indica si se iluminan o no dichas componentes.

BIBLIOGRAFÍA Y REFERENCIAS

Bibliografía y Referencias

Libros

David Seal. *ARM Architecture Reference Manual (2nd Edition).* Addison-Wesley.

Steve Furber. *ARM system-on-chip architecture (2nd Edition)*. Addison-Wesley.

David A. Patterson; John L. Hennessy. *Estructura y diseño de computadores. La interfaz hardware/software (4ª Edición)*. Reverte, 2011.

Andrew N. Sloss; Dominic Symes; Chris Wright. *ARM System developer´s guide. Designing and optimizing system software*. Elsevier, 2004.

Radu Muresan. *Embedded System Development and Labs for ARM*.

Enlaces

Leonid Ryzhyk. *The ARM Architecture*. 2006. http://www.cse.unsw.edu.au/~cs9244/06/seminars/08-leonidr.pdf

Arquitecturas y procesadores ARM. http://es.scribd.com/doc/55525832/Arquitecturas-y-Procesadores-ARM

Marcelo E. Romeo; Eduardo A. Martínez. *Microcontroladores de 32 bits ARM… O como no temerle al cambio!!.* http://www.edudevices.com.ar/download/articulos/comentarios/Microcontroladores_de_32_bits_ARM.pdf

Martin Hinner. *ARM Microcontroller HOWTO*. 2007. http://martin.hinner.info/ARM-Microcontroller-HOWTO/ARM-Microcontroller-HOWTO.html

Arquitectura ARM. http://es.wikipedia.org/wiki/Arquitectura_ARM

ARM. The Architecture for the Digital World. HUhttp://www.arm.com/U

Embest IDE para ARM. HUhttp://www.armkits.com/Product/idemain.aspU

Montando un toolchain con crosstool-ng.
HUhttp://menudoproblema.es/blog/entries/2011/03/01/montando-un-toolchain-con-crosstool-ng/U

How To Setup Cross Compile Environment.
HUhttp://biffengineering.com/wiki/index.php?title=HowToSetupCrossCompileEnvironmentU

Vijay Kumar B. *Embedded Programming with de GNU Toolchain*.
HUhttp://www.bravegnu.org/gnu-eprog/U

Debian ARM Linux on Qemu. HUhttp://909ers.apl.washington.edu/~dushaw/ARM/U

Debian on an emulated ARM machine.
HUhttp://www.aurel32.net/info/debian_arm_qemu.phpU

Virtual ARM Linux environment. Testing with Linux on ARM architecture using QEMU.
HUhttps://developer.mozilla.org/en-US/docs/Developer_Guide/Virtual_ARM_Linux_environmentU

Blog Software and Hardware Embedded: http://balau82.wordpress.com/

Definición de QEMU. HUhttp://es.wikipedia.org/wiki/QEMUUH

Virtualización. HUhttp://es.wikipedia.org/wiki/Virtualizaci%C3%B3nUH

jEdit. HUhttp://www.jedit.orgU

Enlaces Manuales

Manual de Embest ARM Development System. Embedded System Development and Labs for ARM

Manual de usuario de la placa emulada:RealView Platform Baseboard for ARM926J-S User Guide

Manual del vector de interrupciones:PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual

Manual de referencia para la manipulación de los Timer:ARM Dual-Timer Module (SP804). Technical Reference Manual

Manual de referencia para la manipulación de las UARTs:PrimeCell UART (PL011) Technical Reference Manual