Sistemas Embebidos: Teoría, Implementación y Metodologías de Diseño

UNIVERSIDAD NACIONAL DE COLOMBIA

Carlos Iván Camargo Bareño

22 de agosto de 2011

INFORME FINAL
Sistemas Embebidos: Teoría, Implementación y Metodologías de Diseño
AUTHOR: C. Camargo
E-MAIL: cicamargoba@unal.edu.co

Published by the Universidad Nacional de Colombia

Índice general

Capítulo 1

Introducción

La industria de los semiconductores ha crecido velozmente durante los últimos años, su campo de acción se ha extendido a casi todas las actividades del ser humano (Entretenimiento, salud, seguridad, transporte, educación, etc); los tiempos de los procesos de diseño son cada vez mas cortos, lo cual require herramientas Hardware, Software y metodologías de diseño que ayuden a cumplir con las exigencias impuestas al sistema.

Esta *invasión* digital ha sido posible gracias a la industria de los semiconductores y a las empresas desarrolladoras de software, las primeras haciendo uso de un altísimo grado de integración ponen a disposición de los diseñadores *Systems On Chip* (SoC) en los cuales se integran procesadores de 32 bits con una gran variedad de periféricos tales como: Controladores de dispositivos de red (cableada e inalámbricos), controladores de video, procesadores aritméticos, controladores de memorias (Flash, SDRAM, DDR, USB, SD), codecs de audio, controladores de touch screen, etc; lo cual permite la implementación de aplicaciones completas dentro de uno de estos SoCs.

Por otro lado, las empresas desarrolladoras de SW crean herramientas de programación que permiten manejar toda la capacidad de estos SoCs. colocando a disposición de los diseñadores: compiladores, simuladores, emuladores, librerías, Sistemas Operativos y drivers. Lo cual permite realizar desarrollos complejos en tiempos cortos.

En la actualidad existe un gran número de sistemas operativos (OS) disponibles tanto comerciales como *open source*. La figura 1.1 muestra la utilización actual de OS comerciales en aplicaciones embebidas; si sumamos los sistemas operativos basados en linux se obtiene un valor del 19.3 % lo cual hace ganador al sistema operativo de libre distribución [1]. La figura 1.2 muestra un cuadro comparativo de la utilización de herramientas comerciales y linux; de nuevo se observa que linux es el preferido por los diseñadores. Este resultado es interesante ya que uno de los supuestos puntos débiles del software de libre distribución es el soporte, lo cual no lo hace tan agradable a la hora de realizar aplicaciones comerciales, sin embargo, esto es solo un mito, ya que gracias a que el código fuente está disponible, es posible comprender perfectamente su funcionamiento, lo cual no sucede con el software comercial; además, existen muchos foros de diseñadores y desarrolladores que se encargan de responder las inquietudes, estos foros almacenan todos los mensajes recibidos e incluyen herramientas de búsqueda para poder consultarlos, por regla general de estos foros, se debe buscar primero en estos archivos históricos, muy seguramente alguien más preguntó lo mismo antes que nosotros.

El caracter gratutito de las herramientas de libre distribución no significa, ni mucho menos, mala calidad, todo lo contrario, existen muchas personas que escudriñan su código fuente en búsqueda de posibles

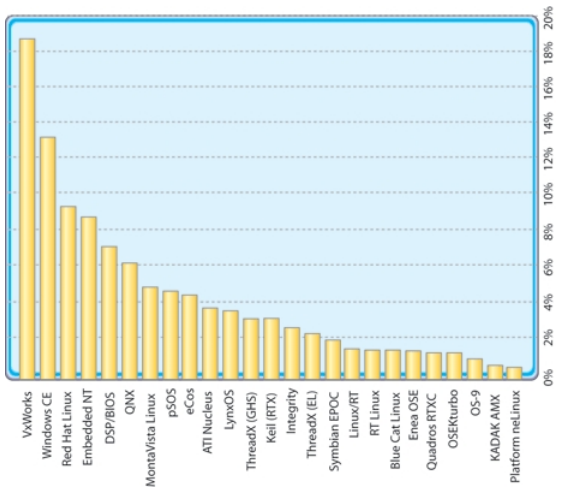

Figura 1.1: Utilización actual de OS para aplicaciones embebidas: Fuente [2].

errores, y realizan cambios con el fín de eliminarlo; por lo tanto, se cuenta con miles de personas que están constantemente depurando y perfeccionando una determinada aplicación; esto no ocurre con el software comercial, normalmente el soporte hay que pagarlo y las perosnas involucradas en el desarrollo no son tantas como las que tienen acceso al código fuente del software libre. Por esta razón empresas como PALM, han dejado de lado productos propietarios como el PALM OS para utilizar linux, SUN Microsystems, liberó el código fuente de su sistema operativo Solaris, ya que no era comparable con linux y está en el proceso de liberar uno de sus procesadores.

De lo anterior se deduce que el mundo de los sistemas digitales ha cambiado en forma considerable, y que en la actualidad existen grandes facilidades para el desarrollo de productos de forma rápida y económica [1]. Desafortunadamente estas tendencias no se han aplicado aún en Colombia; existen varias razones para esto:

1. Desactualización de los programas acedémicos: En muchas de las Universidades de Colombia se utilizan tecnologías obsoletas que impiden la aplicación de metodologías de diseño modernas además de impedir el desarrollo de aplicaciones comerciales, un ejemplo de este tipo de tecnologías es la lógica TTL (74XX) y sus equivalentes en CMOS. Aunque vale la pena aclarar que este tipo de tecnología es útil a la hora de enseñar conceptos básicos, no puede ser utilizada como única herramienta de implementación. Un ejemplo de desactualización de metodologías de diseño lo encontramos en las herramientas de programación para microcontroladores, una gran cantidad de Universidades utilizan lenguaje ensamblador, lo cual impide la re-utilización de código y crea dependencias con el HW utilizado.

2. Falta de interés de la Industria: Un gran porcentaje de la industria Colombiana es consumidora de tecnología, es decir, no generan sus propias soluciones, no cuentan con departamentos de Investigación y Desarrollo; esto se debe a la falta de confianza en los productos nacionales y en algunos casos a la inexistencia de producción nacional. Por otro lado, la cooperación entre la Universidad y la industria es muy reducida, debido a falta de políticas en las Universidades que regulen esta actividad y a la poca inversión por parte de las empresas.

[1] En la actualidad cerca del 60 % de los ingenieros trabajan en compañías con menos de 10 desarrolladores de SW [3]

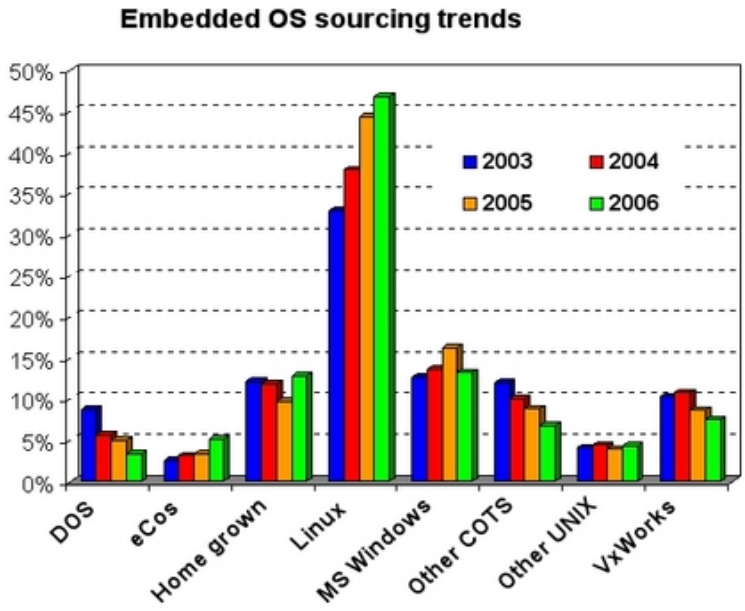

Figura 1.2: Utilización actual de OS para aplicaciones embebidas: Fuente http://www.linuxdevices.com/articles/AT7070519787.html.

3. Políticas del Estado: Casi la totalidad de estos nuevos dispositivos semiconductores deben ser importados, ya que en Colombia no existen distribuidores, por lo tanto, es necesario pagar una serie de impuestos que no desalientan su utilización, estos impuestos están por el orden del 26 % del valor del producto. Por otro lado la apertura económica permite que ingresen productos a bajo precio, con los que no pueden competir los pocos productos desarrollados en el pais, eso sumado a la falta de protección por parte de las políticas estatales condena a los desarrolladores de estos sistemas a la quiebra económica.

Este proyecto resume el trabajo realizado durante los últimos cuatro años en el área de la electrónica digital y más exactamente en el estudio de las metodologías de diseño modernas con aplicaciones comerciales y es presentado para cumplir parcialmente los requisitos de cambio de categoría de profesor asistente a asociado. El presente informe está dividido de la siguiente forma:

- En el capítulo 1 se realiza una breve descripción de los sistemas embebidos, se enumeran sus características, aplicaciones y se hace una descripción de las herramientas HW y SW necesarias para el diseño de los mismos. En los capítulos siguientes se desarrollan casos de estudio encaminados a la comprensión de estos sistemas:
- En el capítulo dos se trabaja con una plataforma comercial de bajo costo: El GameBoy de Nintendo, (gracias a los elevados volúmenes de producción el costo de este dispositivo es bajo alrededor de 40 USD) esta plataforma nos permite desarrollar conceptos básicos como la compilación cruzada, la interfaz HW-SW y los sistemas operativos.
- En el capítulo tres se muestra la implementación de la primera plataforma de desarrollo para sistemas embebidos diseñada en la Universidad Nacional, a pesar de ser muy sencilla permite dar un gran paso en el proceso de fabricación de este tipo de dispositivos.

- En el capítulo cuatro se realiza la implementción de una plataforma de desarrollo que utiliza el sistema operativo linux sobre un arreglo de compuertas programable en campo (FPGA) y se muestran los resultados de una aplicación en el área del Hardware Reconfigurable.
- El capítulo cinco es una guía de adaptación del sistema operativo linux para una arquitectura ARM.

Por último se muestra como algunos de los resultados de este estudio han sido introducidos a los cursos del área de los sistemas digitales en la Universidad Nacional de Colombia y como se ha realizado su difusión en otras Universidades de Colombia.

Capítulo 2

Diseño de Sistemas Embebidos

2.1. Definición

Un Sistema Embebidos es un sistema de propósito específico en el cual, el computador es encapsulado completamente por el dispositivo que el controla. A diferencia de los computadores de propósito general, un Sistema Embebido realiza tareas pre-definidas, lo cual permite su optimización, reduciendo el tamaño y costo del producto [4]

2.2. Características

- Los sistemas embebidos son diseñados para una aplicación específica, es decir, estos sistemas realizan un grupo de funciones previamente definidasm y una vez el sistema es diseñado, no se puede cambiar su funcionalidad. Por ejemplo, el control de un asensor siempre realizará las mismas acciones durante su vida útil.
- Debido a su interacción con el entorno los ES deben cumplir esctríctamente restricciones temporales. El término *Sistemas de Tiempo Real* es utilizado para enfatizar este aspecto.
- Los Sistemas Embebidos son heterogéneos, es decir, están compuestos por componentes Hardware y Software. Los componentes Hardware, como ASICs y Dispositivos Lógicos Programables (PLD) proporcionan la velocidad de ejecución y el cosumo de potencia necesarios en algunas aplicaciones.
- Los Sitemas Embebidos tienen grandes requerimientos en términos de confiabilidad. Errores en aplicaciones como la aviación y el automovilismo, pueden tener consecuencias desastrosas.

2.3. Arquitectura

Una arquitectura típica para un Sistema Embebido se muestra en la Figura 2.1; La cual integra un componente hardware, implementado ya sea en un PLD (CPLD, FPGA) o en un ASIC, conocido con el nombre de periféricos y un componente software (procesador o DSP) capáz de ejecutar software, la parte del procesador está dividida en la CPU (En algunos casos posee una caché) y las unidades de Memoria.

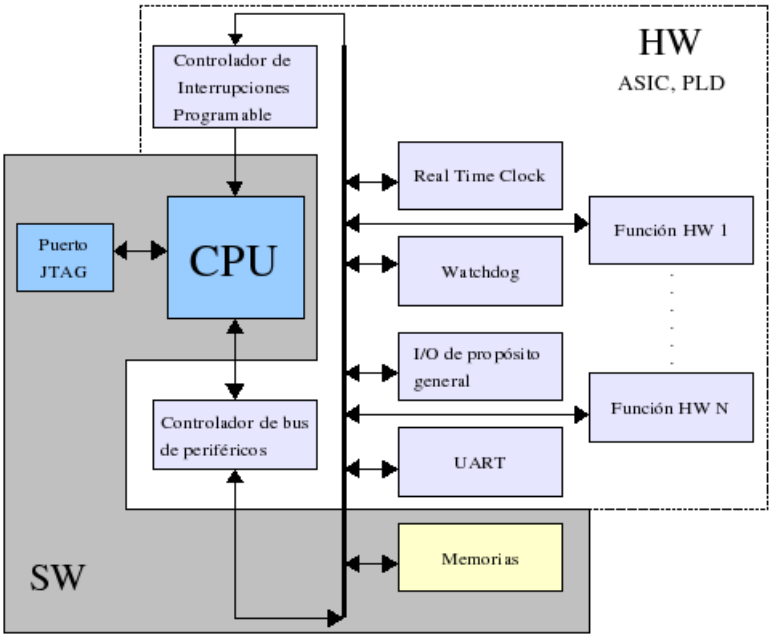

Figura 2.1: Arquitectura de un Sistema Embebido

Podemos utilziar

- Componente HW y SW Integrado en un dispositivo semiconductor (SoC): En la actualidad existen muchas compañías que fabrican procesadores de 32 bits integrados a una gran variedad de periféricos, lo cual simplifica el diseño y reduce costos (menos componentes y menos área de circuito impreso) [1].

- Componente SW en un SoC y componente HW en una FPGA: Cuando no existen en el mercado SoC con la cantidad de periféricos requerida para una determinada aplicación, es necesario recurrir a la utilización de dispositivos comerciales que implementen dicha operación, en algunas ocaciones el periférico puede relizar funciones muy específicas de modo que no existe en el mercado, la solución es entonces implementar estos dispositivos en una FPGA, también se recomienda la utilización de FPGAs en sistemas que requieren una gran cantidad y variedad de periféricos ya que reduce la complejidad y costo del sistema.

- Componente SW y HW en una FPGA: Esta es tal vez la opción más económica y flexible, pero la de menor desempeño, ya que al utilizar los recursos lógicos de la FPGA para la implementación del procesador (softcore) la lngitud de los caminos de interconexión entre los bloques lógicos aumentan el retardo de las señales . Los procesadores *softcore* más populares en la actualidad son:

  - Microblaze de Xilinx[2]
  - Leon de Gaisler Research [3]

[1] http://www.sharpsma.com, http://www.atmel.com, http://www.cirrus.com, http://www.samsung.com, http://www.freescale.com, etc

[2] http://www.xilinx.com

[3] http://www.gaisler.com/

- LatticeMico32 de Lattice Semiconductors[4]
- OpenRisc [5]

2.4. Metodología de Diseño

La Figura 2.2, muestra un diagrama de flujo genérico para diseño para sistemas embebidos [**?**]

El proceso comienza con la *especificación del sistema*, en este punto se describe la funcionalidad y se definen las restricciones físicas, eléctricas y económicas. Esta especificación debe ser muy general y no deben existir dependencias (tecnológicas, metodológicas) de ningún tipo, se suele utilizar lenguajes de alto nivel, como UML, C++. La especificación puede ser verificada a través de una serie de pasos de análisis cuyo objetivo es determinar la validez de los algorítmos seleccionados, por ejemplo, determinar si el algoritmo siempre termina, los resultados satisfacen las especificaciones. Desde el punto de vista de la re-utilización, algunas partes del funcionamiento global deben tomarse de una librería de algorítmos existentes.

Una vez definidas las especificaciones del sistema se debe realizar un modelamiento que permita extraer de estas la funcionalidad. El modelamiento es crucial en el diseño ya que de él depende el paso existoso de la especificación a la implementación. Es importante definir que modelo matemático debe soportar el entorno de diseño. Los modelos más utilizados son: Máquinas de estados finitos, diagramas de flujos de datos, Sistemad de Eventos Discretos y Redes de Petri. Cada modelo posee propiedades matemáticas que pueden explotarse de forma eficiente para responder preguntas sobre la funcionalidad del sistema sin llevar a cabo dispendiosas tareas de verificación. Todo modelo obtenido debe ser verificado para comprobar que cumple con las restricciones del sistema.

Una vez se ha obtenido el modelo del sistema se procede a determinar su *arquitectura*, esto es, el número y tipo de componentes y su inter-conexión. Este paso no es más que una exploración del espacio de diseño en búsqueda de soluciones que permitan la implementación de una funcionalidad dada, y puede realizarse con varios criterios en mente: Costos, confiabilidad, viabilidad comercial.

Utilizando como base la arquitectura obtenida en el paso anterior las tareas del modelo del sistemas son mapeadas dentro de los componentes. Esto es, asignación de funciones a los componentes de la arquitectura. Existen dos opciones a la hora de implementar las tareas o procesos:

1. Implementación Software: La tarea se va a ejecutar en un procesador.
2. Implementación Hardware: La tarea se va a ejecutar en un dispositivo lógico programable.

Para cumplir las especificaciones del sistema algunas tareas deben ser implementadas en Hardware, esto con el fín de no ocupar al procesador en tareas cíclicas, un ejemplo típico de estas tareas es la generación de bases de tiempos. La decisión de que tareas se implementan en SW y que tareas se implementan en HW recibe el nombre de *particionamiento*, esta selección es fuertemente dependiente de restricciones económicas y temporales.

Las tareas Software deben compartir los recursos que existan en el sistema (procesador y memoria), por lo tanto se deben hacer decisiones sobre el orden de ejecución y la prioridad de estas. Este proceso recibe el nombre de *planificación*. En este punto del diseño el modelo debe incluir información sobre el mapeo, el particionamiento y la planificación del sistema.

[4] http://www.latticesemi.com
[5] http://www.opencores.com

Las siguientes fases corresponden a la implementación del modelo, para esto las tareas hardware deben ser llevadas al dispositivo elegido (ASIC o FPGA) y se debe obtener el ″ejecutable″ de las tareas software, este proceso recibe el nombre de *síntesis* HW y SW respectivamente, así mismo se deben sintetizar los mecanismos de comunicación.

El proceso de prototipado consiste en la realización física del sistema, finalmente el sistema físico debe someterse a pruebas para verificar que se cumplen con las especificaciones iniciales.

Como puede verse en el flujo de diseño existen realimentaciones, estas realimentaciones permiten depurar el resultado de pasos anteriores en el caso de no cumplirse las especificaciones iniciales

2.5. Herramientas de Diseño Software

En el mercado existe una gran variedad de herramientas de desarrollo para Sistemas Embebidos, sin embargo, en este estudio nos centraremos en el uso de las herramientas de libre distribución; esta elección se debe a que la mayoría de los productos comerciales utilizan el toolchain de GNU[6] internamente y proporcionan un entorno gráfico para su fácil manejo. Otro factor considerado a la hora de realizar nuestra elección es el económico, ya que la mayoría de los productos comerciales son costosos y poseen soporte limitado. Por otro lado, el toolchain de GNU es utilizado ampliamente en el medio de los diseñadores de sistemas embebidos y se encuentra un gran soporte en múltiples foros de discusión (ver Figura 2.3).

2.5.1. Componentes del *GNU toolchain*

GNU binutils[5]

Son una colección de utilidades para archivos binarios y estan compuestas por:

- **addr2line** Convierte direcciones de un programa en nombres de archivos y números de línea. Dada una dirección y un ejecutable, usa la información de depuración en el ejecutabe para determinar que nombre de atchivo y número de lpinea está asociado con la dirección dada.
- **ar** Esta utilidad crea, modifica y extrae desde ficheros. Un fichero es una colección de otros archivos en una estructura que hace posible obtener los archivos individuales miembros del archivo.
- **as** Utilidad que compila la salida del compilador de C (GCC).
- **c++filt** Este program realiza un mapeo inverso: Decodifica nombres de bajo-nivel en nombres a nivel de usuario, de tal forma que el linker pueda mantener estas funciones sobrecargadas (overloaded) “from clashing”.
- **gasp** GNU Assembler Macro Preprocessor
- **ld** El *linker* GNU combina un número de objetos y ficheros, re-localiza sus datos y los relaciona con referencias. Normalmente el último paso en la construcción de un nuevo programa compilado es el llamado a ld.
- **nm** Realiza un listado de símbolos de archivos tipo objeto.

[6] http://www.gnu.org

- **objcopy** Copia los contenidos de un archivo tipo objeto a otro. *objcopy* utiliza la librería GNU BFD para leer y escribir el archivo tipo objeto. Permite esccribibr el archivo destino en un formato diferente al del archivo fuente.
- **objdump** Despliega información sobre archivos tipo objeto.
- **ranlib** Genera un índice de contenidos de un fichero, y lo almacena en él.
- **readelf** Interpreta encabezados de un archivo ELF.
- **size** Lista el tamaño de las secciones y el tamaño total de un archivo tipo objeto.
- **strings** Imprime las secuencias de caracteres imprimibles de almenos 4 caracteres de longitud.
- **strip** Elimina todos los símbolos de un archivo tipo objeto.

GNU Compiler Collection[4]

El *GNU Compiler Collection* normalmente llamado GCC, es un grupo de compiladores de lenguajes de programación producido por el proyecto GNU. Es el compilador standard para el software libre de los sistemas operativos basados en Unix y algunos propietarios como Mac OS de Apple.

Lenguajes

GCC soporta los siguientes lenguajes:

- **ADA**
- **C**
- **C++**
- **Fortran**
- **Java**
- **Objective-C**
- **Objective-C++**

Arquitecturas

- **Alpha**
- **ARM**
- **Atmel AVR**
- **Blackfin**
- **H8/300**
- **System/370, System/390**

- **IA-32 (x86) and x86-64**
- **IA-64 i.e. the Ïtanium"**
- **Motorola 68000**
- **Motorola 88000**
- **MIPS**
- **PA-RISC**
- **PDP-11**
- **PowerPC**
- **SuperH**
- **SPARC**
- **VAX**
- **Renesas R8C/M16C/M32C**
- **MorphoSys**

Como puede verse GCC soporta una gran cantidad de lenguajes de programación, sin embargo, en el presente estudio solo lo utilizaremos como herramienta de compilación para C y C++. Una característica de resaltar de GCC es la gran cantidad de plataformas que soporta, esto lo hace una herramienta Universal para el desarrollo de sistemas embebidos, el código escrito en una plataforma (en un lenguaje de alto nivel) puede ser implementado en otra sin mayores cambios, esto elimina la dependencia entre el código fuente y el HW[7], lo cual no ocurre al utilizar lenguaje ensamblador.

Por otro lado, el tiempo requerido para realizar aplicaciones utilizando C o C++ disminuye, ya que no es necesario aprender las instrucciones en assembler de una plataforma determinada; además, la disponibilidad de librerías de múltiples propósitos reduce aún más los tiempos de desarrollo, permitiendo de esta forma tener bajos tiempos *time to market* y reducir de forma considerable el costo del desarrollo. Una consecuencia de esto se refleja en el número de desarrolladores en un grupo de trabajo, en la actualidad casi el 60 % de las empresas desarrolladoras de dispositivos embebidos tiene grupos con menos de 10 desarrolladores 2.4.

GNU Debugger

El depurador oficial de GNU (GDB), es un depurador que al igual que GCC tiene soporte para múltiples lenguajes y plataformas. GDB permite al usuario monitorear y modificar las variables internas del programa y hacer llamado a funciones de forma independiente a la ejecución normal del mismo. Además, permite establecer sesiones remotas utilizando el puerto serie o TCP/IP. Aunque GDB no posee una interfaz gráfica, se han desarrollado varios front-ends como DDD o GDB/Insight. A continuación se muestra un ejemplo de una sesión con gdb.

[7] Esto recibe el nombre de re-utilización de código

```
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) run
Starting program: /home/sam/programming/crash
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xc11000
This program will demonstrate gdb

Program received signal SIGSEGV, Segmentation fault.
0x08048428 in function_2 (x=24) at crash.c:22
22          return *y;
(gdb) edit
(gdb) shell gcc crash.c -o crash -gstabs+
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
warning: cannot close "shared object read from target memory": File in wrong format
`/home/sam/programming/crash' has changed; re-reading symbols.
Starting program: /home/sam/programming/crash
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xa3e000
This program will demonstrate gdb
24
Program exited normally.
(gdb) quit
```

C Libraries

Adicionalmente es necesario contar con una librería que proporcione las librerías standard de C: stdio, stdlib, math; las más utilizadas en sistemas embebidos son:

- **glibc**[8] Es la librería C oficial del proyecto GNU. Uno de los inconvenientes al trabajar con esta librería en sistemas embebidos es que genera ejecutables de mayor tamaño que los generados a partir de otras librerías, lo cual no la hace muy atractiva para este tipo de aplicaciones.
- **uClibc**[9] Es una librería diseñada especialmente para sistemas embebidos, es mucho más pequeña que **glibc**.
- **newlib**[10] Al igual que **uClibc**, está diseñada para sistemas embebidos. El típico "Hello, world!" ocupa menos de 30k en un entorno basado en newlib, mientras que en uno basado en glibc, puede ocupar 380k [6].
- **diet libc**[11] Es una versión de *libc* optimizada en tamaño, puede ser utilizada para crear ejecutables estáticamente enlazados para linux en plataformas alpha, arm, hppa, ia64, i386, mips, s390, sparc, sparc64, ppc y x86_64.

[8] http://www.gnu.org/software/libc/
[9] http://uclibc.org/
[10] http://sources.redhat.com/newlib/
[11] http://www.fefe.de/dietlibc/

2.6. Desarrollo Software

El primer paso en nuestro estudio consiste en tener una cadena de herramientas funcional que soporte la familia de procesadores a utilizar. La arquitectura sobre la cual realizaremos nuestro estudio inicial es la ARM (Advanced Risc Machines), ya que la más utilizada en la actualidad por los diseñadores de sistemas embebidos (ver figura 2.5) y se encuentran disponibles una gran variedad de herramientas para esta arquitectura. Sin embargo, lo contenido en esta sección es aplicable a cualquier familia de procesadores soportada por la cadena de herrmientas GNU. Existen dos formas de obtener la cadena de herramientas GNU:

1. Utilizar una distribución precompilada: Esta es la via más rápida, sin embargo, hay que tener cuidado al momento de instalarlas, ya que debe hacerse en un directorio con el mismo *path* con el que fueron creadas. por ejemplo */usr/local/gnutools*; si esto no se cumple, las herramientas no funcionarán de forma adecuada.

2. Utilizar un script de compilación: Existen disponibles en la red una serie de *scripts* que permiten descargar, configurar, compilar e instalar la cadena de herramientas, la ventaja de utilizar este método es que es posible elegir las versiones de las herramientas instaladas, al igual que el directorio de instalación. En este estudio utilizaremos los *scripts* creados por Dan Kegel [8].

2.6.1. Flujo de diseño software

En la figura 2.6 se ilustra la secuencia de pasos que se realizan desde la creación de un archivo de texto que posee el código fuente que implementa la funcionalidad de una tarea software utilizando un lenguaje de alto nivel como C o C++, hasta su programación en una memoria permanente en la plataforma física. Los pasos necesarios para crear un archivo que pueda ser programado en dicha memoria son:

1. **Escritura del código fuente:** Creación del código fuente en cualquier editor de texto utilizando un lenguaje de alto nivel como C o C++.

2. **Compilación:** Utilizando un compilador (GCC en nuestro caso) se crea un *objeto* que contiene las instrucciones en *lenguaje de máquina* del procesador a utilizar (uno diferente al que realiza la compilación que normalmente es de la familia x86); en este punto el compilador solo busca en los encabezados (*headers*) la definición de una determinada función, esto es, la forma en que debe ser utilizada, el tipo de datos y el número de parámetros con que debe ser invocada, por ejemplo, la función *printf* esta declarada en el archivo *stdio.h* como: *int printf (const char *template, ...)*. Esta declaración es utilizada por el compilador para verificar el correcto uso de esta función.

3. **Enlazado:** En esta etapa se realizan dos tareas:

   a) Se enlazan los archivos tipo objeto del proyecto, junto con librerías precompiladas para el procesador de la plataforma, si una determinada función no es definida en ninguna de estas librerías, el *enlazador* generará un error y no se generará el ejecutable.

   b) Se definen las posiciónes físicas de las secciones que componen el archivo ejecutable (tipo ELF), esto se realiza a través de un link de enlazado en el que se define de forma explícita su localización.

4. **Extracción del archivo de programación** En algunas aplicaciones (cuando no se cuenta con un sistema operativo) es necesario extraer las secciones del ejecutable que residen en los medios de

almacenamiento no volátil, que como veremos más adelante representan el conjunto de instrucciones que debe ejecutar el procesador de la plataforma y las constantes del programa. Esto se realiza con la herramiento *objcopy*, la cual, nos permite generar archivos en la mayoría de los formatos soportados por los programadores de memorias y procesadores, como S19 e Intel Hex.

5. **Descarga del programa a la plataforma**. Dependiendo de la plataforma existen varios métodos para descargar el archivo de programación a la memoria de la plataforma:

   a) Utilizando un *loader*: El *loader* es una aplicación que reside en un medio de almacenamiento no volátil y permite la descarga de archivos utilizando el puerto serie, USB, o una interfaz de red.

   b) Utilizando el puerto JTAG: El puerto JTAG (Joint Test Action Group) proporciona una interfaz capaz de controlar los registros internos del procesador, y de esta forma, acceder a las memorias de la plataforma y ejecutar un programa residente en una determinada posición de memoria.

6. **Depuración** Una vez se descarga la aplicación a la plataforma es necesario someterla a una serie de pruebas con el fín de probar su correcto funcionamiento. Esto se puede realizar con el depurador GNU (GDB) y una interfaz de comunicación que puede ser un puerto serie, USB o un adaptador de red.

2.6.2. El formato ELF

El formato ELF (*Executable and Linkable Format*) Es un stándard para objetos, librerías y ejecutables y es el formato que genera las herramientas GNU. Como puede verse en la figura 2.7 está compuesto por varias secciones (*link view*) o segmentos (*execution view*). Si un programador está interesado en obtener información de secciones sobre tablas de símbolos, código ejecutable específico o información de enlazado dinámico debe utilizar *link view*. Pero si busca información sobre segmentos, como por ejemplo, la localización de los segmentos *text* o *data* debe utilizar *execution view*. El encabezado describe el layout del archivo, proporcionando información de la forma de acceder a las secciones [9].

Las secciones pueden almacenar código ejecutable, datos, información de enlazado dinámico, datos de depuración, tablas de símbolos,comentarios, tablas de strings, y notas. Las secciones más importantes son las siguientes:

- **.bss** Datos no inicializados. (RAM)
- **.comment** Información de la versión.
- **.data y .data1** Datos inicializados. (RAM)
- **.debug** Información para depuración simbólica.
- **.dynamic** Información sobre enlace dinámico
- **.dynstr** Strings necesarios para el enlacedinámico
- **.dynsym** Tabla de símbolos utilizada para enlace dinámico.
- **.fini** Código de terminación de proceso.
- **.init** Código de inicialización de proceso.
- **.line** Información de número de línea para depuración simbólica.

- **.rodata y .rodta1** Datos de solo-lectura (ROM)
- **.shstrtab** Nombres de secciones.
- **.symtab** Tabla de símbolos.
- **.text** Instrucciones ejecutables (ROM)

Para aclarar un poco este concepto, escribamos una aplicación sencilla:

```
#include <stdio.h>

int global;
int global_1 = 1;

int main(void)
{
  int i;                                  // Variable no inicializada
  int j = 2;                              // Variable inicializada
  for(i=0; i<10; i++){
    printf("Printing %d\n", i*j);         // Caracteres constantes
    j = j + 1;
    global   = i;
    global_1 = i+j;
  }
  return 0;
}
```

Generemos el objeto compilándolo con el siguiente comando: *arm-none-eabi-gcc -c hello.c*

Examinemos que tipo de secciones tiene este ejecutable *arm-none-eabi-readelf -S hello.o*

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00009c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000484 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 0000d0 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 0000d4 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        00000000 0000d4 000010 00   A  0   0  4
  [ 6] .comment          PROGBITS        00000000 0000e4 00004d 00      0   0  1
  [ 7] .ARM.attributes   ARM_ATTRIBUTES  00000000 000131 00002e 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 00015f 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 000368 0000f0 10     10  11  4
  [10] .strtab           STRTAB          00000000 000458 00002b 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

La sección *.text*, como se dijo anteriormente contiene las instrucciones ejecutables, por esta razón se marca como ejecutable *"X"* en la columna *Flg*. Es posible ver las instrucciones que se ejecutan en esta sección:

arm-none-eabi-objdump -d -j .text hello.o

```
00000000 <main>:
   0:   e92d4800        stmdb   sp!, {fp, lr}
   4:   e28db004        add     fp, sp, #4      ; 0x4
   8:   e24dd008        sub     sp, sp, #8      ; 0x8
   c:   e3a03002        mov     r3, #2  ; 0x2
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e3a03000        mov     r3, #0  ; 0x0
  18:   e50b300c        str     r3, [fp, #-12]
  1c:   ea000013        b       70 <main+0x70>
  20:   e51b200c        ldr     r2, [fp, #-12]
  24:   e51b3008        ldr     r3, [fp, #-8]
```

```
  28:   e0030392        mul     r3, r2, r3
  2c:   e59f005c        ldr     r0, [pc, #92]   ; 90 <.text+0x90>
  30:   e1a01003        mov     r1, r3
  34:   ebfffffe        bl      0 <printf>
  38:   e51b3008        ldr     r3, [fp, #-8]
  3c:   e2833001        add     r3, r3, #1      ; 0x1
  40:   e50b3008        str     r3, [fp, #-8]
  44:   e59f2048        ldr     r2, [pc, #72]   ; 94 <.text+0x94>
  48:   e51b300c        ldr     r3, [fp, #-12]
  4c:   e5823000        str     r3, [r2]
  50:   e51b200c        ldr     r2, [fp, #-12]
  54:   e51b3008        ldr     r3, [fp, #-8]
  58:   e0822003        add     r2, r2, r3
  5c:   e59f3034        ldr     r3, [pc, #52]   ; 98 <.text+0x98>
  60:   e5832000        str     r2, [r3]
  64:   e51b300c        ldr     r3, [fp, #-12]
  68:   e2833001        add     r3, r3, #1      ; 0x1
  6c:   e50b300c        str     r3, [fp, #-12]
  70:   e51b300c        ldr     r3, [fp, #-12]
  74:   e3530009        cmp     r3, #9  ; 0x9
  78:   daffffe8        ble     20 <main+0x20>
  7c:   e3a03000        mov     r3, #0  ; 0x0
  80:   e1a00003        mov     r0, r3
  84:   e24bd004        sub     sp, fp, #4      ; 0x4
  88:   e8bd4800        ldmia   sp!, {fp, lr}
  8c:   e12fff1e        bx      lr
```

La sección *.data* mantiene las variables inicializadas, y contiene:

arm-none-eabi-objdump -d -j .data hello.o

```
00000000 <global_1>:
   0:   01 00 00 00
```

Como vemos, la sección *.data* contiene únicamente el valor de inicialización de la variable *global_1* (1) y no muestra informacióón acerca de la variable *j*, esto se debe a que la información está en el *stack* del proceso. Si observamos el contenido de la sección *.text* observamos que esta variable es asignada en tiempo de ejecución, en la línea *0c:* se ve la asignación de esta variable.

```
0c:   e3a03002        mov     r3, #2  ; 0x2
10:   e50b3008        str     r3, [fp, #-8]
```

La sección *.bss* mantiene la información de las variables no incializadas (En Linux todas las variables no inicializadas, se inicializadan en cero):

arm-none-eabi-objdump -d -j .bss hello

```
000145c4 <global>:
    145c4:      00000000
```

La sección *.rodata* mantiene los datos que no cambian durante la ejecución del programa, es decir, de solo lectura, si examinamos esta sección obtenemos:

hexdump -C hello.o — grep -i 000000d0 (la sección *.rodata* comienza en la posición de memoria 0xd4)

```
000000d0  01 00 00 00 50 72 69 6e  74 69 6e 67 20 25 64 0a  |....Printing %d.|
000000e0  00 00 00 00 00 47 43 43  3a 20 28 43 6f 64 65 53  |.....GCC: (CodeS|
```

En el contenido de esta sección aparece la cadena de caracteres *Printing %d \n*, es decir, los datos que no cambian durante la ejecución.

2.6.3. Herramienta de compilación make

Como pudo verse en la sección es necesario realizar una serie de pasos para poder descargar una aplicación a una plataforma embebida. Debido a que las herramientas GNU solo poseen entrada por consola, es necesario esribir una serie de comandos cada vez que se realiza un cambio en el código fuente, lo cual resulta poco práctico. Para realizar este proceso de forma automática se creó la herramienta *make*, la cual recibe como entrada un archivo que normalmente tiene el nombre *Makefile* o *makefile* y determina que archivos han sido modificados desde la última compilación y ejecuta los comandos necesarios para recompilarlos. Un ejemplo de este tipo de archivo se muestra a continuación:

```
1  SHELL = /bin/sh
2
3  basetoolsdir = /home/at91/gcc-3.4.5-glibc-2.3.6/arm-softfloat-linux-gnu
4  bindir   = ${basetoolsdir}/bin
5  libdir   = ${basetoolsdir}/lib/gcc/arm-softfloat-linux-gnu/3.4.5
6
7  CC       = arm-softfloat-linux-gnu-gcc
8  AS       = arm-softfloat-linux-gnu-as
9  LD       = arm-softfloat-linux-gnu-ld
10 OBJCOPY = arm-softfloat-linux-gnu-objcopy
11
12 CFLAGS  =-mcpu=arm920t -I. -Wall
13 LDFLAGS =-L${libdir} -l gcc
14
15 OBJS = \
16   main.o       \
17   debug_io.o    \
18   at91rm9200_lowlevel.o   \
19   p_string.o
20
21 ASFILES = arm_init.o
22
23 LIBS=${libdir}/
24
25 all: hello_world
26
27 hello_world: ${OBJS} ${ASFILES} ${LIBS}
28   ${LD} -e 0 -o hello_world.elf -T linker.cfg ${ASFILES} ${OBJS} ${LDFLAGS}
29   ${OBJCOPY} -O binary hello_world.elf hello_world.bin
30
31 clean:
32   rm -f *.o *~ hello_world.*
33
34 PREPROCESS.c = $(CC) $(CPPFLAGS) $(TARGET_ARCH) -E -Wp,-C,-dD,-dI
35
36 %.pp : %.c FORCE
37   $(PREPROCESS.c) $< > $@
```

En las líneas 3-5 se definen algunas variables globales que serán utilizadas a lo largo del archivo; en las líneas 7 - 10 se definen las herramientas de compilación a utilizar, específicamente el compilador C (CC), el ensamblador (AS), el enlazador (LD) y la utilidad objcopy. A partir de la línea 15 se definen los objetos que forman parte del proyecto, en este caso: *main.o, debug_io.o, at91rm9200_lowlevel.o y p_string.o*; en la línea 21 se definen los archivos en assembler que contiene el proyecto, para este ejemplo *arm_init.o*. Las líneas 12 y 13 definen dos variables especiales que se pasan directamente al compilador C (CFLAGS) y al enlazador (LDFLAGS) y definen parámetros de comportamiento de estas herramientas.

```
CFLAGS  =-mcpu=arm920t -I. -Wall
```

El parámetro *-mcpu* le indica al compilador C para arquitecturas *ARM* que utilice la familia *arm920*; el parámetro *-I* le indica un directorio donde puede buscar los encabezados, en este caso el caracter "."le indica que busque en el mismo sitio donde se encuentran los archivos fuente; el parámetro *-Wall* le indica que imprima todos los mensajes de errores y advertencias.

```
LDFLAGS =-L${libdir} -l gcc
```

El parámetro *-L* le indica al enlazador la ruta del directorio donde se encuentran las librerías, en este ejemplo apunta a la variable textitlibdir que se encuentra declarado como *$basetoolsdir/lib/gcc/arm-softfloat-linux-gnu/3.4.5*; el parámetro *-l* le indica al enlazador que debe utilizar la librería *gcc* que se encuentra en el directorio definido previamente. En realidad el archivo de la librería tienen el nombre *libgcc.a*, pero como todas las librerías tienen el nombre *libXXXX.a* se eliminan el encabezado y la extensión del archivo.

En las líneas 25, 27 y 31 aparecen unas etiquetas de la forma: *nombre:* estos labels permiten ejecutar de forma independiente el conjunto de instrucciones asociadas a ellas, por ejemplo, si se ejecuta:
make clean

make ejecutará el comando:

*rm -f *.o * hello_world.**

Observemos los comandos asociados a la etiqueta *hello_world:* En la misma línea aparecen *$OBJS $ASFILES $LIBS* esto le indica a la herramienta *make* que antes de ejecutar los comandos asociados a este label, debe realizar las acciones necesarias para generar *$OBJS $ASFILES $LIBS* o lo que es lo mismo: *main.o, debug_io.o, at91rm9200_lowlevel.o, p_string.o, arm_init.o y libgcc.a*. Para esto, *make* tiene predefinidas una serie de reglas para compilar los archivos .c la regla es de la forma:

```
.c.o:
        $(CC) $(CFLAGS) -c $<
.c:
        $(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@
```

Lo cual le indica a la herramienta make que para generar un archivo *.o* a partir de uno *.c* es necesario ejecutar *$(CC) $(CFLAGS) -c $¡*; de aqui la importancia de definir bien la variable de entorno *CC* cuando trabajamos con compiladores cruzados[12]. Hasta este punto al ejecutar el comando: *make hello_world*, *make* realizaría las siguientes operaciones:

```
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o main.o main.c
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o debug_io.o debug_io.c
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o at91rm9200_lowlevel.o at91rm9200_lowlevel.c
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o p_string.o p_string.c
arm-softfloat-linux-gnu-as    -o arm_init.o arm_init.s
```

En las líneas 28 se realiza el proceso de enlazado; al *enlazador* se le pasan los parámetros:

- **-e 0**: Punto de entrada , utiliza la dirección de memoria 0 como punto de entrada.
- **-o hello_world.elf**: Nombre del archivo de salida *hello_world*
- **-T linker.cfg**: Utilice el archivo de enlace *linker.cfg* para definir las posiciones de memoria de las secciones del ejecutable.
- **$ASFILES $OBJS $LDFLAGS**: Lista de objetos y librerías para crear el ejecutable.

En la línea 29 se utiliza la herramienta *objcopy* para generar un archivo binario (*-O binary*) con la información necesaria para cargar en una memoria no volátil.

[12]Un compilador cruzado genera código para una plataforma diferente en la que se está ejecutando, por ejemplo, genera ejecutables para ARM pero se ejecuta en un x86

2.6.4. Archivo de enlace

Como vimos anteriormente, el enlazador o *linker* es el encargado de agrupar todos los archivos objeto *.o*, y las librerías necesarias para crear el ejecutable, adicionalmente, permite definir donde serán ubicados los diferentes segmentos del archivo ELF en un archivo de enlace *linker script*. De esta forma podemos ajustar el ejecutable a plataformas con diferentes configuraciones de memoria, lo que proporciona un grado mayor de flexibilidaad de la cadena de herramientas GNU. Cuando se dispone de un sistema operativo como Linux no es necesario definir este archivo, ya que el sistema operativo se encarga del manejo de las diferentes secciones, sin embargo, es necesario tenerlo presente ya que como veremos más adelante existe un momento en el que el sistema operativo no ha sido cargado en la plataforma y las aplicaciones que se ejecuten deben proporcionar esta información. A continuación se muestra un ejemplo de este archivo:

```
 /* identify the Entry Point  (_vec_reset is defined in file crt.s)  */
ENTRY(_vec_reset)

/* specify the memory areas  */
MEMORY
{
    flash : ORIGIN = 0,          LENGTH = 256K    /* FLASH EPROM          */
    ram   : ORIGIN = 0x00200000, LENGTH = 64K     /* static RAM area      */
}

/* define a global symbol _stack_end */
_stack_end = 0x20FFFC;

/* now define the output sections  */
SECTIONS
{
  . = 0;            /* set location counter to address zero  */
  .text :           /* collect all sections that should go into FLASH after startup  */
  {
    *(.text)        /* all .text sections (code)  */
    *(.rodata)      /* all .rodata sections (constants, strings, etc.)  */
    *(.rodata*)     /* all .rodata* sections (constants, strings, etc.)  */
    *(.glue_7)      /* all .glue_7 sections  (no idea what these are) */
    *(.glue_7t)     /* all .glue_7t sections (no idea what these are) */
    _etext = .;     /* define a global symbol _etext just after the last code byte */
  } >flash          /* put all the above into FLASH */

  .data :           /* collect all initialized .data sections that go into RAM  */
  {
    _data = .;      /* create a global symbol marking the start of the .data section  */
    *(.data)        /* all .data sections  */
    _edata = .;     /* define a global symbol marking the end of the .data section  */
  } >ram AT >flash  /* put all the above into RAM (but load the LMA initializer copy into FLASH)  */

  .bss :            /* collect all uninitialized .bss sections that go into RAM  */
  {
    _bss_start = .; /* define a global symbol marking the start of the .bss section */
    *(.bss)         /* all .bss sections  */
  } >ram            /* put all the above in RAM (it will be cleared in the startup code */
  . = ALIGN(4);     /* advance location counter to the next 32-bit boundary */
  _bss_end = . ;    /* define a global symbol marking the end of the .bss section */
}
_end = .;           /* define a global symbol marking the end of application RAM */
```

En las primeras líneas del archivo aparece la declaración de las memorias de la plataforma, en este ejemplo tenemos una memoria RAM de 64kB que comienza en la posición de memoria 0x00200000 y una memoria flash de 256k que comienza en la posición 0x0. A continuacion se definen las secciones y el lugar donde serán almacenadas; En este caso, las secciones *.text* (código ejecutable) y *.rodata* (datos de solo lectura) se almacenan en una memoria no volátil la flash. Cuando el sistema sea energizado el procesador ejecutará el código almacenado en su memoria no volátil. Las secciones *.data* (variables inicializadas) y *.bss* (variables no inicializadas) se almacenarán en la memoria volátil RAM, ya que el acceso a las memorias no volátiles son más lentas y tienen ciclos de lectura/escritura finitos.

2.7. Herramientas hardware

En esta subsección se realizará una breve descripción de los dispositivos semiconductores más utilizados para la implementación de dispositivos digitales, esto, con el fín de determinar el estado actual de la industria de los semiconductores y entender los componentes básicos con los que se pueden implementar dispositivos digitlaes modernos.

2.7.1. SoC

La Figura 2.8 muestra la arquitectura de un SoC actual, específicamente del AT91RM920 de Atmel. En este diagrama podemos observar el núcleo central un procesador ARM920T de 180MHz y los periféricos asociados a él. En la actualidad podemos encontrar una gran variedad de SoC diseñados para diferentes aplicaciones: Multimedia, Comunicaciones, Asistentes Digitales; los periféricos incluidos en cada SoC buscan minimizar el número de componentes externos, y de esta forma reducir los costos. Este SoC en particular fué uno de los primeros que diseño ATMEL y está enfocado a tareas en las que se requiere una conexión de red. La arquitectura de estos SoC evoluciona muy rápido acomodándose a los requerimientos de nuevas aplicaciones, básicamente, los cambios se producen en la velocidad del procesador central y la adición de periféricos que permiten el control directo de nuevos periféricos, como por ejemplo, la adición de controladores de pantallas de cristal líquido o salidas de video. Dentro de estos periféricos encontramos:

- Controlador para memorias: NAND flash, DataFlash, SDRAM, DDR, SD/MMC
- Puertos USB host o device.
- Puerto I2C
- Interfaz Ethernet 10/100.
- Interfaz high speed USB 2.0
- Puertos SPI.
- Puertos seriales (RS232).
- Soporte JTAG.
- Interfáz de Bus externo (EBI).
- Controlador de LCD.
- Driver de video.
- Tarjetas de sonido.

Existen una serie de periféricos que son indispensables en todo Sistema Embebido, los cuales facilitan la programación de aplicaciones y la depuración de las mismas. Uno de los más importantes y que están presentes en todos los SoC actuales es el controlador de memorias, este periférico permite controlar los medios de almecenamiento que son vitales para la correcta oepración del dispositivo, a continuación se hace un breve recuento de las memorias disponibles para el desarrollo de aplicaciones embebidas.

2.7.2. Memorias Volátiles

Como se estudió anteriormente existen secciones de un ejecutable que deben ser almacenadas en memorias volátiles o en memorias no volátiles. Debido a esto, la mayoría de los SoC incluyen periféricos dedicados a controlar diferentes tipos de memoria, las memorias volátiles son utilizadas como memoria de acceso aleatorio (RAM) gracias a su bajo tiempo de accesso y al ilimitado número de ciclos de lectura/escritura.

memoria SDRAM

El tipo de memoria más utilizado en los sistemas embebidos actuales es la memoria SDRAM; la cual está organizada como una matriz de celdas, con un número de bits dedicados al direccionamiento de las filas y un número dedicado a direccionar columnas (ver Figura 2.9).

Una celda de memoria SDRAM esta compuesta por un transistor y un condensador; el transistor suministra la carga y el condensador almacena el estado de cada celda, esta carga en el condensador desaparece con el tiempo, razón por la cual es necesario recargar estos condensadores periódicamente, este proceso recibe el nombre de *Refresco*. Un ciclo de refresco es un ciclo especial en el que no se escribe ni se lee información, solo se recargan los condensadores para mantener la información.

Un ejemplo simplificado de una operación de lectura es el siguiente: Una posición de memoria se determina colocando la dirección de la fila y la de la columna en las líneas de dirección de fila y columna respectivamente, un tiempo después el dato almacenado aparecerá en el bus de datos. El procesador coloca la dirección de la fila en el bus de direcciones y después activa la señal *RAS* (Row Access Strobe). Después de un retardo de tiempo predeterminado para permitir que el circuito de la SDRAM capture la dirección de la fila, el procesador coloca la dirección de la columna en el bus de direcciones y activa la señal *CAS* (Column Access Strobe). El periférico que controla la SDRAM está encargado de garantizar los ciclos de refresco de acuerdo con los requerimientos de la SDRAM [**?**].

2.7.3. Memorias No Volátiles

La memorias no volátiles almacenan por largos períodos de tiempo la información necesaria para la operación de un Sistema Embebido, pueden ser vistos como discos duros de estado sólido. Existen dos tipos de memoria las NOR y las NAND; las dos poseen la capacidad de ser escritas y borradas utilizando control de software, con lo que no es necesario utilizar programadores externos y pueden ser modificadas una vez sean instaladas en el circuito impreso. Una desventaja de estas memorias es que los tiempos de escritura y borrado son muy largos en comparación con los de las memorias RAM y que presentan un número finito de ciclos de borrado y exritura.

Memorias NOR

Las memorias NOR poseen buses de datos y dirección, con lo que es posible acceder de forma fácil a cada byte almacenado en ella. Los bits almacenados pueden ser cambiados de 0 a 1 utilizando el control de software un byte a la vez, sin embargo, para cambiar un bit de 1 a 0 es necesario borrar una serie de unidades de borrado que reciben el nombre de bloques, lo que reduce el tiempo de borrado de la memoria. Debido a que el borrado y escritura de una memoria ROM se puede realizar utilizando el control software (ver Figura 2.10) no es necesario contar con un periférico especializado para su manejo. Las memorias

NOR son utilizadas en aplicaciones donde se necesiten altas velocidades de lectura y baja densidad, debido a que los tiempos de escritura y lectura son muy grandes, se utilizan como memorias ROM.

Memorias NAND

Las memorias NAND disminuyen los tiempos de escritura y aumentan la capacidad de almacenamiento, ideales para aplicaciones donde se requiera almacenamiento de información. Adicionalmente las memorias NAND consumen menos potencia que las memorias NAND, por esta razón este tipo de memorias son utilizadas en casi todos los dispositivos de almacenamiento modernos como las memorias SD y las memorias USB, los cuales integran una memoria NAND con un circuito encargado de controlarlas e implementar el protocolo de comunicación. A diferencia de las flash tipo NOR, los dispositivos NAND se acceden de forma serial utilizando interfaces complejas; su operación se asemeja a un disco duro tradicional. Se accede a la información utilizando bloques (más pequeños que los bloques NOR). Los ciclos de escritura de las flash NAND son mayores en un orden de magnitud que los de las memorias NOR.

Un problema al momento de trabajar con las memorias tipo NAND es que requieren el uso de un *manejo de bloques defectuosos*, esto es necesario ya que las celdas de memoria pueden dañarse de forma espontánea durante la operación normal. Debido a esto se debe tener un determinado número de bloques que se encargen de almacenar tablas de mapeo para manejar los bloques defectuosos; o puede hacerse un chequeo en cada inicialización del sistema de toda la RAM para actualizar esta lista de sectores defectuosos. El algoritmo de ECC (Error-Correcting Code) debe ser capáz de corregir errores tan pequeños como un bit de cada 2048 bits, hasta 22 bits de cada 2048. Este algorítmo es capaz de detectar bloques defectuosos en la fase de programacíón comparando la información almacenada con la que debe ser almacenada (verificación), si encuentra un error marca el bloque como defectuoso y utiliza un bloque sin defactos para almacenar la información.

La tabla 2.1 resume las principales características de los diferentes tipos de memoria flash y muestra sus aplicaciones típicas.

	SLC NAND	**MLC NAND**	MLC NOR
Densidad	512Mbits - 4GBits	1Gbits - 16GBits	16MBits - 1GBit
Velocidad de Lectura	24MB/s	18.6MB/s	103MB/s
Velocidad de escritura	8 MB/s	2.4MB/s	0,47MB/s
Tiempo de borrado	2ms	2ms	900ms
Interfaz	Acceso Indirecto	Acceso Indirecto	Acceso Aleatorio
Aplicación	Almacenamiento	Almacenamiento	Solo lectura

Cuadro 2.1: Cuadro de comparación de las memorias flash NAND y NOR

Adicionalmente, se encuentran dispoibles las memorias DATAFLASH, estos dispositivos son básicamente una memoria flash tipo NOR con una interfaz SPI, permite una velocidad de lectura de hasta 66MHz utilizando solamente 4 pines para la comunicación con el procesador.

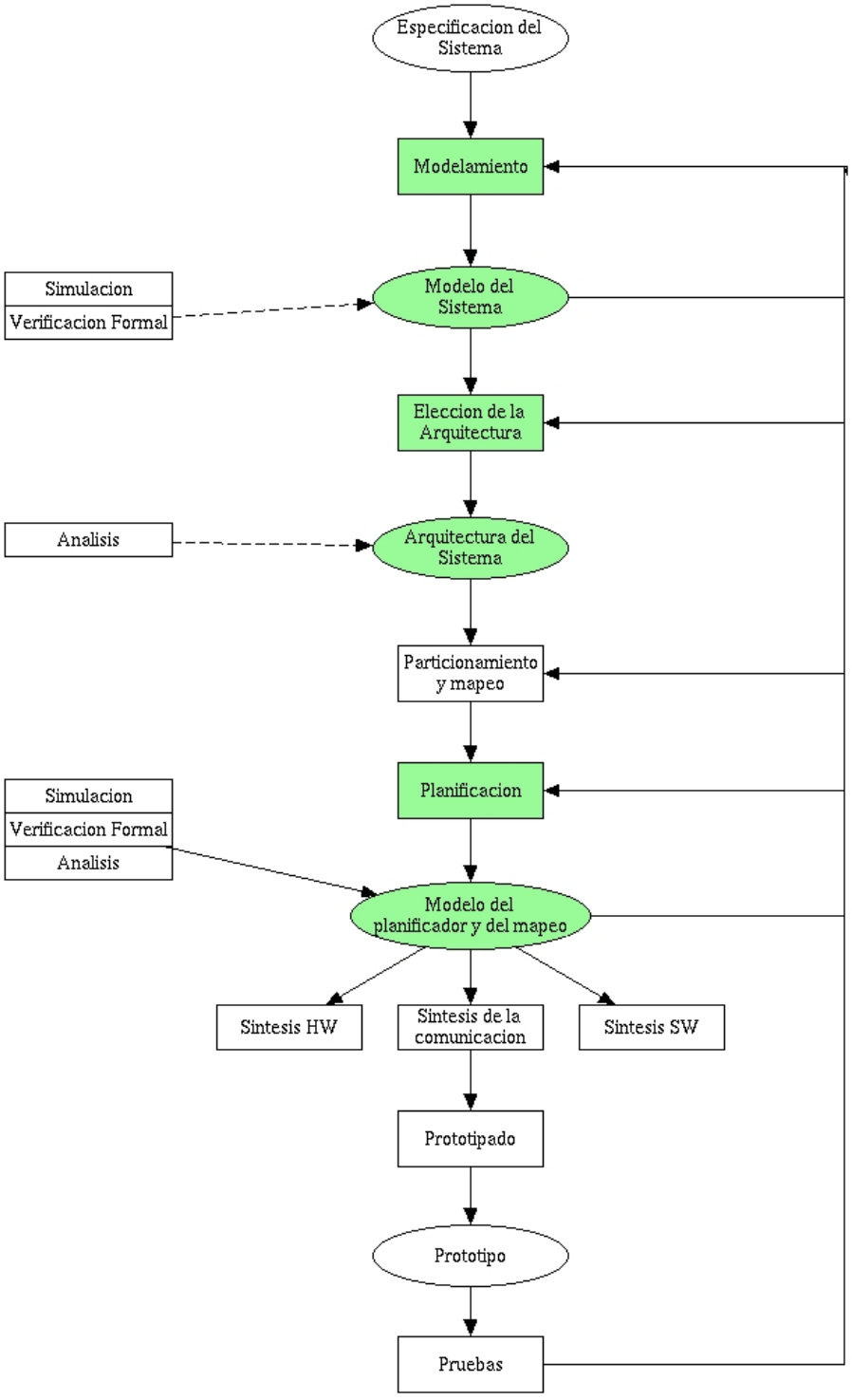

Figura 2.2: Flujo de Diseño de un Sistema Embebido

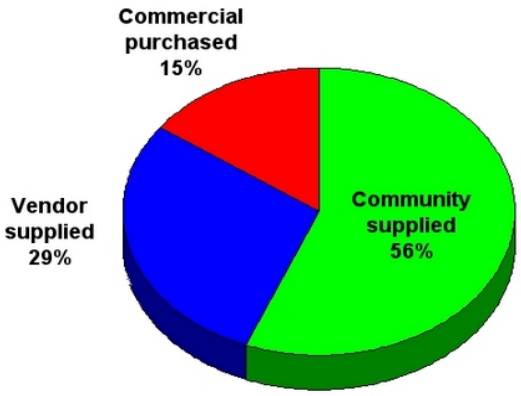

Figura 2.3: Tendencia de utilización de herramientas de desarrollo

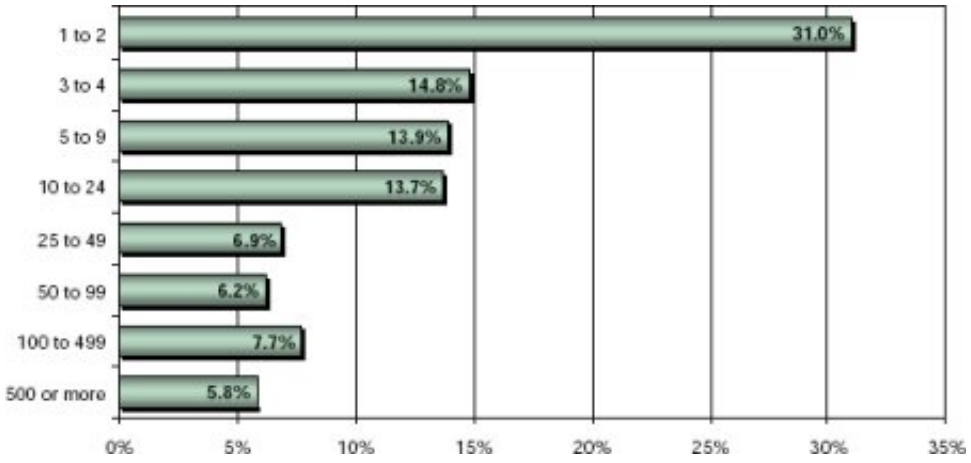

Figura 2.4: Número promedio de desarrolladores por compañía. Fuente Venture Development Corp

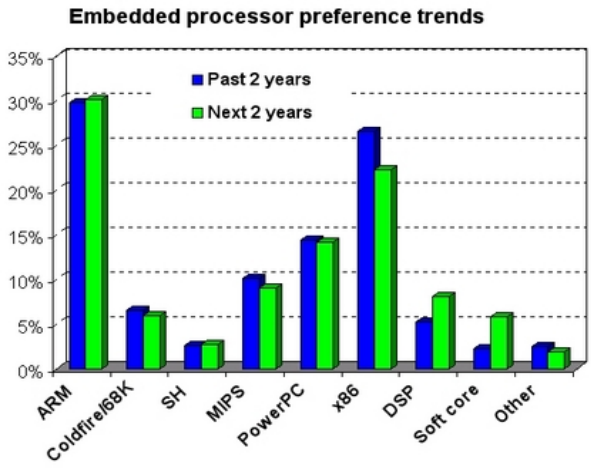

Figura 2.5: Tendencia del mercado de procesadores para sistemas embebidos. Fuente:[7]

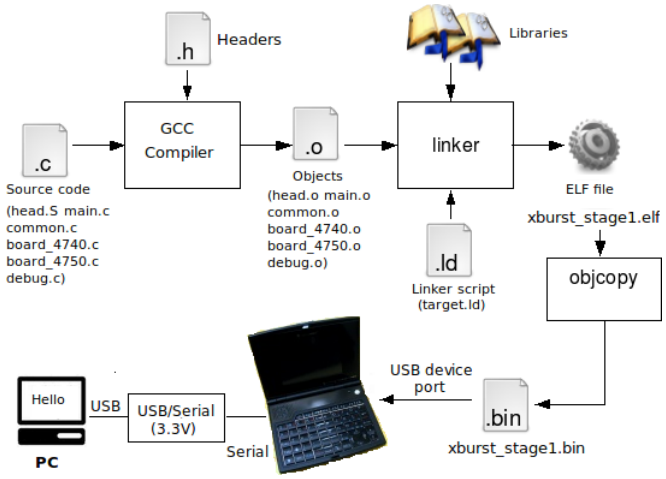

Figura 2.6: Tendencia del mercado de procesadores para sistemas embebidos. Fuente:[7]

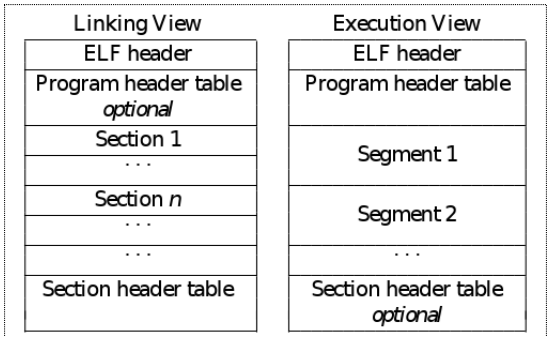

Figura 2.7: Formato ELF

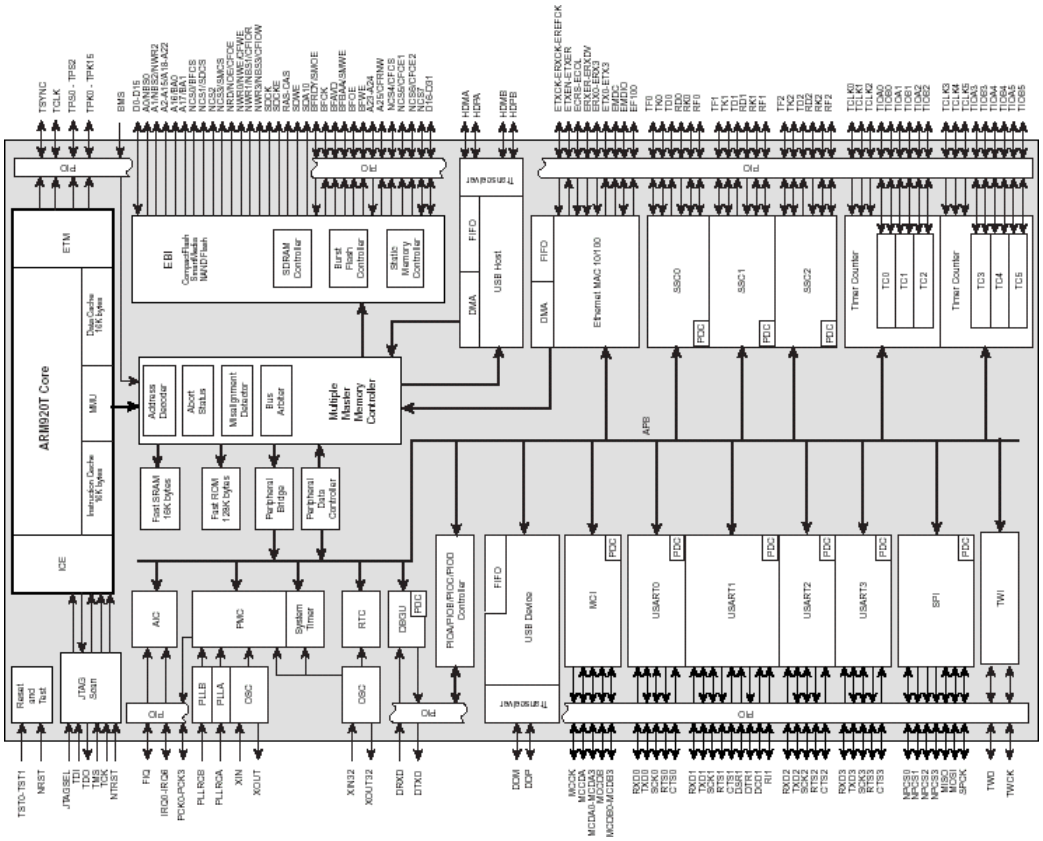

Figura 2.8: SoC AT91RM9200 fuente: Hoja de Especificaciones AT91RM9200, ATMEL

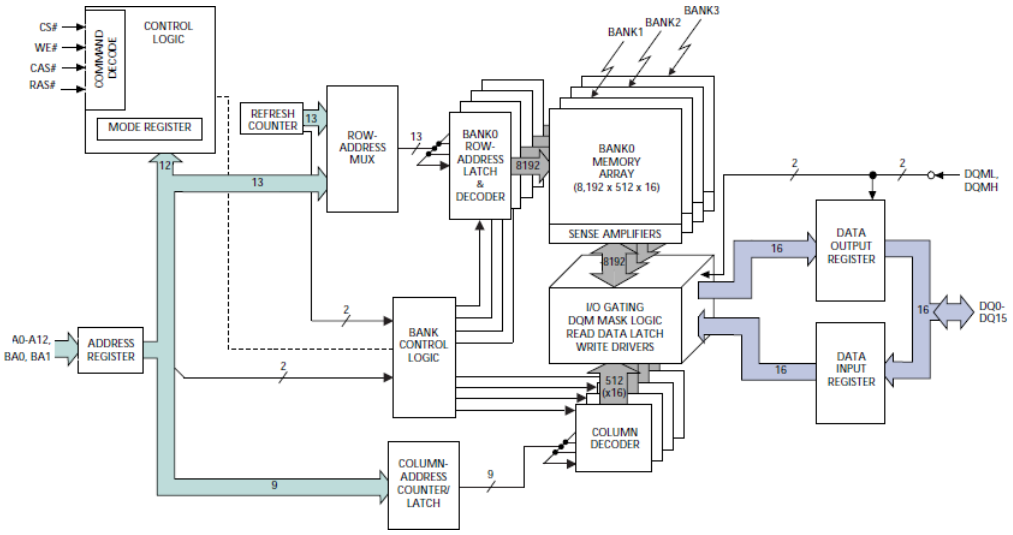

Figura 2.9: Diagrama de Bloques de una memoria SDRAM fuente: Hoja de Especificaciones MT48LC16M16, Micron Technology

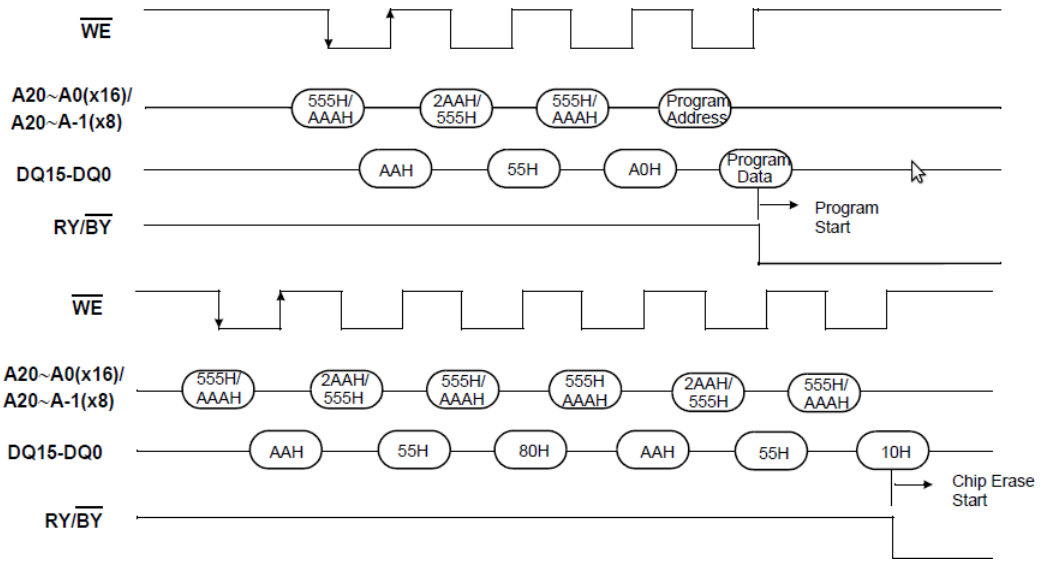

Figura 2.10: Ciclos de escritura y borrado de una memoria flash NOR

Capítulo 3

Sistema en un Chip (SoC)

En esta sección estudiaremos la arquitectura básica de un SoC moderno, componentes, funcionamiento, programación y operación. Como se mencionó anteriormente, la tendencia actual de la industria de los semiconductores es integrar en un solo dispositivo las funcionalidades necesarias para la implementación de dispositivos digitales modernos. Esto es posibe gracias a los grandes avances en las técnicas de fabricación de circuitos integrados y a la demanda de nuevas características exigidas por los fabricantes de dispositivos digitales de consumo masivo como teléfonos celulares, PDAs, consolas de juegos y reproductores multimedia. Para utilizar estos avances tecnológicos es necesario conocer su arquitectura, principio de funcionamiento, programación e implementación, para esto, se estudiarán dos proyectos abiertos que implementan un SoC en un PLD utilizando lenguaje de descripción de hardware y herramientas GNU; proporcionan el código fuente, lo que permite un estudio profundo de su arquitectura. El proyecto *Plasma* [10] y el proyecto Mico32[11].

3.1. Arquitectura

Un SoC, integra un conjunto de periféricos, memorias y una o varias unidades de procesamiento (CPUs) en un solo chip, lo cual facilita el desarrollo de aplicaciones. Comercialmente encontramos una gran variedad de configuraciones CPU - periféricos, dependiendo de la aplicación; dentro de los más comunes se encuentran: controladores de memorias externas (NOR, NAND, SDRAM, DDR), puertos de comunicación (I2C, SPI), puerto de depuración (UART), timers, reloj de tiempo real, codecs de audio, controladores de LCD, controladores de red, controlador de puerto USB host, controlador para sensores de imágenes, etc. La figura 3.1 muestra una arquitectura de un SoC sencillo con los componentes necesarios para implementar tareas simples.

Unidad de Procesamiento Central (CPU)

La unidad de procesamiento central (CPU), como su nombre lo indica, está encargada de centralizar las tareas del sistema coordinando las acciones de los periféricos. Puede ser vista como una máquina de estados programable que controla un camino de datos compuesto por bloques aritméticos, lógicos y un banco de registros. Cada CPU es capaz de realizar una serie de operaciones sobre variables almacenadas en el banco de registros, estas operaciones reciben el nombre de *conjunto de instrucciones*. El programador utiliza estas instrucciones para hacer que la CPU realice una función específica, indicandole paso a paso

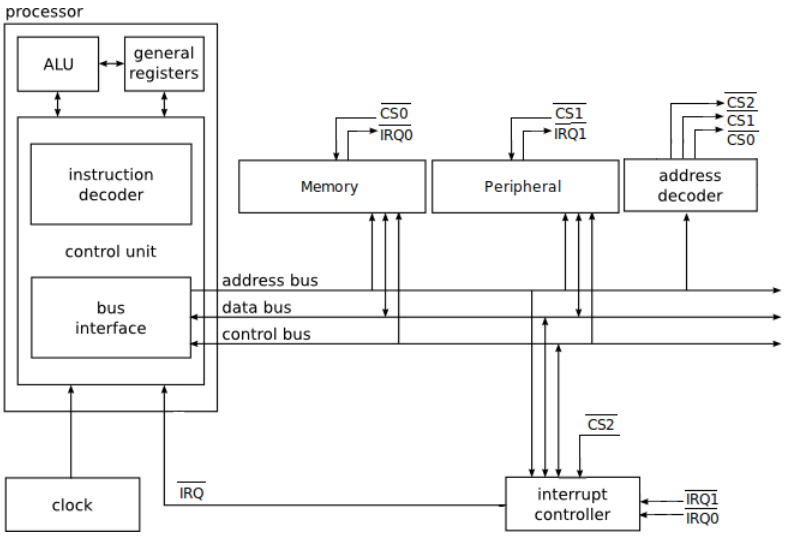

Figura 3.1: Arquitectura mínima de un SoC

donde debe leer la información, como procesarla y como entregar el resultado. A Esta función se le conoce con el nombre de programa o tarea Software.

Periféricos

Los periféricos proporcionan la comunicación con el exterior, permiten el ingreso, almacenamiento y procesamiento de la información

Bibliografía

[1] C. Lanfear, S. Balacco, and M. Volckmann. The 2005 Embedded Software Strategic Market Intelligence Program. *Venture Development Corporation http://www.vdc-corp.com*, August 2005.

[2] J. Turley. Embedded systems survey: Operating systems up for grabs. *Embedded Systems Programming*, May 2004.

[3] Venture Development Corp. Small teams dominate embedded software development. http://www.linuxdevices.com/articles/AT7019328497.html, 1 June 2006.

[4] Wikipedia. Wikipedia, the free encyclopedia. http://en.wikipedia.org/.

[5] Aleph 1. Building the Toolchain. http://www.aleph1.co.uk/node/279.

[6] Bill Gatliff. Porting and Using Newlib in Embedded Systems. http://venus.billgatliff.com/node/3.

[7] Linuxdevices. Embedded Linux market snapshot, 2005. *http://www.linuxdevices.com*, 4 May 2005.

[8] Dan Kegel. Building and Testing gcc/glibc cross toolchains. http://www.kegel.com/crosstool/, 20 February 2006.

[9] Michael L. Haungs. The Executable and Linking Format (ELF). http://www.cs.ucdavis.edu/ haungs/-paper/node1.html, 21 September 1998.

[10] S. Rhoads. Plasma - most MIPS I(TM) opcodes. URL: http://opencores.org/project,plasma, 2008.

[11] Lattice Semiconductor Corporation. LatticeMico32 Open, Free 32-Bit Soft Processor. URL: http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm, 2008.