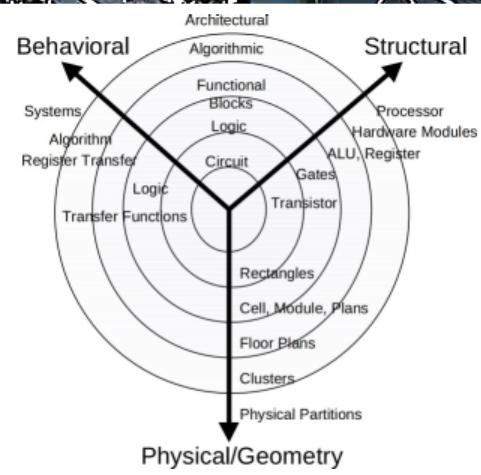
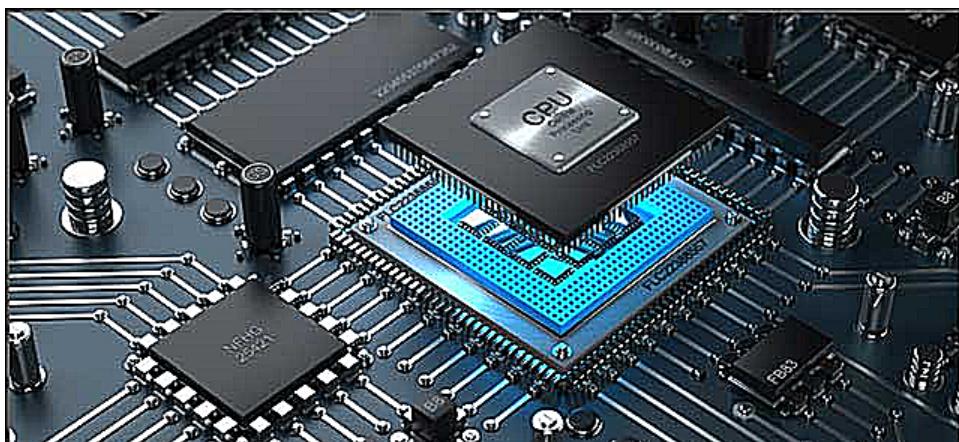


# DISEÑO DE SISTEMAS DIGITALES

Carlos Iván Camargo Bareño

Universidad Nacional de Colombia

21 de enero de 2025



DISEÑO DE SISTEMAS DIGITALES

AUTOR: C. Camargo

E-MAIL: cicamargoba@unal.edu.co

Universidad Nacional de Colombia  
<http://www.unal.edu.co>

Se garantiza el permiso para distribuir, y/o modificar este documento  
bajo los términos de la licencia Creative Commons CC BY-SA

Publicado por la Universidad Nacional de Colombia

# Índice general

<b>1. Síntesis de Sistemas Digitales .....</b>	<b>7</b>
1.1. Niveles de Abstracción .....	7
1.1.1. Síntesis .....	8
1.2. Sistema en Silicio - Descripción Estructural a nivel arquitectura .....	8
1.2.1. Sistemas sobre Silicio SoC .....	10
1.3. Ejemplo de implementación de una tarea usando un periférico .....	15
1.3.1. Lectura del periférico .....	16
1.3.2. Escritura del periférico .....	17
1.3.3. Implementación de la funcionalidad del periférico usando Lógica de transferencia de registros .....	17
1.3.4. Otra forma de implementación del multiplicador .....	27
1.4. Implementación de un divisor de n bits sin signo .....	27
1.4.1. Identificación de componentes del camino de datos e interconexión .....	28
1.4.2. Unidad de control .....	31
1.4.3. Diagrama de bloques del divisor .....	31
1.4.4. Implementación del algoritmo de división usando solo una máquina de estados .....	31
1.4.5. Simulacion del divisor .....	32
1.5. Flujo de diseño Hardware - Software .....	32
1.5.1. simulación .....	36
1.5.2. Estructura de los ejemplos .....	36
1.5.3. Aplicación software para utilizar el multiplicador .....	37
1.6. Programación en lenguaje C .....	38
<b>2. Implementación de tareas Software utilizando procesadores Soft Core .....</b>	<b>41</b>
2.1. Introducción .....	41
2.2. Arquitectura del procesador LM32 .....	41
2.2.1. Banco de Registros .....	42
2.2.2. Registro de estado y control .....	43
2.3. Set de Instrucciones del procesador Mico32 .....	44
2.3.1. Instrucciones aritméticas .....	44
2.3.2. Saltos .....	46
2.3.3. Comunicación con la memoria de datos .....	52
2.3.4. Interrupciones .....	54
2.3.5. Retorno de función y de excepción .....	58
2.4. Arquitectura del SoC LM32 .....	61
2.4.1. Bus wishbone .....	62
2.4.2. Arquitectura de los periféricos .....	64
2.4.3. Interfaz Software .....	68
2.5. Programación del SoC LM32 .....	71
2.5.1. Ejemplo de programación .....	74

<b>3. Procesador RISCV .....</b>	<b>77</b>
3.1. Introducción .....	77
3.2. Arquitectura del procesador RV32I .....	77
3.2.1. Diagrama de Bloques del FemtoRV .....	79
3.2.2. Set de Instrucciones .....	80
3.2.3. Saltos .....	81
3.2.4. Arquitectura del SOC basado en RV32I .....	82
3.2.5. Extensión RV32M .....	82
3.2.6. Flujo Hardware y Software para programar el RV32I .....	85
3.2.7. Ejemplos .....	85
3.3. Procesador RV32IMC .....	85
<b>4. Implementación de SoC usando Herramientas de Automatización .....</b>	<b>99</b>
4.1. Herramienta de automatización litex .....	99
4.1.1. Plataformas en litex .....	101
4.2. <i>migen</i> como herramienta de diseño .....	103
4.2.1. Blink en migen .....	104
4.2.2. Fading LED en migen .....	104
4.2.3. UART en migen .....	105
4.2.4. Prueba de periféricos mediante Uart.Bridge .....	106
4.2.5. Definición de SoC en litex .....	106
4.2.6. Creación de periféricos en migen .....	116
4.2.7. Creación de periféricos en verilog .....	117
4.3. Herramientas de depuración .....	123
4.3.1. UartBone .....	123
4.3.2. Etherbone .....	124
4.3.3. Litex_cli .....	126
4.3.4. litescope .....	127
<b>5. PLATAFORMAS DE DESARROLLO ECB .....</b>	<b>129</b>
5.1. Introducción .....	129
5.2. Herramientas abiertas para diseño de sistemas embebidos .....	129
5.2.1. Herramientas de Desarrollo .....	129
5.3. Métodos de arranque .....	130
5.3.1. Arranque del procesador AT91RM9200 .....	131
5.3.2. Interfaz JTAG .....	131
5.4. Flujo de diseño software .....	133
5.5. Dispositivos semiconductores .....	139
5.5.1. SoC .....	139
5.5.2. Memorias Volátiles .....	139
5.5.3. Memorias No Volátiles .....	141
5.6. Depuración del core ARM [1] .....	142
5.6.1. Proyecto OpenOCD .....	143
5.6.2. Programación de memorias Flash .....	144
5.7. Arquitectura: SoC, Memorias, periféricos .....	144
5.7.1. Programación .....	146
5.7.2. Programación utilizando el puerto JTAG .....	148
5.8. El sistema Operativo Linux .....	148
5.8.1. Arquitectura de Linux [2] [3] .....	150
5.8.2. Arbol de dispositivos .....	154
5.8.3. Distribuciones Linux .....	155
5.8.4. Yocto .....	157
5.8.5. Debian .....	157
5.9. Adaptando Linux a la plataforma ECB_T8_T113 .....	157

Índice general	5
5.9.1. Creación de la imagen utilizando buildroot .....	158
5.9.2. Arranque del procesador Allwineer T113 .....	160
5.9.3. Configuración del kernel de Linux y de u-boot desde Buildroot .....	162
5.9.4. Creación de la tarjeta SD .....	164
5.9.5. Configuración del puerto serial .....	164
5.10. Creación y adaptación de Debian .....	164
5.10.1. Creación/aumento de tamaño de la partición en la SD .....	164
5.10.2. Descarga del sistema de archivos .....	165
5.10.3. Modificación del kernel para Debian .....	166
5.11. Compilación de u-boot .....	167
5.11.1. Descarga de la cadena de Herramientas .....	167
5.11.2. Compilación de U-BOOT 2018 .....	168
5.11.3. U-BOOT mainline .....	169
5.12. Compilación del kernel sin buildroot .....	171
5.13. Aplicaciones gráficas .....	172
5.13.1. Configuración del driver FBTFT para el LCD ILI9340 .....	172
5.13.2. Adición del LCD al árbol de dispositivos .....	173
5.14. configuración de la red en Debian .....	174
5.14.1. Configuración de ssh en el PC .....	174
5.14.2. Configuración de ssh en ECB_T8_T113 .....	175
5.14.3. Aplicación con LVGL .....	175
5.15. Configurando el sonido con ALSA .....	175
5.16. Módulos del kernel .....	177
5.16.1. Ejemplo de un driver tipo carácter .....	177
5.16.2. Instalación/desinstalación automatizada del driver .....	179
5.16.3. Ejemplo Módulo de driver I2C .....	180
5.17. comunicación con periféricos desde espacio de usuario .....	181
5.18. Interrupciones en módulos .....	181
5.19. Instrumentos musicales electrónicos .....	183
5.19.1. Tecclaldo Matricial .....	183
5.19.2. Faust .....	185
5.19.3. Open Sound Control .....	186
5.19.4. Notas musicales con entrada desde key_pad .....	187
5.19.5. Configuración de amidi .....	188
<b>6. Herramientas libres para desarrollar proyectos Hardware-Software .....</b>	197
6.1. OpenHardware .....	197
6.2. Arquitectura de la tarjeta ECB_T8_T113 .....	197
6.3. Software y Aplicaciones Básicas .....	198
6.3.1. FPGA .....	198
Referencias .....	200



# Capítulo 1

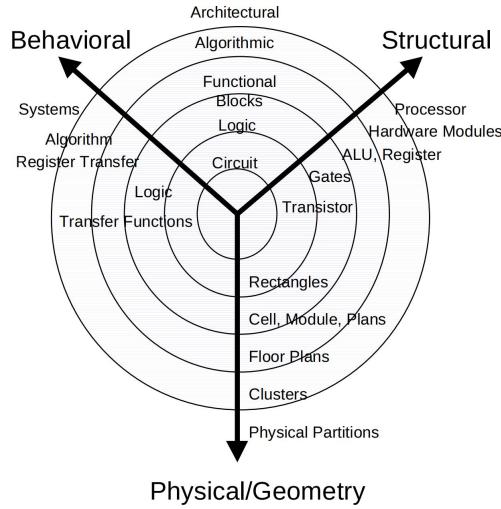
## Síntesis de Sistemas Digitales

Según el diccionario de la lengua española un Sistema es un "Conjunto de cosas que relacionadas entre sí ordenadamente contribuyen a determinado objeto.". Por lo tanto, un sistema digital es un conjunto de componentes electrónicos que de forma ordenada realizan una determinada función.

### 1.1. Niveles de Abstracción

Todo sistema digital puede ser descrito de tres diferentes formas utilizando diferentes dominios de descripción: *Comportamental* - describe el funcionamiento del sistema, *Estructural* - describe su estructura lógica y *Físico* - describe la implementación física, estos dominios se representan como ejes en la figura 1.1 que recibe el nombre de Y de Gajski-Kuhn. Dentro de cada dominio es posible realizar diferentes descripciones dependiendo del nivel de abstracción, un nivel de abstracción alto representa una visión de alto nivel del sistema enfocándose en características globales, mientras que un nivel de abstracción bajo implica una descripción más detallada. Existen 5 niveles de abstracción:

- *Arquitectura*: En este nivel se representa el sistema a alto nivel indicando sus principales componentes/funciones, no se detalla cómo están implementadas. En el dominio comportamental se realiza una descripción de funcionalidades unidas a restricciones físicas, en el dominio Estructural se describen los procesadores, memorias y buses de conexión del sistema y en el dominio físico dependiendo de la implementación se puede mostrar un diagrama del Circuito Integrado o la disposición física en una placa de circuito impreso.
- *Algoritmo*: Se realiza una descripción de las funcionalidades descritas en el nivel arquitectura, en el dominio comportamental se proporciona la descripción de todos los algoritmos que implementan la funcionalidad requerida, en el dominio Estructural se muestran los componentes que implementan un determinado algoritmo (caminos de datos, control, periféricos y procesador), en el dominio físico se realiza una descripción que incluye información de tiempos de respuesta de los circuitos que implementan la descripción estructural.
- *Bloque Funcional*: En el dominio comportamental se describen los algoritmos como el resultado de una transferencia entre registros, lo que la acerca más a la descripción estructural, uno de los métodos utilizados es el uso de máquinas de estado algorítmicas; en el dominio estructural la descripción se realiza utilizando unidades aritméticas-lógicas (ALUs), sumadores, comparadores, contadores, multiplexores y registros todos ellos controlados por una unidad de control. En el dominio físico se realiza una representación de la localización de los componentes, sus caminos de conexión, teniendo en cuenta consideraciones de espacio y potencia.
- *Lógico*: En el dominio funcional se realiza una descripción a nivel de ecuaciones booleanas y autómatas finitos; el dominio estructural se representa con los componentes básicos Flip-Flops, y compuertas.
- *Circuitos*: En el dominio comportamental se realizan representaciones del comportamiento utilizando ecuaciones diferenciales de voltaje y corriente; en el dominio estructural se realiza la representación de cada componente a nivel de transistores, diodos, resistencias y condensadores dependiendo de la tecnología utilizada; el dominio físico se representa con el diagrama físico de las diferentes capas que forman el circuito integrado indicando las dimensiones exactas de y geometría de cada componente.



**Figura 1.1** Niveles de Abstracción Y de Gajksi-Kuhn fuente: D. Gajski, ?Silicon Compilers?, Addison-Wesley, 1987

### 1.1.1. Síntesis

Pasar de un dominio de descripción a otro recibe el nombre de síntesis, es un proceso realizado con ayuda de herramientas de diseño (CAD), las cuales pueden ser lenguajes de programación o de descripción de hardware (verilog, vhdl, systemC, etc) o herramientas gráficas. En la gráfica 1.2 se muestra un flujo de diseño que comienza en una descripción comportamental de alto nivel con la funcionalidad requerida (1), continúa con una descripción estructural a nivel de arquitectura (2), sigue con una descripción comportamental a nivel algorítmico (3), a continuación se realiza una descripción estructural a nivel funcional (4), después se muestra una descripción estructural a nivel lógico (5) y finaliza con una representación física del circuito integrado que implementa la funcionalidad especificada en 1. Procesos de síntesis se pueden observar del paso (3) al (4) donde una herramienta tipo CAD a partir de una descripción comportamental basada en un diagrama de flujo y un diagrama de estados genera una representación en el dominio estructural con componentes básicos (flip-flops, compuertas y registros). Otro ejemplo de síntesis se obtiene cuando se pasa de una descripción **estructural** a nivel lógico (5) a una descripción **física** a nivel de circuito. Estas tareas se realizaban manualmente en el pasado, lo que generaba muchos errores o reprocesos y en algunos casos tomaban mucho tiempo, gracias a los avances de las herramientas computacionales, y a la proliferación de herramientas para este fin, estas tareas se pueden realizar de forma automatizada aumentando la confiabilidad de los diseños al tiempo que se reduce el tiempo para hacerlas y se permite trabajar con circuitos más complejos.

## 1.2. Sistema en Silicio - Descripción Estructural a nivel arquitectura

En el pasado, en ausencia de computadores y herramientas para el diseño, la experiencia del diseñador era clave a la hora de abordar la creación de un nuevo dispositivo digital, decenas de años de experiencia acumulada en el equipo de diseño permitía reducir de forma considerable el tiempo a salir al mercado del producto (time to market), sin embargo, adquirir esta experiencia no era fácil ya que requería mucho tiempo para lograrlo, cuando una persona quiere diseñar un sistema que resuelva un determinado problema, no es factible esperar muchos años para poder hacerlo, lo mismo sucede con los estudiantes en los cursos de diseño digital (que en total en el mejor de los casos suman 2 años), entonces para que estos estudiantes salgan al mundo productivo con las habilidades necesarias ¿Qué se debe hacer?.

1- Alargar las carreras para que los estudiantes adquieran las habilidades necesarias como en el pasado o 2. Utilizar las herramientas modernas y cambiar la forma de trabajo en estos cursos centrándose en las tareas que realizamos mejor los humanos y dejar los procesos tediosos a las máquinas.

Algo en lo que somos buenos los seres humanos es en pensar, esto hasta el momento (2023) no lo ha podido realizar ninguna máquina. Por esta razón, es necesario cambiar la forma de diseño que se realizaba en el pasado, la cual consistía en una metodología en la que a partir de componentes discretos se llegaba a la funcionalidad deseada, de

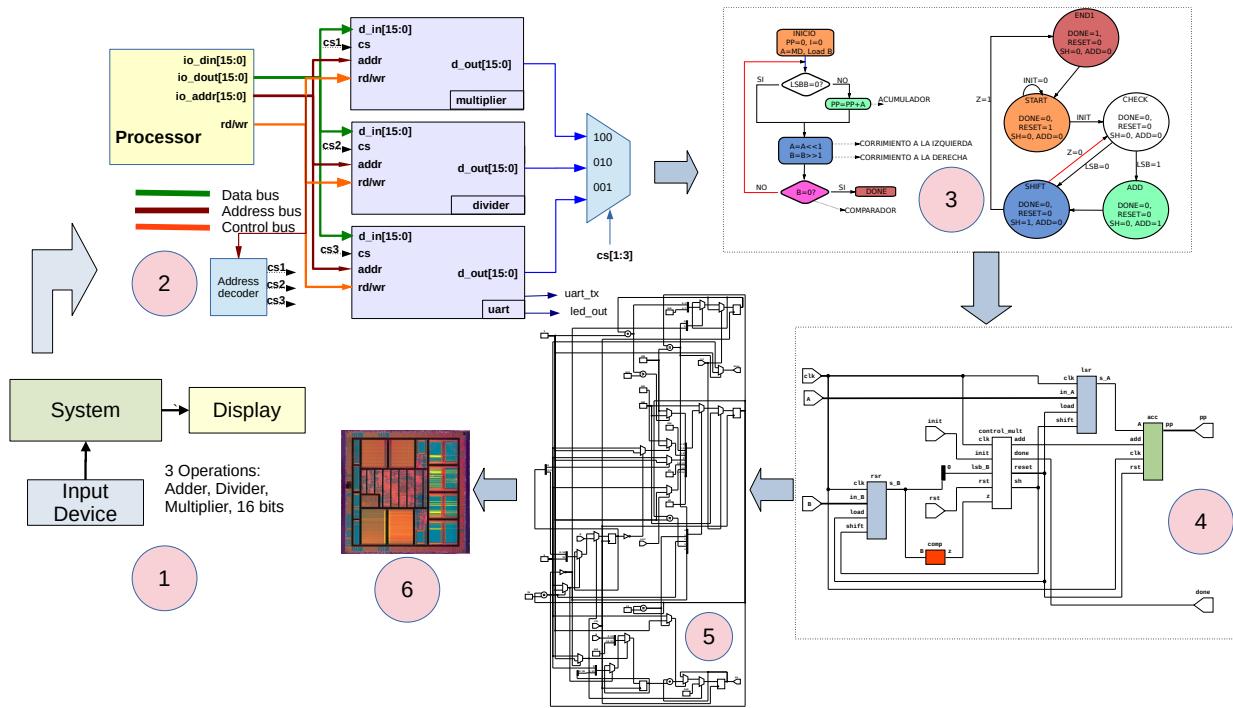


Figura 1.2 Flujo de diseño con diferentes niveles de abstracción y dominios de descripción

nuevo, solo diseñadores con mucha experiencia podían realizar esta tarea. Si se invierte esta metodología partiendo de las especificaciones del sistema y de la funcionalidad que se desea, utilizando los más altos niveles de abstracción para hacer la descripción del sistema y haciendo uso de las herramientas tipo CAD para bajar el nivel de abstracción y para cambiar el dominio de descripción, la experiencia del diseñador no será tan determinante y se podrán formar profesionales con las habilidades que la industria requiere.

Esto debe estar acompañado de un cambio en la metodología de enseñanza ya que la forma tradicional (quiz, parcial, tareas para la casa) no es aplicable a la hora de adquirir habilidades en diseño, para aprender a diseñar se deben enfrentar problemas reales y dar soluciones a problemas reales (lo que no es posible en 2 horas de un parcial o un quiz), es necesario trabajar en equipo de forma efectiva (apoyando a los demás para lograr el objetivo, realizando consensos y articulando el trabajo entre los diferentes miembros o equipos), la teoría debe estar sincronizada con el momento en el flujo de diseño para que se experimente en la práctica logrando aprendizajes significativos.

En este libro se utilizará la descripción a nivel arquitectura asumiendo que todo sistema digital se implementará utilizando procesadores, periféricos y memorias, se utilizarán diferentes procesadores (LM32, RISC-V) y se trabajará en la creación de periféricos dedicados utilizando el lenguaje de descripción de hardware Verilog (aunque no se realizará una presentación de este lenguaje ya que en la web se encuentra mucho material de buena calidad).

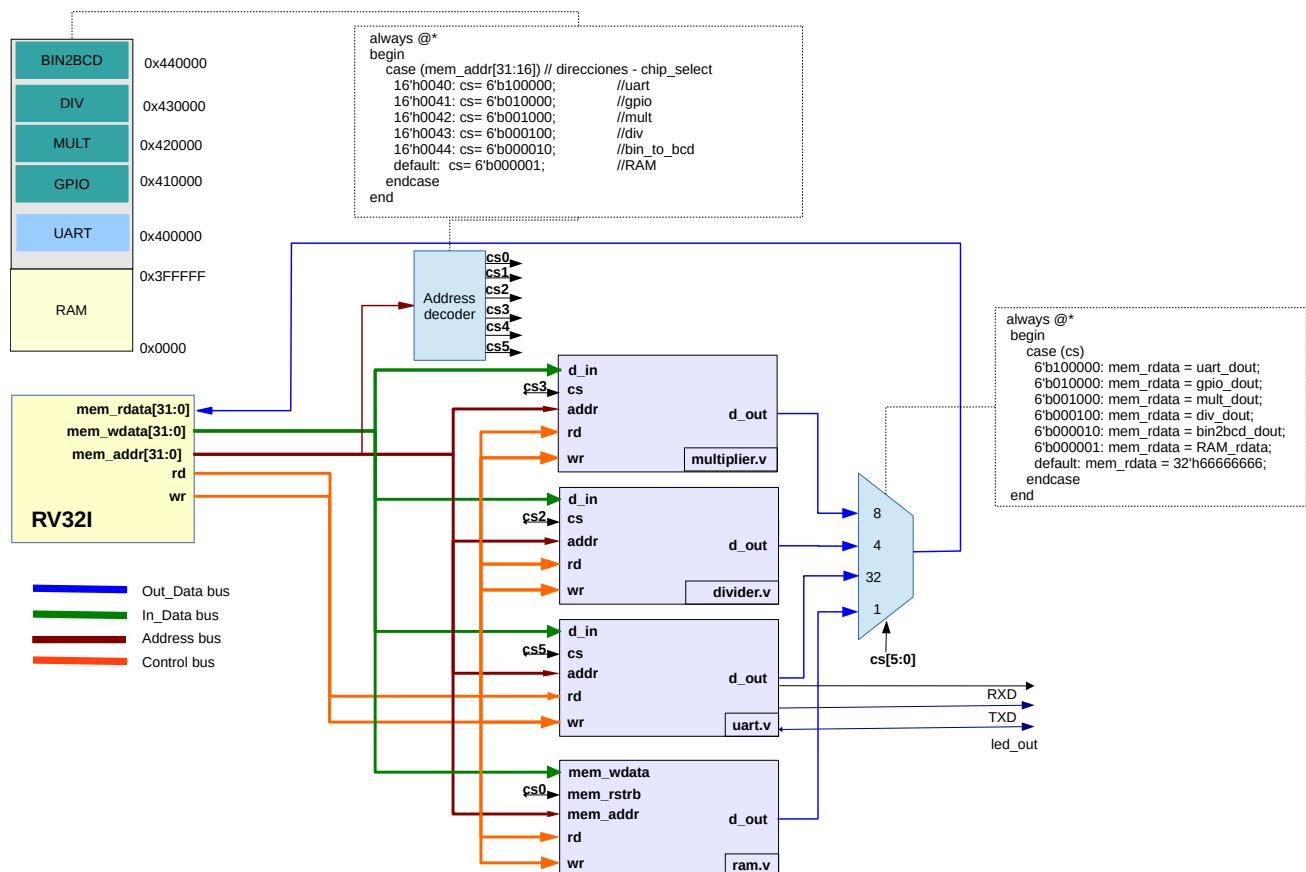
Para el primer curso del área de sistemas digitales se trabajará con un procesador, el cuál será tratado como una caja negra con un conjunto de funcionalidades que puede ser programado usando el lenguaje C. Esto se hace para que se conozca perfectamente la arquitectura de los procesadores modernos donde siempre se incluye una CPU con un grupo de periféricos, al tiempo que se trabajan las habilidades en programación usando un lenguaje de programación de nivel adecuado.

En este primer curso se realizará la descripción de periféricos utilizando máquinas de estado algorítmicas, las cuales, reproducen la arquitectura basada en una unidad que implementa un algoritmo y unos bloques que realizan funciones básicas aritméticas o lógicas.

### 1.2.1. Sistemas sobre Silicio SoC

Un Sistema en silicio (SoC) es la unión de una unidad de procesamiento central (CPU) y un grupo de periféricos (multiplicador, divisor, uart, unidad aritmética de punto flotante, etc); la CPU está encargada de leer, decodificar y ejecutar una lista de instrucciones que implementan una determinada tarea, esta arquitectura permite la ejecución de cualquier algoritmo con solo cambiar las instrucciones que deben ser ejecutadas.

En la figura 1.3 se muestra el diagrama de bloques de un Sistema en Silicio basado en la unidad de procesamiento central RISCV, el cual fue desarrollado por Bruno Levy <sup>1</sup>.



**Figura 1.3** Diagrama de bloques del procesador RISCV

### Procesador RISCVI

La CPU es un módulo que realiza tres tareas de forma continua. *Fetch* lee una instrucción de la memoria de programa, *Decode*, reconoce la instrucción y *Execute*, realiza las acciones definidas para la instrucción detectada. En este capítulo se utilizará una CPU como una caja negra, es decir, un módulo que ejecuta una serie de instrucciones y nos permite controlar otros módulos, los cuales, a su vez implementan una determinada función.

En la actualidad existen una gran variedad de procesadores tanto abiertos como propietarios, sin embargo, el procesador RISCV, ha sido adoptado por una gran cantidad de personas en todo el mundo (que hacen parte de diferentes mundos, industrial, académico, pasatiempo), y gracias a esto se han desarrollado una gran cantidad de herramientas

<sup>1</sup> <https://github.com/BrunoLevy>

que facilitan su uso, en específico, se cuenta con compiladores en diferentes lenguajes, librerías, sistemas operativos y aplicaciones.

Parte de su popularidad se debe a su carácter libre y abierto, esto es, puede utilizarse sin restricciones por quien lo desee. El proyecto inició en la Universidad de Berkeley con la colaboración de voluntarios y diseñadores externos. Este procesador como se puede deducir de su nombre es del tipo **Reduced Instruction Set Computer**, es decir, posee un conjunto de instrucciones reducido, lo que disminuye su complejidad, permite aumentar la velocidad de desempeño al permitir la segmentación (no es necesario esperar a que una instrucción se ejecute para adquirir la siguiente instrucción).

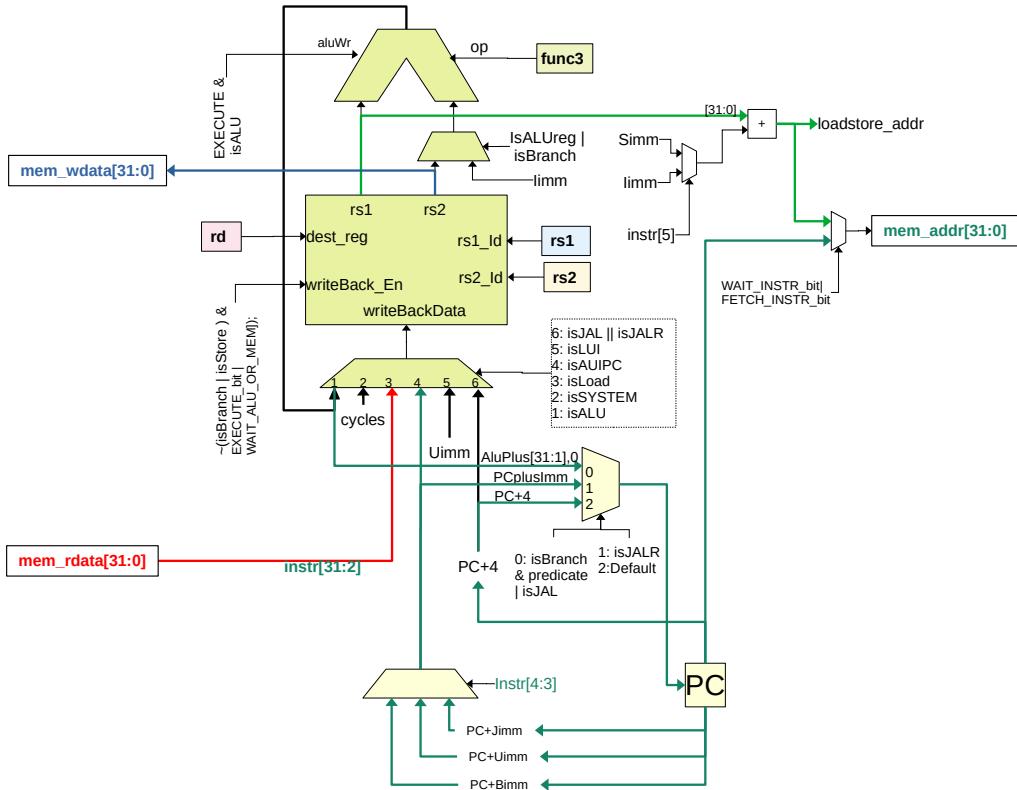
## Set de Instrucciones

En la tabla 3.2 se puede apreciar la lista de las instrucciones del RV32I con su respectivo código y función. Estas instrucciones pueden ser vistas como bloques constructores, y serán parte de los algoritmos implementados para ser ejecutados por el procesador.

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	ADD	R	0110011	0x0	0x00	$rd = rs1 + rs2$
sub	SUB	R	0110011	0x0	0x20	$rd = rs1 - rs2$
xor	XOR	R	0110011	0x4	0x00	$rd = rs1 \wedge rs2$
or	OR	R	0110011	0x6	0x00	$rd = rs1 \mid rs2$
and	AND	R	0110011	0x7	0x00	$rd = rs1 \& rs2$
sll	Shift Left Logical	R	0110011	0x1	0x00	$rd = rs1 << rs2$
srl	Shift Right Logical	R	0110011	0x5	0x00	$rd = rs1 >> rs2$
sra	Shift Right Arith*	R	0110011	0x5	0x20	$rd = rs1 >> rs2$
slt	Set Less Than	R	0110011	0x2	0x00	$rd = (rs1 \mid rs2)?1:0$
sltu	Set Less Than (U)	R	0110011	0x3	0x00	$rd = (rs1 \mid rs2)?1:0$
addi	ADD Immediate	I	0010011	0x0		$rd = rs1 + imm$
xori	XOR Immediate	I	0010011	0x4		$rd = rs1 \wedge imm$
ori	OR Immediate	I	0010011	0x6		$rd = rs1 \mid imm$
andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	$rd = rs1 << imm[0:4]$
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	$rd = rs1 >> imm[0:4]$
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	$rd = rs1 >> imm[0:4]$
slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 \mid imm)?1:0$
sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 \mid imm)?1:0$
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$
sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$
beq	Branch ==	B	1100011	0x0		$if(rs1 == rs2) PC += imm$
bne	Branch !=	B	1100011	0x1		$if(rs1 != rs2) PC += imm$
blt	Branch $\mid$	B	1100011	0x4		$if(rs1 \mid rs2) PC += imm$
bge	Branch $>$	B	1100011	0x5		$if(rs1 \geq rs2) PC += imm$
bltu	Branch $\mid$ (U)	B	1100011	0x6		$if(rs1 \mid rs2) PC += imm$
bgeu	Branch $\geq$ (U)	B	1100011	0x7		$if(rs1 \geq rs2) PC += imm$
jal	Jump And Link	J	1101111			$rd = PC+4; PC += imm$
jalr	Jump And Link Reg	I	1100111	0x0		$rd = PC+4; PC = rs1 + imm$
lui	Load Upper Imm	U	0110111			$rd = imm << 12$
auipc	Add Upper Imm to PC	U	0010111			$rd = PC + (imm << 12)$
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger

Cuadro 1.1 Set de Instrucciones del RV32I. fuente:<https://github.com/jameslzhu/riscv-card/tree/master>

En la Figura 1.4 se muestra el diagrama del camino de datos del procesador, en ella se aprecian los componentes típicos de esta unidad, un banco de registros, una Unidad Aritmética y Lógica (ALU) y los buses de datos (*mem\_wdata*, *mem\_rdata*) y dirección (*mem\_addr*) que van al exterior.



**Figura 1.4** Camino de datos del procesador RISC-V.

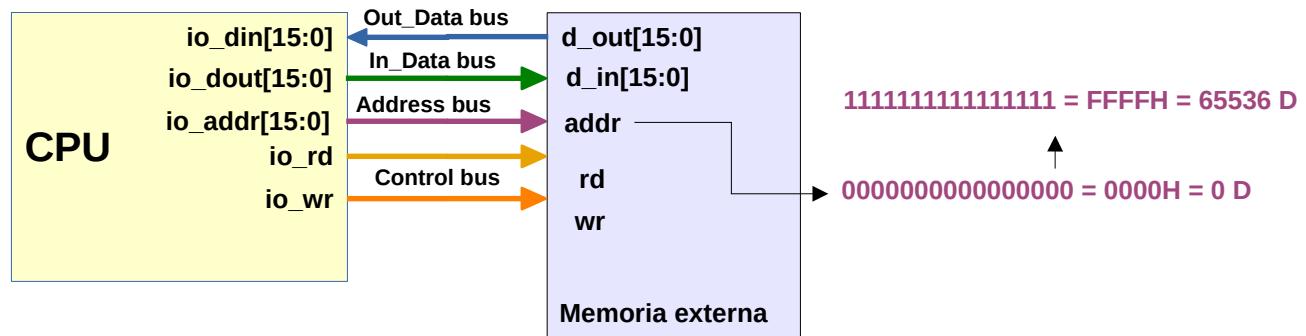
Toda CPU posee un registro especial que recibe el nombre de *contador de programa* (PC), este registro controla el bus de direcciones de la memoria de programa, es decir, controla el flujo de ejecución del programa, permitiendo el llamado a funciones (segmentos de código que se deben ejecutar a lo largo del programa), atención a interrupciones (solicitudes realizadas por dispositivos que se conectan con el mundo exterior), lazos (permiten la ejecución de un segmento de código en repetidas ocasiones), saltos condicionales (permite implementar bloques de decisión), saltos (permiten controlar el flujo de un programa), entre otros.

Para que un algoritmo tenga utilidad en la vida real debe permitir la entrada de datos desde el exterior, así como la salida de los resultados para que sean utilizados externamente, para esto, los procesadores deben contar con dispositivos de entrada/salida que permitan el intercambio de información o realicen tareas específicas, estos dispositivos reciben el nombre de *periféricos*, en la actualidad existe una gran variedad para diversas aplicaciones, lo que permite utilizar diferentes protocolos de comunicación (serial, SPI, I2C, USB, Ethernet, PCI, etc), dispositivos de interfaz con humanos (teclados, monitores, LCDs, etc). Estos periféricos se conectan a la CPU utilizando 3 buses (agrupación de 8, 16, 32, 64, señales) que realizan las siguientes funciones:

- **Datos:** Transmite y recibe información entre el procesador y el periférico, esta información representa valores de variables de un determinado algoritmo, valores para configurar periféricos o valores que se intercambian desde y hacia el exterior.
- **Dirección:** Transmite información que permite seleccionar el periférico con el que se desea comunicar el procesador.
- **Control:** Indica el tipo de operación que va a realizar la CPU, lectura o escritura.

Gracias a estos tres buses la CPU puede acceder a la memoria externa del procesador, esto es, los diferentes valores que puede tomar el bus de direcciones, cada valor del bus de direcciones corresponde a un dato de N bits del bus de

datos; en la figura 1.5 se muestra este concepto, allí podemos observar que existen dos buses para el bus de datos uno para la entrada y otra para la salida de datos (esto se hace ya que los Dispositivos Lógicos Programables no poseen buses tri-estado internamente). El número de señales del bus de direcciones determina el rango de direcciones que puede tomar la memoria externa, en este ejemplo son 16 señales, por lo que este bus puede tomar los valores 0 a 65536, lo que corresponde a un bus de 16 bits. Las señales del bus de control se activan cuando el procesador realiza una operación de lectura (activando la señal *rd*) o de escritura (activando la señal *wr*).



**Figura 1.5** Buses del procesador RISCV y concepto de memoria externa

## Periféricos

Como se dijo anteriormente, los periféricos realizan tareas fijas y permiten adicionar funcionalidad a la CPU, en la figura 1.3 se muestra un SoC con tres periféricos: *multiplier* y *divider* posibilitan operaciones de multiplicación y división, mientras que el periférico *uart* permite la transmisión de información utilizando un protocolo serial asíncrono, el cuál, es el más utilizado en aplicaciones de microcontroladores.

La comunicación entre los periféricos y la CPU se realiza a través de los buses de *datos*, *direcciones* y *control*. La CPU monopoliza el control de las operaciones de lectura y escritura, los periféricos nunca realizarán operaciones a la CPU de forma autónoma, siempre responden a peticiones de la CPU; este proceso centralizado permite coordinar las acciones de los periféricos y cede el control de la ejecución del algoritmo a la CPU, esto no significa que los periféricos no puedan realizar tareas complejas, solo que ellos realizan tareas ordenadas por la CPU.

Para diferenciar con qué periférico se está comunicando la CPU, se debe asignar un rango de las direcciones externas a cada uno de los periféricos que forman el SoC, el rango de direcciones asignado depende del tipo de periférico, así pues, una memoria externa de 1 kB requerirá 1024 B, mientras que un periférico sencillo con 16 registros (de configuración, control y estatus) requerirá 16 posiciones de memoria. En SoCs con pocos periféricos (entre 1-10) es normal asignar el mismo rango de memoria (número de bits) a los periféricos así estos no requieran estas posiciones. En la figura 1.3 podemos observar que se asignan las regiones 0x400000 a 0x40FFFF para la UART, 0x410000 a 0x41FFFF para los GPIOs, 0x420000 a 0x42FFFF para el multiplicador, 0x430000 a 0x43FFFF para el divisor y 0x440000 a 0x44FFFF para el BIN2BCD. Las direcciones 0-0x3FFFFFF son asignadas a la memoria de programa.

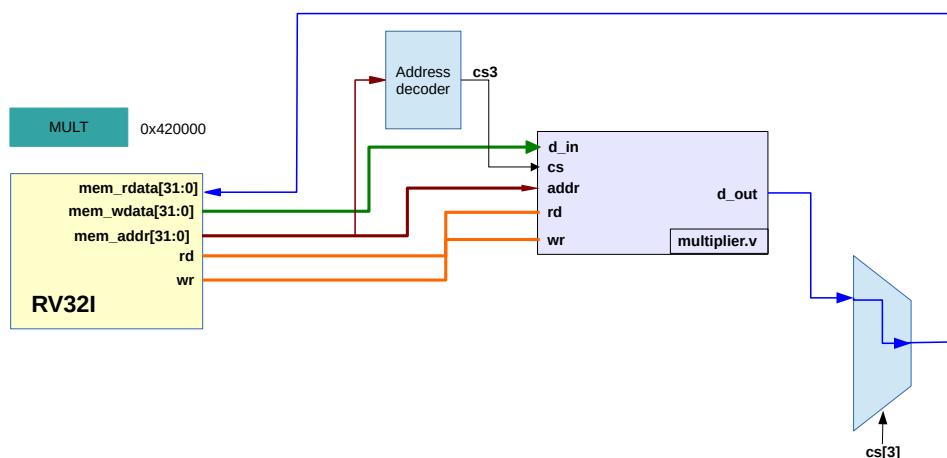
Este listado de rango de direcciones recibe el nombre de *mapa de memoria*, y el módulo que realiza su implementación física recibe el nombre de *decodificador de direcciones* (address decoder), el cual (ver figura 1.3), tiene como entrada las señales del bus de direcciones y proporciona seis salidas (una por cada periférico) *cs0* - *cs6* (*cs* por chip select), estas señales se activan de acuerdo al mapa de memoria establecido: *cs0* se activa (toma un valor lógico alto 1) en el rango 0x000000 a 0x3FFFFFF, *cs1* en el rango 0x400000 a 0x40FFFF y *cs2* para el rango 0x410000 a 0x41FFFF, etc. En el recuadro de la figura se muestra el código en Verilog que implementa este decodificador, se trata de un bloque combinatorio (el valor de las salidas dependen únicamente del valor de las entradas) que tiene como entradas las señales del bus de direcciones *mem\_addr[31:16]* (bits 31 al 16), esto porque las señales *mem\_addr[15:0]* varían de 0000 a FFFF tomando cualquiera de estos valores y no son determinantes. La salida es el bus *cs[5:0]* formado por las seis señales de selección descritas anteriormente. La sentencia *case* examina el valor de *mem\_addr* y asigna el valor a *cs* de acuerdo al mapa de memoria, estos valores son arbitrarios y pueden cambiar a criterio del diseñador, lo único que debe cumplirse es que **no se pueden asignar la misma dirección de memoria a más de un periférico**.

### Ejemplo de comunicación CPU-periférico

Las señales de control *wr* y *rd*, indican el tipo de transmisión; cuando el procesador escribe a un periférico activa la señal *wr* y cuando lee información activa la señal *rd*, dicha información será transmitida por el bus de datos de salida y de entrada respectivamente. Las señales *cs1*, *cs2* ... *cs6* (chip select) activan al periférico con el que se desea realizar la comunicación, y únicamente el que se encuentre activado (solo se activa uno) participará en el intercambio de información con la CPU.

El bus de datos de salida de la CPU se conecta al bus de datos de entrada de todos los periféricos, y sólo el que sea seleccionado (activando su señal *cs*) procesará esa información. El bus de datos de salida de los periféricos se conecta a un *multiplexor* (módulo combinatorio que conecta una de múltiples entradas a la salida dependiendo de la señal de selección) que controla la conexión de estos al bus de datos de entrada del procesador, las señales *cs1*, *cs2* y *cs3* determinan que periférico es conectado. De esta forma cuando se activa la señal de activación del periférico se crea un camino entre el periférico y la CPU.

Para aclarar este proceso, considere que desea escribir los datos 0x0005 en la dirección 0x420004, 0x0003 en la 0x420008 y 0x0001 en la 0x42000C; estas direcciones se encuentran en el rango del multiplicador por lo que la señal *cs3* se activará (1) y las señales *cs0,1,2,4,5* desactivarán (0) a los otros periféricos.



**Figura 1.6** SoC J1 - Comunicación con un periférico

En la figura 1.6 se muestra el circuito equivalente al seleccionar el rango de direcciones del multiplicador, los otros periféricos se desactivan lo que para fines prácticos equivale a eliminarlos, con lo que la CPU se encuentra totalmente conectada al multiplicador.

### Escritura a un periférico

En la figura 1.7 se muestran las formas de onda de las señales asociadas a un ciclo de escritura desde la CPU a las direcciones 0x420004, 0x420008 y 0x42000C. Cuando se realiza una operación de escritura, la CPU coloca la dirección en el bus de direcciones *mem\_addr*, lo que hace que la señal *cs3* sea igual a 1; activa la señal *wr* (1) del bus de control y coloca el dato a escribir en el bus de datos de salida *mem\_wdata*. Para las señales mostradas en la figura escribe 0x0005 en la dirección 0x420004, 0x0003 en la dirección 0x420008 y 0x0001 en la 0x42000C.

### Lectura a un periférico

En la figura 1.8 se muestran las formas de onda de las señales asociadas a un ciclo de lectura desde la CPU a las direcciones 0x420014 y 0x420010. Cuando se realiza una operación de lectura, la CPU coloca la dirección en el bus de direcciones *mem\_addr*, lo que hace que la señal *cs1* sea igual a 1; activa la señal *rd* del bus de control y lee el bus de

## 1.3 Ejemplo de implementación de una tarea usando un periférico

15

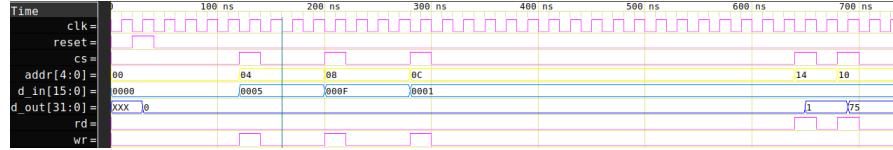


Figura 1.7 SoC J1 - Formas de onda de la escritura un periférico

datos de entrada *mem\_rdata*. Para las señales mostradas en la figura lee 0x0001 en la dirección 0x420014, y 0x000F en la 0x420010.



Figura 1.8 SoC J1 - Formas de onda de la lectura un periférico

### 1.3. Ejemplo de implementación de una tarea usando un periférico

En esta sección mostraremos la estructura interna de los periféricos para permitir la comunicación con la CPU. En la figura 1.9 se muestra el diagrama de caja negra para el multiplicador; este módulo tiene como entradas los operandos de 16 bits **A** y **B** y la señal **init** (que da comienzo al algoritmo de multiplicación) y como salida el resultado de 32 bits **PP** y la señal que indica que ya se realizó la operación **done**.

Se deben crear mecanismos que permitan suministrar los operandos y obtener el resultado de la operación a través de los buses de datos conectados a la CPU, para esto, el primer paso, es identificarlos y asignarles una dirección de memoria a cada uno de ellos para que la CPU pueda leerlos o escribirlos. En este caso (la asignación puede variar a criterio del diseñador) el primer operando se encuentra en la dirección *BASE + 00*, (donde *BASE* es la dirección de memoria del periférico en el mapa de memoria, 0x6700 en este caso), el segundo operando a la dirección *BASE + 02*, la señal *init* a la dirección *BASE + 04*, la señal *done* a la dirección *BASE + 06*, las parte alta y baja del resultado a las direcciones *BASE + 08* y *BASE + 0A* respectivamente y. En total 6 registros, por lo que solo se necesitarán las primeras 4 líneas del bus de direcciones (*io\_addr[3:0]*) para seleccionar a qué registro se quiere acceder.

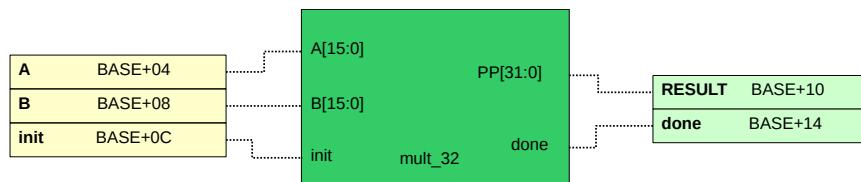


Figura 1.9 Mapa de memoria para el multiplicador

Una vez definida la dirección de memoria para la información de entrada y salida del periférico se debe adaptar el diseño para que permita el intercambio de información con la CPU. En la figura 1.10 se muestra el diagrama de bloques del periférico multiplicador, en él podemos identificar dos multiplexores conectados a los buses de datos (*dout* y *din*) encargados de seleccionar a qué señal se escribirá/leerá la señal desde/hacia la CPU.

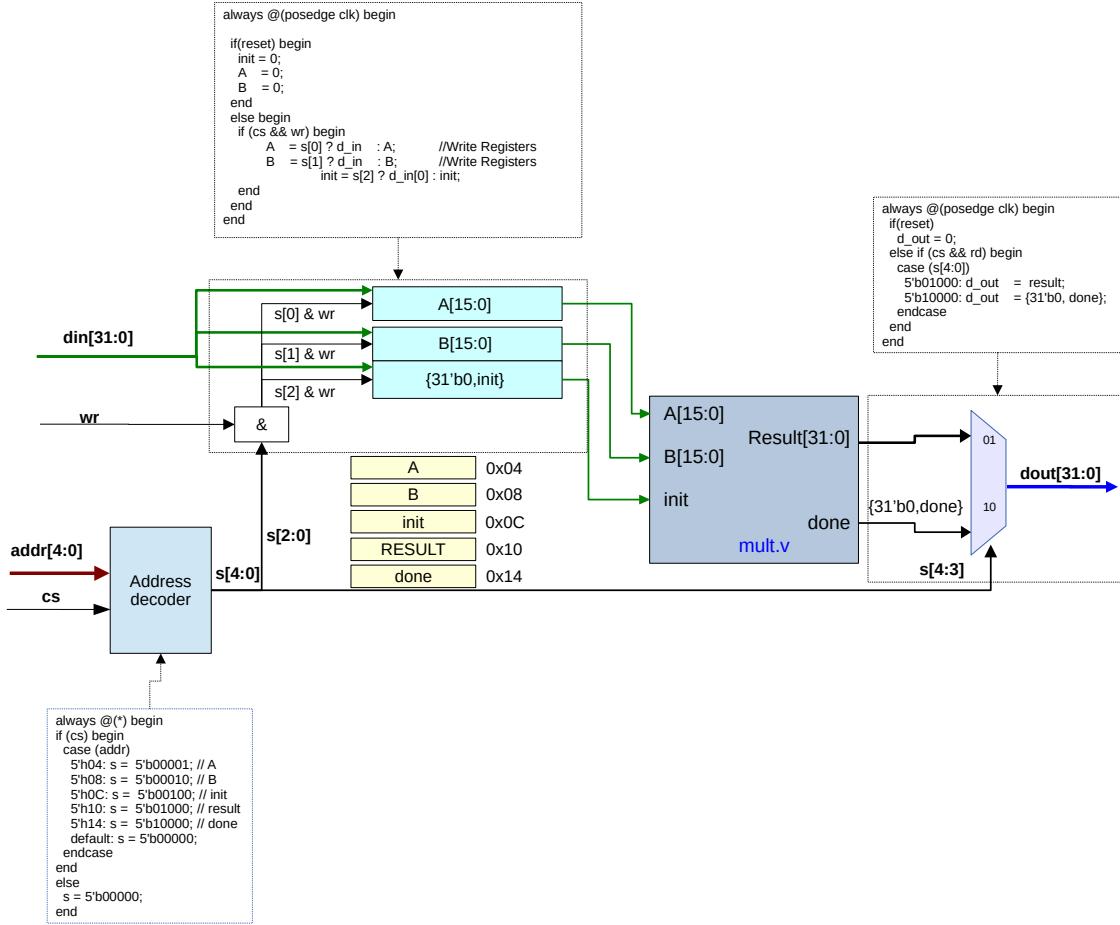


Figura 1.10 Diagrama de bloques del periférico multiplicador

### 1.3.1. Lectura del periférico

En la figura 1.11 se muestra el diagrama de bloques simplificado para la lectura. En ella podemos observar que la señal de control `s[5:3]` selecciona lo que se enviará por el bus de datos de salida `dout`, es decir, `PP[31:0]` o `done`, el decodificador de direcciones interno activa una de estas señales de acuerdo con el mapa de memoria del dispositivo tal como se indica en el código en verilog de la figura 1.11. Este decodificador es un bloque combinatorio que activa las señales de selección `s[5:3]` siempre y cuando las señales `cs` (periférico seleccionado) y `rd` (señal de lectura del procesador) estén activas.

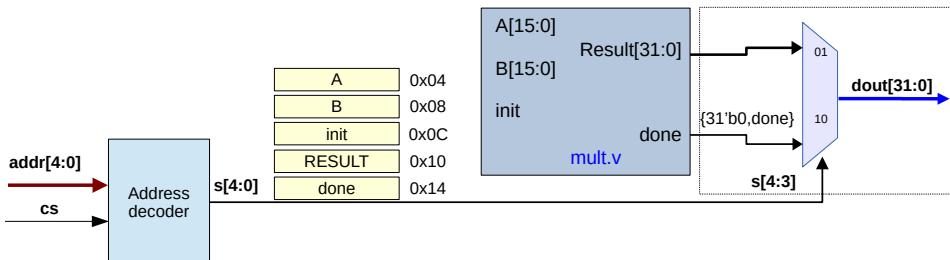
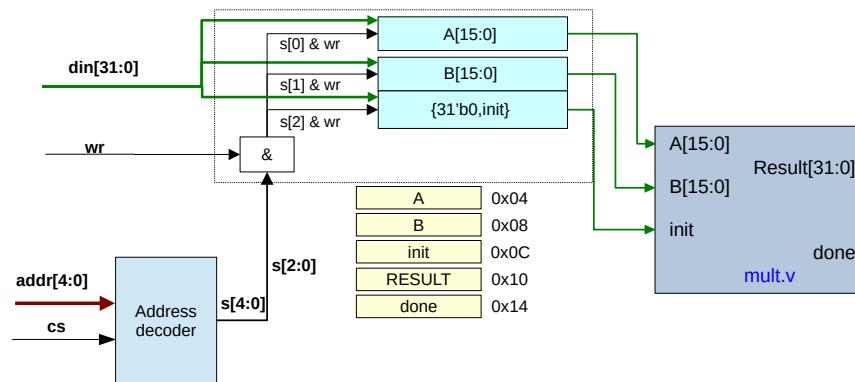


Figura 1.11 Diagrama de bloques para la lectura del multiplicador

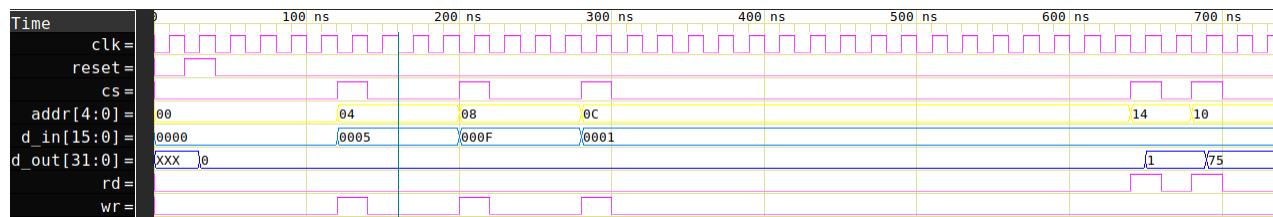
### 1.3.2. Escritura del periférico

En la figura 1.12 se muestra el diagrama de bloques simplificado para la escritura; de forma similar a la lectura de datos desde la CPU a los periféricos, el bus de datos de entrada (*din*) está conectado a todos los registros *A*, *B* e *init*, el codificador de direcciones controla la activación de dichos registros; en este caso se utilizan registros ya que es necesario mantener los operandos en el valor deseado durante la ejecución del algoritmo de multiplicación. Se muestra el código del módulo combinatorio que realiza la decodificación de las direcciones de escritura el cual activa las señales *s[0:2]* siempre y cuando las señales *cs* y *wr* estén activas lo que indica un ciclo de escritura al periférico.



**Figura 1.12** Diagrama de bloques para la escritura del multiplicador

En la figura 1.13 podemos observar las formas de onda de un ciclo completo de operación del multiplicador, en este ejemplo realizamos la multiplicación  $0x5 * 0xF$  (5 \* 16), el bus de datos de entrada al periférico se resaltan en azul, las direcciones en amarillo; aquí podemos observar que cuando las señales *cs* = 1 y *wr* = 1, el bus de direcciones toma el valor **0x04** y el de datos **0x0005** indicando la escritura del primer operando; en el siguiente ciclo de escritura el bus de direcciones toma el valor de **0x08** y el de datos **0x0F** con lo que se carga el segundo operando, en la tercera operación de escritura el bus de direcciones es **0x0C** y el dato 1, con lo que se inicia el algoritmo de multiplicación con estos dos operandos, en este momento inicia la ejecución del algoritmo de multiplicación (lo que demora undeterminado número de ciclos de reloj); al finalizar debemos leer el estado del flag **done** en la dirección de memoria **0x14** (*cs* = *rd* = 1) si su valor es igual a 1 se realiza la lectura (*cs* = *rd* = 1) del resultado en la dirección de memoria **0x10**, que en este caso es el valor 75 (0x4B).



**Figura 1.13** Simulación de la multiplicación  $0x5 * 0xF$

### 1.3.3. Implementación de la funcionalidad del periférico usando Lógica de transferencia de registros

La lógica de transferencia de registros es una descripción comportamental a nivel funcional, que puede ser sintetizada a una descripción estructural utilizando registros, una unidad que realiza operaciones aritméticas/lógicas y una máquina

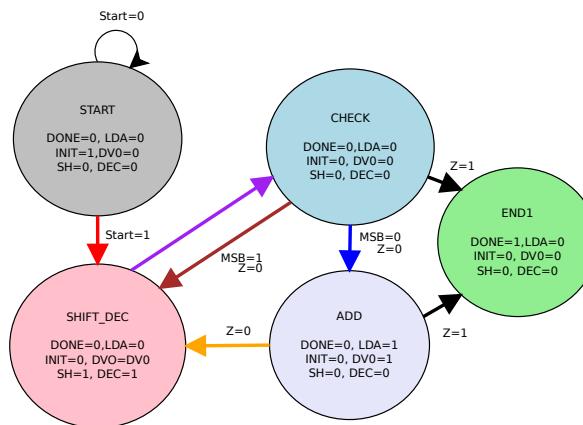
de control que maneja la transferencia o envío de datos de un registro a otro. La secuencia o flujo de datos al realizar ese proceso de transferencia: registros, unidad aritmético-lógica, registros; es lo que permite que se generen los algoritmos. Así, todo sistema secuencial (algoritmo) puede ser visto como la unión de un camino de datos (registros y operaciones) y una unidad de control que determina cuándo y cuáles operaciones se deben realizar, al igual que las transferencia o movimiento de datos entre registros.

En este nivel de descripción los diseñadores abordan la solución de un problema a nivel de operaciones y de transferencia de datos entre registros, lo cual ocurre durante muchos ciclos de reloj [REF GAJSKI BUSCAR]. En este nivel de abstracción no se aborda el problema desde la perspectiva de las dimensiones geométricas o conexiones físicas concretas del sistema, sino que se aborda desde el nivel comportamental desarrollado por cada uno de los pasos que permiten realizar esa manipulación de datos almacenados en registros, hasta la ejecución de operaciones aritmético lógicas; finalizando nuevamente en registros en donde se van guardando los resultados. Es este el punto de partida para abordar el desarrollo estructurado de soluciones en hardware, que para problemas de mediana y baja complejidad hacen que sea una solución acertada.

Una primera alternativa para llegar a diseñar un sistema digital nace de las propias necesidades que entregan las especificaciones del problema, en donde se busca realizar la integración de diferentes bloques, elementos de librería o periféricos (p.e decodificadores, multiplexores, sumadores, restadores, contadores, registros de desplazamiento) que conlleven a tener una arquitectura del sistema. Esta arquitectura puede llegar a tener elementos netamente combinacionales (el valor de las salidas depende únicamente del valor de las entradas), o secuenciales (el valor de las salidas depende de su valor actual y del valor de las entradas), o la combinación de ambos.

En el caso más frecuente de sistemas digitales como lo es el sincrónico, se requiere de elementos secuenciales que siempre están gobernados bajo la misma señal de reloj, la cual tiene una frecuencia de operación muy superior a cualquiera de las frecuencias de operación de las demás señales de entrada del sistema. Adicionalmente uno de los métodos más conocidos para el diseño de circuitos digitales de cierta complejidad se basa en la utilización de máquinas de estado finito “FSM” (en inglés Finite State Machine), a donde llegan todas las señales de control las cuales controlan o manipulan los demás bloques o elementos de librería que componen dicha arquitectura.

Las máquinas de estado finito utilizan una representación (1.14) por medio de círculos los cuales representan los estados o momentos de operación en determinado instante de tiempo, definido por condiciones de operación que controlan los demás bloques en dicho instante y flechas que interconectan los diferentes estados, las cuales representan las condiciones de cambio entre estados o instantes de tiempo.

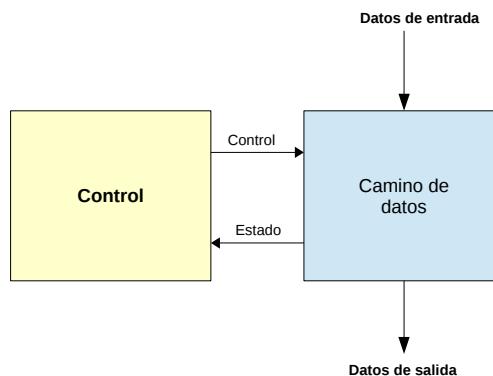


**Figura 1.14** Diagrama de estados de una máquina de estado finito

En el caso de máquinas de estados complejas con gran cantidad de estados y señales, la representación con círculos y flechas puede llegar a ser confusa y muy larga. Es por esto que se hace necesario acompañar siempre cada descripción y de hecho comenzar la etapa de diseño utilizando los algoritmos que permitan manipular con rigor y consistencia las operaciones matemáticas necesarias para la ejecución de los diversos cálculos implicados. Para esto se suelen emplear grafos que muestren dependencia entre operandos y flujos que comunican los datos del algoritmo. A este tipo de notación se le conoce como máquinas de estado algorítmicas, en las cuales se tiene una representación gráfica intuitiva de las operaciones del circuito en cada ciclo de reloj.

### Máquinas de Estados Algorítmicas (ASM)

La lógica de transferencia de registros es una técnica ampliamente utilizada para la descripción a alto nivel del comportamiento de circuitos secuenciales; los cuales pueden ser vistos como un grupo de registros y operaciones aritméticas y/o lógicas que transfieren datos de un registro a otro. Todo sistema secuencial puede ser visto como la unión de un camino de datos (entre registros y la unidad aritmético lógica) y una unidad de control que determina en qué momento se deben realizar las operaciones y las transferencias de los datos entre registros. En la Figura 1.15 se puede ver la estructura fundamental de una máquina de estados algorítmica. Nótese que los datos nunca pasan por la unidad de control y que la comunicación entre el bloque de Control y el bloque Camino de datos se lleva a cabo por señales simples que deciden qué operación se debe realizar (señal control) y señales que dan información al Bloque de control sobre el estado de la operación, lo que llamaremos banderas. Estas señales dan información al control como por ejemplo si el resultado de una operación da un número negativo, el resultado da cero, hay overflow o se genera un acarreo al realizarse la operación entre otras.



**Figura 1.15** Estructura de una máquina de estados algorítmica.

A la unión de una unidad de control y un camino de datos le daremos el nombre de máquina de estados algorítmica (ASM), para diferenciarla de la máquina de estados finitos y para indicar la posibilidad de implementación de cualquier tipo de algoritmo. Los pasos que se realizan para el diseño e implementación de una máquina de estados algorítmica son los siguientes:

1. Elaboración de un diagrama de flujo que describa la funcionalidad deseada ya sea a nivel gráfico o en texto.
2. Identificación de los componentes del camino de datos.
3. Identificación de las señales necesarias para controlar el camino de datos e interconexión.
4. Especificación de la unidad de control utilizando diagramas de estado.
5. Implementación de los componentes del camino de datos y de la unidad de control utilizando lenguajes de descripción de hardware.
6. Simulación y pruebas.

### ASM para la multiplicación de números binarios

El algoritmo de multiplicación que se implementará se basa en productos parciales (PP); el primer producto parcial siempre es cero (ver Figura 1.16), se realiza la multiplicación iniciando con el bit menos significativo (bit de la derecha) del multiplicador, el resultado de la multiplicación se suma al primer producto parcial y se obtiene el segundo producto parcial; si el bit del multiplicador es '0' no se afecta el contenido de PP, por lo que no se realiza la suma. A continuación se realiza la multiplicación del siguiente bit (a la izquierda del LSB) y el resultado se suma al producto parcial pero corrido un bit a la izquierda, esto para indicar que la potencia del siguiente bit tiene un grado más; este corrimiento se debe realizar ya que si un número binario se multiplica por 2 el resultado es el mismo número corrido a la izquierda; este proceso continúa hasta completar todos los bits del multiplicador y el último producto parcial es el resultado final.

$$\begin{array}{r}
 1010 \\
 * 0101 \\
 \hline
 0000 \quad \leftarrow \text{Primer producto parcial} \\
 1010 \\
 \hline
 1010 \quad \leftarrow \text{Segundo producto parcial} \\
 0000 \\
 \hline
 01010 \quad \leftarrow \text{Tercer producto parcial} \\
 1010 \\
 \hline
 110010 \quad \leftarrow \text{Cuarto producto parcial} \\
 0000 \\
 \hline
 110010 \quad \leftarrow \text{Resultado}
 \end{array}$$

**Figura 1.16** Multiplicación de números binarios usando productos parciales.

### Descripción comportamental a nivel algoritmo

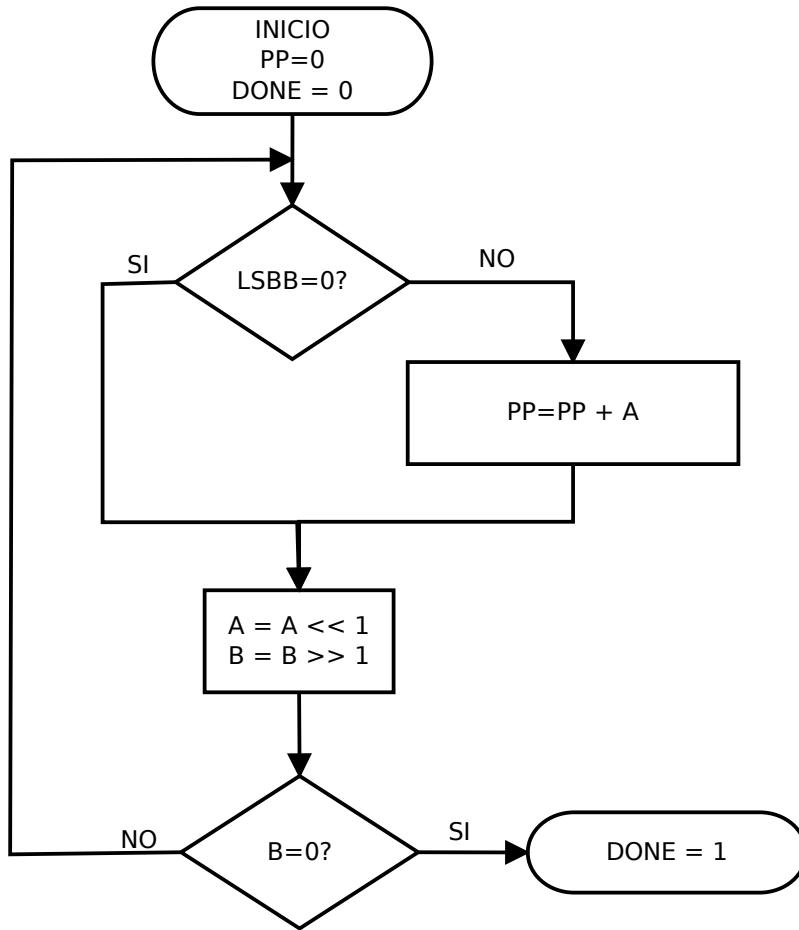
En la Figura 1.17 se muestra un diagrama de flujo del algoritmo que implementa la multiplicación por productos parciales descrita anteriormente. El primer paso para realizar la multiplicación es hacer que el producto parcial (PP) sea igual a cero, a continuación se realiza una verificación del bit menos significativo del multiplicador, esto se hace para sumar únicamente los resultados que no son parcialmente cero. En este caso se utiliza un corrimiento a la izquierda para obtener el siguiente bit del multiplicador, si por ejemplo al número *1010* se le realiza un corrimiento a la derecha se obtiene el número *0101*, con lo que el bit menos significativo corresponde al segundo bit de *1010*, si se realiza otro corrimiento a la derecha se obtiene *0010* y de nuevo el bit menos significativo corresponde al tercer bit de *1010*, al realizar de nuevo un corrimiento se obtiene *0001*, con lo que tendríamos todas las cifras del multiplicador de forma consecutiva en el bit menos significativo. Cuando se realiza un nuevo corrimiento el resultado es *0000* lo que indica que el producto parcial no puede cambiar y podemos terminar el algoritmo. Este método para finalizar el algoritmo produce que el número de iteraciones depende del valor del multiplicador; otra forma de terminar el algoritmo sin que dependa del valor del multiplicador se obtiene al contar el número de bits del multiplicador y realizar el corrimiento *n* veces, donde *n* es el número de bits del multiplicador.

Para indicar que cada vez que se toma un bit del multiplicador, este tiene una potencia mayor que el bit anterior, debemos multiplicar el resultado por la base, la cual es 2 en este caso; como se mencionó anteriormente, multiplicar por 2 equivale a realizar un corrimiento a la izquierda, por lo que siempre que se tome un nuevo bit del multiplicador debemos correr a la izquierda el multiplicando.

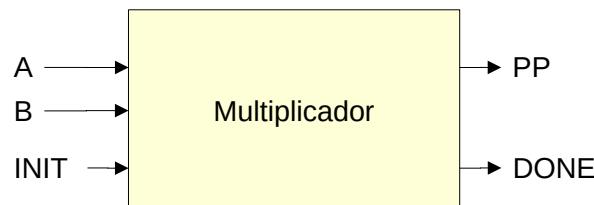
Una vez conocido el funcionamiento del sistema se procede a realizar el diagrama de caja negra de entradas y salidas. En la Figura 1.18 se muestra el multiplicando y el multiplicador (A y B), señales de *m* bits cada una, el resultado de la multiplicación PP (Bus de *2m* Bits), la señal de reloj (CLOCK). Las señales INIT y DONE se utilizan para iniciar el proceso de multiplicación e indicar que el resultado está disponible respectivamente; es importante que todo sistema digital posea la forma de interactuar con el exterior, ya que sin ello el sistema carecería de utilidad.

### Identificación de los componentes del camino de datos

Para identificar los componentes del camino de datos, se recorre el algoritmo para encontrar las operaciones que se realizan, el resultado se aprecia en la figura 1.19, en ella se resaltan las operaciones que se deben realizar para la correcta operación del algoritmo; la primera es una operación de acumulación correspondiente a  $PP = PP + A$ ; la segunda operación que encontramos son los dos corrimientos a la izquierda y derecha del multiplicando (A) y el multiplicador (B) respectivamente, el último módulo es un comparador que indica que el multiplicador es igual a cero, indicando que el algoritmo puede finalizar.



**Figura 1.17** Diagrama de flujo para la multiplicación de números binarios.

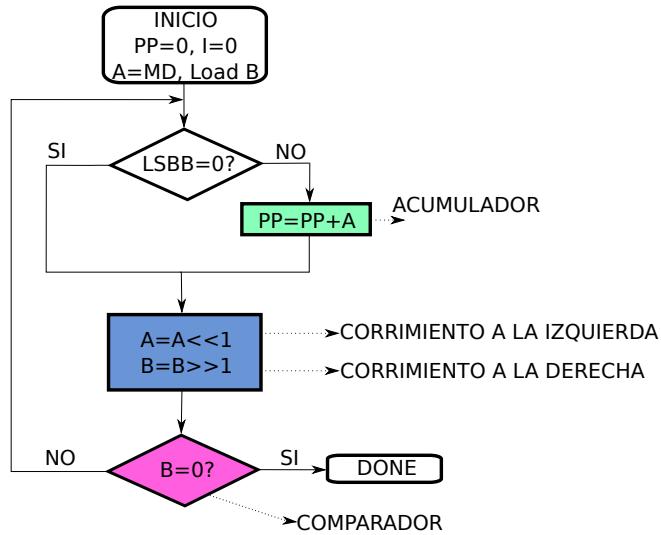


**Figura 1.18** Diagrama de caja negra para el multiplicador de números binarios.

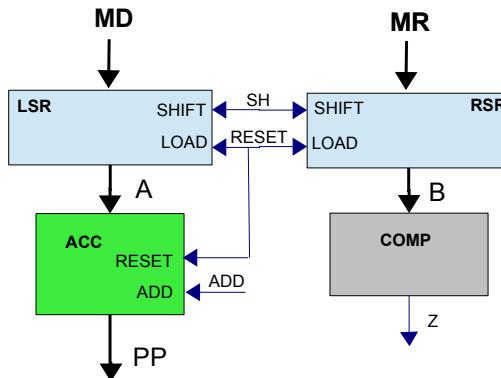
### Identificación de las señales de control e interconexión del camino de datos

En la figura 1.20 se muestra la interconexión de los componentes del camino de datos y las señales que lo controlan. La primera operación que aparece en el diagrama de flujo es la del acumulador ( $PP = PP + A$ ), la cual guarda el valor de la salida del registro de corrimiento que contiene al multiplicando ( $A$ , MD), por lo que debe existir una conexión entre el registro de corrimiento (LSR) a la izquierda y el acumulador (ACC). La segunda operación que aparece es la de los registros de corrimiento, por lo que los valores del multiplicando y multiplicador deben cargarse para su posterior corrimiento a las unidades de corrimiento a la izquierda y derecha respectivamente. La salida del corrimiento a la derecha del multiplicador es comparada en cada ciclo para determinar si se llegó al final del algoritmo, por lo que la entrada del comparador es la salida del registro de corrimiento del multiplicador.

Para determinar las señales de control de cada componente del camino de datos, se debe identificar su función y las operaciones que debe realizar; los registros de corrimiento deben permitir la carga de un valor inicial y el corrimiento de las mismas, esto se realiza con las señales *LOAD* y *SHIFT* respectivamente; el acumulador debe tener la posibilidad



**Figura 1.19** Identificación de los componentes del camino de datos para el multiplicador de números binarios.



**Figura 1.20** Identificación de las señales de control e interconexión del camino de datos.

de inicializarse en cero y una señal para que sume el valor de la entrada al que tiene almacenado, esto se hace con las señales *RESET* y *ADD*; por último el comparador debe proporcionar una señal que indique que el valor de su entrada es igual a cero, *Z* en este caso.

Aunque es posible que la máquina de control maneje todas las señales de control del camino de datos, es mejor agruparlas de acuerdo a su activación; esto es, si una operación se activa al mismo tiempo que otra, se puede utilizar una señal que las controle a ambas; para esto, se utiliza el diagrama de flujo y se observa en qué momento se realizan las operaciones. En el diagrama de flujo se observa que se cargan los valores de los registros de corrimiento y se inicializa en cero el acumulador al comenzar el algoritmo y durante la ejecución del mismo no se vuelve a realizar esta operación, por este motivo utilizaremos la misma señal (*RESET*) para realizar estas operaciones; la señal que controla el momento en que el acumulador se incrementa es única, ya que no se realiza otra operación en ese lugar del algoritmo y recibe el nombre de *ADD*; las operaciones de corrimiento se realizan en el mismo sitio, por lo que se puede utilizar una señal común, a la que llamaremos *SH*; por último la salida del comparador *Z* y el bit menos significativo de *B* *LSB* son señales de salida del camino de datos que le darán a la unidad de control la información necesaria para tomar la acción adecuada en los bloques de decisión.

### Especificación de la unidad de control utilizando diagramas de estado

Una vez que se conoce el camino de datos, las señales que lo controlan y las señales que ayudarán a la unidad de control a tomar decisiones, se procede con la especificación de la unidad de control, la cual, se implementa con una

máquina de estados finitos, por lo que la mejor forma de especificarla es utilizando un diagrama de estados; en la figura 1.21 se muestra la relación entre el diagrama de flujo y el diagrama de estados.

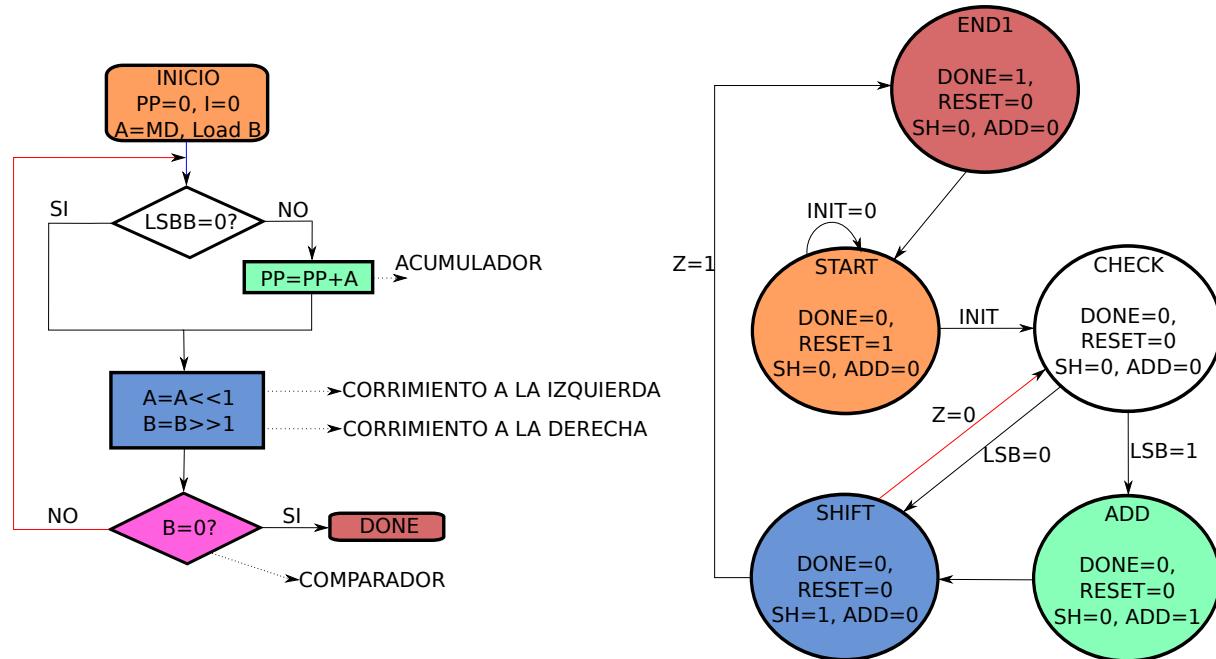


Figura 1.21 Diagrama de estados de la unidad de control del multiplicador binario.

Como se puede observar, existe una relación muy estrecha entre el diagrama de estados y el diagrama de flujo, cada operación del diagrama de flujo corresponde a un estado de la máquina de control y las transiciones entre ellos son idénticas, observe las líneas del mismo color en la figura 1.21

La máquina de estados debe iniciar en START y se queda en este estado siempre que la señal INIT tenga un valor de '0'. En el estado START la señal RESET = '1', con lo que el valor del acumulador se hace cero y se cargan los registros A y B. Cuando INIT = '1', entramos al estado CHECK el cual evalúa la señal LSB, si LSB = '0', no se debe realizar la suma de MD, pero si se debe realizar un corrimiento para obtener el siguiente bit del multiplicador y realizar el corrimiento necesario en MD. Si LSB = '1' se debe sumar el valor de las salidas de LSR al valor del acumulador, y después se debe realizar un corrimiento. En el estado ADD se hace la salida ADD = '1' para que el valor a la entrada del acumulador se sume al valor almacenado en él. En el estado SHIFT se realiza el corrimiento de RSR y LSR haciendo el valor de la señal SH = 1, es necesario declarar el valor de todas las salidas en todos los estados.

### Implementación de los componentes del camino de datos y de la unidad de control

Existe abundante literatura sobre el uso de lenguajes de descripción de hardware para la implementación de sistemas digitales; por este motivo, en este libro no se realizará un estudio profundo de dichos lenguajes, sin embargo, se proporcionarán archivos con las implementaciones para que puedan ser utilizadas como base en los cursos desarrollados alrededor de este texto.

Es muy importante anotar la importancia de la portabilidad del código, como es bien sabido, existen varias empresas que suministran entornos de desarrollo que permiten la entrada de diseño utilizando diferentes medios; las herramientas gráficas utilizadas por ellos no son compatibles entre sí, lo que hace imposible el paso de un diseño implementado en una herramienta gráfica a otra de otro fabricante; sin embargo, todas las herramientas aceptan texto con el estándar del lenguaje; por esto, se recomienda utilizar únicamente entrada de texto en las descripciones.

En la figura 1.22 se muestra el código en verilog para el multiplicador, se muestra la forma de instanciar diferentes módulos y el código para probar el diseño (*testbench*).

```

module mult_32(clk ,rst ,init ,A ,B ,pp ,done);
input rst;
input clk;
input init;
input [15:0]A;
input [15:0]B;
output [31:0]pp;
output done;

wire w_Sh;
wire w_Reset;
wire w_Add;
wire w_Z;

wire [31:0]w_A;
wire [15:0] w_B;

rsr rsr0 (.clk(clk),.in_B(B),.shift(w_sh),.load(w_reset),.s_B(w_B));
lsr lsr0 (.clk(clk),.in_A(A),.shift(w_sh),.load(w_reset),.s_A(w_A));
comp comp0(B(w_B),z(w_z));
acc acc0 (.clk(clk),.A(w_A),.add(w_add),.rst(w_reset),.pp(pp));
control mult control0 (.clk(clk),.rst(rst),.lsb_B(B(w_B)[0]),.init(init),.z(w_z).done(done),
.sh(w_sh),.reset(w_reset),.add(w_add));
endmodule

```

```

module rsr (clk, in_B, shift, load, s_B);
input clk;
input [15:0]in_B;
input load;
input shift;
output reg [15:0]s_B;
always @(posedge clk)
if(load)
s_B = in_B;
else
begin
if(shift) s_B = s_B >> 1;
else s_B = s_B;
end
endmodule

```

```

module lsr (clk , in_A , shift , load , s_A);
input clk;
input [15:0]in_A;
input load;
input shift;
output reg [30:0]s_A;
always @ (posedge clk)
if(load)
s_A = in_A;
else
begin
if(shift) s_A = s_A << 1;
else s_A = s_A;
end
endmodule

```

```

module comp(B, z);
input [15:0]B;
output z;
reg tmp;
initial tmp = 0;
assign z = tmp;
always@(*)
tmp = (B==0) ? 1'b1 : 1'b0;
endmodule

```

```

module acc (clk ,A, add, rst,
pp);
input clk;
input [31:0]A;
input add;
input rst;
output reg [31:0]pp;
initial pp = 0;
always @ (posedge clk)
if (rst)
pp = 32'h00000000;
else
begin
if (add) pp = pp + A;
else pp = pp;
end
endmodule

```

```

module control_mult( clk ,rst ,lsb_B ,init ,z ,done ,sh ,reset ,add );
input clk;
input rst;
input lsb_B;
input init;
input z;
output reg done;
output reg sh;
output reg reset;
output reg add;
parameter START = 3'b000;
parameter CHECK = 3'b001;
parameter SHIFT = 3'b010;
parameter ADD = 3'b011;
parameter END = 3'b100;
reg [2:0] state;
initial begin
done = 0;
sh = 0;
reset = 0;
add = 0;
state = 0;
end
reg [3:0] count;
always @ (posedge clk)
begin
case(state)
START:begin
done = 0;
sh = 0;
reset = 1;
add = 0;
end
CHECK:begin
done = 0;
sh = 1;
reset = 0;
add = 0;
end
SHIFT:begin
done = 0;
sh = 1;
reset = 0;
add = 0;
end
ADD:begin
done = 0;
sh = 0;
reset = 0;
add = 1;
end
END:begin
done = 1;
sh = 0;
reset = 0;
add = 0;
end
default:begin
done = 0;
sh = 0;
reset = 0;
add = 0;
end
endcase
end
endmodule

```

```

timescale 1ns / 1ps
`define SIMULATION
module mult_32_TB;
reg clk;
reg rst;
reg reset;
reg start;
reg [15:0]in_A;
reg [15:0]in_B;
wire [31:0]pp;
wire done;
mult_32 uut (.clk(clk),.rst(rst),.init(start),.A(in_A),.B(in_B),.pp(pp),.done(done));
parameter PERIOD = 20;
parameter real_DUTY_CYCLE = 0.5;
parameter OFFSET = 0;
reg [20:0] i;
event reset_trigger;
event reset_done_trigger;
initial begin
forever begin
@ (reset trigger);
@(posedge clk);
rst = 1;
@ (posedge clk);
rst = 0;
-> reset_done_trigger;
end
end
initial begin // Initialize Inputs
clk = 0; reset = 1; start = 0; in_A = 16'h0005; in_B = 16'h0003;
end
initial begin // Process for clk
#OFFSET;
forever begin
clk = 1'b0;
#(PERIOD*PERIOD*DUTY_CYCLE) clk = 1'b1;
#(PERIOD*DUTY_CYCLE);
end
end
initial begin // Reset the system, Start the image capture process
#10-> reset_trigger;
@ (reset_done trigger);
@(posedge clk);
start = 0;
@ (posedge clk);
start = 1;
for(i=0; i<2; i+=1) begin
@(posedge clk);
end
start = 0;
for(i=0; i<17; i+=1) begin
@(posedge clk);
end
end
initial begin: TEST_CASE
$dumpfile("mult_32_TB.vcd");
$dumpvars(1, uut);
#((PERIOD*DUTY_CYCLE)*120) $finish;
end
endmodule

```

Figura 1.22 Código verilog de la ASM multiplicador.

En la figura 1.23 se muestra el diagrama de bloques sintetizado.

## Simulación

Como se mencionó anteriormente, es posible realizar las simulaciones utilizando las herramientas gráficas de cada uno de los entornos de desarrollo que proporcionan los fabricantes de dispositivos lógicos programables, sin embargo, su uso dificulta la portabilidad del diseño. Por este motivo, se recomienda el uso de *testbench* escritos con el lenguaje estándar. Como parte del proceso de diseño, cada módulo debe ser simulado antes de ser integrado en la descripción de más alto nivel.

Los lenguajes de descripción de hardware permiten generar vectores de pruebas para ser aplicados a los módulos bajo prueba (**Unit Under Test**), para generar estos vectores se pueden utilizar instrucciones que no son sintetizables como el *for*, operaciones matemáticas, funciones trigonométricas, etc.

En la figura 1.24 se muestra el flujo de pruebas para la ASM multiplicador. En el archivo **mult\_32\_TB.v** se declaran las señales que generan los vectores de prueba (entradas de módulo: clk, rst, start, [15:0]in\_A, [15:0]in\_A), así como las que registrarán las salidas del módulo ante dichos vectores (salidas del módulo: [31:0] pp, done). Estas señales deben conectarse al módulo bajo prueba **mult\_32** al que se le dá un nombre arbitrario **uut**.

Las líneas resaltadas en color naranja, morado y azul general las formas de onda para las señales, **rst**, **clk** y **start** respectivamente.

El simulador aplica estas formas de onda a la unidad bajo prueba y almacena los resultados en el archivo **mult\_32\_TB.vcd**, este archivo puede ser visualizado con la herramienta *gtkwave*.

## 1.3 Ejemplo de implementación de una tarea usando un periférico

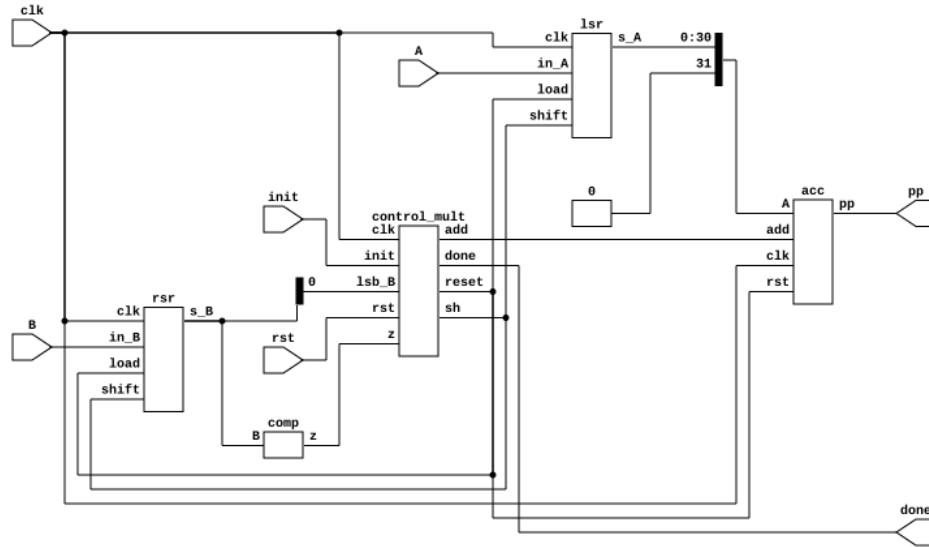


Figura 1.23 Diagrama RTL de la ASM multiplicador.

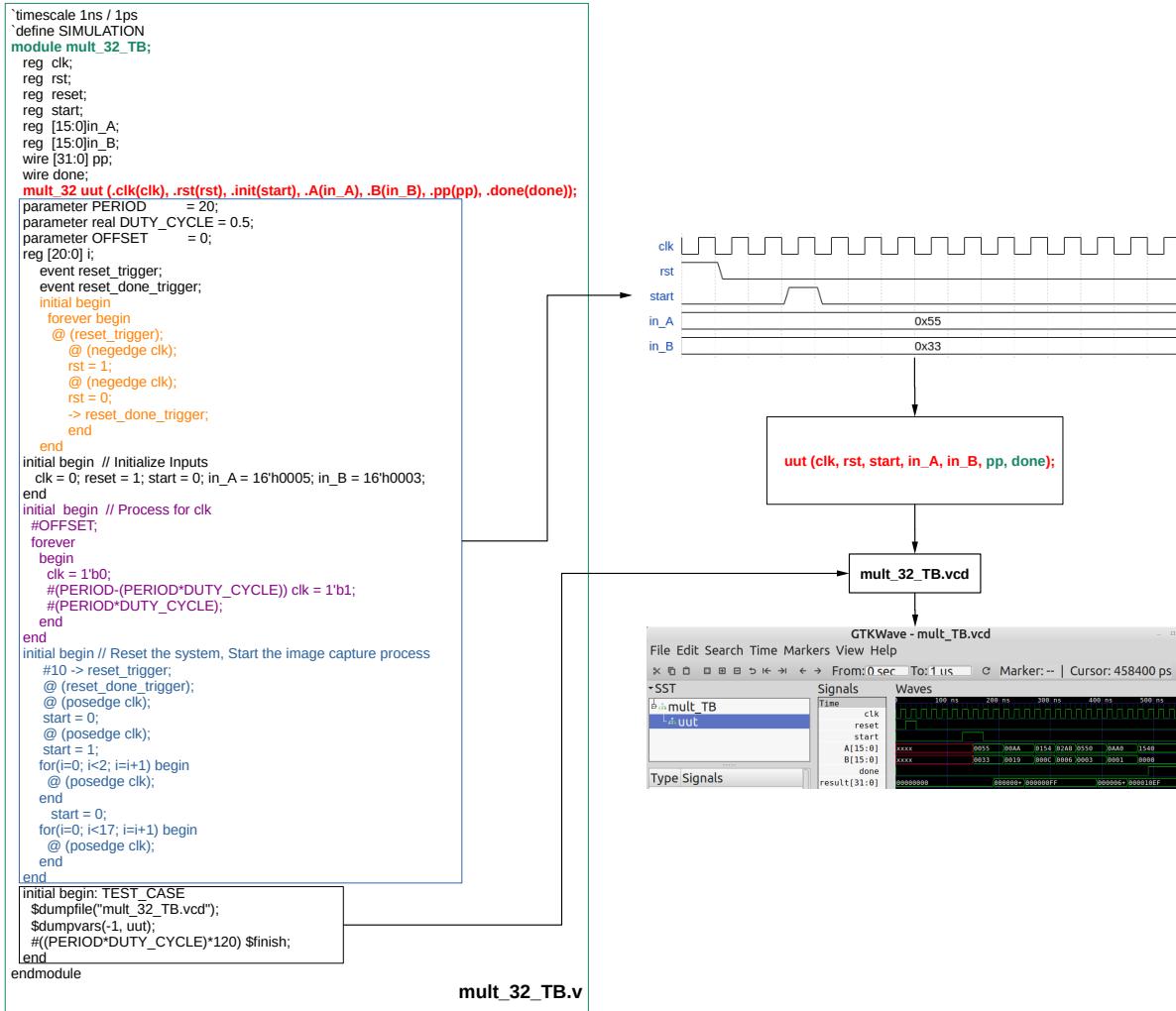
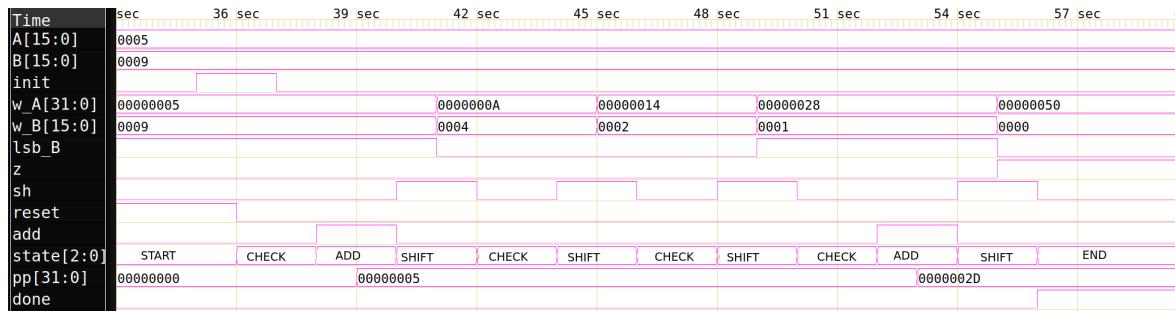


Figura 1.24 Simulación ASM multiplicador.

## Niveles de simulación

Es bueno tener claro los diferentes niveles de simulación que se pueden realizar a un sistema digital; la simulación más rápida es la que tiene en cuenta únicamente el lenguaje de descripción de hardware utilizado, es decir, simula el lenguaje sin realizar ningún proceso de síntesis (convertir HDL a compuertas lógicas); sin embargo, no es posible garantizar que los resultados del circuito sintetizado (en el peor de los casos no es posible sintetizar algunas descripciones) sean los mismos que la simulación del lenguaje; por esto, existe la simulación post-síntesis, en la que se simula el RTL (lógica de transferencia de registros) o las compuertas lógicas básicas obtenidas del proceso de síntesis, esta simulación garantiza que el circuito obtenido del proceso de síntesis se comporta como lo deseamos; el tercer nivel de simulación se obtiene cuando se adiciona un modelo de tiempo al diagrama de compuertas del nivel anterior, en este nivel, se tienen en cuenta las capacitancias de carga y la capacitancia de los caminos de interconexión para obtener el retardo de cada elemento del circuito, esta simulación es la más precisa y permite conocer la velocidad máxima a la que puede operar el sistema, esta simulación en algunos entornos de desarrollo recibe el nombre de *simulación post place & route*.

En la figura 1.25 se muestra el resultado de una simulación funcional utilizando la herramienta *icarus verilog*, en ella se muestran todas las señales involucradas en la máquina de estados algorítmica del multiplicador. En la gráfica se muestra la simulación de multiplicar 5 (0101) \* 9 (1001), la máquina de estados inicia en el estado (*state[2:0]*) *START* y continúa en este estado siempre que la señal *init* sea igual a 0, cuando la señal *init* = 1 se pasa al estado *check*, acá se revisa el LSB de B (1001), en la primera iteración el LSB = 1, por lo que se debe acumular el valor de A en el PP (0101) pasando al estado *ADD*, se realiza un corrimiento activando la señal *sh*, y se pasa al estado *check*, ahora el LSB de B (0100) es 0 por lo que no acumulamos y se realiza de nuevo un corrimiento y una revisión del LSB de B (0010), de nuevo se tiene un valor de 0 y no se realiza suma en el acumulador, al realizar el siguiente corrimiento el LSB de B (0001) es 1 lo que genera una acumulación de A (101000) en el PP; al realizar el siguiente corrimiento de B (0000) z = 1 indicando el fin del algoritmo, con lo que se pasa al estado *END* en el que la señal *DONE* = 1, el valor del PP es el resultado de la operación en este caso 0x2D (45).



**Figura 1.25** Simulación de la máquina de estados algorítmica del multiplicador binario.

En la figura 1.26 observamos los pasos que se deben seguir para realizar una simulación *post-synthesis* y *post-place and route* (proceso mediante el cual una herramienta implementa las funciones lógicas en el dispositivo lógico programable). Para la simulación post-síntesis se generan dos archivos con las interconexiones del circuito sintetizado en formatos JSON (.json) y Verilog (.v), estos archivos contienen la conexión de las compuertas lógicas (de la tecnología seleccionada) con las que la herramienta de síntesis implementa la funcionalidad deseada. La herramienta de simulación (iverilog) recibe como entrada el archivo en formato verilog junto con la descripción en verilog de las compuertas para la tecnología utilizada (ICE40\_SIM\_CELLS) y el archivo de pruebas (que es el mismo utilizado para la simulación funcional).

La simulación post-place&route se hace a partir de un archivo en verilog generado a partir de la salida de la herramienta que realiza el proceso de place&route, este archivo contiene la interconexión de los bloques internos del dispositivo, este archivo junto con el archivo que describe la tecnología es la entrada a la herramienta de simulación iverilog.

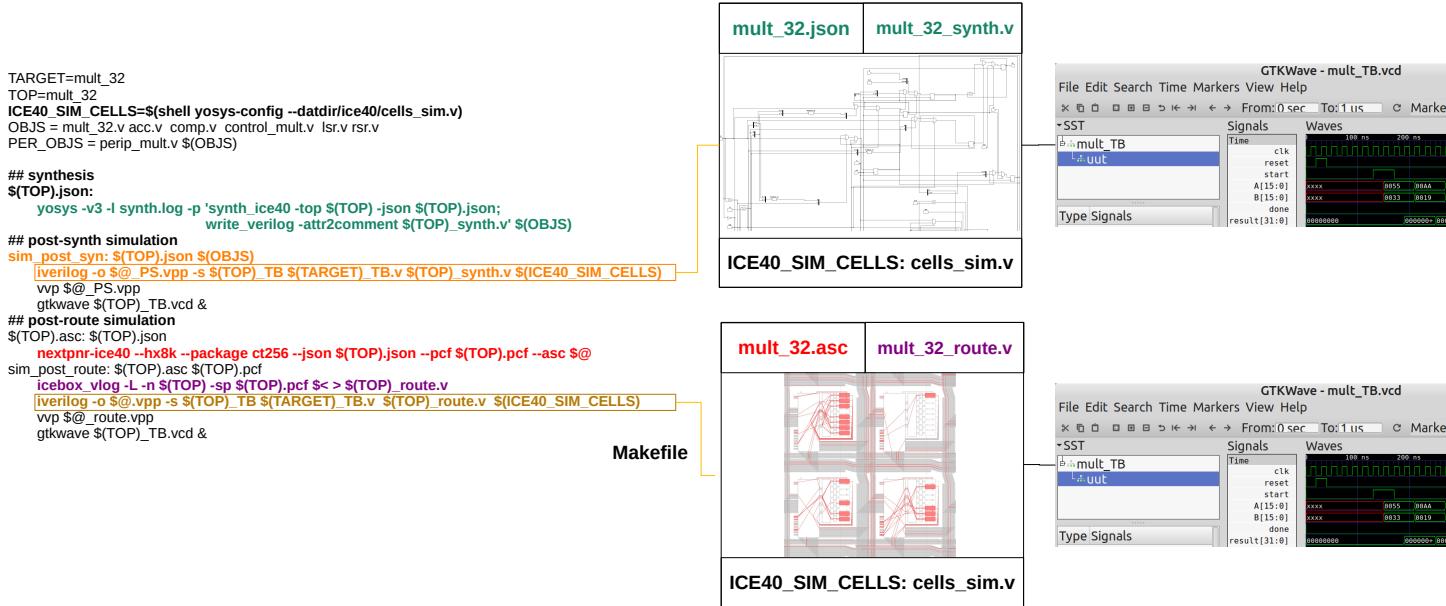


Figura 1.26 Simulación post-síntesis y post-place&route.

### 1.3.4. Otra forma de implementación del multiplicador

La forma de implementación mostrada en la subsección anterior, es muy eficiente, ya que el diseñador le indica a la herramienta de síntesis como debe construir los diferentes componentes y la forma de interconectarlos. Sin embargo, llegar a la forma del camino de datos óptima requiere un poco de experiencia, para evitar frustraciones en los estudiantes, es bueno, dejar que las herramientas de síntesis (que dicho sea de paso, han avanzado mucho en la optimización de recursos) realicen parte del trabajo; para esto, se pueden declarar las operaciones que se deben realizar dentro de cada estado de la máquina de estados, tal como se muestra en la figura 1.27

## 1.4. Implementación de un divisor de n bits sin signo

El proceso de división de números binarios es similar al de números decimales: Inicialmente se separan dígitos del Dividendo de izquierda a derecha hasta que la cifra así formada sea mayor o igual que el divisor. En la figura 1.28 separamos el primer dígito de la derecha (0) y le restamos el divisor (la operación de resta se realizó en complemento a dos), el resultado de esta operación es un número negativo (los números negativos en representación complemento a dos comienzan por 1). Esto indica que el número es menor que el divisor, por lo tanto, colocamos un cero en el resultado parcial de la división (este dígito será el más significativo) y separamos los dos primeros dígitos (00) y repetimos el proceso.

Sólo hasta el sexto resultado parcial obtenemos un cero en la primera cifra de la resta (recuerde que en complemento a dos los números tienen una longitud fija en nuestro caso 4 bits, si una operación provoca un bit adicional este se descarta, los bits descartados se encerraron en líneas punteadas en la Figura 1.28), lo que indica que el número es mayor o igual que el divisor, en este caso, se coloca un '1' en el resultado parcial y se conserva el valor de la operación de resta, el cual se convierte en el nuevo residuo parcial, este proceso se repite hasta haber "bajado" todos los dígitos del dividendo.

```

module mult (
    input      reset, clk, init,
    output reg   done,
    output reg [31:0] result,
    input  [15:0] op_A,
    input  [15:0] op_B
);

parameter START = 3'b000; parameter CHECK = 3'b001;
parameter SHIFT = 3'b010; parameter ADD_ = 3'b011;
parameter END_ = 3'b100; parameter START1 = 3'b101;

reg [2:0] state; reg [15:0] A; reg [15:0] B; reg [4:0] count;

always @ (posedge clk or posedge reset)
begin
    if (reset) begin
        done <= 0; result <= 0;
        state = START;
    end else begin
        case(state)
            START: begin
                count = 0; done <= 0; result = 0;
                if (init)
                    state = START1;
                else
                    state = START;
            end

            START1: begin
                A <= op_A; B <= op_B; done <= 0; result = 0;
                state = CHECK;
            end

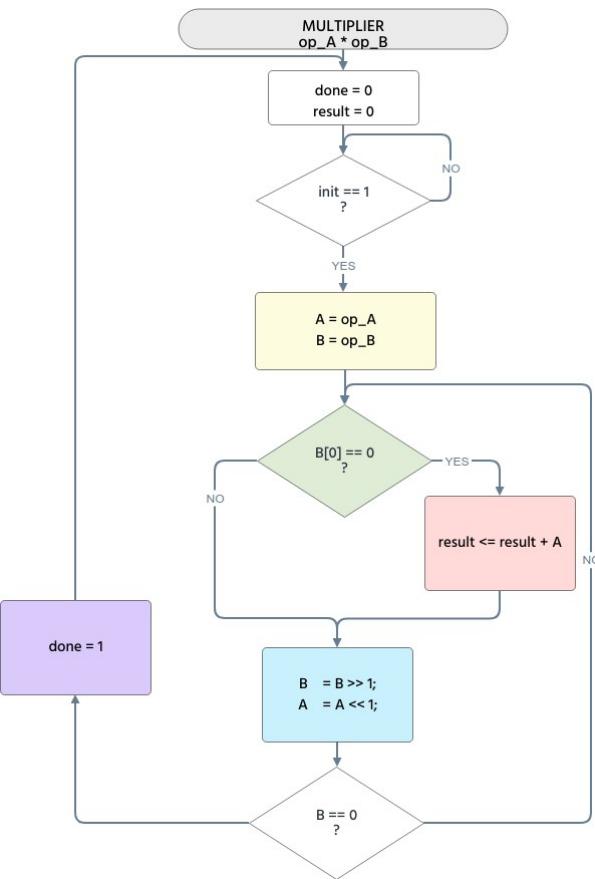
            CHECK: begin
                if (B[0])
                    state = ADD_;
                else
                    state = SHIFT;
            end

            SHIFT: begin
                B = B >> 1; A = A << 1;
                done = 0;
                if (B == 0)
                    state = END_;
                else
                    state = CHECK;
            end

            ADD_: begin
                result <= result + A; done = 0;
                state = SHIFT;
            end

            END_: begin
                done = 1;
                count = count + 1;
                state = (count > 29) ? START : END_;
            end
        endcase
    end
endmodule

```

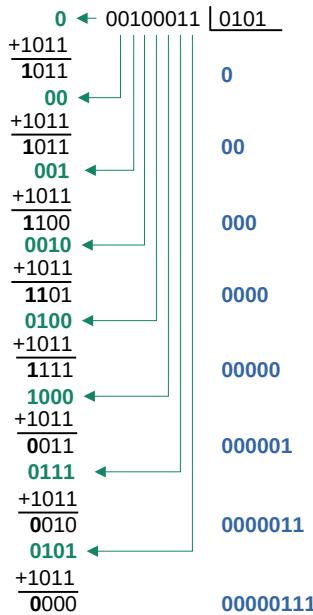
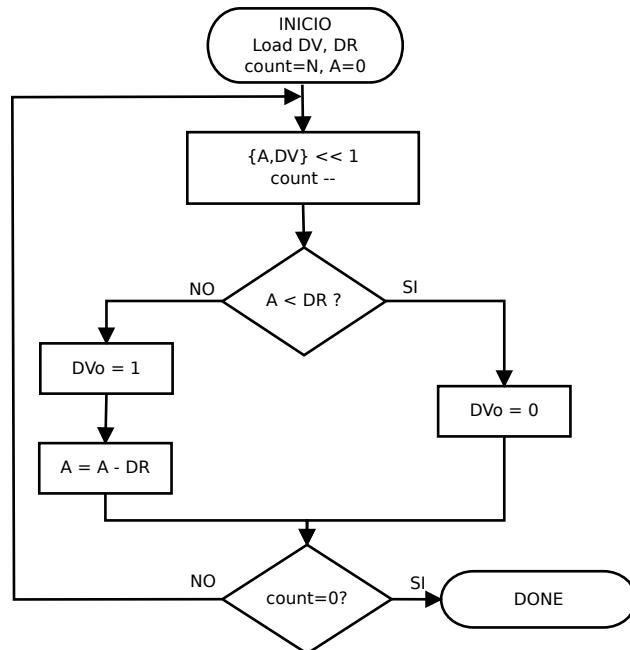


**Figura 1.27** Implementación del multiplicador simplificando el camino de datos en la máquina de estados.

#### 1.4.1. Identificación de componentes del camino de datos e interconexión

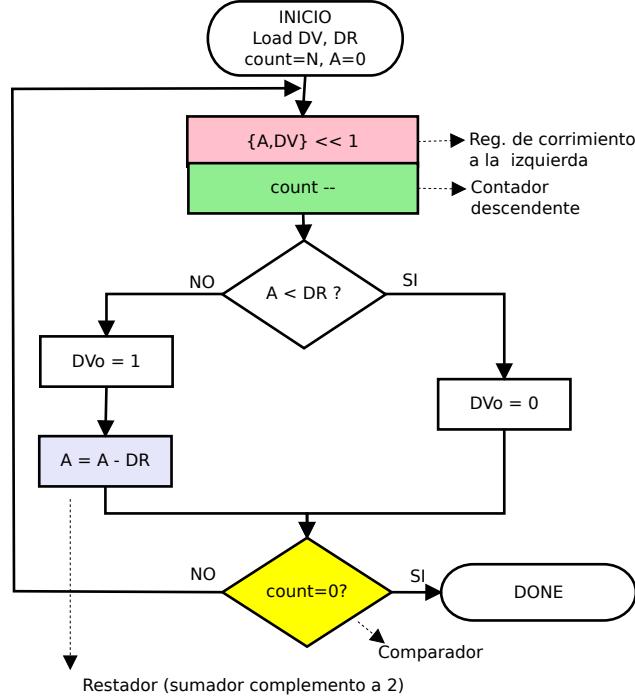
En la figura 1.30 podemos observar cómo se obtienen los componentes del camino de datos a partir del diagrama de flujo del divisor; se necesita un registro de corrimiento a la izquierda donde se almacena el Dividendo (DV) y las cifras que se van separando (A), un contador que cuente el número de bits que se han “bajado”, un restador (sumador en complemento a 2) para determinar si el número separado del dividendo “cabe” en el divisor (observando el bit más significativo MSB), y un comparador que indique que el valor del contador llegó a cero.

En la figura 1.31 se muestra la interconexión de los elementos del camino de datos y se muestran las señales de control. De nuevo, las señales que se activan en el mismo punto del diagrama de flujo pueden agruparse, por esto, la señal de inicialización del registro A, la carga de DV y la inicialización del contador se realizará con la señal INIT; el registro de desplazamiento a la izquierda va almacenando el resultado de la división a medida que se van utilizando los bits más significativos del dividendo, con esto se reduce el número de componentes, la señal DV0 ayuda a formar el resultado; la señal SH realiza el corrimiento a la izquierda del registro {A,DV} con lo que en A queda el número que se va separando del dividendo y en DV el resultado de la división; la señal LDA carga el resultado de la resta entre A y el divisor únicamente cuando el resultado de la resta es positivo, esto es cuando la señal MSB es igual a 1; la señal

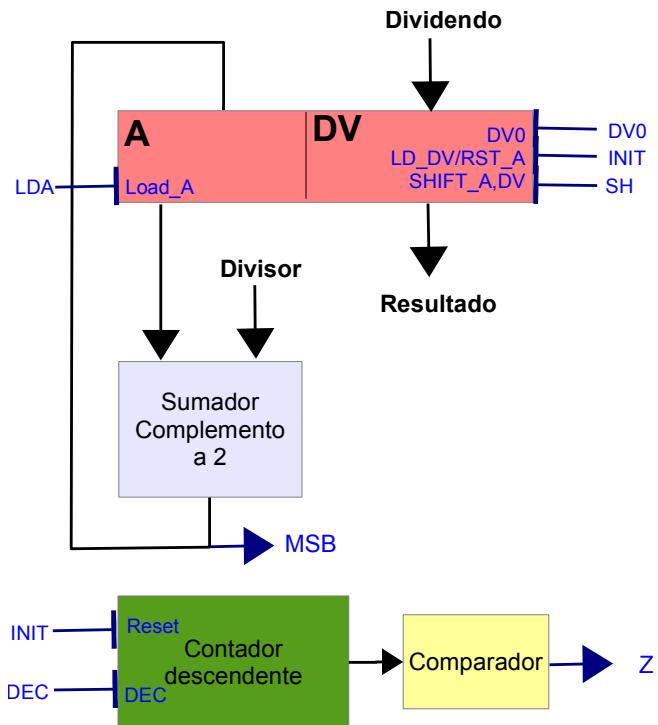
**Figura 1.28** División de numeros binarios.**Figura 1.29** Algoritmo para la división de números binarios.

*DEC* hace que el valor del contador disminuya en 1, y la salida *Z* se hace 1 cuando el valor de este contador llega a cero indicando que el algoritmo terminó.

De lo anterior tenemos que la unidad de control tiene como entradas las señales: *Reset*, *Start*, *MSB* y *Z*; y como salidas: *INIT*, *DV0*, *SH*, *DEC* y *LDA*; de nuevo los bloques de decisión del diagrama de flujo del algoritmo hacen referencia a las entradas de la unidad de control.



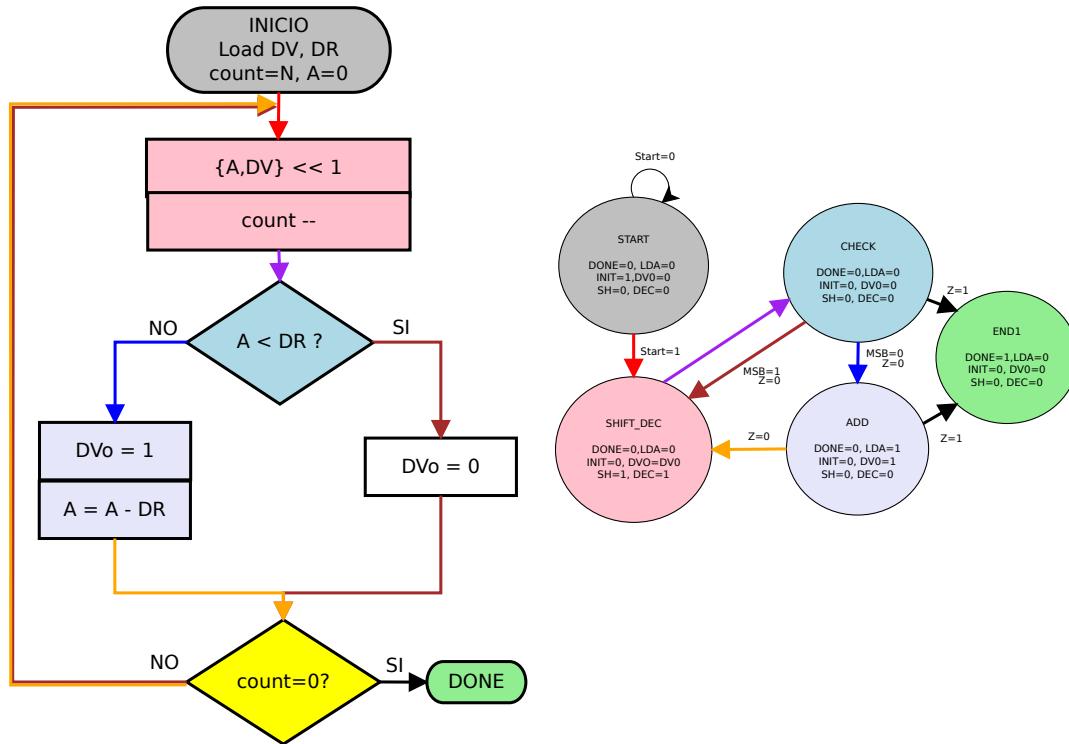
**Figura 1.30** Identificación de componentes del camino de datos para la división de números binarios.



**Figura 1.31** Interconexión del camino de datos para la división de números binarios.

### 1.4.2. Unidad de control

En la Figura 1.32 se muestra la relación existente entre el diagrama de flujo y el diagrama de estados de la unidad de control del divisor binario.



**Figura 1.32** Diagrama de estados de la unidad de control para la división de números binarios.

### 1.4.3. Diagrama de bloques del divisor

En la figura 1.33 se muestra el diagrama de bloques del divisor; su arquitectura es muy similar a la del multiplicador ya que tiene prácticamente las mismas señales, y requieren la misma información para su operación.

Los periféricos en general poseen la misma arquitectura, las variables son almacenadas en registros, multiplexores en los buses de entrada y salida de datos controlan en donde se almacena o de donde proviene la información que llega o sale a la CPU. Esta arquitectura posee un decodificador de direcciones interno que controla los multiplexores de entrada y salida.

### 1.4.4. Implementación del algoritmo de división usando solo una máquina de estados

En la figura 1.34 se muestra la implementación del divisor utilizando una máquina de estados finitos donde se implementan las tareas del camino de datos.

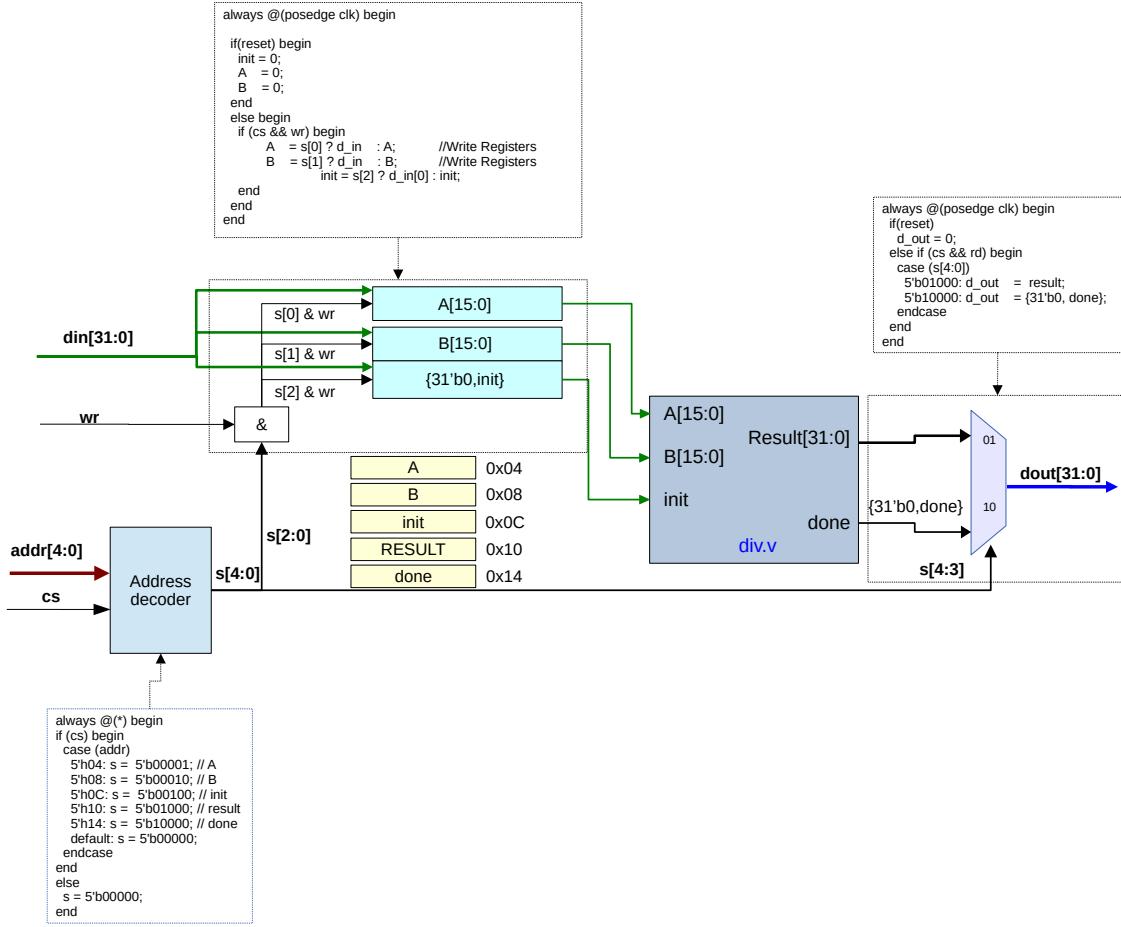


Figura 1.33 Diagrama de bloques del divisor

#### 1.4.5. Simulacion del divisor

En la figura 1.35 se muestra la simulación de la implementación en verilog del divisor.

### 1.5. Flujo de diseño Hardware - Software

Como se dijo anteriormente todo algoritmo puede ser implementado en Software (código secuencial que se ejecuta en un procesador) o en Hardware (unidad de control + camino de datos), la forma más eficiente de implementación dependerá de restricciones de velocidad, consumo de potencia, tamaño y costos.

Hasta el momento hemos descrito dos periféricos que implementan las operaciones binarias multiplicación y división, ahora vamos a mostrar como controlar estos periféricos desde una unidad de procesamiento o CPU. Por ahora la CPU se utilizará como una caja negra que podemos programar, en el siguiente capítulo se estudiará internamente.

En la figura 1.36 se muestra el flujo de diseño que se utiliza con procesadores *softcore*, es decir, procesadores implementados en FPGAs. Como puede verse el procesador *softcore* que utilizamos cuenta con una memoria de programa (ram.v) *mapeada* en las direcciones 0x0 - 0x3FFFFF, cuando se activa la señal de reset, el contador de programa de la CPU se hace igual a 0 y comienza a leer, decodificar y ejecutar las instrucciones en dicha memoria.

Las herramientas de síntesis modernas permiten inicializar el contenido de las memorias con archivos de texto donde se almacena el código en formato hexadecimal tal como se muestra a continuación.

00000293 00000313 00a00393 00128293

## 1.5 Flujo de diseño Hardware - Software

## 33

```

module div (
    input      reset, clk, init,
    output reg   done,
    output reg [31:0] result,
    input  [15:0] op_A,
    input  [15:0] op_B );
parameter START      = 3'b000; parameter CHECK_GREATER = 3'b001;
parameter CHECK_END   = 3'b011; parameter SHIFT        = 3'b010;
parameter END        = 3'b100; parameter START1     = 3'b101;
begin
    reg [2:0] state; reg [31:0] A;      reg [15:0] B;
    reg [15:0] opB; wire [15:0] A_minus_B; reg [4:0] count;
    assign A_minus_B = A[31:16] + (-B + 1); // 2-complement
    always @(posedge clk or posedge reset)
    begin
        if (reset) begin
            done <= 0; result <= 0; state = START;
        end else begin
            case(state)
                START: begin
                    count = 16; done <= 0; result = 0;
                    if(init)
                        state = START1;
                    else
                        state = START;
                end
                START1: begin
                    A <= {16'b0,op_A}; // {A'',op_A}
                    B <= op_B; done <= 0;
                    result = 0;
                    state = SHIFT;
                end
                SHIFT: begin
                    A = A << 1; count = count - 1;
                    done = 0;
                    state = CHECK_GREATER;
                end
                CHECK_GREATER: begin
                    if(A_minus_B[15])
                        A[0] = 0;
                    else begin
                        A[0] = 1; A[31:16] = A_minus_B;
                    end
                    done = 0;
                    state = CHECK_END;
                end
                CHECK_END: begin
                    if(count == 0)
                        state = END;
                    else
                        state = SHIFT;
                end
                END:begin
                    done = 1;
                    state = START;
                end
                default: state = START;
            endcase
        end
    end
end

```

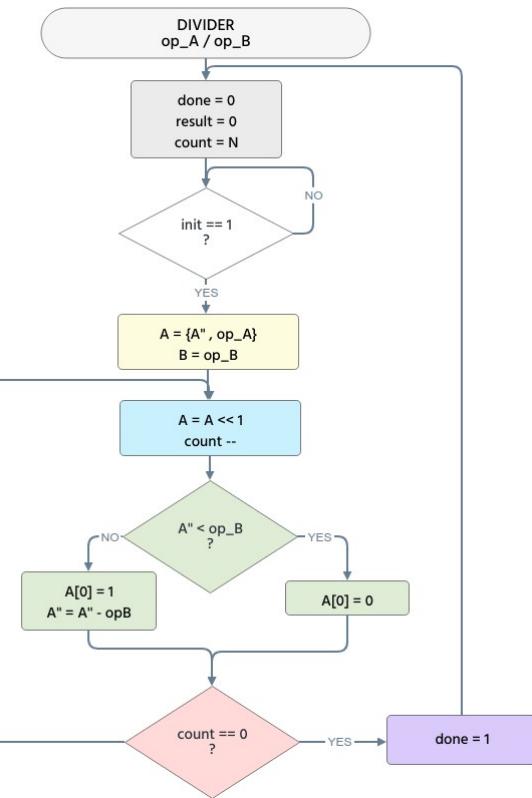


Figura 1.34 Implementación del divisor simplificando el camino de datos en la máquina de estados.

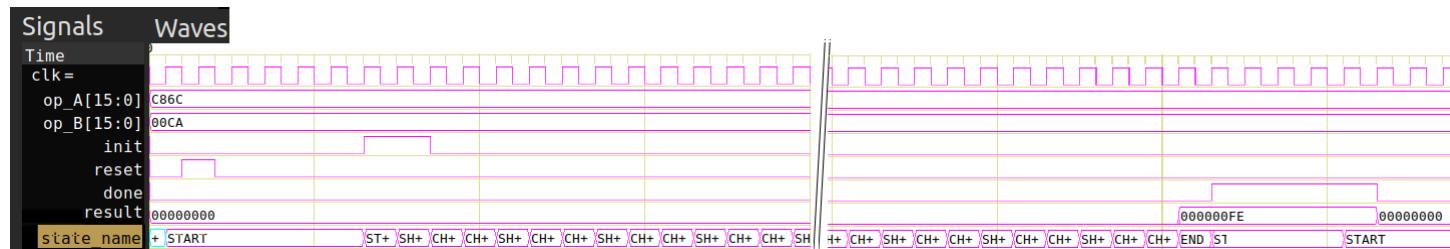


Figura 1.35 Simulación del divisor.

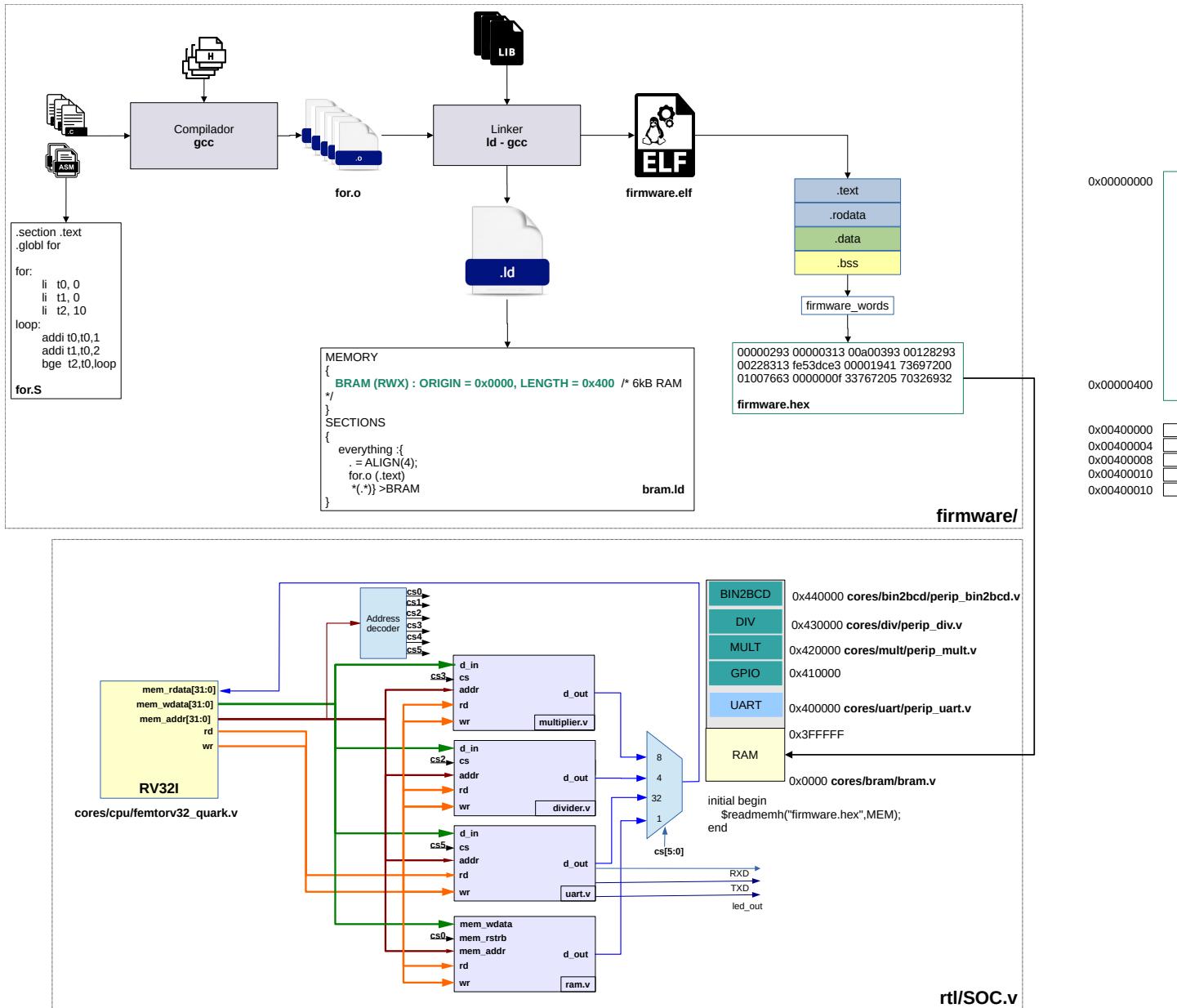


Figura 1.36 Flujo de diseño Hardware - Software.

```
00228313 fe53dce3 00001941 73697200
01007663 0000000f 33767205 70326932
```

Para generar este código es necesario ejecutar el flujo de diseño software, el cual convierte los archivos fuente en las instrucciones que se almacenaran en la memoria de programa. En la figura 1.37 se muestran los archivos necesarios para compilar una aplicación escrita en lenguaje ensamblador.

El archivo **Makefile** ejecuta las instrucciones de forma ordenada y ayuda a realizar el proceso sin necesidad de ejecutar comandos largos. La herramienta **make** ejecuta las opciones disponibles en el archivo **Makefile**, si no se le especifica un parámetro a **make**, esta ejecutará la opción **all**, que en este caso tiene como requisito la opción **firmware.hex** quien a su vez ejecutará la opción **firmware.elf**, quien ejecutará la opción **OBJECTS**, **OBJECTS** es una variable que contiene los objetos necesarios para generar el ejecutable, en este caso el objeto **for.o**, la opción **%.o: %.S** le dice a **make** como generar un archivo **.o** a partir de un archivo **.S**. Por lo que el primer comando en ejecutarse será:

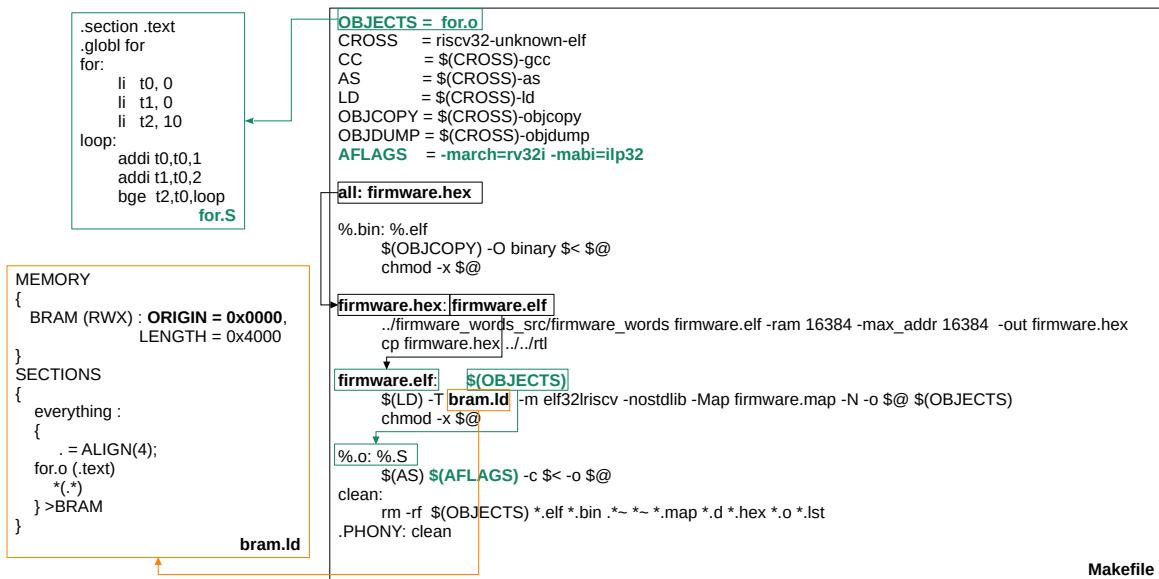


Figura 1.37 Flujo de diseño Software para la implementación del ciclo *for*.

```
riscv32-unknown-elf-as -march=rv32i -mabi=ilp32 -c for.S -o for.o
```

Una vez cumplido el requerimiento de la opción **firmware.elf**, se ejecutan sus tareas:

```
riscv32-unknown-elf-ld -T bram.ld -m elf32lriscv -nostdlib -Map \
firmware.map -N -o firmware.elf for.o
chmod -x firmware.elf
```

El archivo **bram.ld** le indica al enlazador (riscv32-unknown-elf-ld) en donde se encuentra el origen de la memoria de programa (0x0 en este caso) y el objeto que debe estar en esa posición, **for.o** para este ejemplo.

Con el requisito de **firmware.hex** cumplido se ejecutan sus tareas:

```
../firmware_words_src/firmware_words firmware.elf -ram 16384 -max_addr \
16384 -out firmware.hex
```

Acá la herramienta *firmware\_words* genera el archivo de inicialización *firmware.hex* el cual debe tener el tamaño de la memoria declarada en el código en verilog (rtl/cores/cpu/femtory32\_quark.v) 16384 = 0x4000 bytes:

```
module Memory
...
reg [31:0] MEM [0:4095];
initial begin
    $readmemh("./firmware.hex",MEM);
end
...
endmodule
```

Finalmente el archivo *firmware.hex* es copiado al directorio donde se encuentra el código en verilog del procesador para ser incluido en la inicialización de la memoria.

```
cp firmware.hex ../../rtl
```

### 1.5.1. simulación

Una vez generado el archivo de inicialización de la memoria de programa se procede a la simulación, al simular el SoC podemos observar el contenido del banco de registros, del contador de programa y los registros de la ALU y de esta forma verificar el correcto funcionamiento del program implementado.



Figura 1.38 Simulación de la implementación del ciclo *for*.

### 1.5.2. Estructura de los ejemplos

Como parte del material de apoyo se cuenta con un repositorio git donde se aloja código fuente que puede ser utilizado para reproducir los ejemplos aquí mostrados. El directorio que contiene los ejemplos relacionados con la arquitectura HW-SW acá presentada es el siguiente:

```

|-- firmware
|   |-- asm
|   |   |-- bram.ld
|   |   |-- for.S
|   |   |-- Makefile
|   |-- c
|   |   |-- hello.c
|   |   |-- libfemtorv
|   |   |-- linker.ld
|   |   `-- Makefile
|   '-- firmware_words_src
|       |-- firmware_words.cpp
|       `-- Makefile
`-- Makefile
`-- rtl
    |-- bench_quark.v
    |-- cores
        |-- bram
        |-- cpu

```

```

|   | -- div
|   | -- mult
|   '-- uart
|   |   | -- perip_uart.v
|   |   |-- uart.v
|-- Makefile
|-- SOC_TB.v
`-- SOC.v

```

En el directorio firmware aparecen dos carpetas **asm** y **c** para código fuente en lenguaje **ensamblador** y en lenguaje **c** respectivamente.

En el directorio firmware se aloja todo el código relacionado con la descripción a alto nivel de los periféricos y del procesador. En la raíz de este directorio se encuentra el archivo SOC.v (ver figura 1.39) que describe la arquitectura mostrada en la figura 1.36. El subdirectorio **cores** contiene los periféricos del SOC una carpeta por cada periférico con un nombre representativo. Dentro de cada carpeta de periférico se encuentran por lo menos dos archivos uno que implementa la funcionalidad y el que implementa la conexión al procesador este último tendrá el prefijo **perip\_**. Se dispondrá de archivos **Makefile** en cada directorio para facilitar el proceso de síntesis, simulación, compilación y configuración, se recomienda familiarizarse con estos archivos.

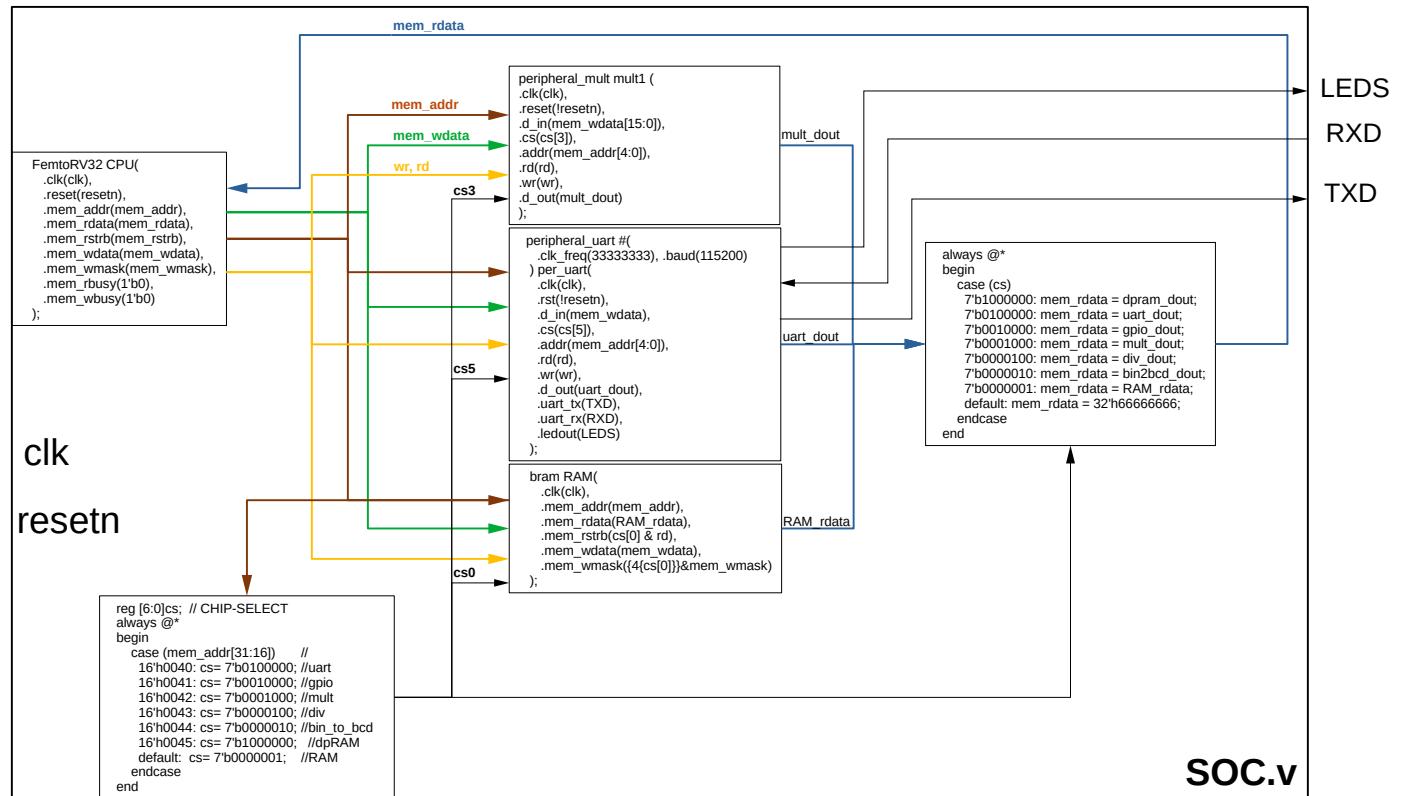


Figura 1.39 Sistem On Chip femtorv

### 1.5.3. Aplicación software para utilizar el multiplicador

En la figura 1.40 se muestra la implementación del multiplicador en el procesador RISC-V-I utilizando las instrucciones (función *mult32*) y usando un periférico que implementa la multiplicación (función *mult\_hw*).

```

.equ MULT_BASE, 0x420000
.equ MULT_OP_A, 0x04
.equ MULT_OP_B, 0x08
.equ MULT_INIT, 0x0C
.equ MULT_RESULT, 0x10
.equ MULT_DONE, 0x14

.section .text
.globl mult
.globl mult_hw

mult:
    addi sp,sp,-12      # store a3, a4, a5 on stack
    sw a3,0(sp)
    sw a4,4(sp)
    sw a5,8(sp)

    mv a4,a0            # a4 * a5
    mv a5,a1            # a4 * a5
    li a0,0              # PP = 0
.L2:
    andi a3,a5,1        #
    beqz a3,.L1           # LSB of a5 = 0 ?
    add a0,a0,a4          # PP = PP + a4
.L1:
    srl a5,a5,1          # a5 = a5/2 right shift
    slli a4,a4,1          # a0 = a0*2 left shift
    bnez a5,.L2           # loop until a5 = 0

    lw a3,0(sp)           #
    lw a4,4(sp)           #
    lw a5,8(sp)           #
    addi sp,sp,12          # restore sp
    ret

mult_hw:
    li gp,MULT_BASE
    sw a0,MULT_OP_A(gp)
    sw a1,MULT_OP_B(gp)
    li a0,1
    sw a0,MULT_INIT(gp)
    sw zero,MULT_INIT(gp)
.L0:
    li t0,1
    lw t1,MULT_DONE(gp)
    and t1,t1,t0
    beqz t1,.L0
    lw a0,MULT_RESULT(gp)
    ret

```

Figura 1.40 Implementación Software (RISCV-I) del multiplicador.

Se presentan estas dos implementaciones con fines de comparación, para analizar la velocidad de ejecución de una tarea software y una tarea Hardware.

En la figura 1.41 se muestra la relación entre las variables del código y el rtl del periférico

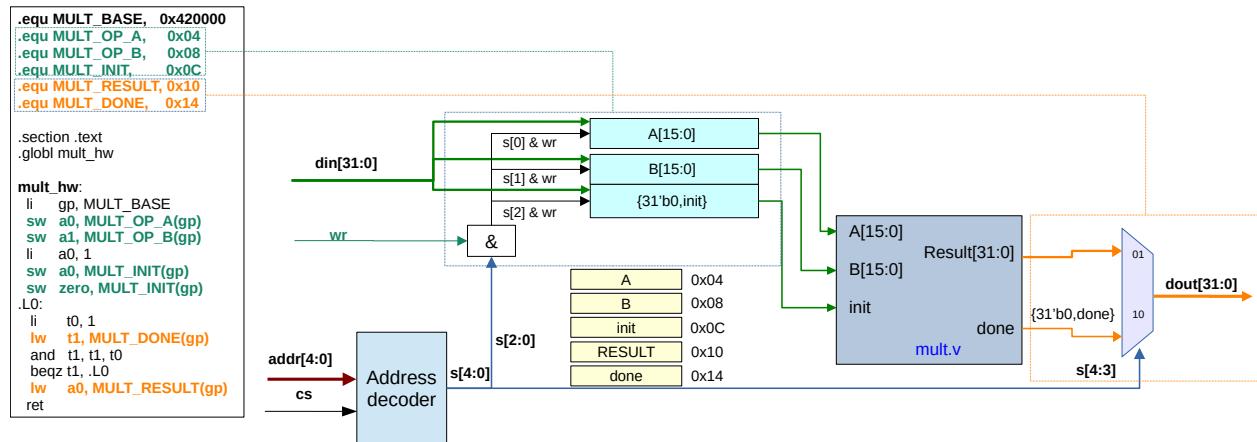


Figura 1.41 Implementación Software (RISCV-I) del multiplicador.

## 1.6. Programación en lenguaje C

El lenguaje ensamblador es muy útil para entender los conceptos básicos de programación y para entender las operaciones internas que realiza el procesador, sin embargo, es poco práctico para la implementación de aplicaciones con

mediana complejidad, además que impide la portabilidad a otras arquitecturas. El lenguaje C facilita la escritura de algoritmos y permite su ejecución en las plataformas que soporte el compilador (siempre que no se utilicen asignaciones a recursos propios del procesador).



## Capítulo 2

# Implementación de tareas Software utilizando procesadores Soft Core

### 2.1. Introducción

En el capítulo anterior se estudió la forma de implementar tareas hardware utilizando máquinas de estado algorítmicas. La implementación de tareas hardware es un proceso un poco tedioso ya que involucra la realización de una máquina de estados por cada tarea; la implementación del camino de datos se simplifica de forma considerable ya que existe un conjunto de bloques constructores que pueden ser tomados de una librería creada por el diseñador. El uso de tareas hardware se debe realizar únicamente cuando las restricciones temporales del diseño lo requieran, ya que como veremos en este capítulo, la implementación de tareas software es más sencilla y rápida.

La estructura de una máquina de estados algorítmica permite entender de forma fácil la estructura de un procesador ya que tienen los mismos componentes principales (unidad de control y camino de datos), la diferencia entre ellos es la posibilidad de programación y la configuración fija del camino de datos del procesador.

En este capítulo se estudiará la arquitectura del procesador MICO32 creado por la empresa Lattice semiconductor y gracias a que fué publicado bajo la licencia GNU, es posible su estudio, uso y modificación. En la primera sección se hace la presentación de la arquitectura; a continuación se realiza el análisis de la forma en que el procesador implementa las diferentes instrucciones, iniciando con las operaciones aritméticas y lógicas siguiendo con las de control de flujo de programa (saltos, llamado a función); después se analizarán la comunicación con la memoria de datos; y finalmente el manejo de interrupciones.

En la segunda sección se abordará la arquitectura de un SoC (System on a Chip) basado en el procesador LM32, se analizará la forma de conexión entre los periféricos y la CPU utilizando el bus wishbone; se realizará una descripción detallada de la programación de esta arquitectura utilizando herramientas GNU.

Este proceso se repetirá para el procesador RISC-V; RISC-V es una arquitectura de conjunto de instrucciones de hardware libre basado en un diseño de conjunto de instrucciones reducido (RISC), su carácter libre le permite ser utilizado sin tener que pagar licencias de ningún tipo. En este capítulo se utilizará la descripción del conjunto de instrucciones RV32I implementada por Bruno Levy

### 2.2. Arquitectura del procesador LM32

La figura 2.1 muestra el diagrama de bloques del soft-core LM32, este procesador utiliza 32 bits y una arquitectura de 6 etapas del pipeline; también cuenta con una lógica de bypass que se encarga de hacer que el caminos de datos entre operaciones sea más corto y se puedan ejecutar en un ciclo sencillo, para que los datos no recorran todo el pipeline para completar instrucciones.

Las 6 etapas del pipeline son:

A *Address*: Se calcula la dirección de la instrucción a ser ejecutada y es enviada al registro de instrucciones.

F *Fetch*: La instrucción se lee de la memoria.

D *Decode*: Se decodifica la instrucción y se toman los operandos del banco de registros o tomados del bypass.

X *Execute*: Se realiza la operación especificada por la instrucción. Para instrucciones simples (sumas y operaciones lógicas), la ejecución finaliza en esta etapa, y el resultado se hace disponible para el bypass.

M *Memory*: Para instrucciones más complejas como acceso a memoria externa, multiplicación, corrimiento, división, es necesaria otra etapa.

D *Write back*: Los resultados producidos por la instrucción son escritas al banco de registros.

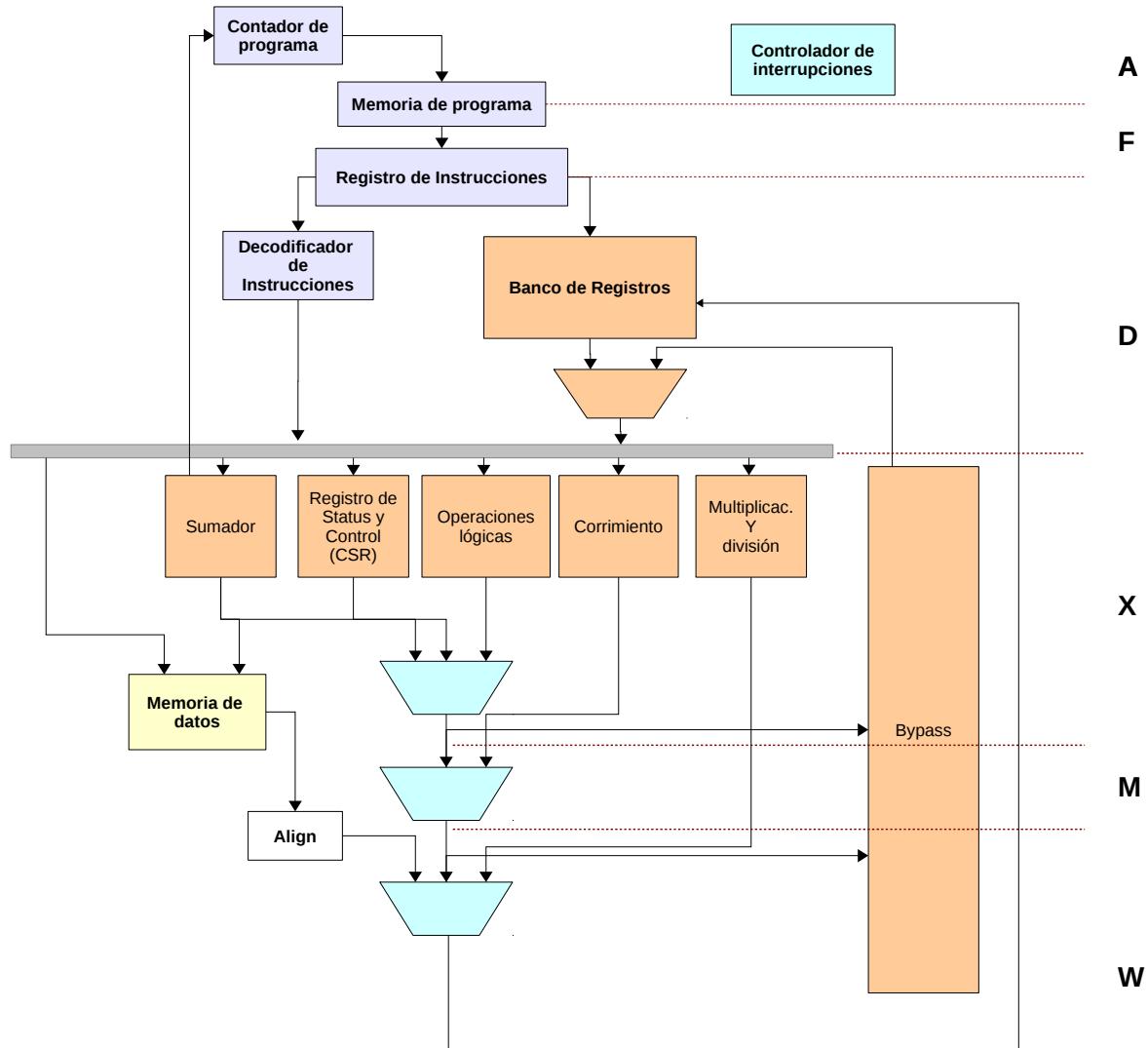


Figura 2.1 Diagrama de bloques del LM32

### 2.2.1. Banco de Registros

El LM32 posee 32 registros de 32 bits; el registro *r0* siempre contiene el valor 0, esto es necesario para el correcto funcionamiento de los compiladores de C y ensamblador; los siguientes 8 registros (*r1* a *r7*) son utilizados para paso de argumentos y retorno de resultados en llamados a funciones; si una función requiere más de 8 argumentos, se utiliza la pila (*stack*). Los registros *r1* - *r28* pueden ser utilizados como fuente o destino de cualquier instrucción. El registro *r29* (*ra*) es utilizado por la instrucción *call* para almacenar la dirección de retorno. El registro *r30* (*ea*) es utilizado para almacenar el valor del *contador de programa* cuando se presenta una excepción. El registro *r31* (*ba*) almacena

el valor del contador de programa cuando se presenta una excepción tipo *breakpoint* o *watchpoint*. Los registros *r26* (*gp*) *r27* (*fp*) y *r28* (*sp*) son el puntero global, de frame y de pila respectivamente.

Después del reset los 32 bits de los registros quedan indefinidos, por lo que la primera acción que debe ejecutar el programa de inicialización es asegurar un cero en el registro *r0*, esto lo hace con la siguiente instrucción *r0 (xor r0, r0, r0)*

### 2.2.2. Registro de estado y control

La tabla 2.1 muestra los registros de estado y control (CSR), indicando si son de lectura o escritura y el índice que se utiliza para acceder al registro.

**Cuadro 2.1** Registro de Estado y Control

Nombre	Index	Descripción
PC		Contador de Programa
IE	0x00	(R/W)Interrupt enable
EID	—	(R) Exception ID
IM	0x01	(R/W)Interrupt mask
IP	0x02	(R) Interrupt pending
ICC	0x03	(W) Instruction cache control
DCC	0x04	(W) Data cache control
CC	0x05	(R) Cycle counter
CFG	0x06	(R) Configuration
EBA	0x07	(R/W)Exception base address

#### Contador de Programa (PC)

El contador de programa es un registro de 32 bits que contiene la dirección de la instrucción que se ejecuta actualmente. Debido a que todas las instrucciones son de 32 bits, los dos bits menos significativos del PC siempre son cero. El valor de este registro después del reset es *h00000000*

#### IE Habilitación de interrupción

El registro IE contiene la bandera IE, que determina si se habilitan o no las interrupciones. Si este flag se desactiva, no se presentan interrupciones a pesar de la activación individual realizada con IM. Existen dos bits *BIE* y *EIE* que se utilizan para almacenar el estado de IE cuando se presenta una excepción tipo breakpoint u otro tipo de excepción; esto se explicará más adelante cuando se estudien las instrucciones relacionadas con las excepciones.

#### EID Exception ID

El índice de la excepción es un número de 3 bits que indica la causa de la detención de la ejecución del programa. Las excepciones son eventos que ocurren al interior o al exterior del procesador y cambian el flujo normal de ejecución del programa. Los valores y eventos correspondientes son:

- **0:** Reset; se presenta cuando se activa la señal de reset del procesador.
- **1:** Breakpoint; se presenta cuando se ejecuta la instrucción break o cuando se alcanza un punto de break hardware.
- **2:** Instruction Bus Error; se presenta cuando falla la captura en una instrucción, regularmente cuando la dirección no es válida.
- **3:** Watchpoint; se presenta cuando se activa un watchpoint.

- **4:** Data Bus Error; se presenta cuando falla el acceso a datos, típicamente porque la dirección solicitada es inválida o porque el tipo de acceso no es permitido.
- **5:** División por cero; Se presenta cuando se hace una división por cero.
- **6:** Interrupción; se presenta cuando un periférico solicita atención por parte del procesador. Para que esta excepción se presente se deben habilitar las interrupciones globales (IE) y la interrupción del periférico (IM).
- **7:** System Call; se presenta cuando se ejecuta la instrucción *scall*.

## IM Máscara de interrupción

La máscara de interrupción contiene un bit de habilitación para cada una de las 32 interrupciones, el bit 0 corresponde a la interrupción 0. Para que la interrupción se presente es necesario que el bit correspondiente a la interrupción y el flag IE sean igual a 1. Despues del reset el valor de IM es *h00000000*

## IP Interrupción pendiente

El registro IP contiene un bit para cada una de las 32 interrupciones, este bit se activa cuando se presenta la interrupción asociada. Los bits del registro IP deben ser borrados escribiendo un 1 lógico.

## 2.3. Set de Instrucciones del procesador Mico32

En esta sección se realizará un análisis del conjunto de instrucciones del procesador Mico32. Para facilitar el estudio se realizó una división en cuatro grupos comenzando con las instrucciones aritméticas y lógicas, siguiendo con las relacionadas con saltos, después se analizará la comunicación con la memoria de datos y finalmente las relacionadas con interrupciones y excepciones. Para cada uno de estos grupos se mostrará el camino de datos (simplificado) asociado al conjunto de instrucciones.

### 2.3.1. Instrucciones aritméticas

#### Entre registros

En la figura 2.2 se muestra el camino de datos simplificado de las operaciones aritméticas y lógicas cuyos operandos son registros, y cuyo resultado se almacena en un registro. En otras palabras son de la forma: **gpr[RX] = gpr[RY] OP gpr[RZ]**, donde: OP puede ser *nor*, *xor*, *and*, *xnor*, *add*, *divu*, *modu*, *mul*, *or*, *sl*, *sr*, *sru*, *sub*. Como puede verse en esta figura la instrucción contiene la información necesaria para direccionar los registros que almacenan los operandos **RY** (instruction\_d 25:21) y **RZ** (instruction\_d 20:16), estas señales de 5 bits direccionan el banco de registros y el valor almacenado en ellos puede obtenerse en dos salidas diferentes (**gpr[rz]** y **gpr[ry]**). En el archivo *rtl/lm32/lm32\_cpu.v* se implementa el banco de registros de la siguiente forma:

```
assign reg_data_0 = registers[read_idx_0_d];
assign reg_data_1 = registers[read_idx_1_d];
```

En este código *reg\_data\_0* y *reg\_data\_1* son las dos salidas **gpr[rz]** y **gpr[ry]**; las señales *read\_idx\_0\_d* y *read\_idx\_1\_d* corresponden a *instruction\_d 25:21* y *instruction\_d 20:16* respectivamente. El contenido de los registros direccionados de esta forma son llevados al modulo *logic\_op* donde se realiza la operación correspondiente a la instrucción y el resultado pasa a través de los estados del pipeline hasta llegar a la señal *w\_result* (parte inferior de la figura). Esta señal entra al banco de registros para ser almacenada en la dirección dada por la señal *write\_idx\_w* la cual es fijada por la instrucción, más específicamente por (*instruction\_d 15:11*). En el archivo *rtl/lm32/lm32\_cpu.v* se implementa esta escritura al banco de registros de la siguiente forma:

```
always @(posedge clk_i)
begin
```

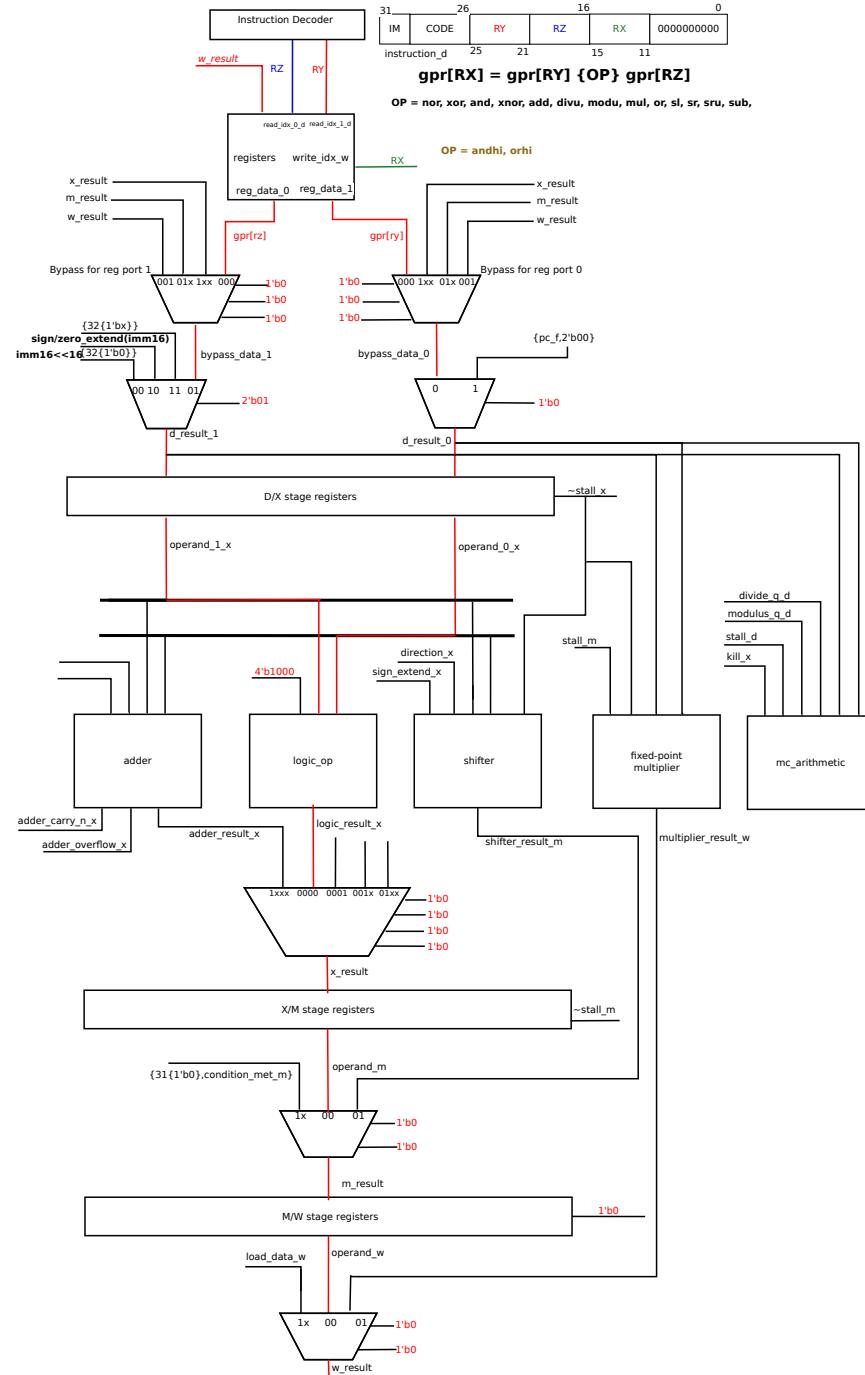
## 2.3 Set de Instrucciones del procesador Mico32

## 45

```

if (reg.write.enable.q.w == 'TRUE)
    registers[write_idx.w] <= w_result;
end

```



**Figura 2.2** Camino de datos de las operaciones aritméticas y lógicas entre registros

## Inmediatas

Existe otro grupo de operaciones lógicas y aritméticas en las que uno de los operandos es un registro y el otro es un número fijo, esto permite realizar operaciones con constantes que no son almacenadas previamente en registros, sino que son almacenadas en la memoria de programa. En la figura 2.3 se muestra como se modifica el camino de datos para este tipo de instrucciones; en ella, podemos observar que *instruction\_d[25:21]* direcciona uno de los operandos que está almacenado en el banco de registros y de forma similar al caso anterior el dato almacenado es llevado al bloque *logic\_op*. El segundo operando es llevado a este bloque desde un multiplexor donde se hace una extensión de signo de *instruction\_d[15:0]* o se hace un corrimiento a la derecha de 16 posiciones; esto, para convertir el número de 16 bits a uno de 32 bits, lo que da como resultado *16instruction\_d[15]*, *instruction\_d[15:0]* y *instruction\_d[15:0]*, *16'h0000* respectivamente; el corrimiento de 16 bits a la derecha se hace para poder realizar las operaciones *andhi* y *orhi*, las cuales sólo operan sobre la parte alta de los operandos.

### 2.3.2. Saltos

Los saltos permiten controlar el flujo de ejecución del programa posibilitando la implementación de ciclos, llamado a funciones, y toma de decisiones. En esta subsección estudiaremos el camino de datos resultante para este tipo de instrucciones. A diferencia de las instrucciones aritméticas y lógicas, en este tipo de instrucciones se modifica el valor del contador de programa.

#### Condicionales

En los saltos condicionales la instrucción se almacena la dirección de los registros que deben ser comparados, específicamente en *instruction\_d[25:21]* y *instruction\_d[20:16]*; los valores almacenados en estos registros son llevados al sumador y a un bloque especial que determina si se cumple o no la condición (señales rojas en la gráfica); la señal *condition\_met\_x* se activa si la condición se cumple. En la figura 2.4 se muestra el camino de datos para las instrucciones condicionales.

Para que el valor del contador de programa se modifique, es necesario que las señales *condition\_met\_x*, *branch\_m* y *valid\_m* se encuentren activas (señales anaranjadas en la gráfica); la señal *branch\_m* se activa cuando la instrucción es de tipo *branch* o *call*; la señal *valid\_m* se activa cuando se presenta una instrucción válida. Adicionalmente, es necesario que el procesador no se encuentre en un estado de *stall*. Si se cumplen las condiciones anteriores, se activará la señal *branch\_taken\_m*, la que le indicará a la unidad de instrucciones que cargue el valor de la señal *branch\_target\_m* en el contador de programa.

El valor de *branch\_target\_m* (señal azul en la gráfica) es fijado por dos diferentes métodos: cuando se produce una excepción o cuando se produce un salto, la señal *exception\_x* selecciona el valor adecuado para cada caso. La señal *branch\_target\_x* es el resultado de la suma de *pc\_d* y de *branch\_offset\_d* (para esta suma no se utiliza el bloque sumador). El valor de *branch\_offset* es seleccionado por la señal *select\_call\_immediate* entre las señales *call\_immediate* (para instrucciones de llamado a función) y *branch\_immediate*; esta última tiene como valor *16inst[15]*, *inst[15:0]*, lo que es una extensión de signo de la constante de 16 bits almacenado en la memoria de programa.

En la figura 2.5 se ilustran 3 ciclos que utilizan condicionales; en color azul se muestra el código en C y en negro se muestra el código implementado por el compilador.

#### Llamado a función y salto incondicional

Existen dos tipos de llamado a función y de salto incondicional; su diferencia radica en la forma de almacenar la dirección a la que deben saltar. En la figura 2.6 se muestra el camino de datos correspondiente a las instrucciones *call* y *bi*, estas almacenan en la instrucción la dirección y en la figura 2.7 se muestra el camino de dato correspondiente a las instrucciones *call* y *b* las que almacenan la dirección en un registro.

Para ambos casos el contador de programa es modificado si se activan las señales *condition\_met\_x*, *branch\_m* y *valid\_m*; la señal *valid\_m* se activa cuando se presenta una instrucción válida; *branch\_m* (color amarillo en los graficos) se activa

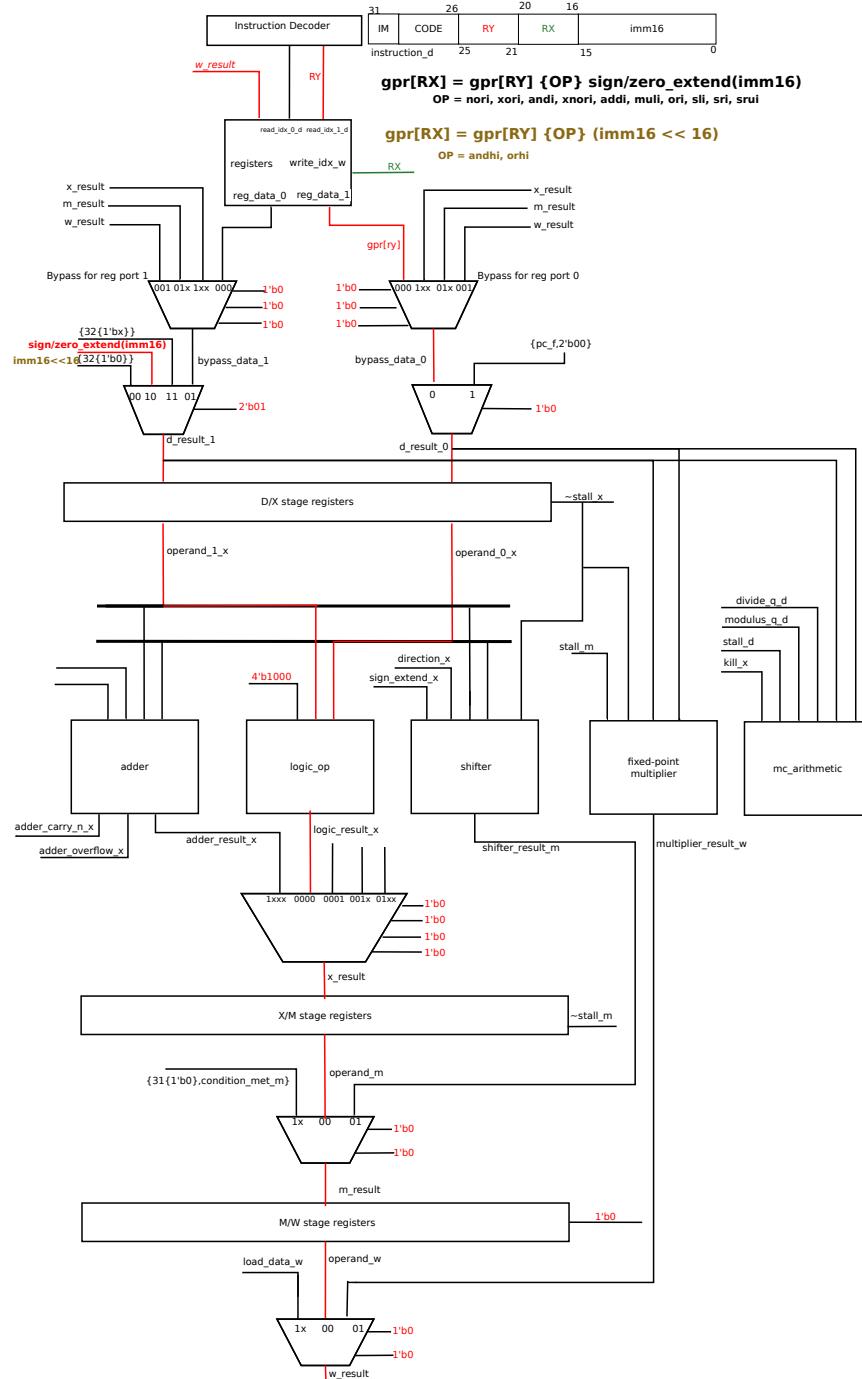


Figura 2.3 Camino de datos de las operaciones aritméticas y lógicas inmediatas

cuando la instrucción que se está ejecutando es un salto o un llamado a función; y *condition\_met\_x* se activa cuando se cumple con la condición para el salto, debido a que estos saltos y llamados son incondicionales, el MICO32 contempla dos casos en los que activa esta señal, tal como se muestra a continuación (tomado de *rtl/lm32/lm32.cpu.v*):

```
always @*
begin
  case (instruction[28:26])
    1111: // Salto
    1110: // Salto
    1101: // Salto
    1100: // Salto
    1011: // Salto
    1010: // Salto
    1001: // Salto
    1000: // Salto
    0111: // Salto
    0110: // Salto
    0101: // Salto
    0100: // Salto
    0011: // Salto
    0010: // Salto
    0001: // Salto
    0000: // Salto
    11111111: // Llamada
    11111110: // Llamada
    11111101: // Llamada
    11111100: // Llamada
    11111011: // Llamada
    11111010: // Llamada
    11111001: // Llamada
    11111000: // Llamada
    11110111: // Llamada
    11110110: // Llamada
    11110101: // Llamada
    11110100: // Llamada
    11110011: // Llamada
    11110010: // Llamada
    11110001: // Llamada
    11110000: // Llamada
  endcase
end
```

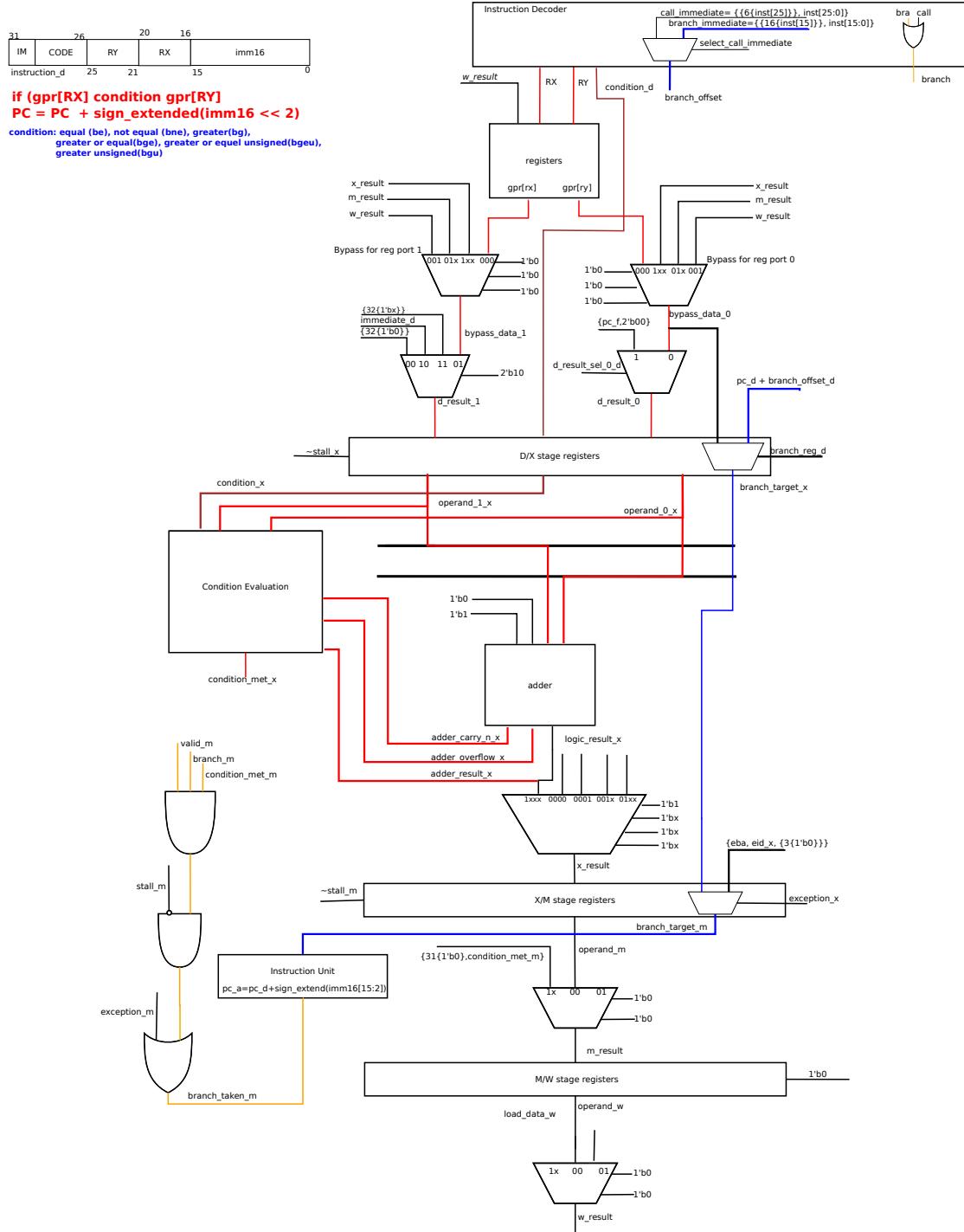


Figura 2.4 Camino de datos de los saltos condicionales

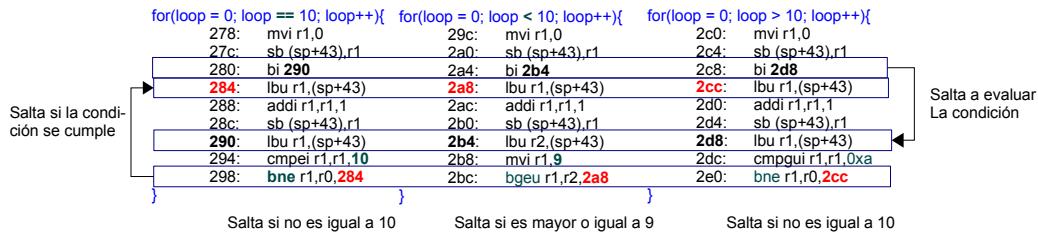


Figura 2.5 Ejemplo de código: saltos condicionales

```

3'b000: condition_met_x = 'TRUE;
3'b110  condition_met_x = 'TRUE;
...
...
...
default: condition_met_x = 1'bx;
endcase
end

```

Los bits *instruction[28:26]* hacen parte del código de la instrucción; el valor para las instrucciones *bi* y *b* es 000 y para *call* y *calli* es 110, lo que activa *condition\_met\_x* cada vez que se presentan estas instrucciones.

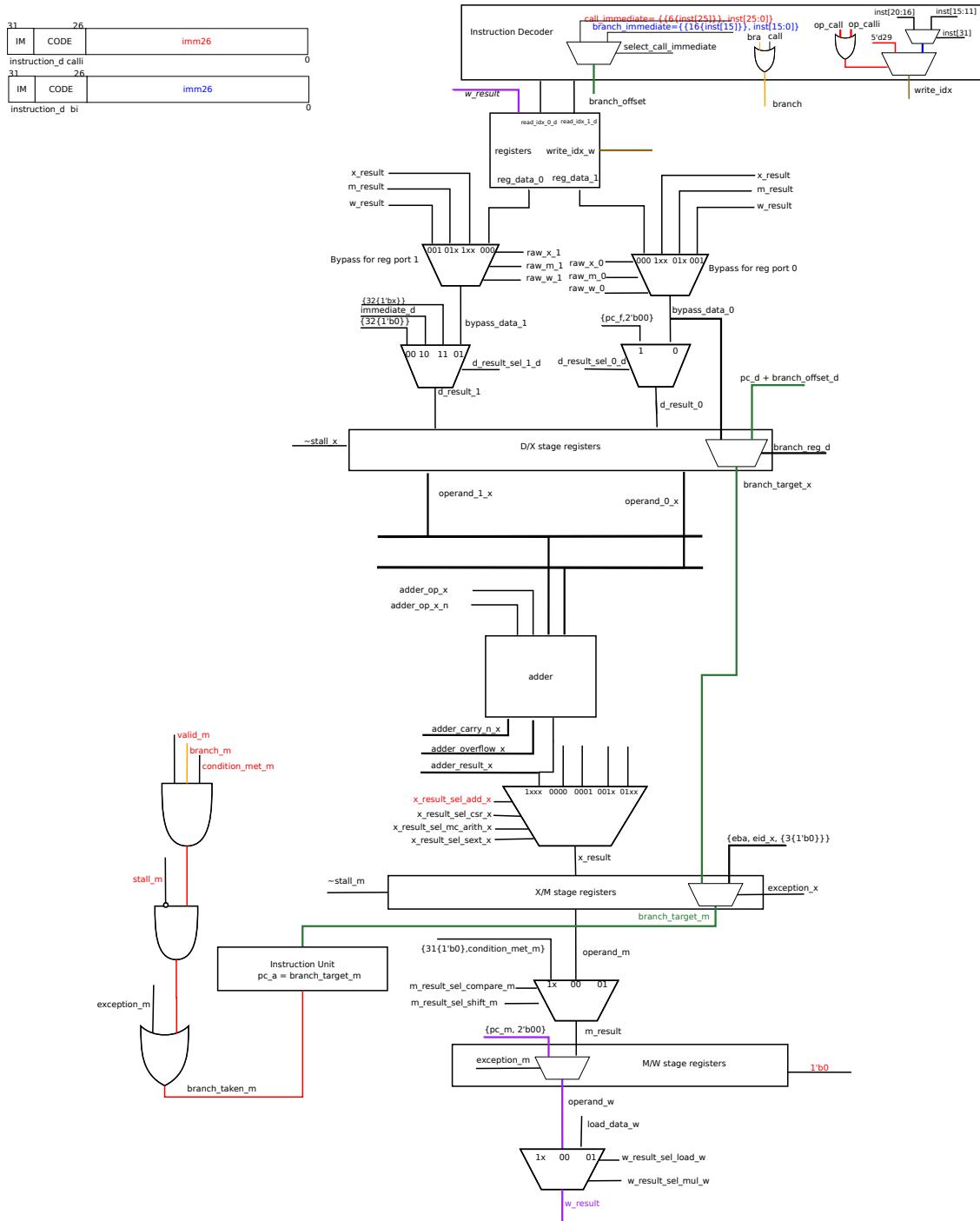
De forma similar a las instrucciones relacionadas con saltos condicionales el valor del contador de programa es igual al valor de la señal *branch\_target\_x* (señal de color verde en las figuras); el valor de esta señal para las instrucciones *call* y *b* proviene del valor almacenado en el registro seleccionado por *instruction\_d[25:21]*. Para las instrucciones *calli* y *bi* el valor está dado por la señal *branch\_offset* la que toma como valor *6ins[25],ins[25:0]* o *16ins[15],ins[15:0]* para una instrucción *call* o *b* respectivamente.

Adicionalmente, para las instrucciones de llamado a función *call* y *calli* se debe almacenar en el registro *R29* la dirección de memoria siguiente a la que se realizó el llamado a la función, esto con el fin de retornar al flujo de programa principal, esto se logra haciendo uso del pipeline y se utiliza el valor del contador de programa *pc\_m* cuyo valor contiene el valor adecuado para el retorno del llamado a función; el valor de *pc\_m* (señal color morado en las figuras) es asignado a la señal *w\_result* del banco de registros para ser almacenado en el registro indicado por *write\_idx* (señal marrón en los gráficos); la que toma el valor de 29 cuando se presenta una instrucción *calli* o *call*.

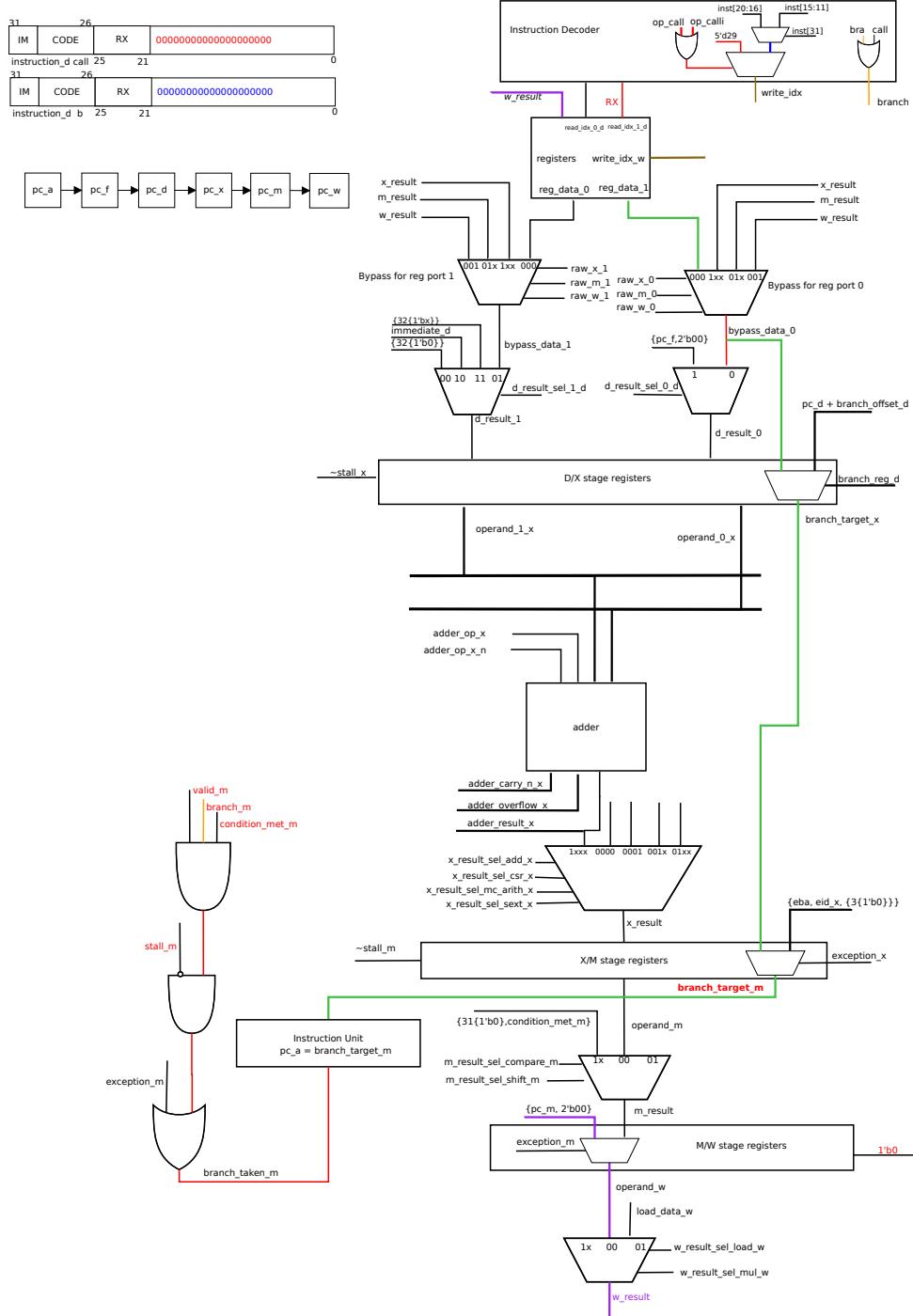
En la figura 2.8 se muestra un ejemplo de uso de la función *call*. El código en C utilizado para este ejemplo se muestra en color azul. La línea de código *result1 = function(0x30)* hace el llamado a la función *function* pasándole el parámetro *0x30* (*decimal 48*); el código implementado por el compilador se muestra junto al código en C; como se mencionó anteriormente, los primeros registros del banco de registros se usan para pasar parámetros entre funciones, en el paso 1, se almacena el valor *0x30* (*48 decimal*) en *r1*; en el paso dos se hace un llamado inmediato a función a la dirección de memoria *0x8C*, lo que hace que el valor del contador de programa tome el valor *0x8C* y se almacene el valor *0x310* en el registro *ra*.

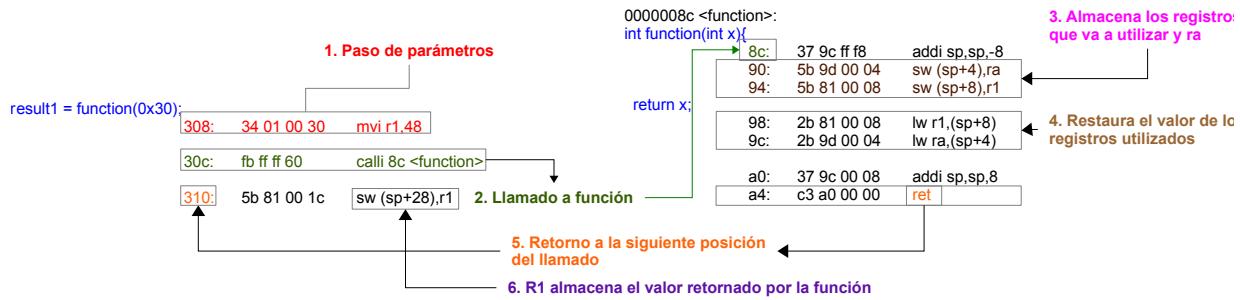
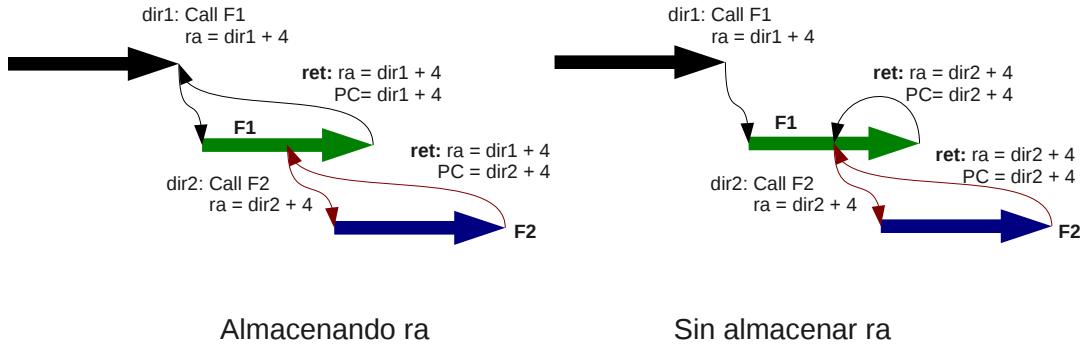
La función *function* está declarada como *int function(int x)* y reside en la posición de memoria *0x8C*. En el paso 3, se almacena el valor de los registros que se utilizan en la función con el fin de restaurarlos antes de retornar al programa donde fué llamada, esto se hace debido a que solo existe un banco de registros en el procesador y si no se hace esto el valor de los registros antes y después del llamado será diferente lo que ocasionará errores en los algoritmos implementados. El registro *ra* almacena el valor de la dirección de retorno, y se almacena para asegurar que cuando se hagan llamados a función anidados se retorne a la dirección adecuada. En el paso 4 se restaura el valor de los registros, garantizando la continuidad del programa principal; finalmente, en el paso 6 la función *ret* carga el valor del *ra* en el contador de programa.

En la figura 2.9 se ilustra la importancia del almacenamiento de los registros en los llamados a funciones, para este ejemplo se consideró el registro *ra*; cuando se almacena el registro *ra* en la función *F2* para ser restaurado al finalizar la función el flujo de programa retorna a la función *F1*, lo que se ejecuta correctamente en los dos casos. Cuando finaliza *F1* el valor de *ra* varía; cuando no se almacena el valor la dirección de retorno de *F1* es modificada por lo que cuando se retorna el contador de programa se hace igual a la dirección de retorno de *F2*.

**Figura 2.6** Camino de datos de los saltos y llamado a funciones inmediatos

## 2.3 Set de Instrucciones del procesador Mico32

**Figura 2.7** Camino de datos de los saltos y llamado a funciones

**Figura 2.8** Ejemplo de código: llamado a función**Figura 2.9** Llamado a función anidado

### 2.3.3. Comunicación con la memoria de datos

Antes de estudiar el camino de datos correspondiente a este grupo de instrucciones, hablaremos de los tipos de datos que soporta el procesador MICO32. En la figura 2.10 se muestran ejemplos de manipulación de diferentes tipos de datos y como estos son tratados en la memoria del procesador.

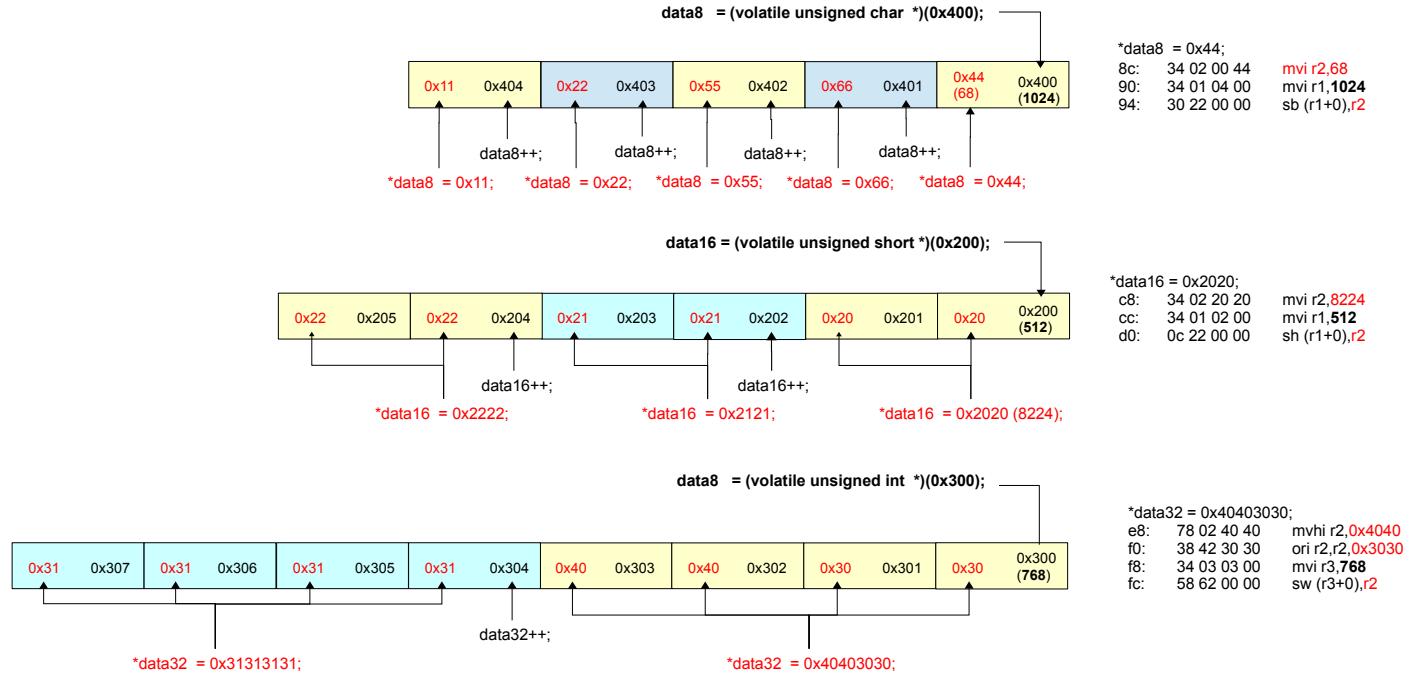
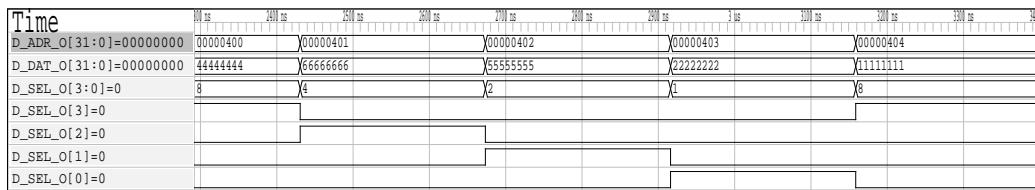
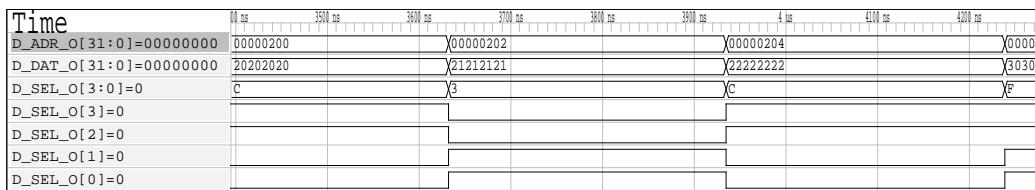
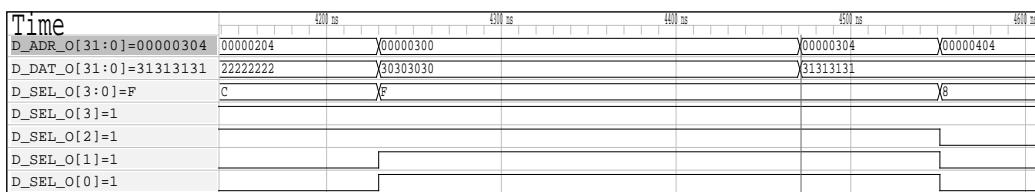
#### Tipos de datos

El primer tipo de datos que se muestra en esta figura es el *char*, la variable *data8* es declarada como un *volatile unsigned char* \*, es decir un puntero a un *char* sin signo tipo *volatile*; los tipos de datos *volatile* le indican al compilador que no realice optimizaciones sobre esta variable, lo que es importante cuando se direccionan periféricos. Al puntero *data8* se le asigna la dirección *0x400* y el valor *0x44*. Si se aumenta el valor de la dirección del puntero en una posición *data8++* la nueva dirección será *0x401* y si se aumenta de nuevo pasará a ser *0x402*; lo que indica que el procesador a pesar de ser de 32 bits puede realizar direccionamiento con granularidad byte; esto es muy conveniente para un almacenamiento eficiente de información, de no ser así se utilizaría una palabra de 32 bits para almacenar 8 bits.

La segunda parte de la figura 2.10 ilustra el manejo del tipo de dato *short* el cual es de 16 bits; para esto se utiliza en puntero *data16* con una dirección inicial de *0x200* y un valor de *0x2020*; al aumentar la dirección del puntero en 1 (*data16++*) la dirección resultante es *0x202*, lo que permite el almacenamiento eficiente de este tipo de dato.

Finalmente se ilustra el tipo de datos *int* y se observa como las direcciones de memoria inicial y final después de aumentar el valor del puntero son *0x300* *0x304*; lo que muestra que el direccionamiento interno de la memoria depende del tipo de datos.

El procesador MICO32 posee 4 señales *D\_SEL\_O[3:0]* que son utilizadas para indicarle a los periféricos el tipo de operación de lectura/escritura que se está efectuando; en la figura 2.11 se observa que estas señales se activan de forma individual indicando el byte que se está direccionando; en la figura 2.12 las señales se activan por parejas indicando el grupo de 2 bytes que se está direccionando; finalmente en la figura 2.13 las 4 señales se activan al tiempo lo que indica un acceso a los 4 bytes al mismo tiempo.

**Figura 2.10** Tipos de datos soportados por el procesador Mico32**Figura 2.11** Acceso a un dato tipo *char***Figura 2.12** Acceso a un dato tipo *short***Figura 2.13** Acceso a un dato tipo *int*

### Escritura a la memoria de datos

El acceso a memoria de datos permite extender las capacidades del procesador posibilitando la conexión de periféricos; los que a su vez, realizan la comunicación con el exterior utilizando diferentes protocolos de comunicación y medios físicos. En esta subsección se describirá la forma en la que el MICO32 implementa las operaciones de lectura y escritura a la memoria de datos.

En la figura 2.14 se ilustra el camino de datos asociado a las instrucciones *sb*, *sh* y *sw*. En las tres, el valor contenido en el registro direccionado por *instruction\_d[25:21]* (RX señales color rojo en la figura) más el valor de 16 bits (con signo extendido a 32 bits) forman la dirección a la que se desea escribir. El valor contenido en el registro direccionado por *instruction\_d[20:16]* (RY señales color azul en la figura) corresponde al dato que será escrito en esa posición de memoria; de esta forma se construyen los buses de datos y direcciones del procesador. Cómo se dijo anteriormente el MICO32 direcciona con granularidad de byte, por esta razón en las instrucciones *sh* y *sw* se indica el valor escrito en las direcciones +1 y +1, +2 y +3 respectivamente; indicando el tamaño en bytes del tipo de dato escrito. Estas señales ingresan a un módulo llamado *load\_store\_unit* que se encarga de generar las señales correspondientes al bus *wishbone*, más adelante estudiaremos en detalle el funcionamiento de este bus.

En la figura 2.11, y 2.13 se muestran las formas de onda cuando se escribe un dato tipo *char* (0x44) a la dirección 0x400, el dato escrito en el bus es 0x44444444 para que el periférico pueda utilizar cualquiera de las cuatro partes del bus de datos *D\_DAT\_O[7:0]*, *D\_DAT\_O[15:8]*, *D\_DAT\_O[23:16]*, *D\_DAT\_O[31:24]*, algo similar ocurre en la escritura del tipo de dato *short* (con valor 0x2020) mostrado en la figura 2.12, aquí se repite el dato para poder utilizar dos partes del bus de datos *D\_DAT\_O[15:0]* y *D\_DAT\_O[31:16]*.

### Lectura

En la figura 2.15 se muestra el camino de datos asociado a las instrucciones *lb/lbu*, *lh/lhu* y *lw*. La dirección de la cual se leerá se calcula de forma similar al caso de la escritura; el valor contenido en el registro direccionado por *instruction\_d[25:21]* (RY señales color rojo en la figura). El dato leído por el módulo *load\_store\_unit* (señal morada en la figura) es almacenado en el registro cuya dirección está dada por *instruction\_d[20:16]*.

#### 2.3.4. Interrupciones

Existen dos formas de conocer si un periférico conectado al procesador requiere atención por parte del procesador; examinando de forma constante los registros de estado del periférico o utilizando interrupciones. La consulta constante de los registros de estado del periférico requiere incluir en el código una rutina que realice esta operación, la cual debe ser llamada de forma regular en el programa principal, la velocidad con que se realice esta consulta debe ser la adecuada para que no se pierdan eventos; debido a esto; uno de los problemas de esta técnica es que al aumentar el número de periféricos aumenta el tiempo entre consultas para un periférico, lo que aumenta la posibilidad de pérdida de eventos; adicionalmente, aumenta el tiempo dedicado a la consulta, lo que disminuye el tiempo disponible para ejecutar las tareas software en el procesador.

Las interrupciones modifican el flujo normal de ejecución del sistema y son originadas por señales dedicadas, lo que hace que su atención ocurra de forma inmediata. Cuando se presenta una interrupción, el valor del contador de programa toma un valor fijo que recibe el nombre de vector de interrupción, el valor del vector de interrupción está formado por (ver figura 2.16 señales de color rojo) una dirección base *EBA* (Exception Base Address) que por defecto es 0x00, un índice que indica la excepción que se presentó *eid\_x* (6 para la interrupción) y tres ceros; estos tres ceros hacen que el espacio entre vectores de excepción sea de 8 palabras de 32 bits, por lo que la rutina de atención a la interrupción debe tener máximo 8 instrucciones (esta rutina se explicará más adelante); para la interrupción el valor del vector es de 0x30.

Como se puede observar en la figura 2.16 para que se genere una excepción (señales de color azul), se debe activar cualquiera de las señales *instruction\_bus\_error\_exception*, *sysrem\_call\_exception\_exception*, *data\_bus\_error\_exception*, *divide\_by\_zero\_exception* o *interrupt\_exception*; lo que activará la señal *branch\_taken\_m* quien a su vez realiza el cambio en el contador de programa *pc\_a*. Para que la señal *interrupt\_exception* se active es necesario: 1- habilitar la generación de interrupciones, es decir, que el flag *ie* (interrupt enable) está activo; 2- habilitar la generación de la inte-

## 2.3 Set de Instrucciones del procesador Mico32

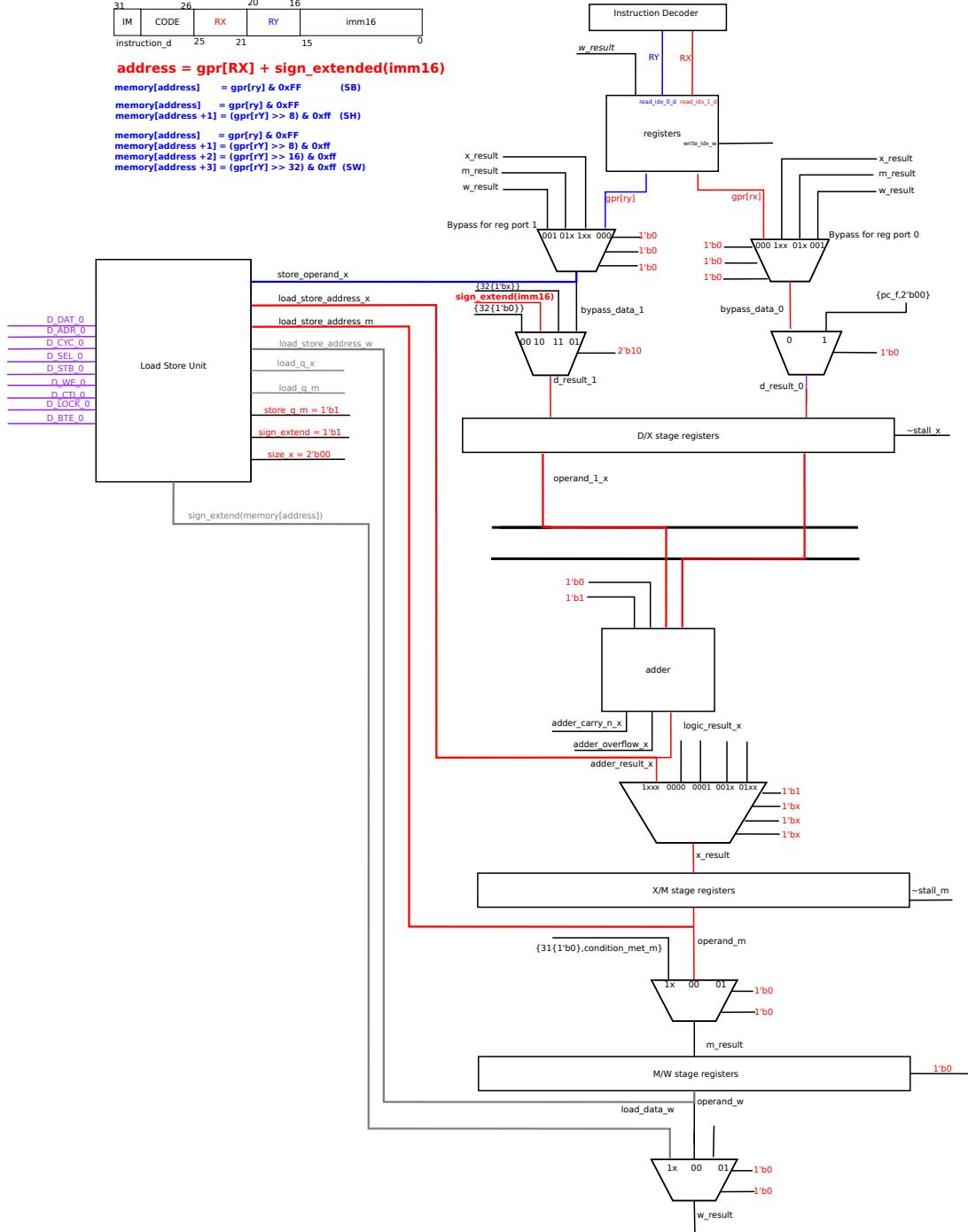


Figura 2.14 Camino de datos de las instrucciones de escritura a memoria

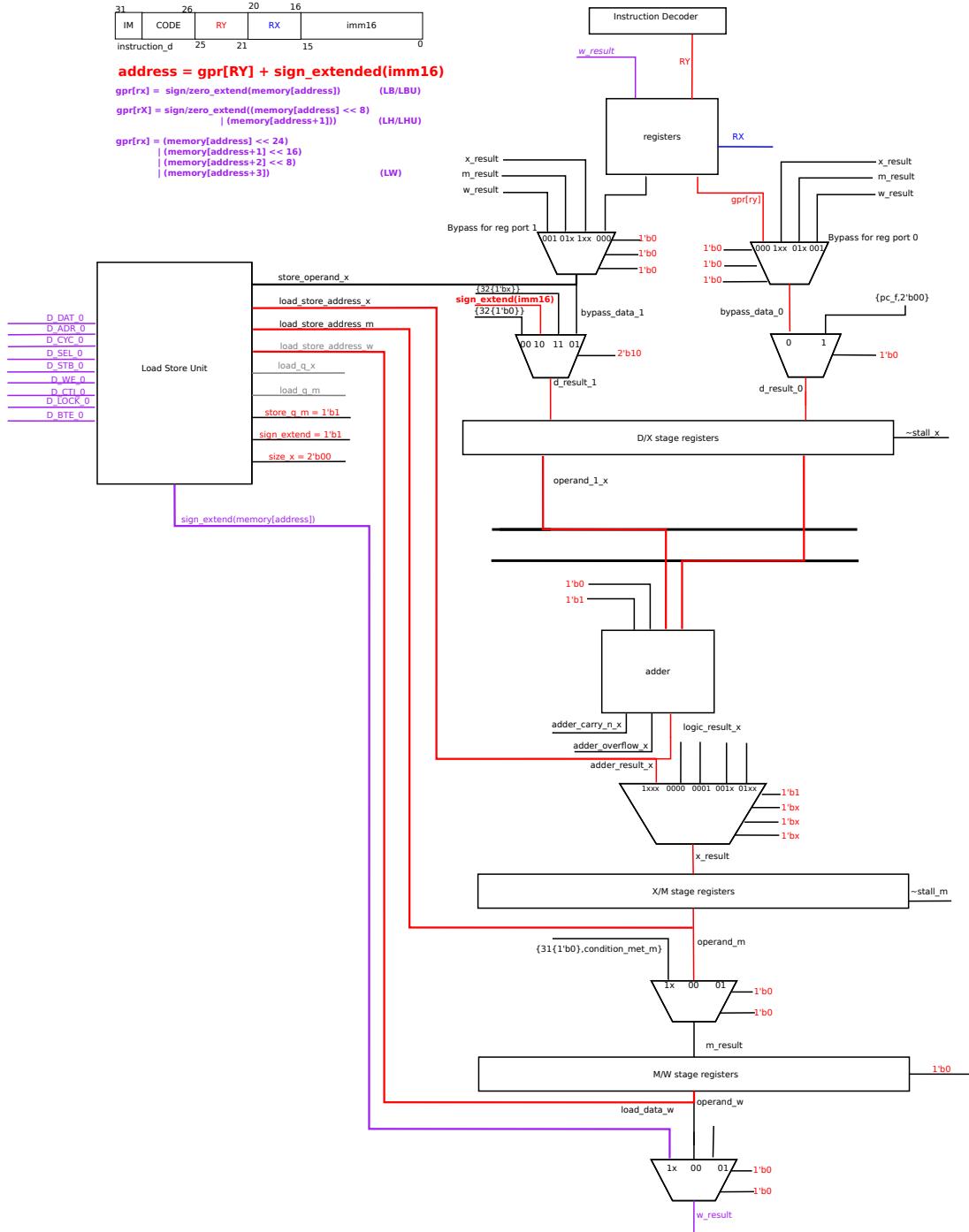
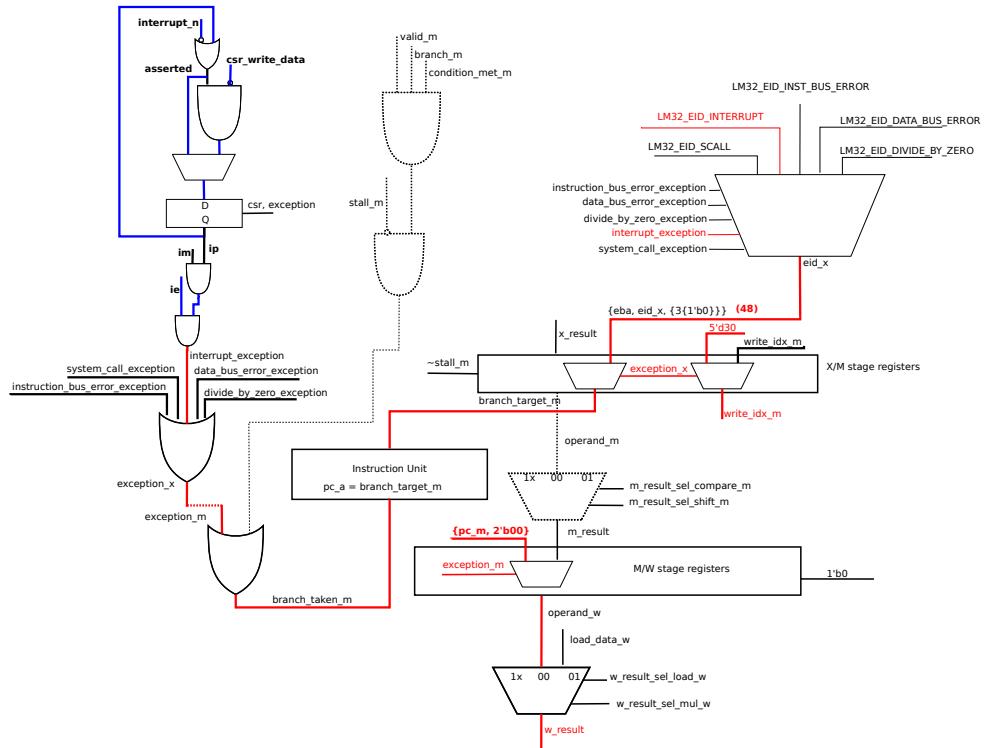


Figura 2.15 Camino de datos de las instrucciones de escritura a memoria

rrupción deseada, para esto el bit correspondiente a la interrupción debe ser igual a 1 en la señal *im* (interrupt mask), lo que recibe el nombre de *enmascaramiento* y 3 - Que el periférico asociado a la interrupción realice una solicitud de atención activando su señal de interrupción, lo que origina una activación de la señal correspondiente en *ip* (interrupt pending).

Al activarse la señal *exception\_x* la variable que direcciona el registro a ser escrito en el banco de registros *write\_idx* toma el valor 30 decimal (*ea* - exception address) y el valor a ser escrito (*w\_result*) será *pc\_m*, 2'b00 (los saltos en el contador de programa es de a 4 bytes, debido a que las instrucciones son de 32 bits, por esta razón los dos bits menos significativos no son tomados en cuenta); lo que garantiza que al salir de la interrupción, el programa principal continuará donde se interrumpió.



**Figura 2.16** Camino de datos correspondiente a las generación de excepciones

### Rutina de atención a la interrupción

A continuación se lista la rutina que se ejecuta cada vez que se presenta una interrupción; como se dijo anteriormente, la dirección del vector de interrupción debe ser 0x48, por lo que este código debe residir en la memoria de programa en dicha dirección.

```

48      sw      (sp+0), ra
49      calli   .save_all
50      resr   r1, IP
51      calli   irq_handler
52      mvhi   r1, 0xffff
53      ori    r1, r1, 0xffff
54      wcsr   IP, r1
55      bi    .restore_all_and_ere

```

En la línea 48 se almacena el valor del registro *ra* en la pila (la pila es una región de la memoria RAM que se utiliza para diferentes propósitos en la ejecución de un programa), esto se hace para que al salir de la rutina de atención a la interrupción el programa continúe de forma adecuada, de no hacer esto, si la interrupción se produjo cuando se estaba ejecutando una función el valor de retorno de la interrupción se modificaría.

En la línea 49 se hace un llamado a la función *save\_all*:

```

    addi    sp, sp, -128
    sw     (sp+4), r1
    ...
    sw     (sp+108), r27
#endif
    sw     (sp+120), ea
    sw     (sp+124), ba
/* ra and sp need special handling, as they have been modified */
    lw     r1, (sp+128)
    sw     (sp+116), r1
    mv     r1, sp
    addi   r1, r1, 128
    sw     (sp+112), r1
    ret

```

En esta función, toma una “fotografía” del estado del procesador en el instante en que se presenta la interrupción, almacenando el valor de todos los registros en la pila, esto se hace para garantizar que el estado del procesador antes y después de la interrupción sea el mismo.

En la línea 49 se almacena el valor de la señal *ip* (interrupt pending) en el registro *r1*, esto se hace para pasar parámetros a la función que será llamada en la línea 50. *irq\_handler* es la función que realizará las acciones correspondientes a una determinada interrupción, esta función debe ser declarada en C en cualquier archivo que haga parte del código fuente de la aplicación (en los ejemplos del repositorio se declara en el archivo *soc-hw.c*) como: *void irq\_handler(uint32\_t pending)*.

En las líneas 51 - 53 se llena con unos la señal *IP*, lo que equivale a una restauración de esta señal, y puede verse como una forma de informarle al procesador que las interrupciones ya fueron atendidas. Finalmente en la línea 54 se hace un llamado a la función *\_restore\_all\_and\_ere*:

```

    lw     r1, (sp+4)
    ...
    lw     r27, (sp+108)
    lw     ra, (sp+116)
    lw     ea, (sp+120)
    lw     ba, (sp+124)
/* Stack pointer must be restored last, in case it has been updated */
    lw     sp, (sp+112)
    eret

```

Esta función: restaura el valor de todos los registros del procesador, incluyendo los registros *ra*, *ea* y *ba*, el registro *ea* se almacena para asegurar el correcto funcionamiento ante el caso de excepciones anidadas; y ejecuta la instrucción *eret* la que hace que el contador de programa tome el valor almacenado en el registro *ea* con lo que el programa retorna a la siguiente instrucción del punto donde se generó la interrupción.

Como se mencionó anteriormente, para que la interrupción se presente es necesario habilitar las interrupciones globales y la máscara asociada al periférico. Para esto, el archivo *crt0ram.S* suministra las siguientes funciones:

```

irq_enable:
    mvi   r1, 1
    wcsr  IE, r1
    ret

irq_disable:
    mvi   r1, 0
    wcsr  IE, r1
    ret

irq_set_mask:
    wcsr  IM, r1
    ret

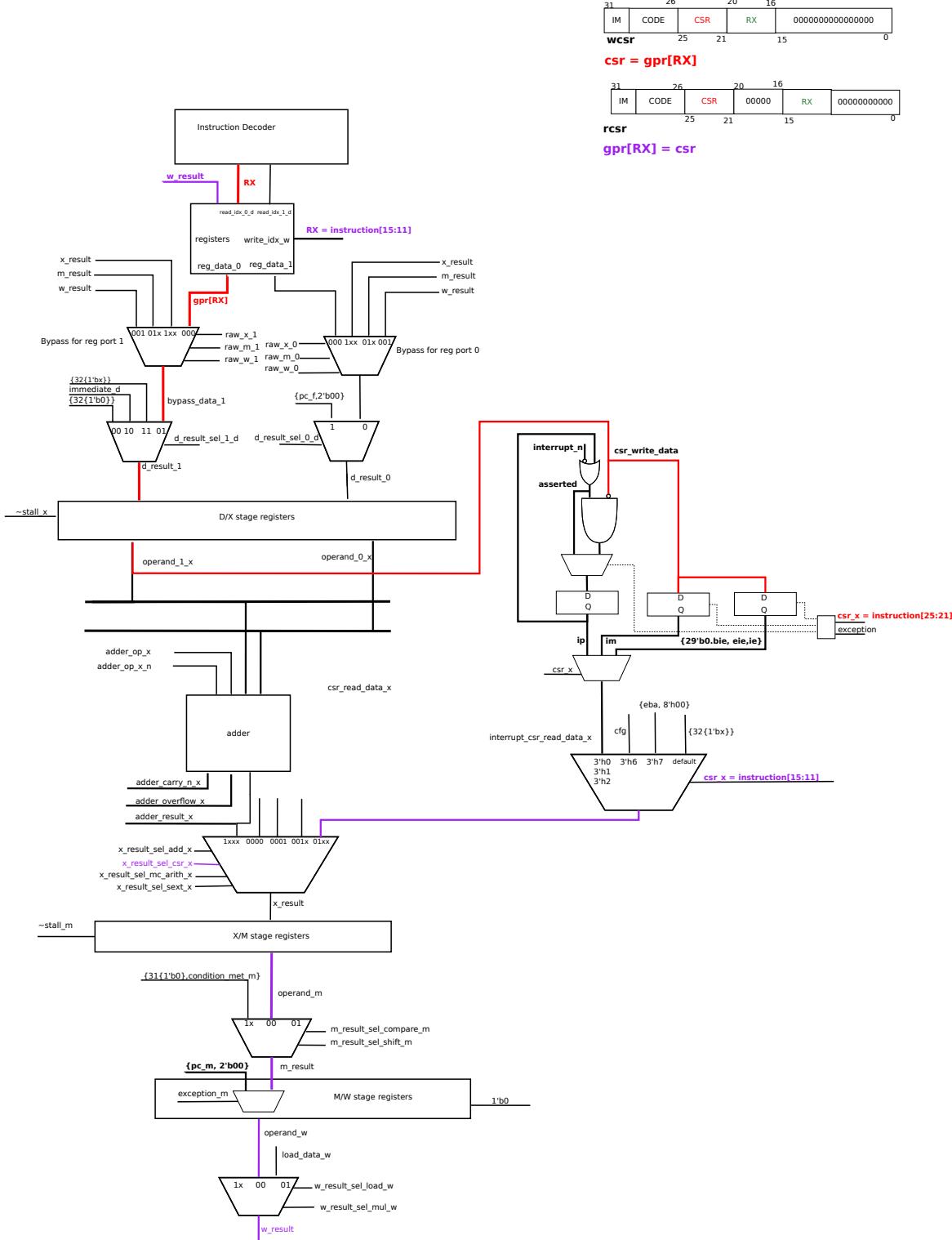
```

En este código se utiliza la instrucción *wcsr* y en la función de atención a la interrupción ya se había utilizado la instrucción *rcsr* (*rcsr r1, IP*); estas instrucciones realizan operaciones de escritura y lectura sobre los registros de estatus y control del procesador. En la figura 2.17 se muestra el camino de datos relacionado con estas instrucciones. El camino de color rojo muestra la escritura utilizando la instrucción *wcsr*; *instruction\_d[20:16]* contiene la dirección del registro a ser escrito en *csr*; y *instruction\_d[25:21]* el registro de estatus y control a escribir.

La lectura de los registros de estado y control se muestra en color morado en la figura; de forma similar a la escritura *instruction\_d[25:21]* direccióna el registro a leer y *instruction\_d[15:11]* la dirección del registro que almacenará el valor leído.

### 2.3.5. Retorno de función y de excepción

La figura 2.18 muestra el camino de datos asociado a las instrucciones de retorno de excepción y de función *eret* y *ret*; en estas instrucciones, el valor de la dirección del registro que va a ser almacenado en el contador de programa es fijo (*instruction\_d[25:21]*), siendo 30 para la instrucción *eret* y 29 para la instrucción *ret*. El valor contenido en estos

**Figura 2.17** Camino de datos correspondiente al acceso de los registros asociados a las excepciones

registros pasa a la señal *branch\_target* y su valor será almacenado en el contador de programa retornando a la dirección siguiente a la que se produjo la excepción o el llamado a función.

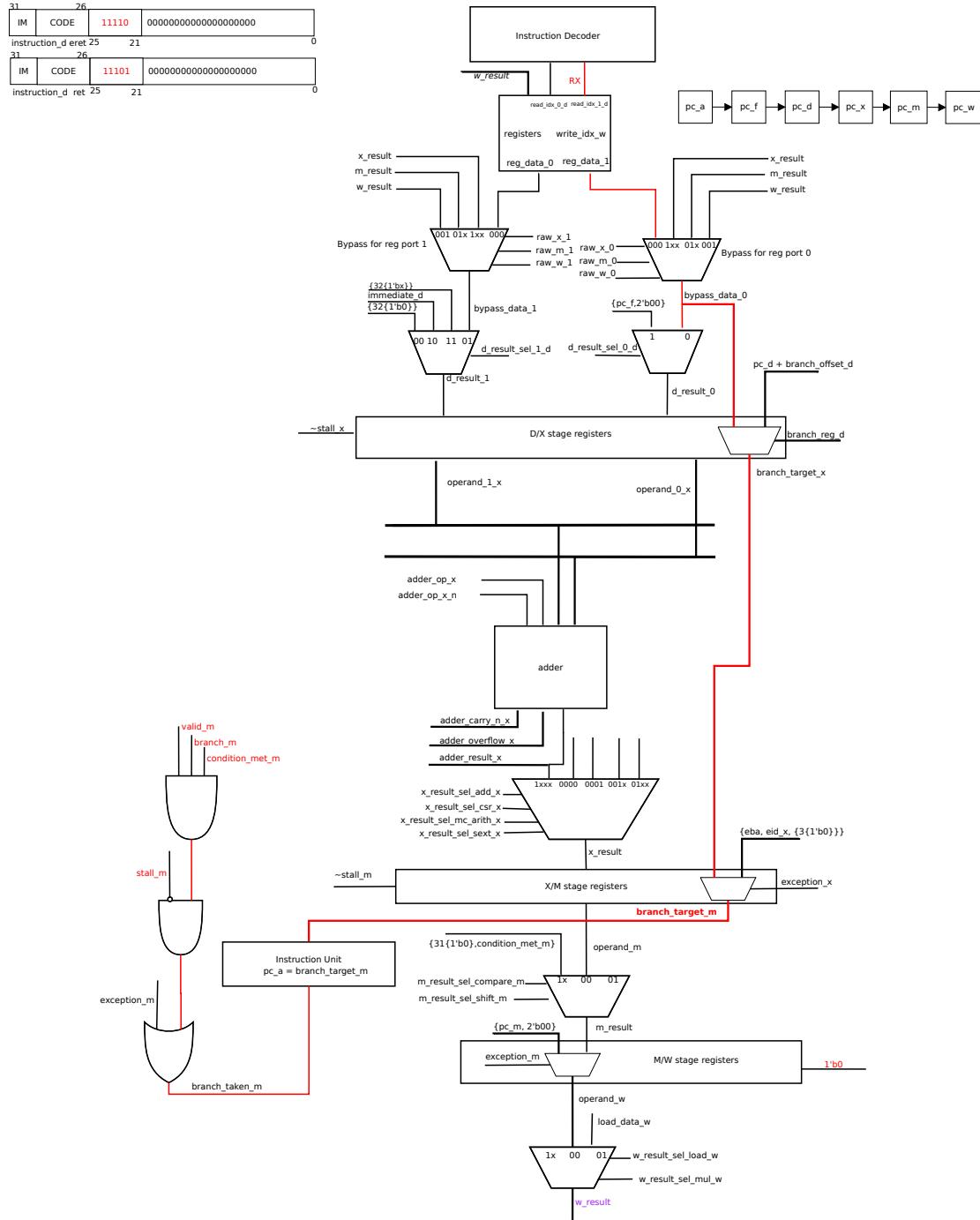


Figura 2.18 Camino de datos asociado al retorno de función y de excepción

En la Figura 2.19 se resume el proceso de atención a la interrupción. La solicitud de atención por parte de un periférico recibe el nombre de **IRQ** (interrupt request) y la rutina que atiende esta solicitud recibe el nombre de **ISR** (interrupt service routine)

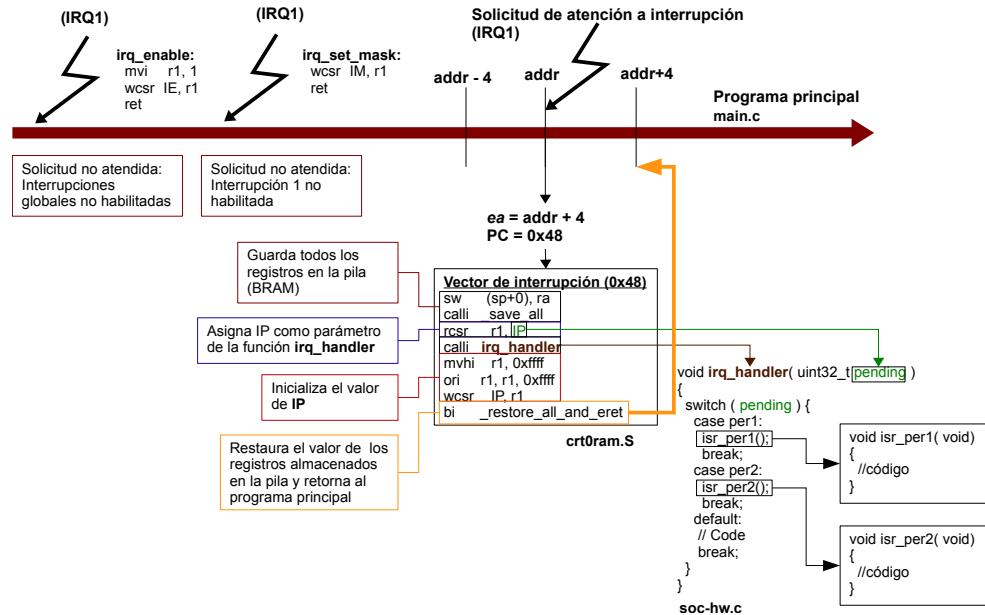


Figura 2.19 Flujo asociado a la atención de una interrupción

## 2.4. Arquitectura del SoC LM32

En la sección anterior se explicó el funcionamiento detallado de cada grupo de instrucciones del procesador MICO32; en esta sección se realizará una descripción de un SoC (sistema sobre silicio) basado en el procesador MICO32; esta arquitectura permitirá entender los SoC modernos desde el punto de vista estructural y de programación

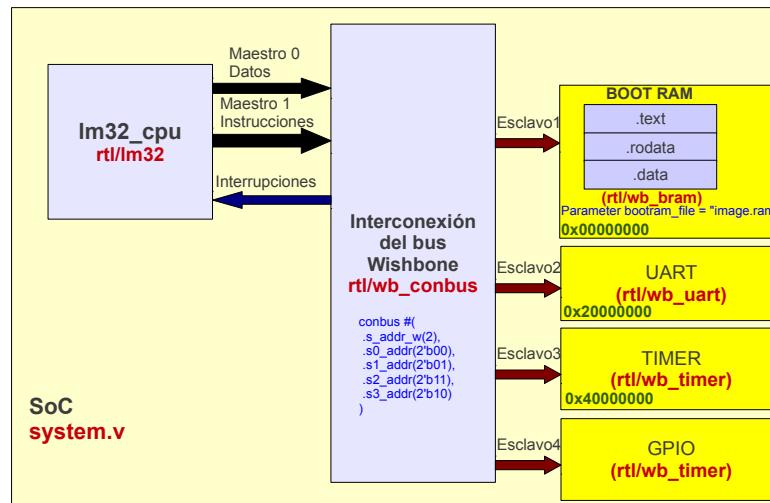


Figura 2.20 Diagrama de bloques del SoC LM32

En la figura 2.20 se muestra el diagrama de bloques del SoC *LM32*, el cual tiene como unidad de procesamiento central el procesador MICO32; esta CPU se conecta a una serie de periféricos a través de el bus *wishbone*. La funcionalidad del SoC está determinada por los periféricos implementados, en esta sección se realizará una descripción de cuatro periféricos básicos para el desarrollo de operaciones básicas de entrada/salida:

- Boot-RAM: Esta memoria almacena la aplicación que se ejecutará al inicializar el SoC.
- UART (Universal Asynchronous Receiver-Transmitter): Puerto serie que permite comunicarse con el exterior y es utilizado como medio de depuración.
- TIMER: Encargado de generar bases de tiempo precisas, de vital importancia en el funcionamiento de la mayoría de las aplicaciones.
- GPIO: Pines de entrada/salida de propósito general.

Adicionalmente, existe un módulo llamado *conbus* que realiza la interconexión entre los periféricos y el procesador, su arquitectura y funcionamiento se explicarán más adelante.

### 2.4.1. Bus wishbone

El bus *wishbone* es un bus diseñado para comunicar los diferentes componentes de un SoC, este bus es abierto y puede ser utilizado libremente. A continuación se listan las señales que componen este bus:

- *ack\_o*: La activación de esta señal indica la terminación normal de un ciclo del bus.
- *addr\_i*: Bus de direcciones.
- *cyc\_i*: Esta señal se activa cuando un ciclo de bus válido se encuentra en progreso.
- *sel\_i*: Estas señales indican cuando se coloca un dato válido en el bus *dat\_i* durante un ciclo de escritura, y cuando deberían estar presentes en el bus *dat\_o* durante un ciclo de lectura. El número de señales depende de la granularidad del puerto. El LM32 maneja una granularidad de 8 bits sobre un bus de 32 bits, por lo tanto existen 4 señales para seleccionar el byte deseado (*sel\_i[3:0]*).
- *stb\_i*: Esta señal se activa cuando se selecciona un esclavo; el cual debe responder a las otras señales únicamente cuando se activa esta señal. El esclavo debe activar la señal *ack\_o* como respuesta a la activación de *stb\_i*.
- *we\_i*: Esta señal indica la dirección del flujo de datos; en un ciclo de lectura tiene un nivel lógico bajo y en escritura tiene un nivel lógico alto.
- *dat\_i*: Bus de datos de entrada.
- *dat\_o*: Bus de datos de salida.

En la figura 2.21 se muestra un ciclo de lectura típico a un periférico con dirección de memoria *0xF0000000*, en ella podemos observar la activación de las señales *wb\_cyc\_i* y *wb\_stb\_i* indicando un ciclo de bus válido y la selección del esclavo, el valor de *wb\_we\_i* indica que el acceso es de lectura, a lo que el esclavo debe responder colocando el dato requerido por el procesador en el bus de salida *wb\_dat\_o* y con la activación de la señal *wb\_ack\_o*

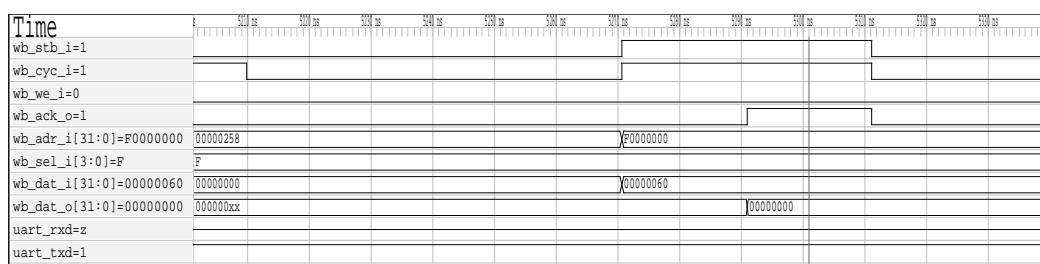


Figura 2.21 Ciclo de lectura del bus wishbone

En la figura 2.32 se muestra la escritura del valor *0x2A* a la dirección de memoria *0xF0000004*, las formas de onda son similares a las del ciclo de lectura, salvo que el valor de la señal *wb\_we\_i* es uno indicando la escritura.

### Interface del bus wishbone (conmax)

El bus wishbone tiene una arquitectura maestro/esclavo en la que solo los maestros pueden iniciar las operaciones de lectura y escritura y únicamente el esclavo al que se le hace el requerimiento debe responder. Para coordinar la

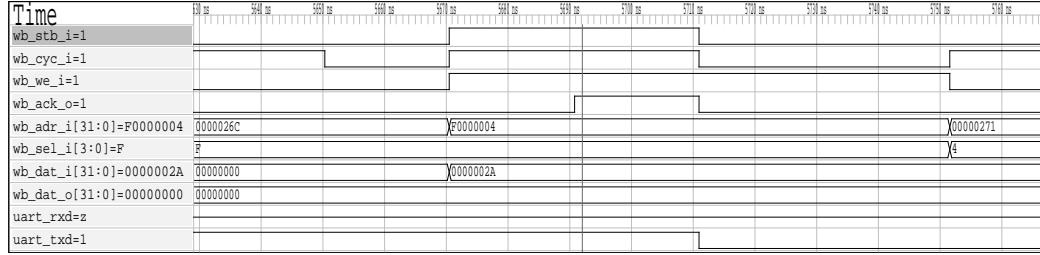


Figura 2.22 Ciclo de escritura del bus wishbone

comunicación entre múltiples maestros se debe incluir un árbитro, que en el LM32 recibe el nombre de *conmax*. La figura 2.23 muestra el diagrama de bloques de este árbitro.

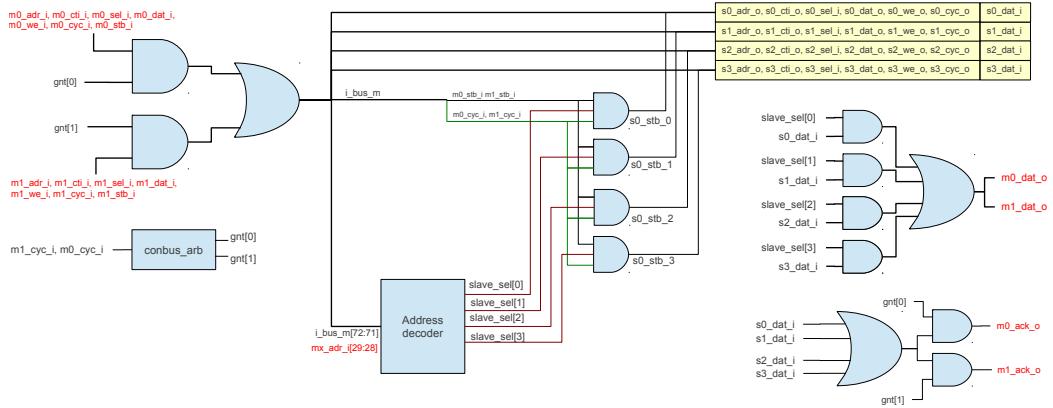


Figura 2.23 Circuito de interconexión del bus wishbone

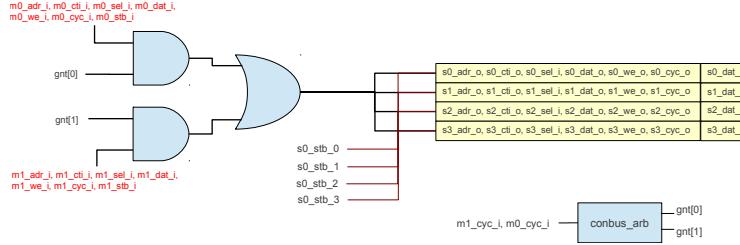
Una de las funciones del árbitro *conmax* es fijarle un rango de direcciones único a cada periférico, por esta razón todo árbitro debe tener un decodificador de direcciones (módulo *ADDRESS DECODER* en la figura 2.23) que tiene como entradas los bits más significativos del bus de direcciones, en este caso solo se usan dos bits (*mx\_adr\_i[29:28]*) ya que solo se cuenta con cuatro periféricos; este decodificador activa las señales *slave\_sel[3:0]* de acuerdo a la definición en el archivo *system.v*:

```
conbus #(
    .s_addr_w(2),
    .s0_addr(2'b00), // bram      0x00000000
    .s1_addr(2'b01), // uart       0x20000000
    .s2_addr(2'b11), // timer      0x60000000
    .s3_addr(2'b10)  // gpio        0x40000000
) conbus0(
```

Asignado las direcciones de memoria *0x00000000 - 0x1FFFFFFF*, *0x20000000 - 0x3FFFFFFF*, *0x60000000 - 0x7FFFFFFF* y *0x40000000 - 0x5FFFFFFF* a la BRAM, UART, TIMER y GPIO respectivamente, la activación de *slave\_sel[3:0]* hace que se active su correspondiente señal *s0\_stb\_[3:0]* (y se presenta un ciclo válido de bus) indicándole al periférico que ha sido seleccionado para una operación de lectura o escritura.

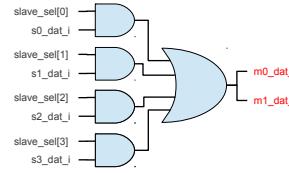
En la figura 2.24 se muestra el circuito simplificado del árbitro *conmax* para una operación de escritura; en ella se puede observar que todos los esclavos comparten las señales *s0\_adr\_o*, *s0\_cti\_o*, *s0\_sel\_i*, *s0\_dat\_o*, *s0\_we\_o*, *s0\_cyc\_o* y *sx\_dat\_i*, las cuales son la salida de un multiplexor que selecciona entre las señales correspondientes a los diferentes maestros del SoC (m0 y m1 en este caso); las señales *gnt[0]* y *gnt[1]* seleccionan al maestro que se conectará con todos los esclavos, por esta razón nunca se activarán las dos al tiempo. Las únicas señales que no comparten los esclavos wishbone son las que indican a los periféricos que han sido seleccionados para una transferencia de información *s0\_stb\_[3:0]*, *slave\_sel[3:0]*.

En la figura 2.25 se muestra el circuito simplificado del árbitro *conmax* para una operación de lectura; en ella podemos observar que los buses de datos de los periféricos *s[3:0].dat\_i* se conectan a los buses de datos de los maestros



**Figura 2.24** Circuito equivalente a una operación de escritura para el árbitro del bus wishbone

$m[1:0].dat_o$ ; las señales  $slave\_sel[3:0]$  se activan una a la vez y seleccionan el esclavo que se conectará con el maestro.

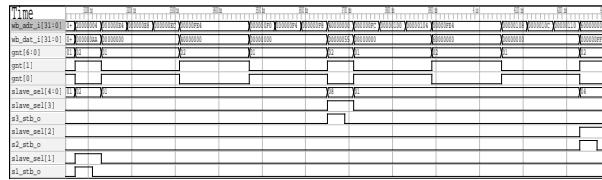


**Figura 2.25** Circuito equivalente a una operación de lectura para el árbitro del bus wishbone

Para ilustrar de forma gráfica la operación del árbitro se implementó un programa que escribe los siguientes valores a las direcciones de los periféricos UART, TIMER y GPIO:

1.  $0xAA$  a la dirección del esclavo 1  $0x20000004$
2.  $0x55$  a la dirección del esclavo 3  $0x40000000$
3.  $0xFF$  a la dirección del esclavo 2  $0x60000000$

Como podemos ver en la figura 2.26, las señales  $gnt[0]$  y  $gnt[1]$  se activan de forma alterna y solo está activa una de ellas, cuando se escribe el valor  $0xAA$  a la dirección  $0x20000004$  se activan las señales  $slave\_sel[1]$  y  $s1\_stb_o$  indicando la activación del primer periférico; similarmente, cuando se escribe el valor  $0x55$  a la dirección  $0x40000000$  se activan las señales  $slave\_sel[3]$  y  $s3\_stb_o$  indicando la activación del tercer periférico y finalmente, cuando se escribe el valor  $0xFF$  a la dirección  $0x60000000$  se activan las señales  $slave\_sel[2]$  y  $s2\_stb_o$  indicando la activación del segundo periférico.



**Figura 2.26** Formas de onda del proceso de comunicación entre la CPU y los periféricos usando el bus wishbone

## 2.4.2. Arquitectura de los periféricos

En esta subsección se realizará un estudio de la arquitectura de los esclavos wishbone, se analizarán tres periféricos: GPIO, UART y TIMER

## Periférico GPIO

En todo SoC es necesario contar con pines de entrada/salida de propósito general, este sencillo periférico permite controlar la dirección de un pin, controlar el valor de un pin de salida y leer el valor de un pin de entrada; en la figura 2.27 se muestra el diagrama de bloques de este periférico.

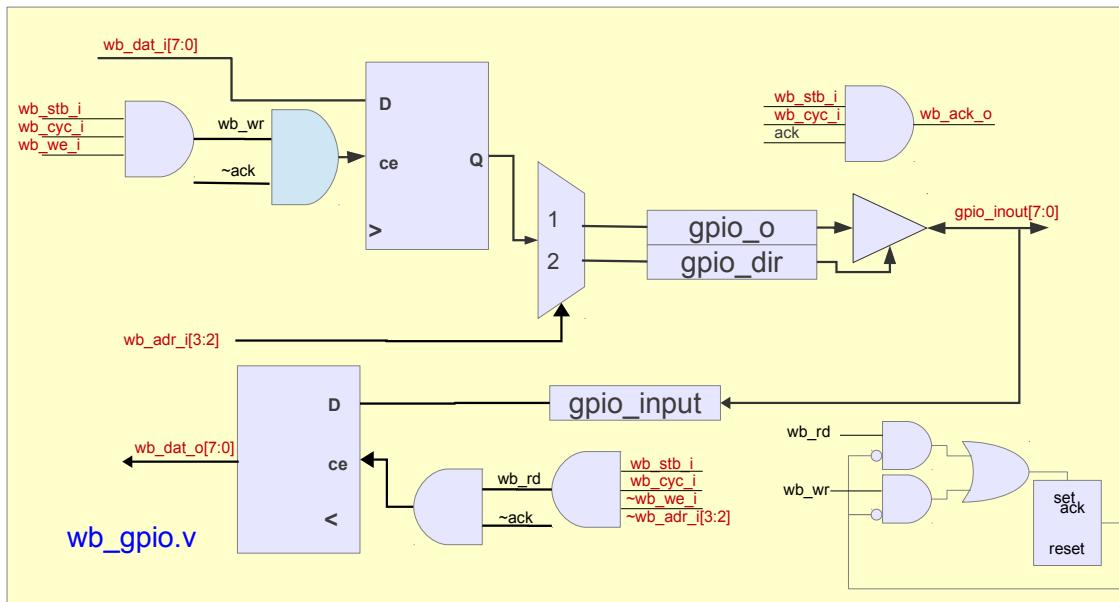


Figura 2.27 Ejemplo de periférico wishbone: GPIO

La dirección del pin es fijada con un buffer tri-estado que a su vez es controlado por el valor almacenado en el registro `gpio_dir`. Los registros `gpio_o` y `gpio_input` almacenan los valores escritos y leídos de los pines respectivamente. Para entender el comportamiento de este periférico analizaremos los circuitos de lectura y escritura de forma separada. En la figura 2.28 se muestra el circuito de lectura; el valor del registro `gpio_input` es almacenado en un registro que está conectado al bus de datos de salida del periférico `wb_dat_o` cuando la señal `wb_rd` sea igual a 1 y la señal `ack` sea igual a cero. `wb_rd` es igual a 1 cuando se presente un ciclo de bus válido, se seleccione el periférico y se realice una operación de lectura.

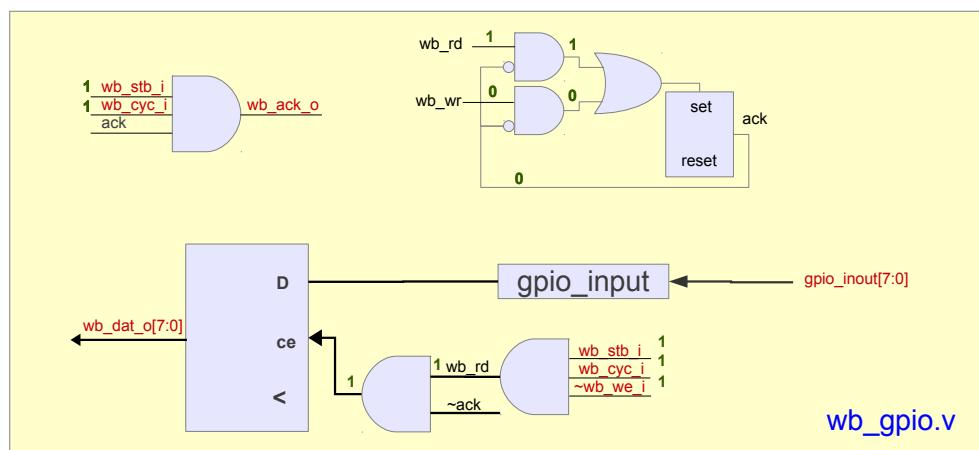
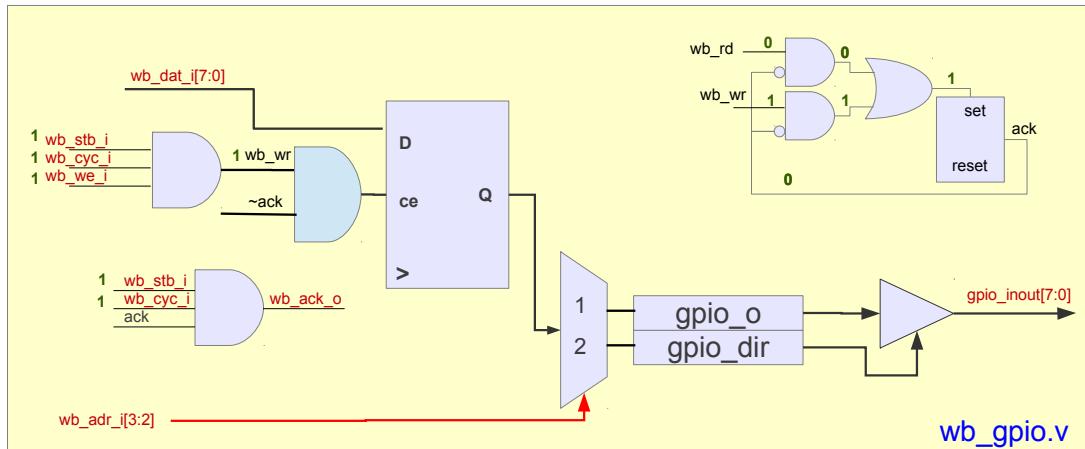


Figura 2.28 Circuito equivalente de lectura del periférico GPIO

El circuito simplificado de escritura se muestra en la figura 2.29; en este periférico el bus de datos proveniente del maestro puede almacenarse en los registros *gpio.dir* y *gpio.o*; el multiplexor controlado por *wb\_adr\_i[3:2]* selecciona donde será almacenado el dato. La transferencia al registro seleccionado se realiza únicamente cuando la señal *wb\_wr* sea igual a 1 y la señal *ack* sea igual a cero. *wb\_wr* es igual a 1 cuando se presente un ciclo de bus válido, se seleccione el periférico y se realice una operación de escritura.



**Figura 2.29** Circuito equivalente de escritura del periférico GPIO

Tanto en la operación de lectura como en la de escritura se debe generar la señal *wb\_ack\_o* para indicarle al maestro que la solicitud de comunicación ha sido recibida y atendida; para esto se implementó el circuito compuesto de las 2 compuertas AND, una compuerta OR y un FLIP FLOP, este circuito hace que la señal *ack* sea igual a 1 cuando cualquiera de las señales *wb\_rd* o *wb\_wr* sea igual a 1 y el estado de la señal *ack* sea igual a 0; es decir, cuando el dispositivo pasa del estado no seleccionado a ser seleccionado para una operación de lectura o escritura.

### Periférico UART

En la figura 2.30 se muestra el diagrama de bloques de un periférico un poco más complejo una UART, su arquitectura se basa en un módulo que implementa las tareas de comunicación que se encuentra descrito en el archivo *uart.v*; en el archivo *wb\_uart.v* se hace la adaptación de esta unidad funcional al bus wishbone, esta arquitectura permite que el módulo funcional pueda ser conectado a diferentes buses sin tener que reescribir todo el código.

Del diagrama de bloques de la UART podemos observar que su arquitectura es similar a la del GPIO, existen los mismos bloques de interconexión con los buses de entrada y de salida y se utiliza el mismo circuito para generar la señal *ack*.

El circuito simplificado de salida se muestra en la figura 2.31, se observa que existen dos valores que pueden ser leídos desde el procesador: la señal *rx\_data* y los bits de estado *tx\_error* y *tx\_avail*; en este caso la línea de dirección *wb\_adr\_i[2]* selecciona la información que será transmitida al procesador.

*wb\_adr\_i[3:2]*

El circuito de escritura se muestra en la figura 2.32, en este ejemplo el bus de datos proveniente del maestro se conecta directamente a la señal *tx\_data* ya que este periférico no permite modificar otros parámetros. Por esta razón el circuito solo transmite un 1 a la señal *uart\_txd* lo que hace que la uart transmita el valor fijado por la señal *tx\_data*.

### Periférico TIMER

En la figura 2.33 se muestra el diagrama de bloques resumido del periférico TIMER; el cual posee 6 registros que pueden ser modificados y leídos por el procesador. De nuevo la arquitectura de este periférico es similar a los anteriores así como el circuito de generación de la señal *ack*.

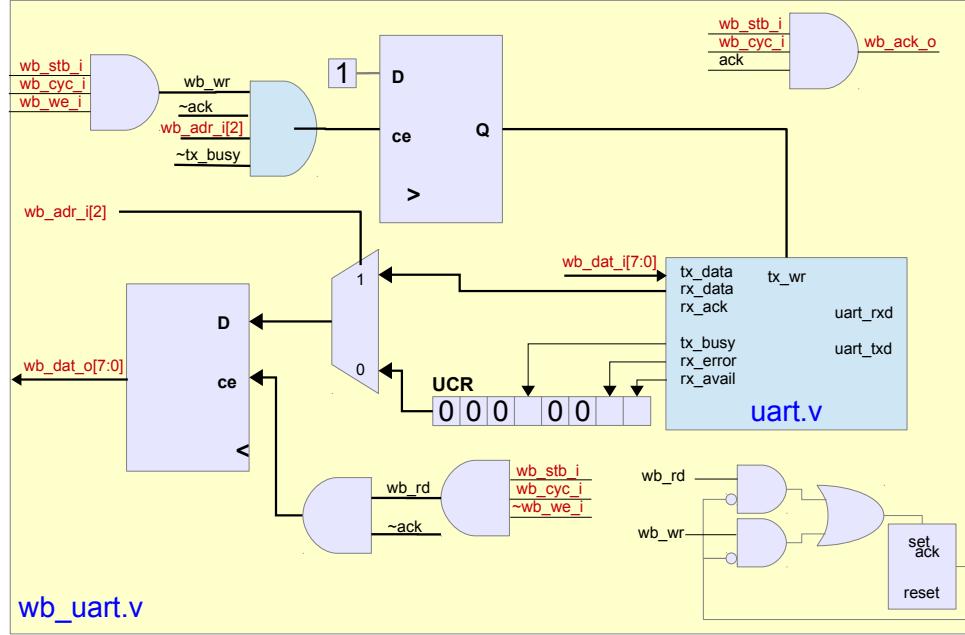


Figura 2.30 Ejemplo de periférico wishbone: UART

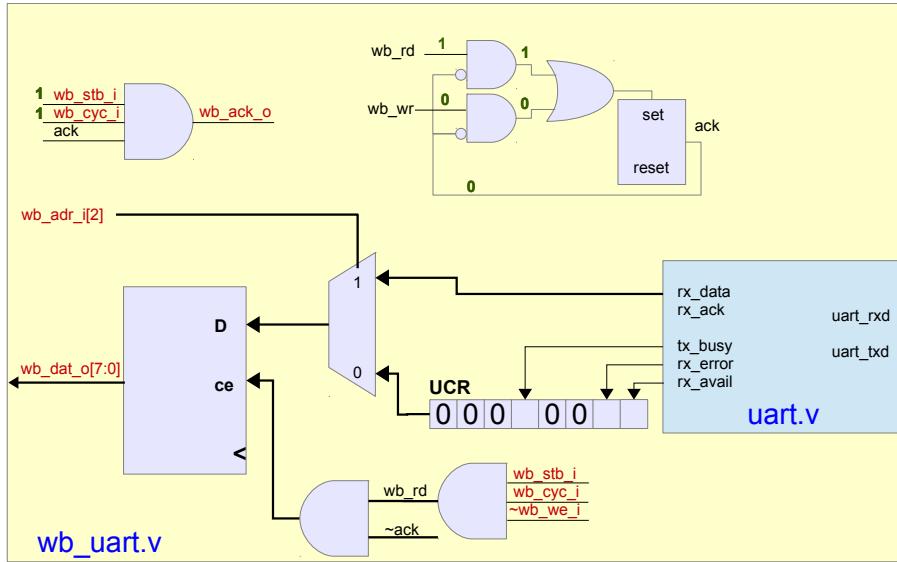


Figura 2.31 Circuito equivalente de lectura de la UART

En la figura 2.34 se muestra el circuito de lectura del periférico timer. La diferencia frente a los anteriores es la posibilidad de leer 6 diferentes variables; por esta razón se utilizan tres señales del bus de direcciones `wb_adr_i[5:3]`. En la figura 2.35 se muestra el diagrama de escritura del timer, de forma similar al circuito de lectura la señal `wb_adr_i[5:3]` selecciona el registro que almacenará el valor proveniente del procesador.

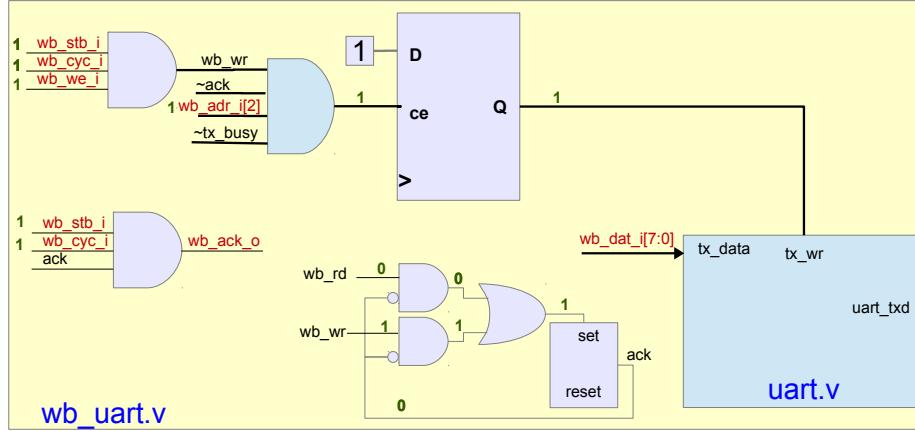


Figura 2.32 Circuito equivalente de escritura de la UART

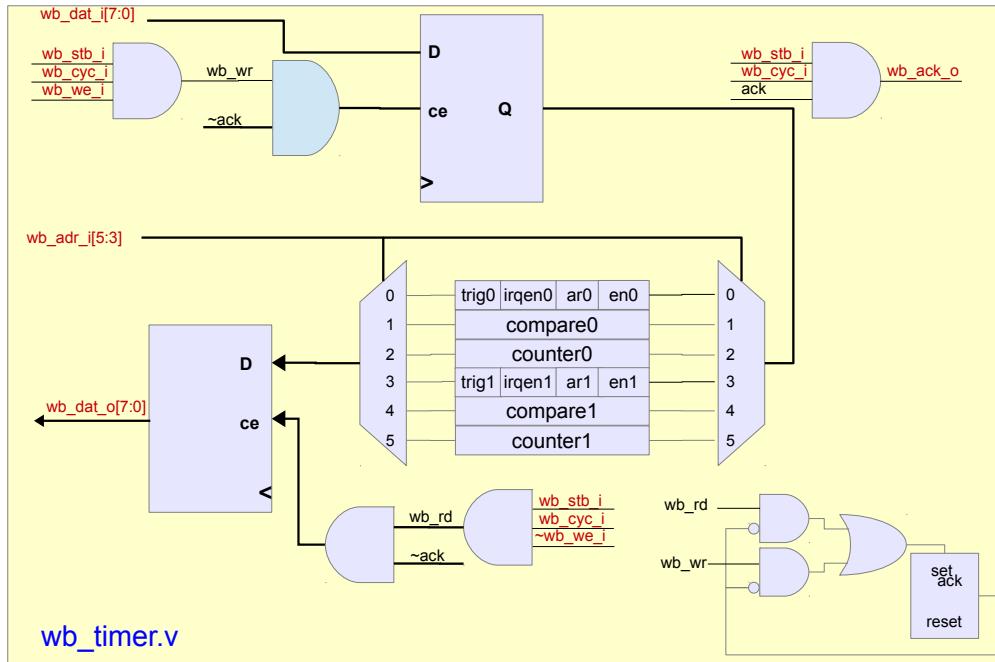


Figura 2.33 Ejemplo de periférico wishbone: TIMER

#### 2.4.3. Interfaz Software

En la subsección anterior se hizo una descripción de los diferentes componentes de la configuración básica del SoC LM32; aquí, se explicará cómo controlar desde un programa en C la comunicación con los periféricos.

##### Estructura de datos del periférico

Para facilitar el acceso a los diferentes registros de un periférico es conveniente declarar un nuevo tipo de dato que haga una representación de su mapa de memoria.

En la figura 2.36 se muestra el diagrama de bloques del GPIO y la declaración del tipo de dato `gpio_t`; el multiplexor que selecciona el sitio donde se almacenará el dato proveniente del procesador está controlado por las líneas de

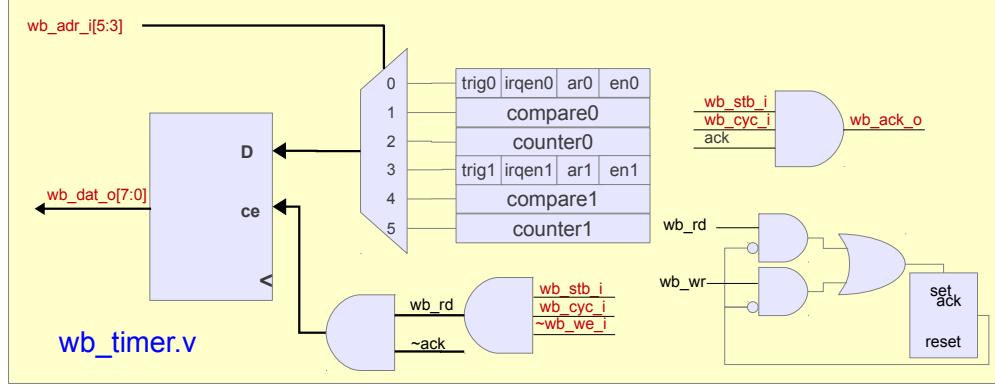


Figura 2.34 Circuito equivalente de lectura del periférico TIMER

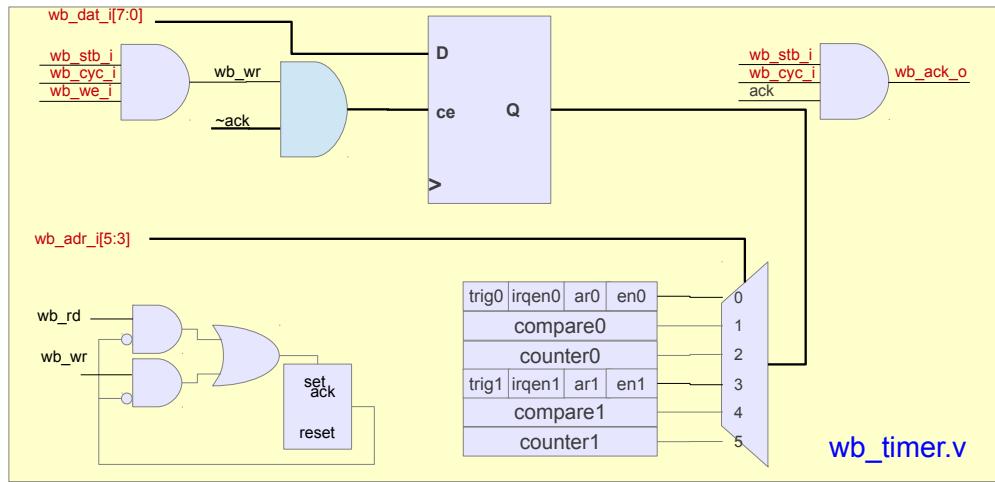


Figura 2.35 Circuito equivalente de escritura del periférico TIMER

dirección  $wb\_adr\_i[3:2]$  cuando estas señales tengan el valor de 01 o lo que es lo mismo la dirección termine en 0x04 se seleccionará el registro  $gpio\_o$ ; si estas señales tengan el valor de 10 o lo que es lo mismo la dirección termine en 0x08 se seleccionará el registro  $gpio\_dir$ . De aquí la posición de los elementos *write* y *w\_dir* de la estructura  $gpio\_t$ ; al ser declarada la variable *read* como  $uint32\_t$  se reservan cuatro bytes (0x00, 0x01, 0x02 y 0x03) para almacenar esta variable, la siguiente posición de memoria (0x04) corresponde a la variable *write*, la cual es declarada como un tipo de dato  $uint32\_t$  por lo que se reservan cuatro bytes para su almacenamiento (0x04, 0x05, 0x06 y 0x07), en la siguiente posición de memoria (0x08) se almacenará la variable *w\_dir* y se reservarán cuatro bytes (0x08, 0x09, 0x0A y 0x0B) para su almacenamiento.

Como puede observarse en la figura 2.36, el contenido del registro  $gpio\_input$  siempre está disponible sin importar el valor de la dirección, lo que indica que el dato estará disponible siempre que se seleccione el periférico para una operación de lectura, en este caso se colocó en la primera posición de memoria por conveniencia.

Todos los tipos de datos declarados en la estructura  $gpio\_t$  son del tipo *volatile*. Este tipo de dato le indica al compilador que no realice optimizaciones sobre esta variable.

En la figura 2.37 se observa la declaración del tipo de dato  $uart\_t$  y su relación con el circuito interno de la UART. Aquí,  $wb\_adr\_i[2]$  controla el valor que será pasado al bus de datos del maestro; si  $wb\_adr\_i[2]$  es 0, se transmite el valor del registro *UCR*, si el valor de  $wb\_adr\_i[2]$  es 1, se transmitirá el valor de la señal *rx\_data*. Al definir la variable *ucr* al comienzo de la estructura y al asignarle el tipo  $uint32\_t$  se reservan los bytes 0x00, 0x01, 0x02 y 0x03 para su almacenamiento; al declarar a continuación la variable  $uint32\_t$  *rxtx* se reservan los siguientes cuatro bytes (0x04, 0x05, 0x06 y 0x07) para su almacenamiento.

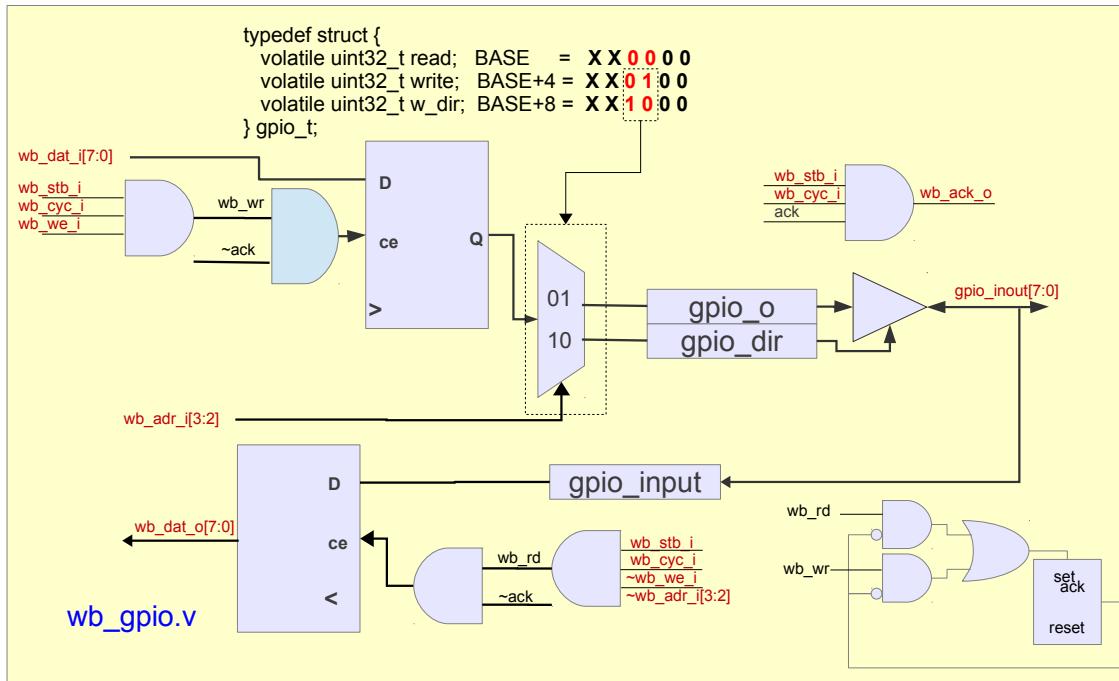


Figura 2.36 Definición de la dirección de los registros internos del GPIO

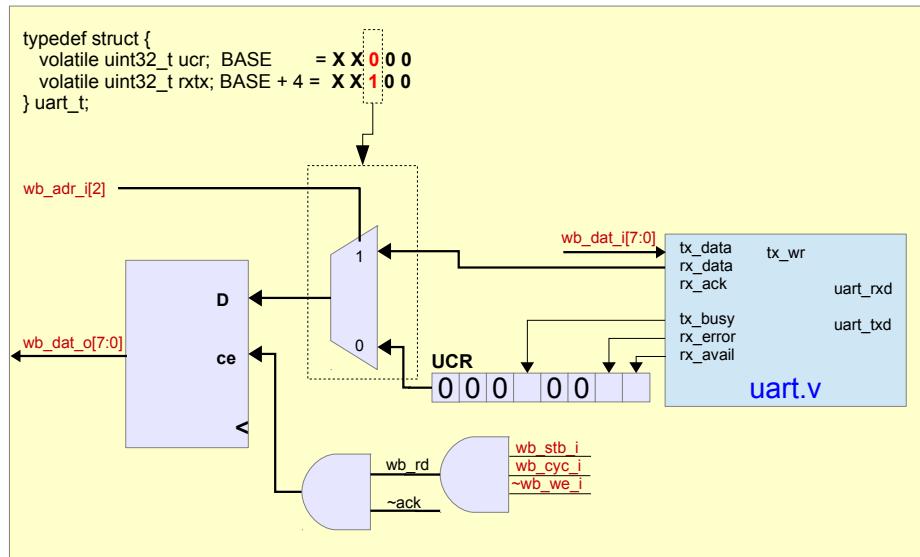


Figura 2.37 Definición de la dirección de los registros internos de la UART

Finalmente, en la figura 2.38 se muestra la declaración del tipo de dato *timer\_t* y su relación con los registros internos del periférico.

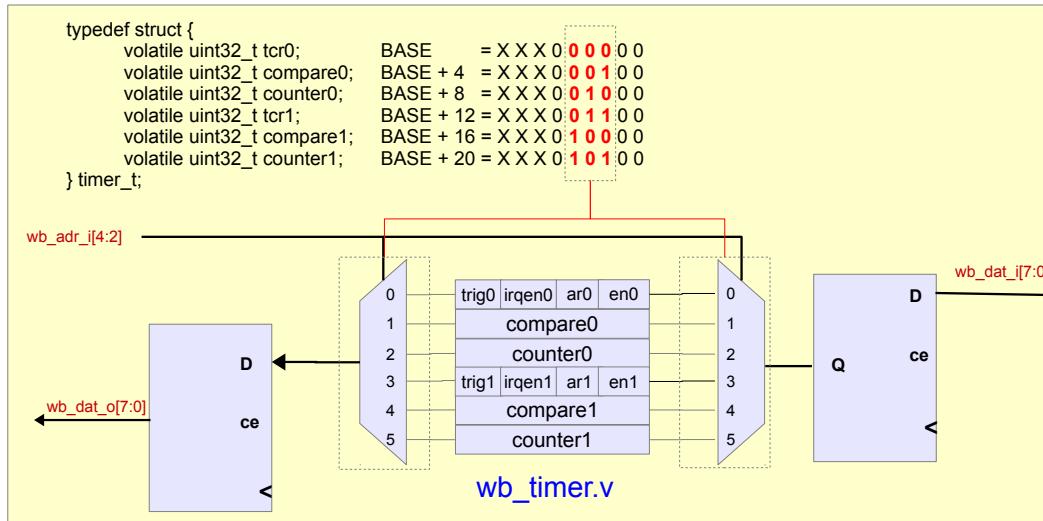


Figura 2.38 Definición de la dirección de los registros internos del TIMER

### Dirección de memoria de los periféricos

Una vez creados los tipos de datos que representan los registros internos de los periféricos se debe asignar un valor a la dirección base de cada uno de ellos, esta dirección debe ser la misma que le asigna el decodificador de direcciones del árbito wishbone. En la figura 2.39 se muestra el valor que deben tomar estas direcciones.

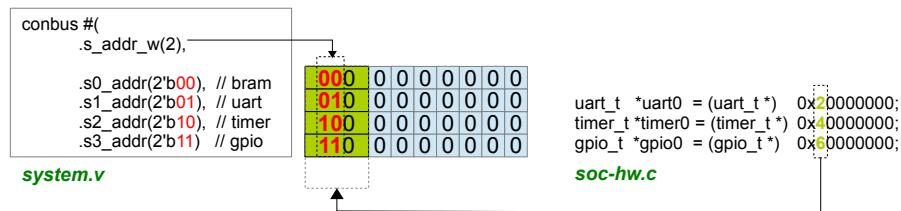
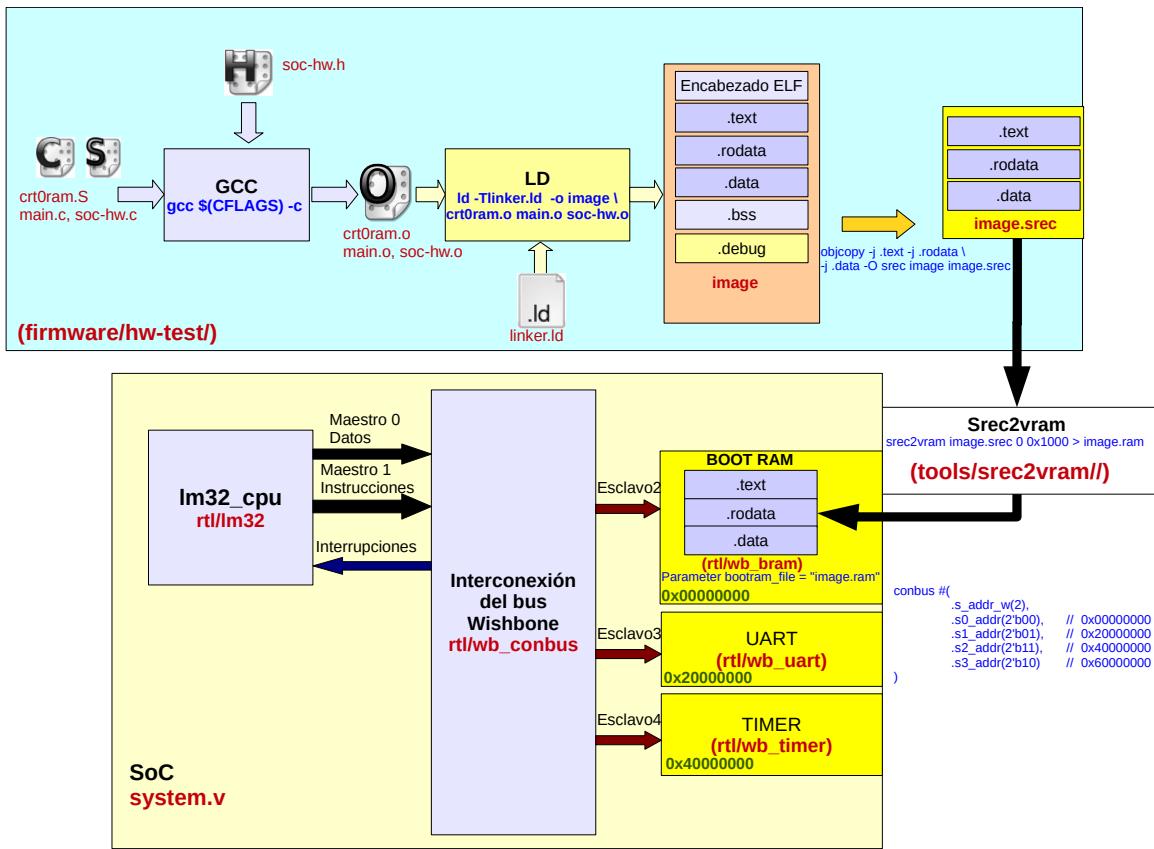


Figura 2.39 Asignación de la dirección de memoria a los periféricos

## 2.5. Programación del SoC LM32

En esta sección se realizará una explicación detallada del flujo de diseño software que debe seguir para implementar aplicaciones en el SoC LM32 (ver figura 4.13). El desarrollo de aplicaciones en el SoC posee dos componentes: hardware y software, el componente hardware implementa tareas en forma de periféricos, mientras que el componente software las implementa como una serie de instrucciones que son ejecutadas de forma secuencial en el procesador MICO32.

El SoC LM32 ejecuta las instrucciones que están almacenadas en la memoria *BOOT RAM* en la posición de memoria *0x00*; para generar las instrucciones que serán ejecutadas es necesario generar un archivo de inicialización (*boot-*



**Figura 2.40** Flujo de diseño para el procesador LM32

*tram\_file = image.ram) para la *BOOT RAM*, la herramienta de síntesis lee este archivo de inicialización y al momento de crear el bloque de memoria inicializa el contenido con dicho archivo.*

En la figura 4.13 se muestra cómo se genera el archivo *image.srec* que contiene las instrucciones que forman la aplicación. Todas las aplicaciones software se encuentran dentro del directorio *firmware* y existe un subdirectorio para cada aplicación *hw-test* para este caso.

## Compilación

La aplicación *hw-test* tiene tres archivos fuente: *crt0ram.S*, *soc-hw.c* y *main.c*, estos archivos deben ser compilados por la herramienta *GCC* para generar archivos que contengan la funcionalidad de cada archivo fuente implementado en el lenguaje del MICO32, estos archivos reciben el nombre de objetos y sus nombres son los mismos que los archivos fuentes pero con la extensión *.o* (*crt0ram.o* *main.o* *soc-hw.o*). Adicionalmente, los objetos suministran información sobre las funciones y señales globales que se declaran en ellos, esta información recibe el nombre de *símbolo*.

## Enlazado

A continuación se debe generar un archivo ejecutable que integre el código de los diferentes objetos, esta tarea la realiza el *enlazador LD*. El enlazador analiza todos los objetos y crea una lista de símbolos clasificándolos en símbolos resueltos o no resueltos; un símbolo se considera resuelto cuando se encuentra el código que lo implementa en cualquiera de los objetos que se van a enlazar, si al finalizar de enlazar todos los objetos no se resuelve un símbolo el enlazador no podrá generar el ejecutable y emitirá un mensaje de error.

El archivo ejecutable que se genera utiliza el formato *ELF (Executable and Linkable Format)* el cuál es un estándar para objetos, librerías y ejecutables. Un ejecutable *ELF* está compuesto por diferentes 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*. Este archivo posee un encabezado que describe el layout del archivo, proporcionando información de la forma de acceder a las secciones. Las secciones pueden almacenar código ejecutable, datos, información de enlazado dinámico, datos de depuración, tablas de símbolos, comentarios, tablas de cadenas, y notas. Las secciones más importantes son:

- **.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 enlace dinámico
- **.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 **.rodata1** Datos de solo-lectura (ROM)
- **.shstrtab** Nombres de secciones.
- **.symtab** Tabla de símbolos.
- **.text** Instrucciones ejecutables (ROM)

## Script de enlazado

El enlazador permite definir donde serán ubicados los diferentes segmentos del archivo ELF por medio de un archivo de enlace que recibe el nombre de *script de enlazado*; de esta forma podemos ajustar el ejecutable a plataformas con diferentes configuraciones de memoria, lo que brinda un grado mayor de flexibilidad de la cadena de herramientas GNU. El archivo de enlazado en este ejemplo tiene el nombre de *linker.ld* y se muestra a continuación.

```
OUTPUT_FORMAT("elf32_lm32")
ENTRY(_start)
.DYNAMIC = 0;
.RAMSTART = 0x00000000;
.RAMSIZE = 0x1000;
.RAMEND = .RAMSTART + .RAM.SIZE;

MEMORY {
    ram : ORIGIN = 0x00000000, LENGTH = 0x1000      /* 4k */
}
SECTIONS
{
    .text :
    {
        _ftext = .;
        *(.text .stub .text.* .gnu.linkonce.t.*)
        _etext = .;
    } > ram
    .rodata :
    {
        . = ALIGN(4);
        _frodata = .;
        *(.rodata .rodata.* .gnu.linkonce.r.*)
        *(.rodata1)
        _erodata = .;
    } > ram
    .data :
    {
        . = ALIGN(4);
        _fdata = .;
        *(.data .data.* .gnu.linkonce.d.*)
        *(.data1)
        _gp = ALIGN(16);
        *(.sdata .sdata.* .gnu.linkonce.s.*)
        _edata = .;
    } > ram
    .bss :
    {
        . = ALIGN(4);
        _fbss = .;
        *(.dynbss)
        *(.sbss .sbss.* .gnu.linkonce.sb.*)
        *(.scommon)
        *(.dynbss)
        *(.bss .bss.* .gnu.linkonce.b.*)
        *(COMMON)
    }
}
```

```

    .ebss = .;
    .end = .;
} > ram
}PROVIDE(_fstack = ORIGIN(ram) + LENGTH(ram) - 4);

```

En este archivo se declara la posición inicial de la memoria (*\_RAM\_START*), su longitud (*\_RAM\_SIZE*) y se declaran las diferentes memorias que posee el sistema, para este caso solo existe la memoria *BOOT RAM* que recibe el nombre de *ram*. Las secciones son declaradas en el orden que serán almacenadas, en la *ram* iniciando con la sección *text* en la posición de memoria *0x00000000*; el enlazador coloca todo el lenguaje de máquina extraído de los objetos generados a partir del código fuente a partir de esta dirección; a continuación coloca las constantes utilizadas en el código (como los #define, las cadenas de caracteres a ser impresas) que hacen parte de la sección *rodata*; después las variables inicializadas que hacen parte de la sección *data* y finalmente las variables sin inicializar de la sección *bss*. Adicionalmente, este archivo declara un espacio de memoria llamado *fstack*, el cual realizará las funciones de la pila. Al finalizar el proceso de enlazado se genera un archivo ejecutable llamado *image*.

## Utilitarios binarios

El formato ELF no puede ser utilizado directamente para generar la memoria de programa del procesador; para ello, se debe extraer de este la información correspondiente a las secciones *text*, *rodata* y *data*. Para realizar esta tarea la cadena de herramientas GNU proporciona la utilidad *objcopy*, la cual puede generar un archivo de salida en varios formatos, en este caso se genera el archivo *image.srec* con formato S-record.

## Inicialización de la memoria BRAM

El bloque de memoria bram se utiliza como memoria de programa, memoria RAM y pila; por este motivo es necesario inicializarla con los datos de la aplicación. La inicialización de esta memoria se realiza en el archivo *rtl/wb\_bram/wb\_bram.v*:

```

initial
begin
    if (mem_file.name != "none")
    begin
        $readmemh(mem_file.name, ram);
    end
end

```

Donde el parámetro *mem\_file.name* es pasado desde el archivo *system.v*

```

parameter bootram_file = "../firmware/hw-test/image.ram",
...
wb_bram #(
    .adr_width( 12 ),
    .mem_file_name( bootram_file )
)

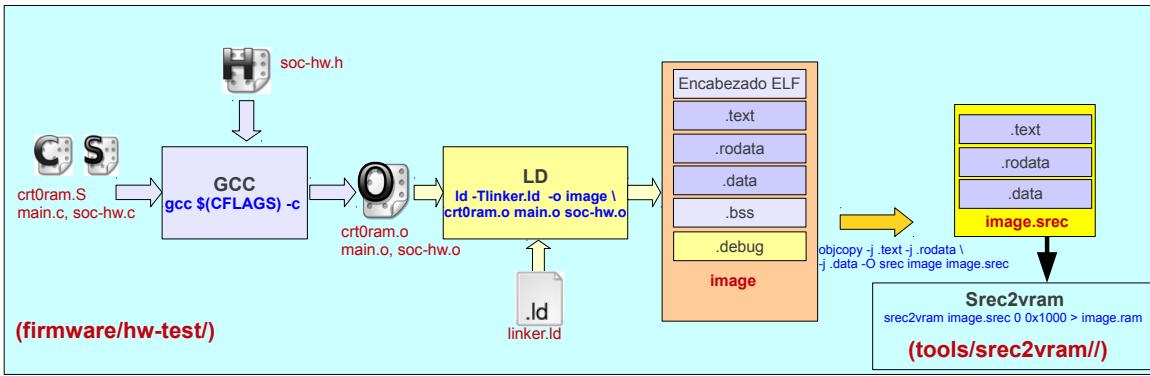
```

El archivo *image.ram* es un archivo de texto plano donde se inicializan todas las 0x1000 posiciones de memoria, este archivo es generado por la utilidad *srec2vram*.

### 2.5.1. Ejemplo de programación

En esta sección se explicará de forma detallada un ejemplo que ayudará a entender cómo se deben escribir las aplicaciones para este tipo de arquitectura. El código fuente de este ejemplo se encuentra en el directorio *firmware/hw-test* y está compuesto por los archivos que se muestran en la figura 2.41.

El proceso de compilación requiere ejecutar una serie de comandos que pueden llegar a ser un poco engorrosos en la fase de diseño; por esta razón, la comunidad del software libre creó una aplicación que permite automatizar este proceso, esta utilidad recibe el nombre de *make*; *make* lee los pasos que debe ejecutar para compilar y generar los archivos necesarios para la aplicación desde un script que recibe el nombre de *Makefile*.



**Figura 2.41** Flujo de diseño para el procesador LM32

### Makefile

En esta subsección se explicará el contenido del archivo Makefile para el ejemplo bajo estudio; en el siguiente listado se muestra la primera parte del archivo:

```
LM32_CC=lm32-linux-gcc
LM32_LD=lm32-linux-ld
LM32_OBJCOPY=lm32-linux-objcopy
LM32_OBJDUMP=lm32-linux-objdump

SREC2VRAM ?= ../../tools/srec2vram/srec2vram
VRAMFILE=image.ram

CFLAGS=-MMD -O2 -Wall -g -fomit-frame-pointer -mbarrel-shift-enabled
-mmultiply-enabled -mdivide-enabled -msign-extend-enabled
LDFLAGS=-nostdlib -nodefaultlibs -Tlinker.ld
SEGMENTS = -j .text -j .rodata -j .data
```

En este segmento del archivo Makefile se declaran variables locales que serán utilizadas posteriormente. Es costumbre crear una variable global con el nombre de los ejecutables de: el compilador (*LM32\_CC*), el enlazador (*LM32\_LD*), manipulador de binarios (*LM32\_OBJCOPY*) y utilidad para generar el listado del código en assembler del ejecutable (*LM32\_OBJDUMP*); esto con el fin de que el mismo Makefile pueda ser usado con diferentes cadenas de herramientas. Adicionalmente, se define el sitio del ejecutable *srec2vram* que como se mencionó anteriormente genera el archivo de inicialización de la memoria BRAM; se define el nombre de la imagen que contiene la memoria de programa (*srec2vram*).

Las variables *CFLAGS* y *LDFLAGS* son parámetros que se pasan al compilador y al enlazador respectivamente; *-Tlinker.ld*, le indica al enlazador que utilice el archivo *linker.ld* para definir la distribución de memoria del ejecutable. En la siguiente sección del archivo Makefile se realizan las tareas necesarias para generar el archivo *image.ram*

```
all: image.srec $(VRAMFILE)

crt0ram.o: crt0ram.S
    $(LM32_CC) $(CFLAGS) -c crt0ram.S

main.o: main.c
    $(LM32_CC) $(CFLAGS) -c main.c

soc-hw.o: soc-hw.c
    $(LM32_CC) $(CFLAGS) -c soc-hw.c

image: crt0ram.o main.o soc-hw.o
    $(LM32_LD) $(LDFLAGS) -Map image.map -N -o image crt0ram.o main.o soc-hw.o

image.lst: image
    $(LM32_OBJDUMP) -h -S $<> $@

image.srec: image.lst
    $(LM32_OBJCOPY) $(SEGMENTS) -O srec image image.srec

$(VRAMFILE): image.srec
    $(SREC2VRAM) image.srec 0x00000000 0x1000 > $(VRAMFILE)

clean:
    rm -f image image.lst image.bin image.srec image.map *.o *.d
```

Cada cadena de caracteres que termina con dos puntos : es un posible parámetro que puede ser pasado a la herramienta *make*, la que se encarga de leer el Makefile y ejecutar las operaciones en él asignadas. Al ejecutar el comando *make* sin parámetros este busca en el directorio donde fué ejecutado un archivo con el nombre *Makefile* o *makefile* y ejecutará lo que encuentre en la etiqueta *all*; los nombres que aparecen después de los dos puntos, son dependencias que se deben

ejecutar antes de realizar las acciones propias; en este caso se deben buscar las etiquetas *image.srec* y *\$(VRAMFILE)* y ejecutarlas.

Para ejecutar *image.srec* es necesario ejecutar antes *image.lst*, la cual a su vez requiere que se ejecute *image*, este tipo de encadenamientos son típicos en estos archivos y son necesarios para seguir el flujo de compilación. Para ejecutar las acciones de *image* es necesario ejecutar *crt0ram.o*, *main.o* y *soc-hw.o* las cuales compilan el código fuente *crt0ram.S*, *main.c* y *soc-hw.c* para generar los objetos correspondientes; una vez creados los objetos se pueden ejecutar la acción de *image*: *\$(LM32\_LD) \$(LDFLAGS) -Map image.map -N -o image crt0ram.o main.o soc-hw.o*, este comando llama al enlazador para que cree el ejecutable *image* a partir de los objetos *crt0ram.o*, *main.o* y *soc-hw.o*. A continuación se ejecutarán las operaciones de *image.lst*, seguidas por las de *image.srec* y *\$(VRAMFILE)*.

El siguiente es el resultado de ejecutar el comando *make*:

```
Im32-linux-gcc -MD -O2 -Wall -g -s -fomit-frame-pointer -mbarrel-shift-enabled -mmultiply-enabled -mdivide-enabled -msign-extend-enabled -c crt0ram.S
Im32-linux-gcc -MD -O2 -Wall -g -s -fomit-frame-pointer -mbarrel-shift-enabled -mmultiply-enabled -mdivide-enabled -msign-extend-enabled -c main.c
Im32-linux-gcc -MD -O2 -Wall -g -s -fomit-frame-pointer -mbarrel-shift-enabled -mmultiply-enabled -mdivide-enabled -msign-extend-enabled -c soc-hw.c
Im32-linux-ld -nostdlib -nodefaultlibs -Tlinker.ld -Map image.map -N -o image crt0ram.o main.o soc-hw.o
Im32-linux-objcopy -j .text -j .rodata -j .data -O srec image image.srec
../../../../tools/srec2vram/srec2vram image.srec 0x00000000 0x1000 > image.ram
Extracting [00000000,00001000) (size=0x1000)
```

En la figura 2.42 se muestra la relación de los diferentes archivos involucrados en este ejemplo, mostrando las funciones relacionadas con la UART.

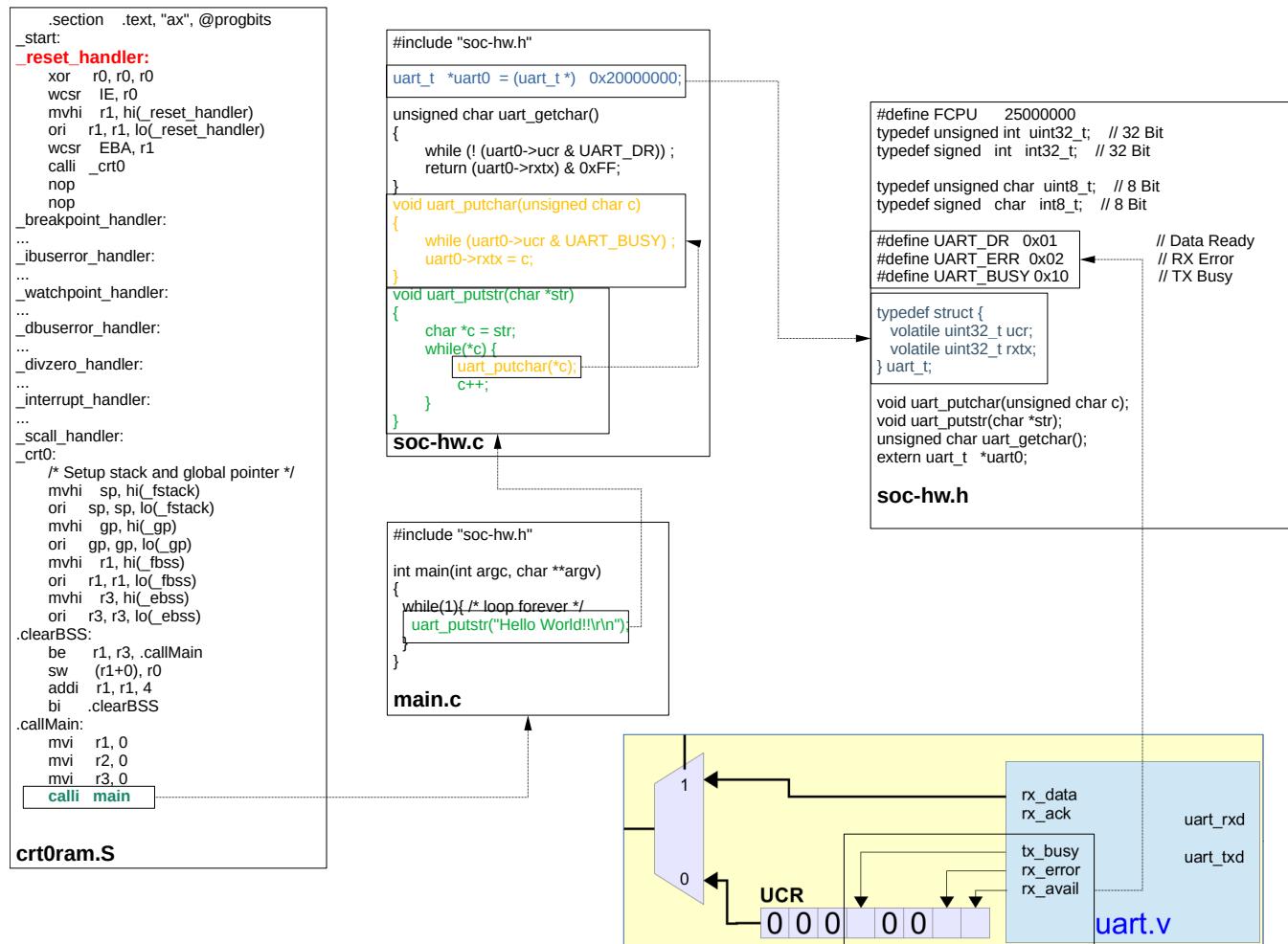


Figura 2.42 Aplicación Hello World

## Capítulo 3

# Procesador RISCV

### 3.1. Introducción

En el capítulo anterior se estudió la forma de implementar tareas hardware utilizando máquinas de estado algorítmicas. La implementación de tareas hardware es un proceso un poco tedioso ya que involucra la realización de una máquina de estados por cada tarea; la implementación del camino de datos se simplifica de forma considerable ya que existe un conjunto de bloques constructores que pueden ser tomados de una librería creada por el diseñador. El uso de tareas hardware se debe realizar únicamente cuando las restricciones temporales del diseño lo requieran, ya que como veremos en este capítulo, la implementación de tareas software es más sencilla y rápida.

La estructura de una máquina de estados algorítmica permite entender de forma fácil la estructura de un procesador ya que tienen los mismos componentes principales (unidad de control y camino de datos), la diferencia entre ellos es la posibilidad de programación y la configuración fija del camino de datos del procesador.

En este capítulo se estudiará la arquitectura del procesador MICO32 creado por la empresa Lattice semiconductor y gracias a que fué publicado bajo la licencia GNU, es posible su estudio, uso y modificación. En la primera sección se hace la presentación de la arquitectura; a continuación se realiza el análisis de la forma en que el procesador implementa las diferentes instrucciones, iniciando con las operaciones aritméticas y lógicas siguiendo con las de control de flujo de programa (saltos, llamado a función); después se analizarán la comunicación con la memoria de datos; y finalmente el manejo de interrupciones.

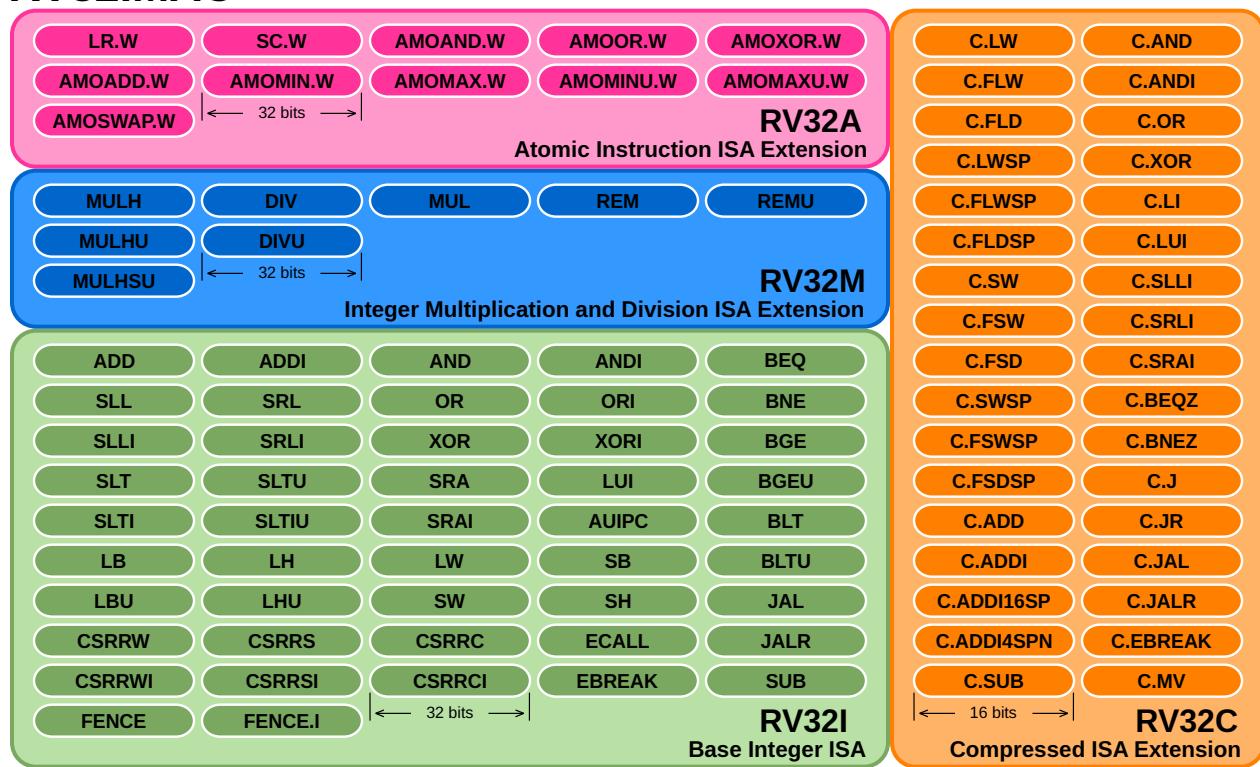
En la segunda sección se abordará la arquitectura de un SoC (System on a Chip) basado en el procesador LM32, se analizará la forma de conexión entre los periféricos y la CPU utilizando el bus wishbone; se realizará una descripción detallada de la programación de esta arquitectura utilizando herramientas GNU.

Este proceso se repetirá para el procesador RISC-V; RISC-V es una arquitectura de conjunto de instrucciones de hardware libre basado en un diseño de conjunto de instrucciones reducido (RISC), su carácter libre le permite ser utilizado sin tener que pagar licencias de ningún tipo. En este capítulo se utilizará la descripción del conjunto de instrucciones RV32I implementada por Bruno Levy

### 3.2. Arquitectura del procesador RV32I

El RISC-V posee una estructura modular que permite adaptar la arquitectura a requerimientos funcionales y económicos, en la figura 3.1 se muestra el concepto de modularidad, el módulo base es un procesador de 32 bits de base entera (RV32I) al que se le pueden integrar módulos para realizar multiplicaciones y divisiones enteras (RV32M), instrucciones atómicas (instrucciones que automáticamente modifican la lectura-escritura a memoria para permitir sincronización entre múltiples procesadores RISC-V) (RV32A) e instrucciones comprimidas (Instrucciones que engloban una serie de instrucciones, se usa para reducir el tamaño de la memoria de programa) (RV32C). Adicionalmente existen otras 24 extensiones entre las que se encuentran las de operaciones aritméticas de punto flotante de precisión simple (RV32F) y las de doble precisión (RV32D).

## RV32IMAC



**Figura 3.1** Set de instrucciones modular de la variante RV32IMAC. CPU de 32 bits de Base entera (RV32I), con extensión ISA para multiplicación y división entera (RV32M), instrucciones Atómicas (RV32A) e instrucciones Comprimidas (RV32C): fuente: Eduardo Corpeño (<https://github.com/kuashio>)

## Registros

El RISC-V posee 32 registros (o 16 en la variante embebida). En la tabla 3.1 se muestran sus nombres.

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

**Cuadro 3.1** Registros del procesador RV32I.

## Set de Instrucciones

En la tabla 3.2 se puede apreciar la lista de las instrucciones del RV32I con su respectivo código y función.

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	ADD	R	0110011	0x0	0x00	$rd = rs1 + rs2$
sub	SUB	R	0110011	0x0	0x20	$rd = rs1 - rs2$
xor	XOR	R	0110011	0x4	0x00	$rd = rs1 \wedge rs2$
or	OR	R	0110011	0x6	0x00	$rd = rs1 \mid rs2$
and	AND	R	0110011	0x7	0x00	$rd = rs1 \& rs2$
sll	Shift Left Logical	R	0110011	0x1	0x00	$rd = rs1 << rs2$
srl	Shift Right Logical	R	0110011	0x5	0x00	$rd = rs1 >> rs2$
sra	Shift Right Arith*	R	0110011	0x5	0x20	$rd = rs1 >> rs2$
slt	Set Less Than	R	0110011	0x2	0x00	$rd = (rs1 \mid rs2) ? 1 : 0$
sltu	Set Less Than (U)	R	0110011	0x3	0x00	$rd = (rs1 \mid rs2) ? 1 : 0$
addi	ADD Immediate	I	0010011	0x0		$rd = rs1 + imm$
xori	XOR Immediate	I	0010011	0x4		$rd = rs1 \wedge imm$
ori	OR Immediate	I	0010011	0x6		$rd = rs1 \mid imm$
andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	$rd = rs1 << imm[0:4]$
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	$rd = rs1 >> imm[0:4]$
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	$rd = rs1 >> imm[0:4]$
slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 \mid imm) ? 1 : 0$
sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 \mid imm) ? 1 : 0$
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$
sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm
bge	Branch ≥	B	1100011	0x5		if(rs1 ≥ rs2) PC += imm
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm
bgou	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm
jal	Jump And Link	J	1101111			$rd = PC+4; PC += imm$
jalr	Jump And Link Reg	I	1100111	0x0		$rd = PC+4; PC = rs1 + imm$
lui	Load Upper Imm	U	0110111			$rd = imm << 12$
auipc	Add Upper Imm to PC	U	0010111			$rd = PC + (imm << 12)$
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger

Cuadro 3.2 Set de Instrucciones del RV32I. fuente:<https://github.com/jameslzhu/riscv-card/tree/master>

### 3.2.1. Diagrama de Bloques del FemtoRV

Como se mencionó anteriormente, en este capítulo se utilizará el procesador **femtoRV** desarrollado por Bruno Levy<sup>1</sup>. En la figura 3.2 se muestra el diagrama de bloques del procesador.

En esta figura podemos observar el camino de datos típico del procesador sin etapas de *pipeline*; es un camino compuesto por la Unidad Aritmética y Lógica que a su vez se encarga de calcular los valores del contador de programa (PC) ante instrucciones de salto, y se encarga de realizar las operaciones para los saltos condicionales. La interfaz del procesador con el mundo exterior se realiza a través de los buses:

<sup>1</sup> <https://github.com/BrunoLevy>

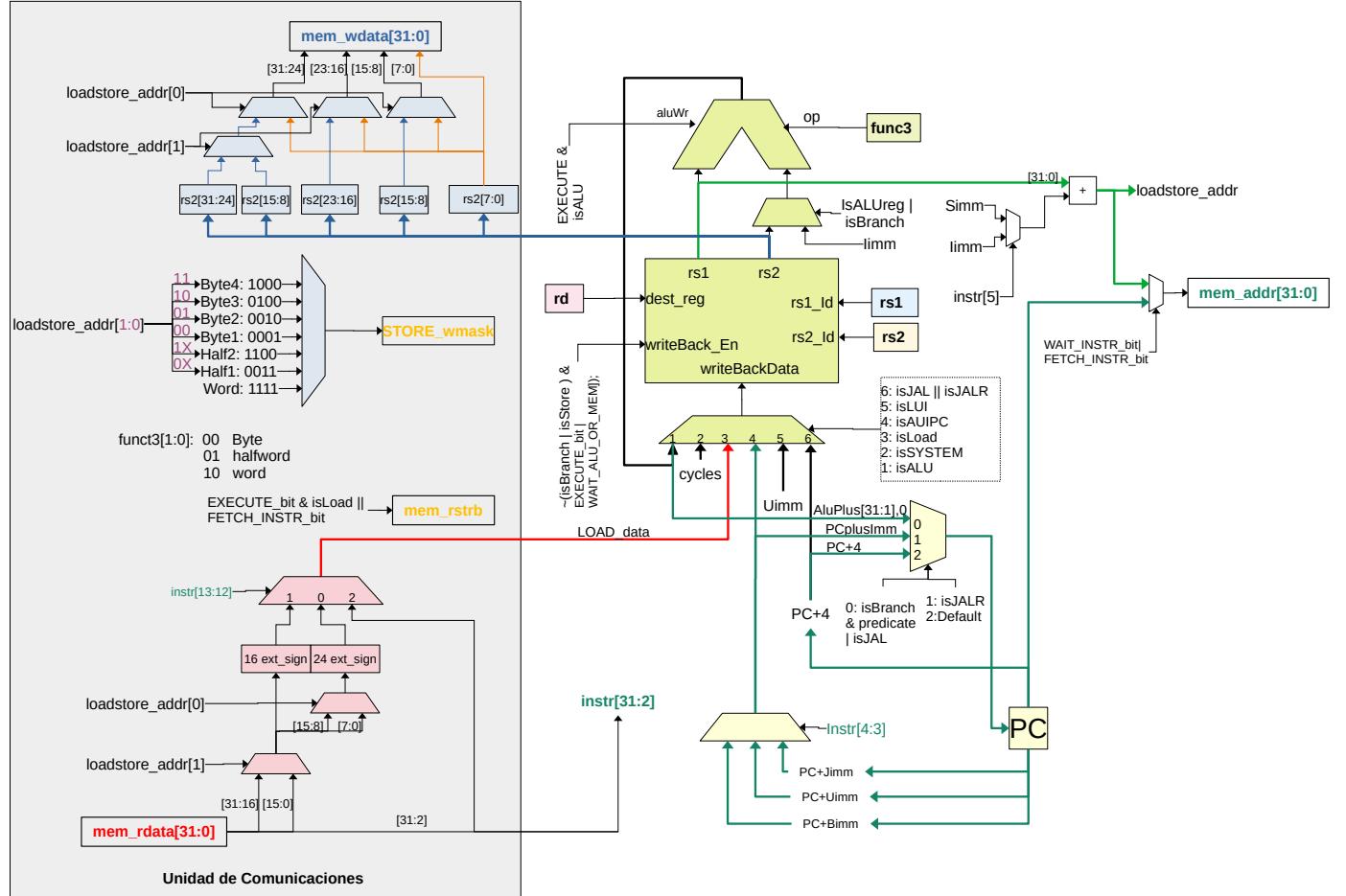


Figura 3.2 Diagrama de bloques del RV32I

- Dirección: **mem\_addr**: Encargado de direccionar la memoria externa donde se almacenan las instrucciones que componen el programa y de realizar operaciones de lectura y escritura a los periféricos.
- Salida de Datos: **mem\_wdata**: Por este bus se envían los datos hacia los periféricos.
- Entrada de Datos: **mem\_rdata**: Por este bus se reciben los datos provenientes de los periféricos y las instrucciones almacenadas en la memoria de programa.
- Control **mem\_rstb**, **store\_wmask**: Señales que indican a los periféricos y a la memoria de programa cuando se pueden realizar las operaciones de lectura y escritura.

### 3.2.2. Set de Instrucciones

Como se mencionó anteriormente, el RISC-V posee un set de instrucciones reducido que permite implementar las diferentes aplicaciones, en esta sub-sección mostraremos cómo se implementan las diferentes instrucciones en el camino de datos mostrado en la figura 3.2.

### Instrucciones aritméticas y lógicas

Las instrucciones aritméticas y lógicas que puede implementar la variante RV32I pueden verse en la figura 3.3. Existen dos formatos para este tipo de operaciones, en el primero los operandos y el resultado se almacenan en el banco de registros, mientras que en el segundo uno de los operandos proviene de la misma instrucción, en las figuras 3.3 y 3.4 se muestran los componentes del camino de datos que intervienen en estas instrucciones y el formato de la instrucción.

**Operaciones Aritméticas y lógicas entre registros** Como puede verse en la figura 3.3 en la instrucción existen campos donde se indica el ID de los registros fuente y destino **rs1**, **rs2** y **rd**, cada uno de estos campos tiene 5 bits, lo que permite colocar cualquiera de los 32 registros del RISC-V. El camino de datos se simplifica al conectar la salida del banco de registros a la entrada de la ALU y la salida de esta a la entrada de datos del banco de registros.

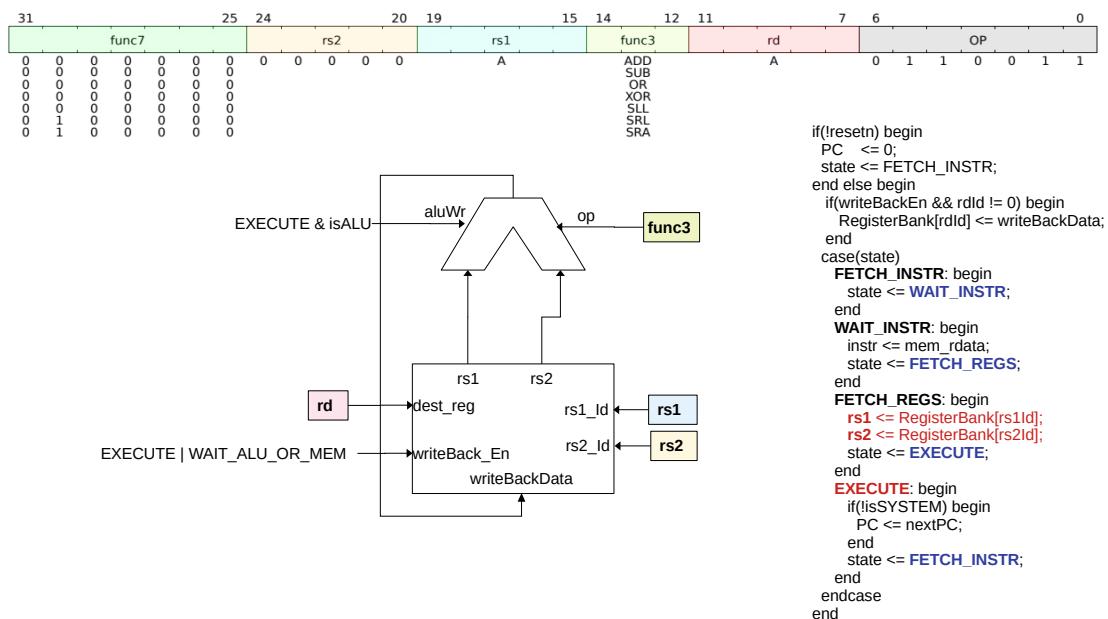


Figura 3.3 Instrucciones aritméticas entre registros del RV32I

**Operaciones Aritméticas y Lógicas con un operador inmediato** Las operaciones aritméticas pueden realizarse con operandos que están inmersos en la instrucción a este tipo de operaciones se les conoce con el nombre de inmediatas; como puede verse en la figura 3.4, la instrucción posee 12 bits reservados para este operando, 5 bits para el ID del registro donde se almacena el segundo operando **rs1** y 5 bits para el ID del registro donde se almacenará el resultado **rd**. El camino de datos se modifica para que la ALU reciba directamente el operando desde la instrucción.

### Simulación de las instrucciones aritméticas y lógicas.

En la figura 3.8 se muestra la simulación

#### 3.2.3. Saltos

No es posible realizar algoritmos sin que se realicen saltos (cambios del flujo de ejecución del procesador), estos permiten realizar decisiones, ciclos, etc. Existen dos tipos de saltos:

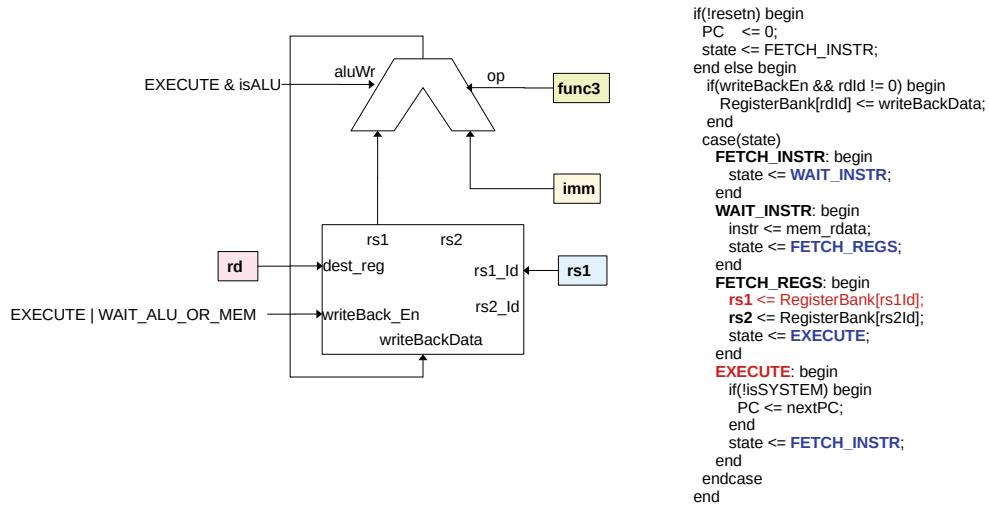


Figura 3.4 Instrucciones aritméticas inmediatas del RV32I

- Incondicionales: El salto se realiza sin ninguna condición.
- Condicionales: Se debe cumplir una condición para que se pueda realizar el salto.

### Saltos Incondicionales

Estas instrucciones permiten modificar el flujo de ejecución del programa y existen dos tipos:

- JAL: Valor a donde se salta almacenado en la Instrucción.
- JALR: Valor a donde se salta almacenado en la Instrucción y en un Registro.

En ambos casos la acción de la instrucción es modificar el valor del contador de programa (PC) para que la siguiente instrucción a ejecutar se encuentre en un lugar diferente al actual. En ambos casos se almacena el valor de la siguiente instrucción que se debería ejecutar si no existiera el salto ( $PC+4$ ) lo que permite volver al sitio del llamado del salto (así operan las funciones).

En la figura 3.6 se muestra la instrucción **JAL** en ella podemos ver que el contador de programa se modifica a  $PC + Imm$ .

En la figura 3.7 se muestra la instrucción **JALR**, aquí el valor actual del PC se almacena en el banco de registros y la suma del valor del registro **rs1** se suma al valor imm proveniente de la instrucción se almacena en el contador de programa (PC).

### 3.2.4. Arquitectura del SOC basado en RV32I

### 3.2.5. Extensión RV32M

El set de instrucciones RV32I no posee un multiplicador ni un divisor implementado en hardware, por esta razón es necesario incluir la librería matemática al momento de realizar el enlazado.

Si se observa el archivo firmware.lst generado en los ejemplos que se simularon para el RV32I se observan las funciones `_divsi3`, `_umodsi3`, y `_modsi3`. El problema de realizar estas operaciones por software es que toman muchos

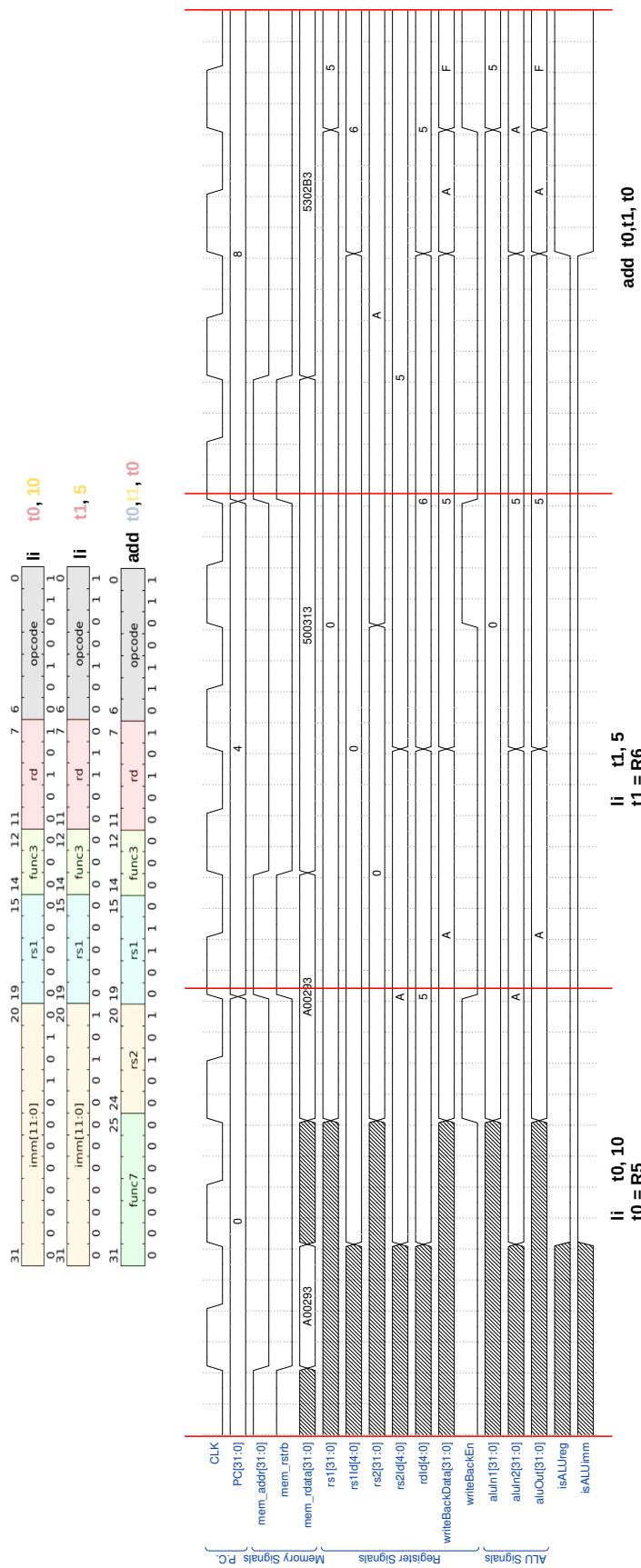
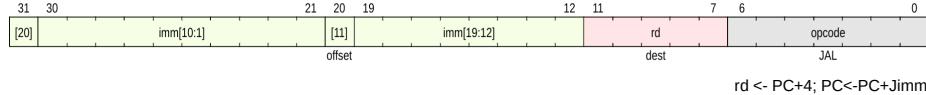


Figura 3.5 Formas de onda de la simulación de las instrucciones aritméticas inmediatas y entre registros del RV32I



```

if(resetn) begin
    PC <= 0;
    state <= FETCH_INSTR;
end else begin
    if(writeBackEn && rdId != 0) begin
        RegisterBank[rdId] <= writeBackData;
    end
    case(state)
        FETCH_INSTR: begin
            state <= WAIT_INSTR;
        end
        WAIT_INSTR: begin
            instr <= mem_rdata;
            state <= FETCH_REGS;
        end
        FETCH_REGS: begin
            rs1 <= RegisterBank[rs1Id];
            rs2 <= RegisterBank[rs2Id];
            state <= EXECUTE;
        end
        EXECUTE: begin
            if(!isSYSTEM) begin
                PC <= nextPC;
            end
            state <= isLoad ? LOAD :
            isStore ? STORE :
            FETCH_INSTR;
        end
        LOAD: begin
            state <= WAIT_DATA;
        end
        WAIT_DATA: begin
            state <= FETCH_INSTR;
        end
        STORE: begin
            state <= FETCH_INSTR;
        end
    endcase
end

```

Figura 3.6 Instrucción JAL en el RV32I

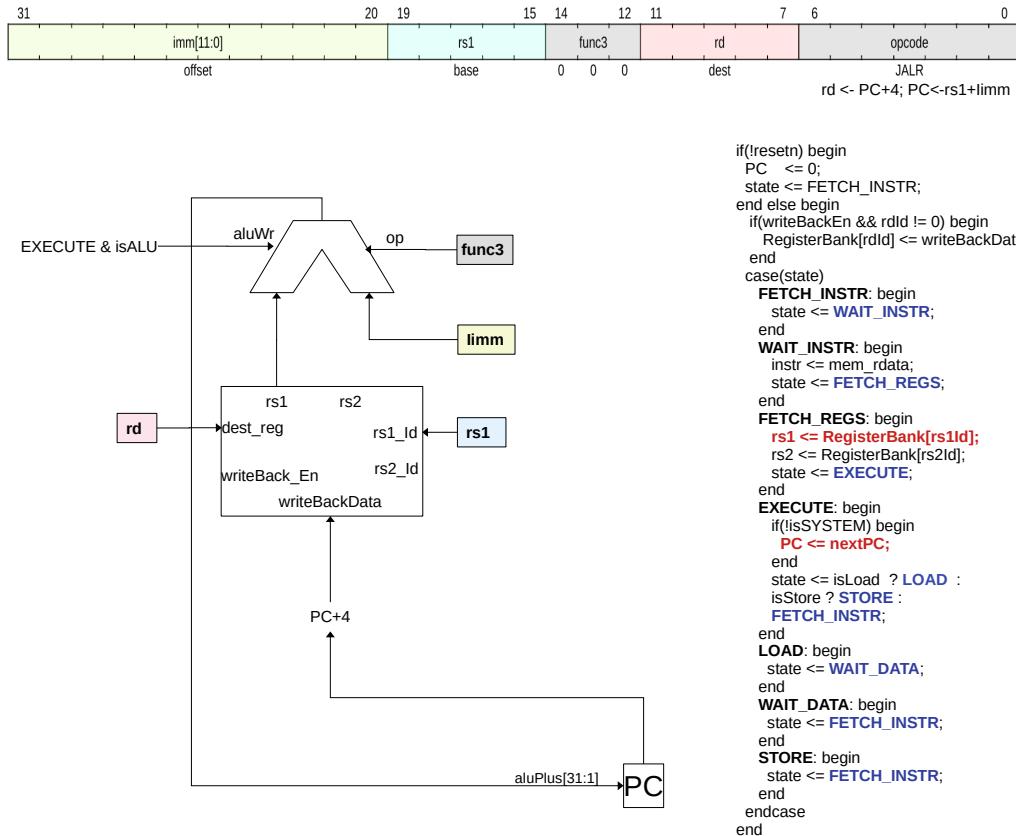
ciclos de reloj, lo que reduce el tiempo de respuesta del procesador y puede poner en riesgo el cumplimiento de las restricciones temporales. La extensión RV32M (ver tabla 3.3) añade estas operaciones al incluir módulos hardware que las implementan. En la tabla

Cuadro 3.3 Extensión RM32M fuente: <https://github.com/jameslzhu/riscv-card/>

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

Bruno Levy modifica la ALU para incluir estas instrucciones adicionando los bloques que se muestran en la figura 3.16

En la figura 3.19 se muestra el diagrama de bloques del RV32M la ALU contiene los bloques representados en la figura 3.16, en este diagrama se elimina la entrada directa del contador de ciclos del RV32I (*cycles*) y se pasa a una estructura **CSR** (Registro de Status y Control), el funcionamiento es el mismo y es necesario ya que con este módulo se puede implementar la función *delay*.

**Figura 3.7** Instrucción JALR del RV32I

La máquina de control es casi igual a la del RV32I, se introduce una nueva condición para needToWait, para el RV32I dependía de isLoad e isStore, para el RV32M se adiciona isDivide ya que el algoritmo de división se ejecuta en varios ciclos de reloj.

### 3.2.6. Flujo Hardware y Software para programar el RV32I

En la Figura

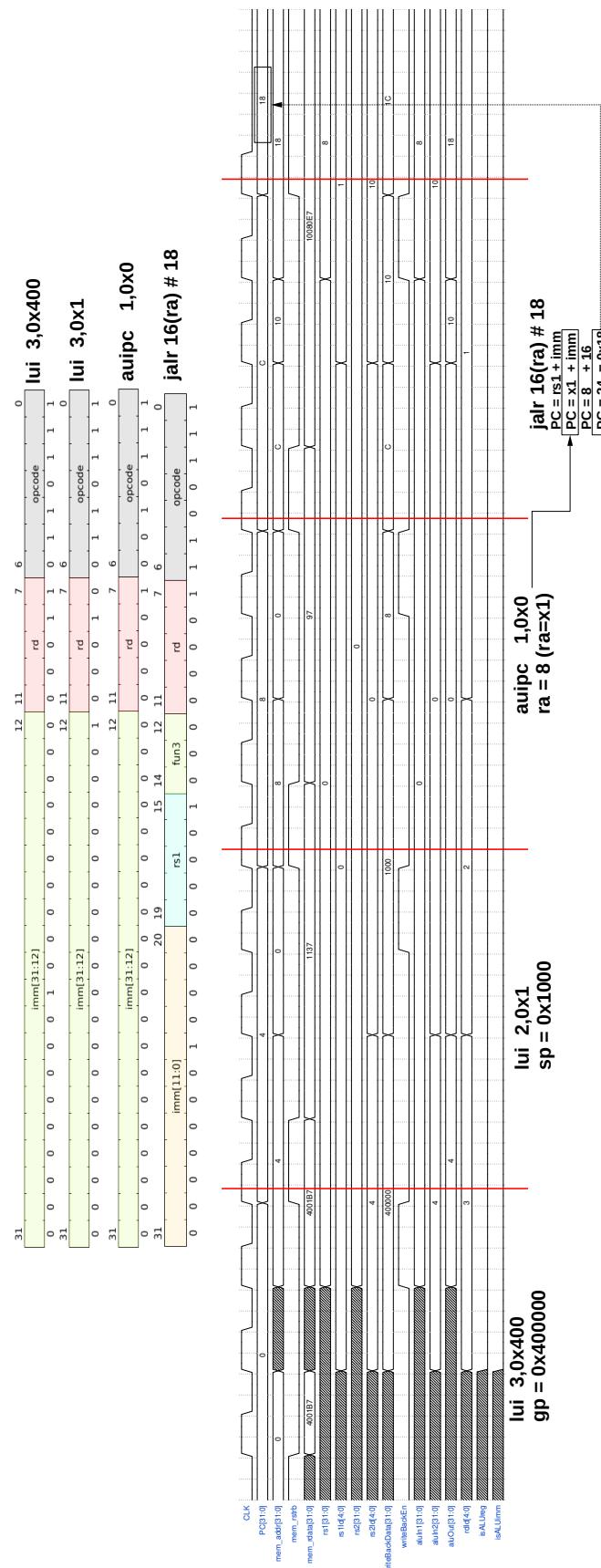
En la página <https://luplab.gitlab.io/rvcodecjs/> se encuentra una utilidad que permite decodificar las instrucciones y facilita el entendimiento de los programas

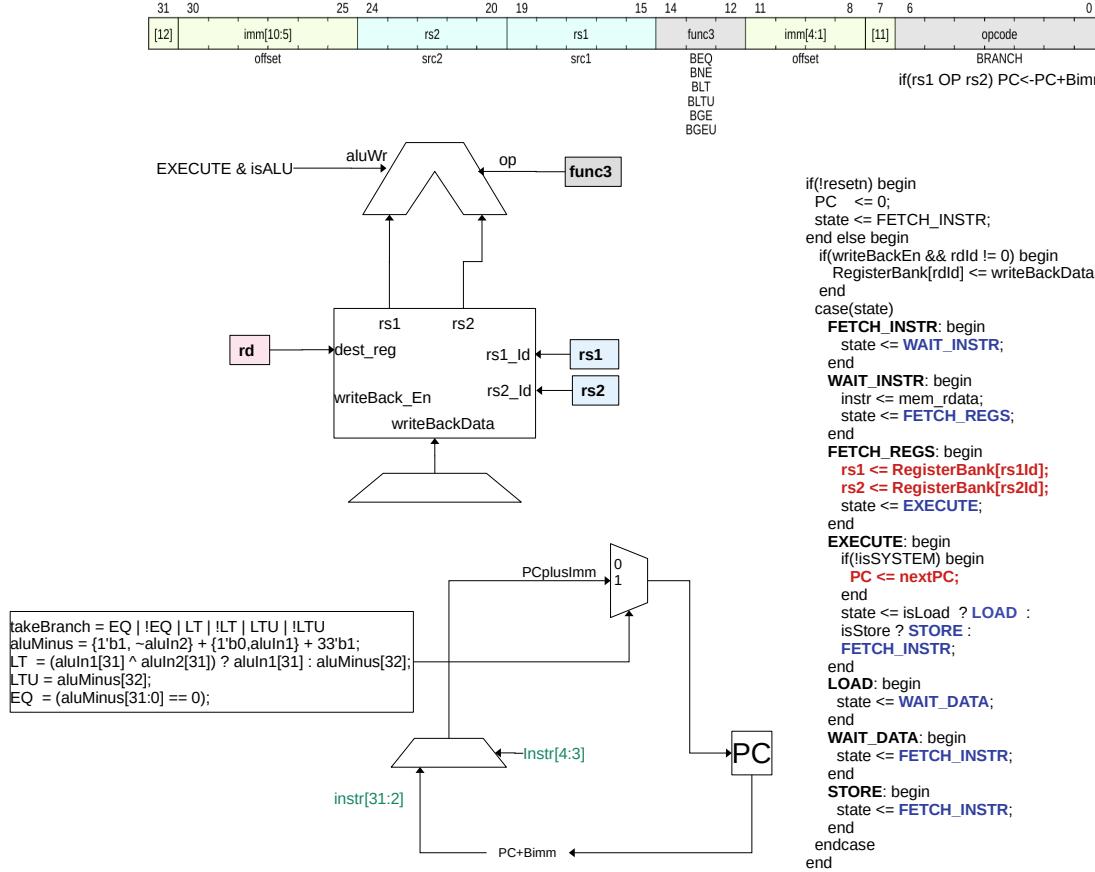
### 3.2.7. Ejemplos

#### Conversor Binario a BCD

### 3.3. Procesador RV32IMC

En la figura 3.21 se muestra el diagrama de bloques del procesador RV32IMC, este procesador soporta operaciones de multiplicación y división, instrucciones comprimidas e Interrupciones.

**Figura 3.8** Formas de onda de la simulación de la instrucción JALR del RV32I

**Figura 3.9** Saltos condicionales en el RV32I

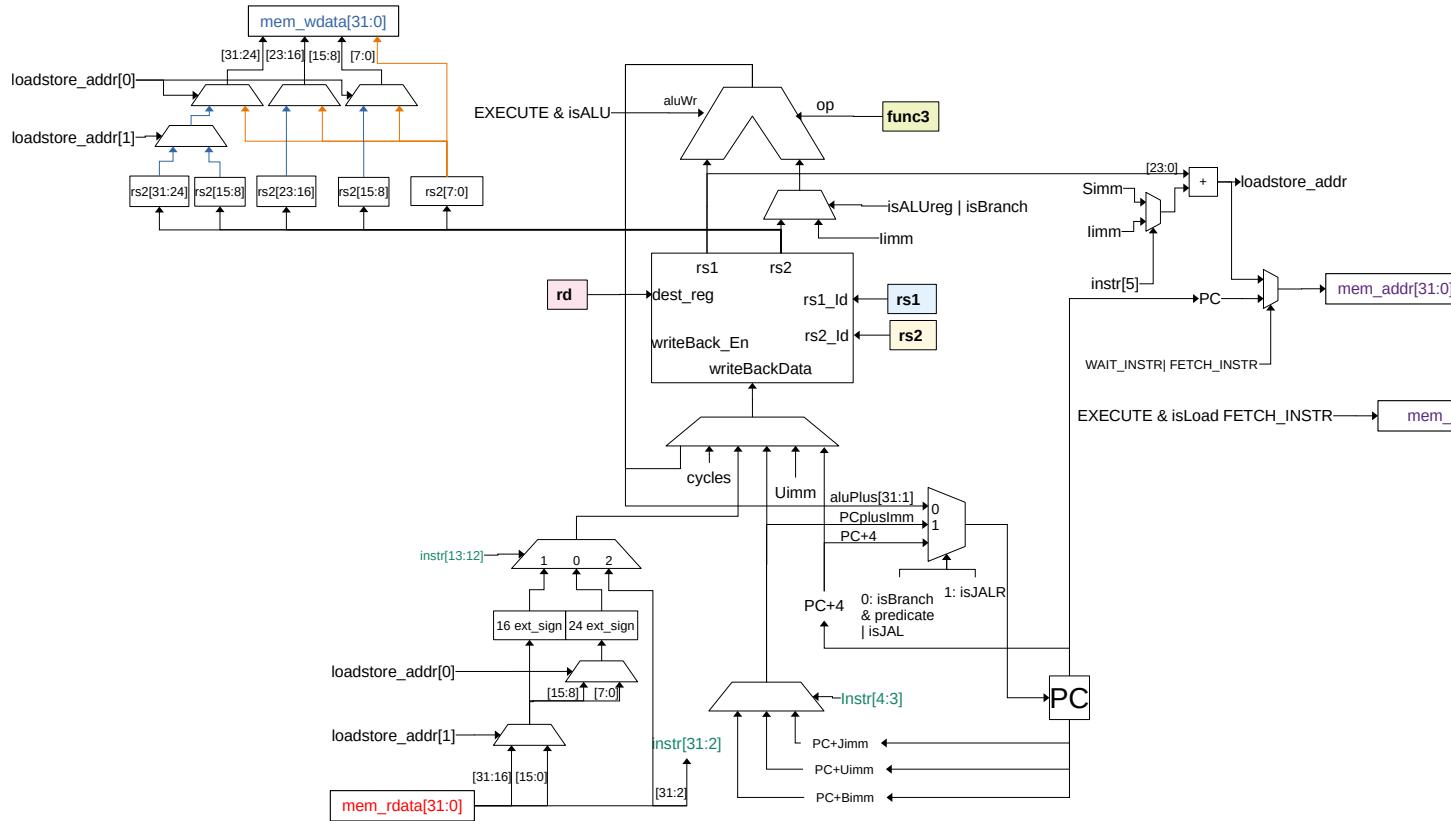


Figura 3.10 Lectura desde memoria externa en el RV32I

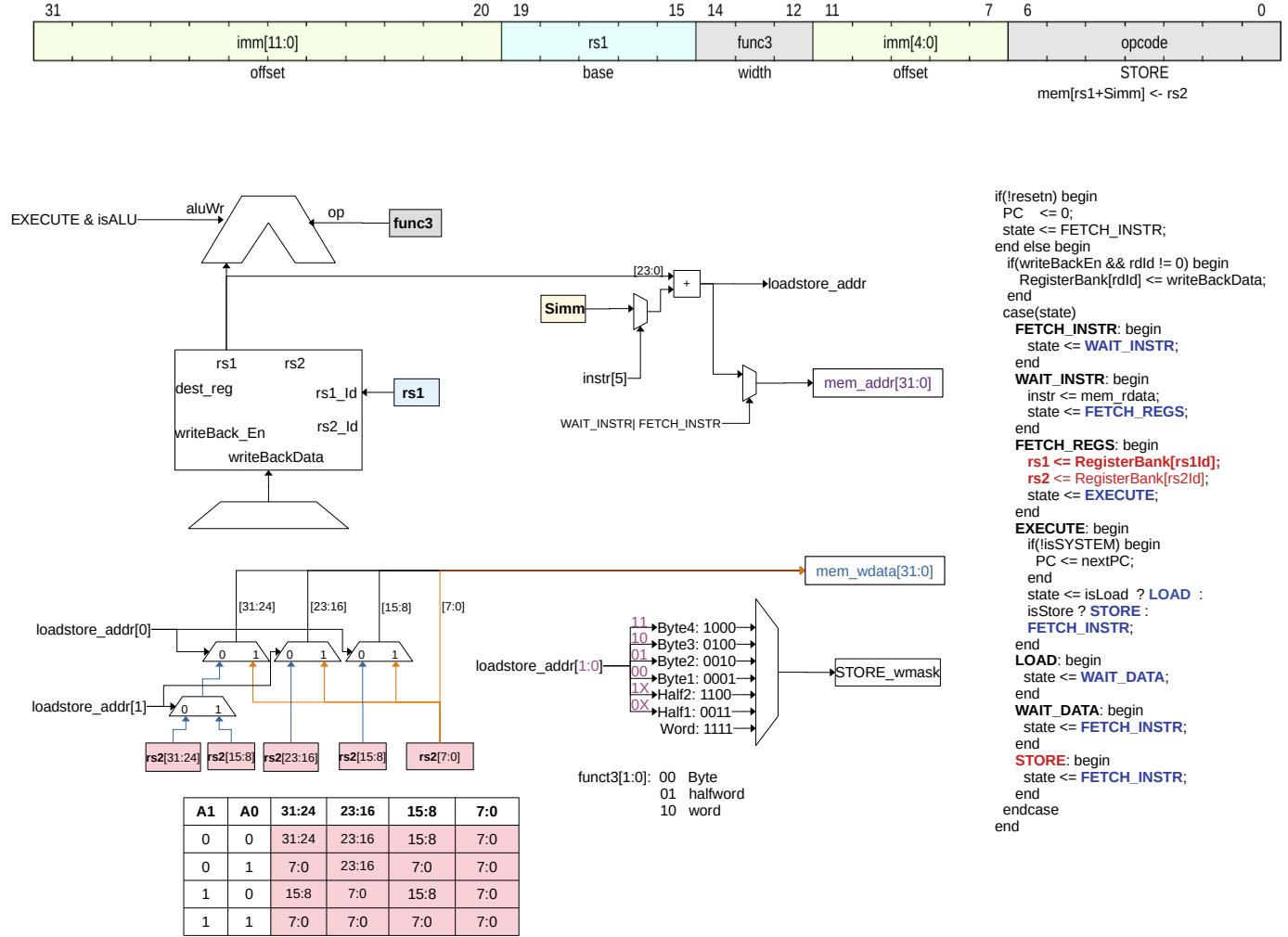


Figura 3.11 Escritura a memoria externa en el RV32I

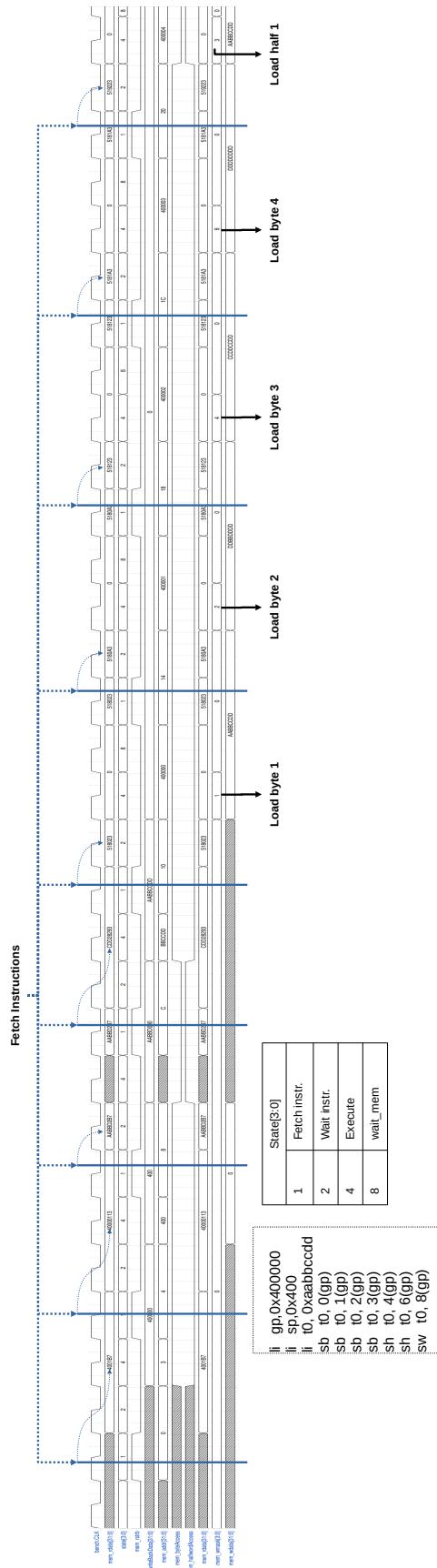


Figura 3.12 Simulación de una escritura a memoria externa en el RV32I

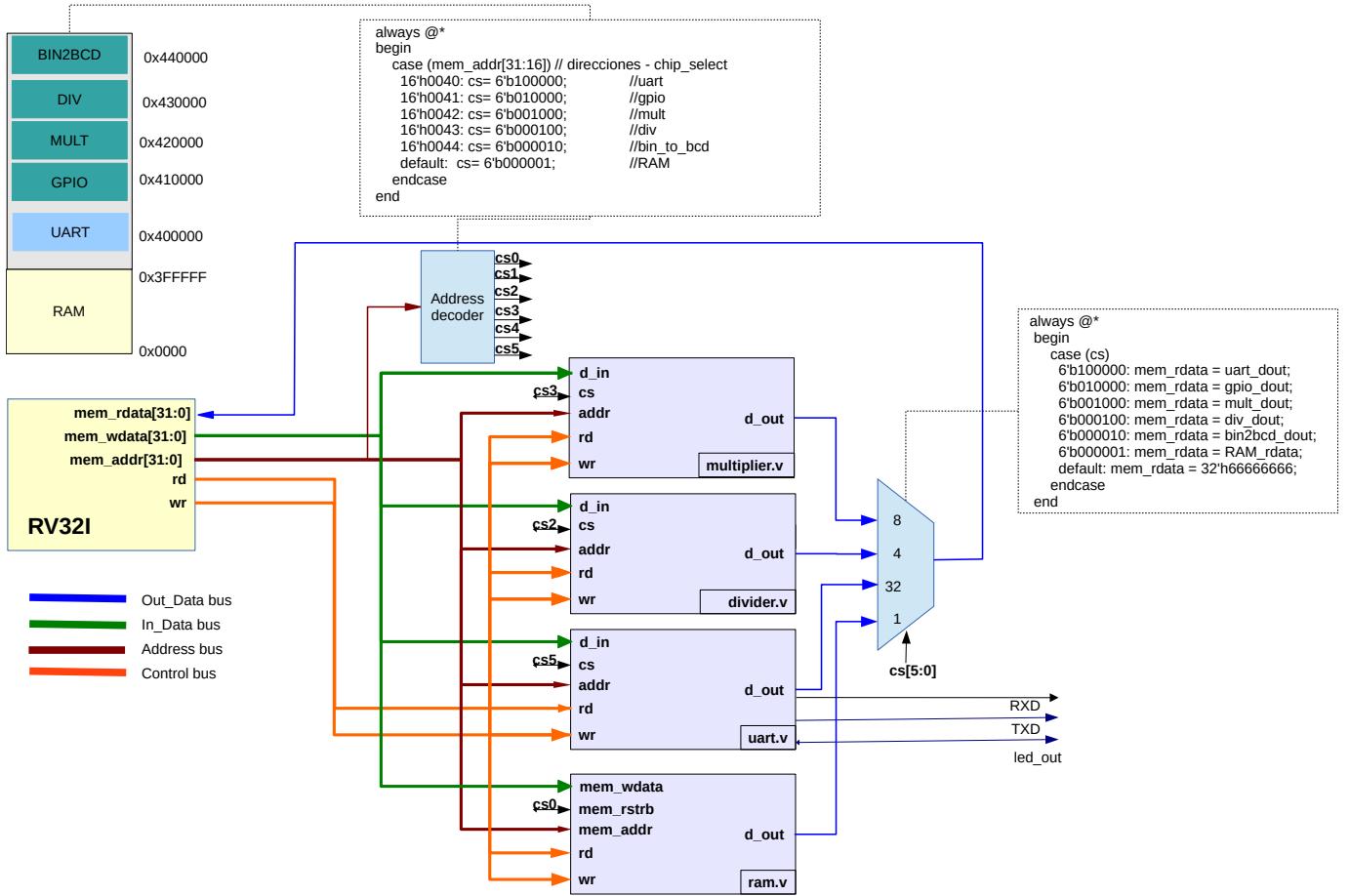


Figura 3.13 Diagrama de bloques del SoC basado en el RV32I

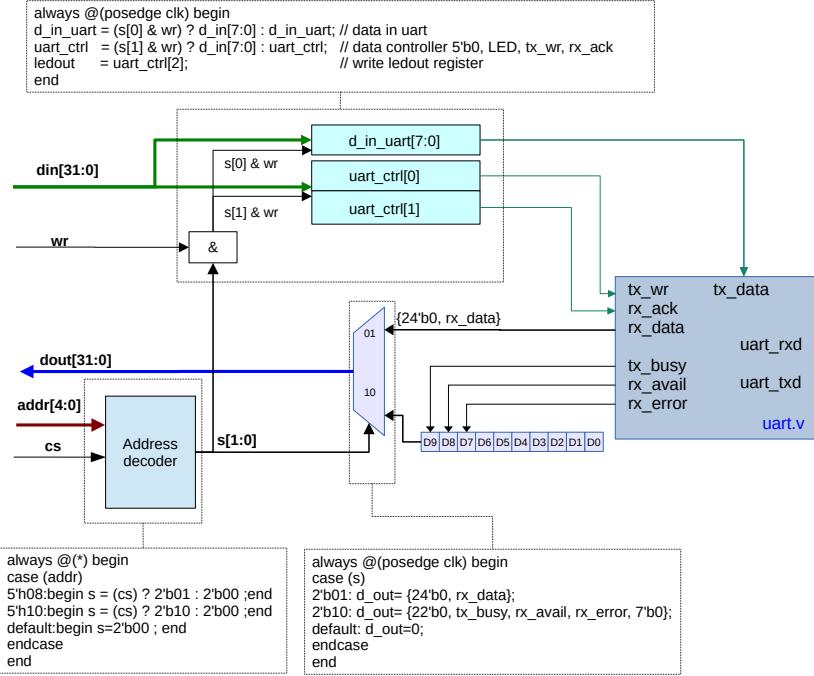


Figura 3.14 Diagrama de bloques del periférico uart

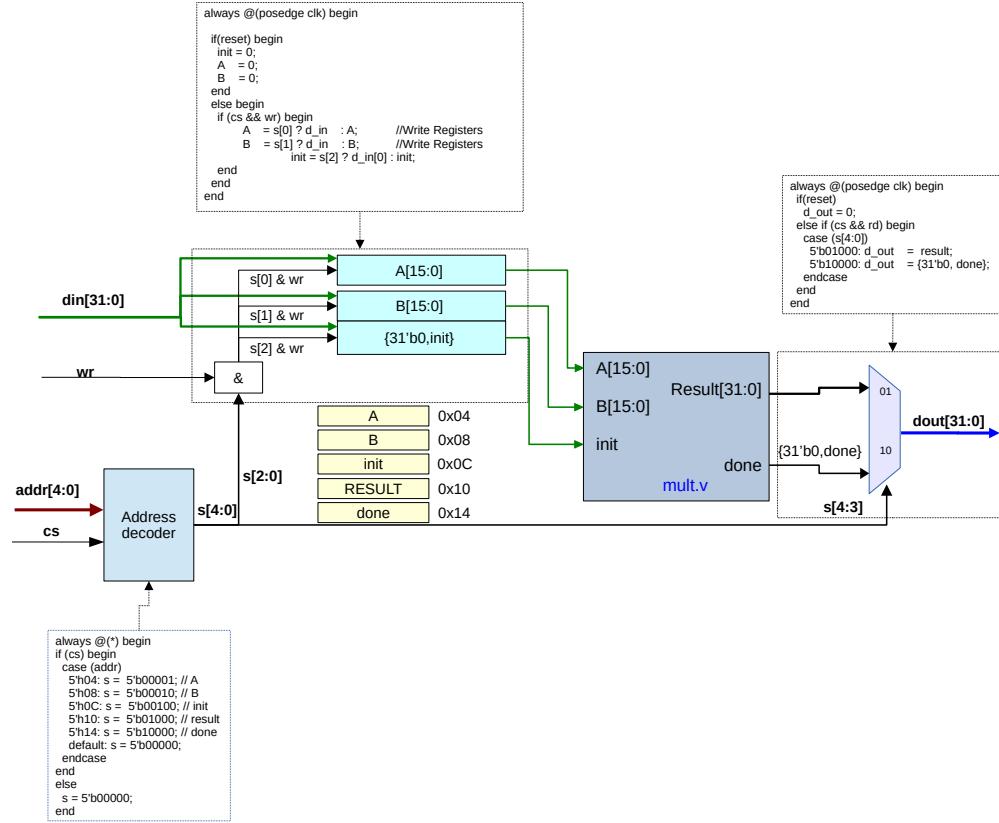


Figura 3.15 Diagrama de bloques del periférico mult

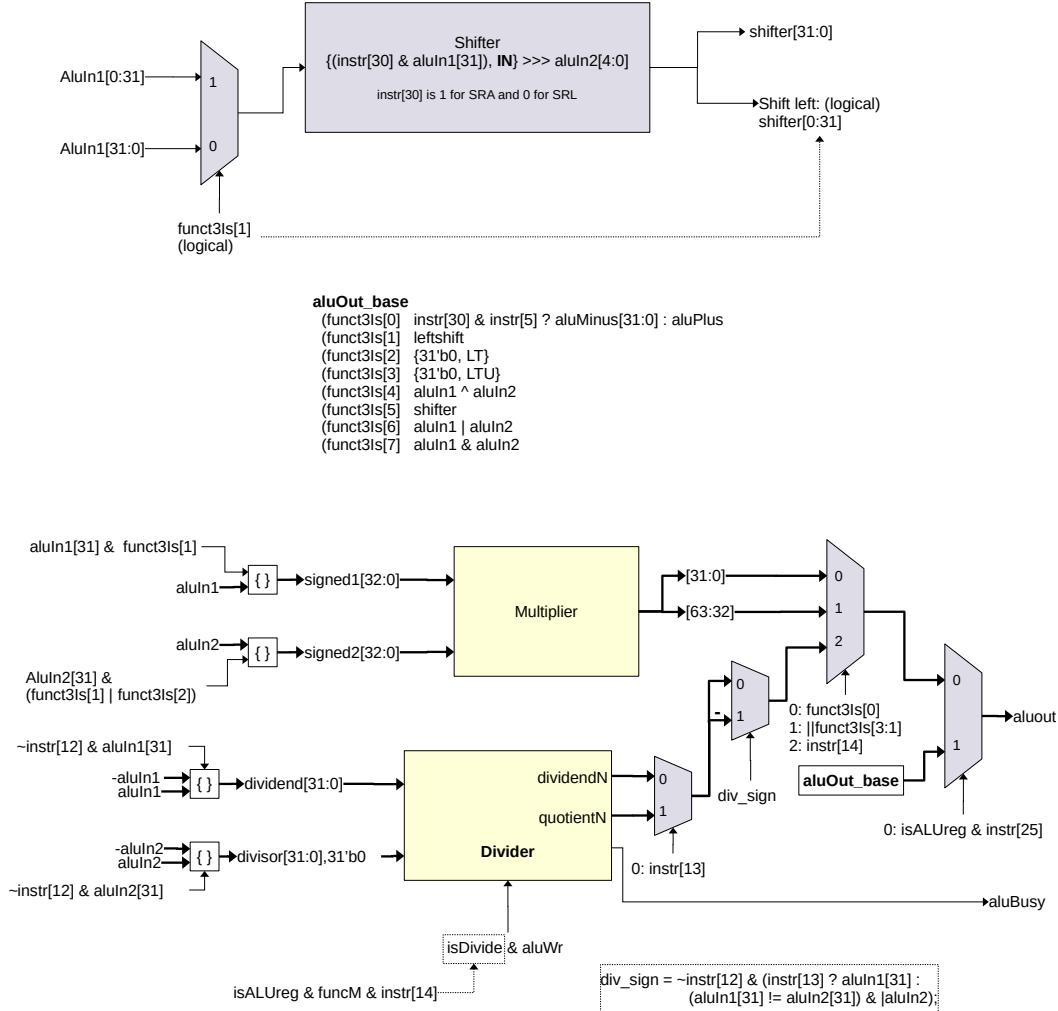


Figura 3.16 Diagrama de bloques del módulo aritmético del RV32M

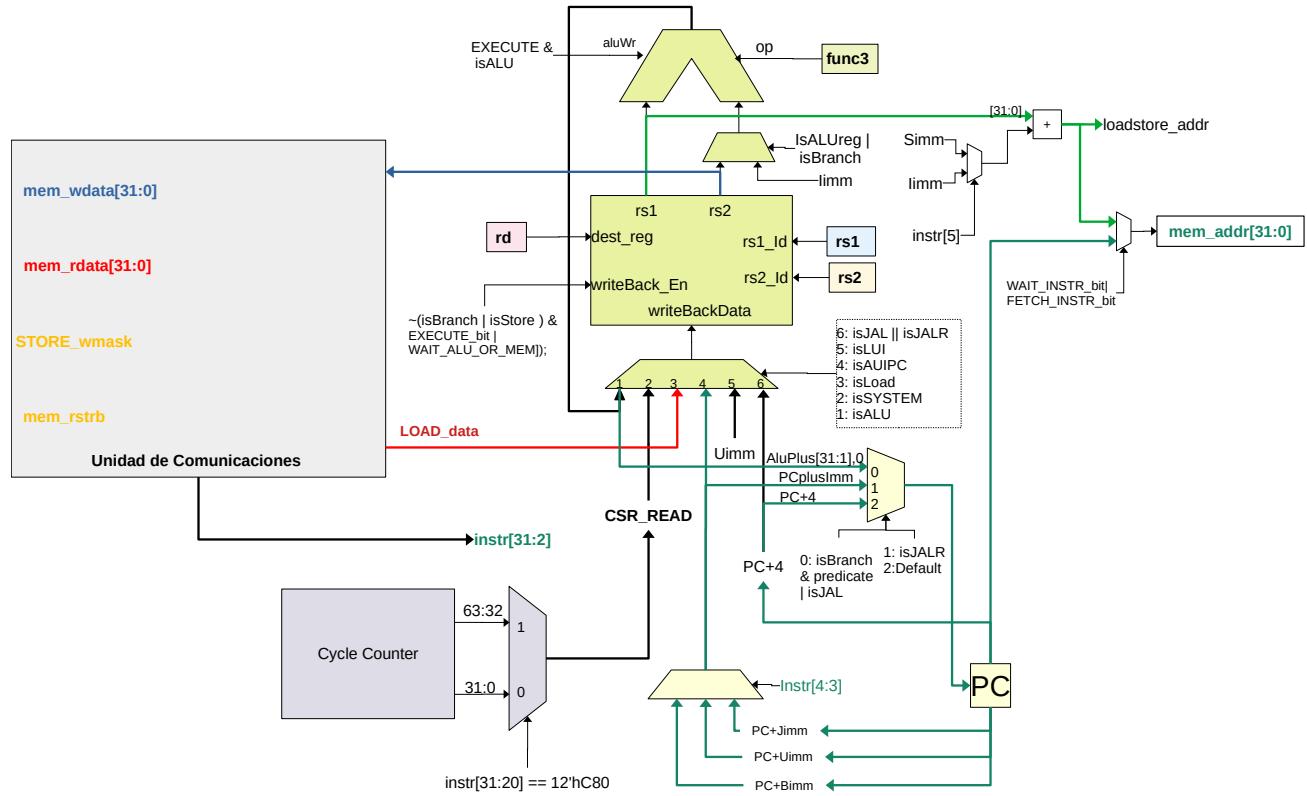


Figura 3.17 Diagrama de bloques del RV32M

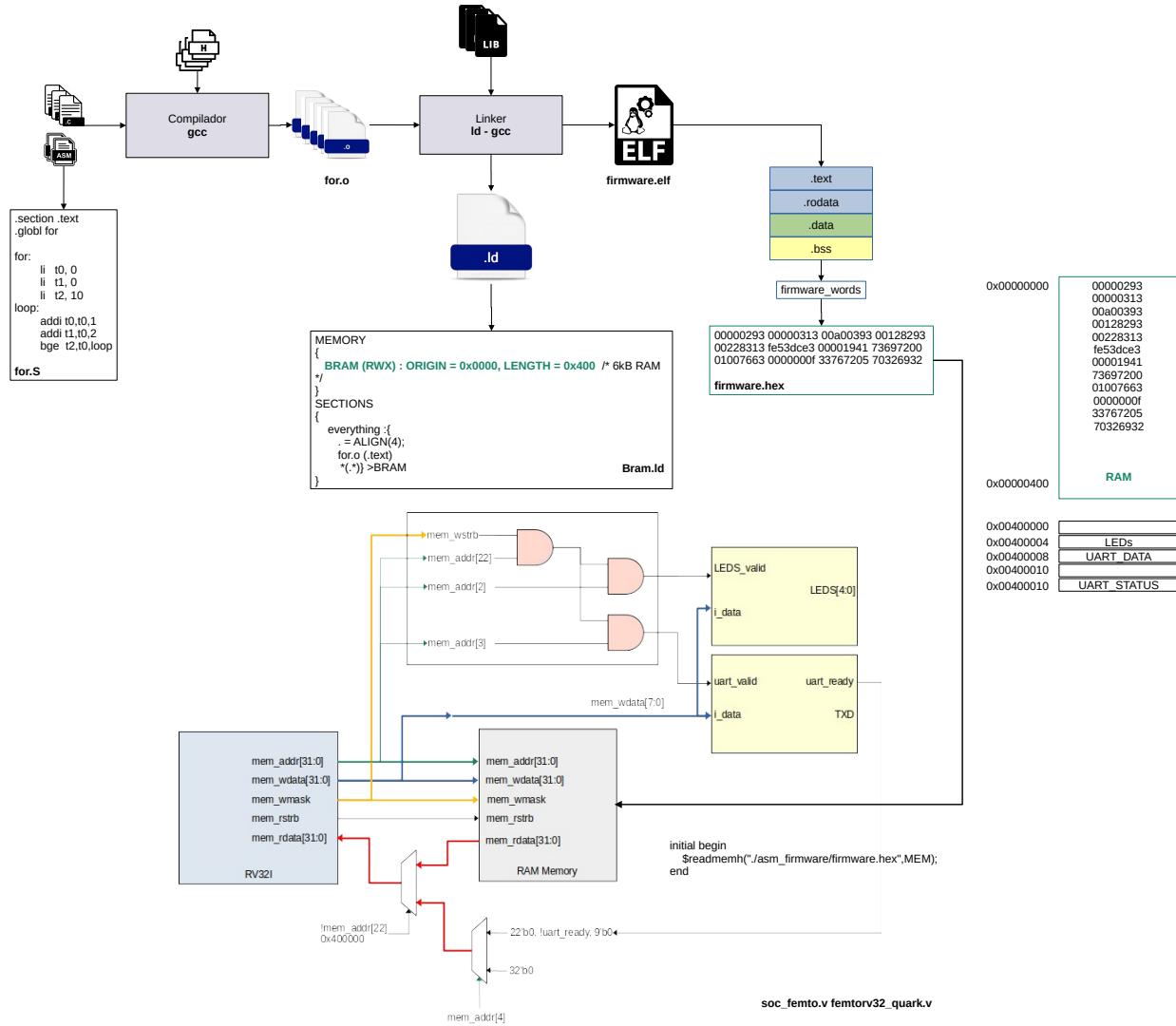


Figura 3.18 Flujo software (assembler) para el procesador femtoRV

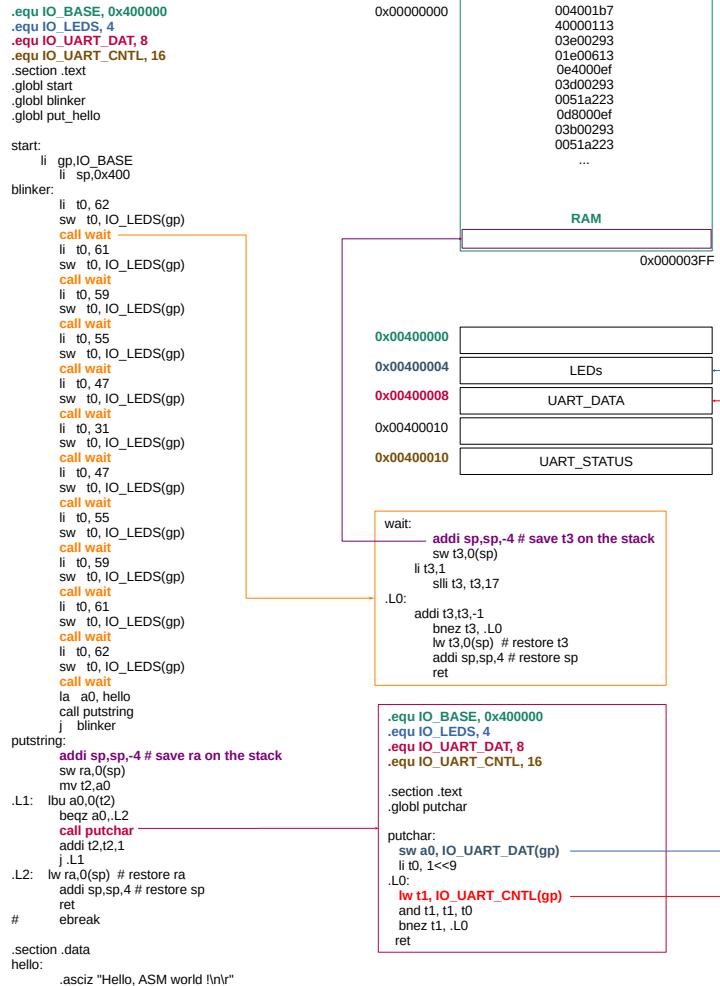
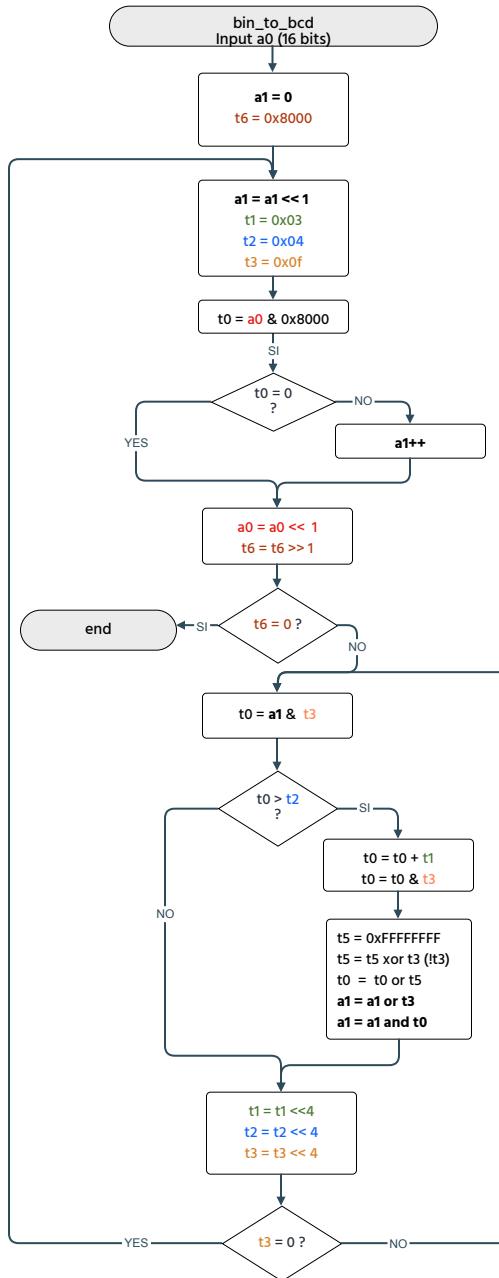


Figura 3.19 Manejo de periféricos LEDs y UART en el RV32I

**Figura 3.20** Diagrama del algoritmo double dabble

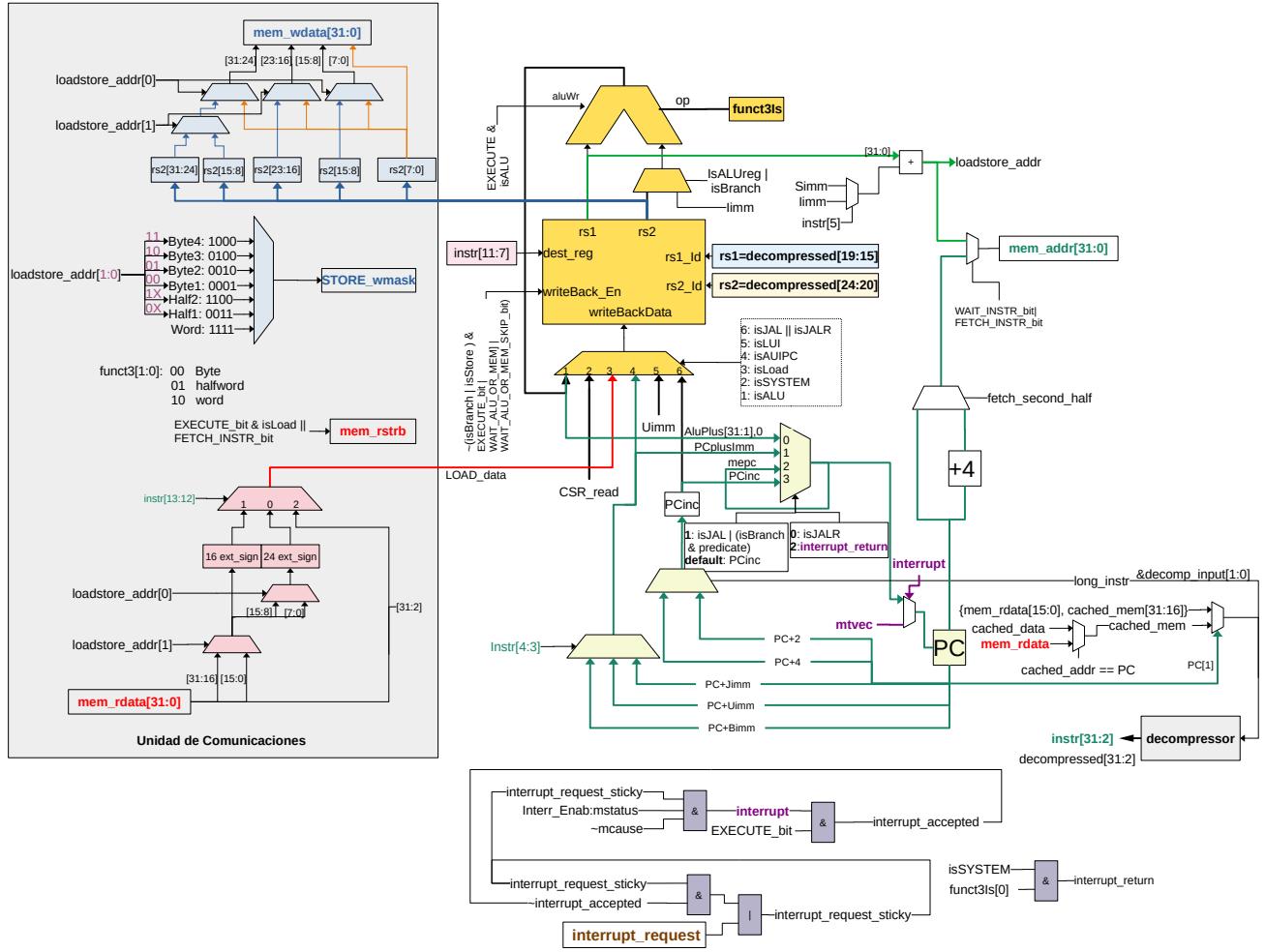


Figura 3.21 Diagrama de Bloques del procesador RV32IMC

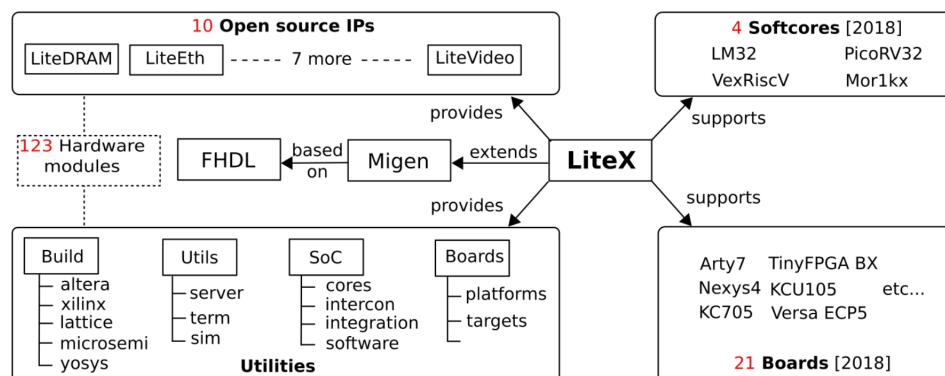
# Capítulo 4

## Implementación de SoC usando Herramientas de Automatización

### 4.1. Herramienta de automatización litex

En las secciones anteriores se presentó el flujo de diseño *hardware/software* para generar un Sistema Sobre Silicio (*SoC*) utilizando los procesadores LM32 de Lattice y femtoRV - RISCV. La implementación de periféricos dedicados puede resultar tediosa ya que se trabaja con archivos de cientos de líneas, lo que puede llevar a errores humanos que impactarían los tiempos de desarrollo; adicionalmente, agregar, modificar o cambiar periféricos implica la edición de varios archivos que deben estar articulados para que el sistema funcione, estas tareas pueden realizarse de forma automatizada eliminando errores y aumentando la flexibilidad.

Una de las herramientas con mayor uso y soporte es *litex*: un constructor de SoCs y librería de componentes IP escritos a nivel de Lógica de Transferencia de Registros (RTL), unido a unas utilidades que facilitan el diseño de SoCs, *litex* se basa en *migen* una caja de herramientas de código abierto basado en python que a su vez está formado por un lenguaje de descripción de hardware (HDL), una librería de IPs, un simulador y un sistema de generación, síntesis y compilación.



**Figura 4.1** Diagrama de bloques del LM32 Tomado de: LiteX: an open-source SoC builder and library based on Migen Python DSL Florent Kermarec, Sébastien Bourdeauducq, Jean-Christophe Le Lann and Hannah Badier e first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy

La figura 4.1 muestra la arquitectura de *litex*, decenas de componentes hacen parte de sus recursos, dentro de los cuales se encuentran:

- Softcores<sup>1</sup>: LM32, Mor1kx, PicoRV3, VexRiscv, femtoRV, LM32.
- IPs Open Source: litedram liteeth litesata litesdcards litepcie litejesd204b, spi, uart, timer, video, xadc, usb, pwm, gpio.
- Soporte a FPGAs: Xilinx(AMD), Altera(Intel), Lattice, Efinix, Gowin.

<sup>1</sup> Procesadores descritos en HDL

- Lenguaje de descripción de hardware en python - migen.
- Banco de herramientas y archivos para soportar una gran variedad de plataformas existentes de las FPGAs soportadas.

Con estos recursos Litex permite el diseño e implementación de sistemas digitales de forma rápida comparada con la forma tradicional en la que se debe especificar todos los componentes.

### **Instalación del entorno Litex**

Conda permite administrar las herramientas de desarrollo de manera sencilla ya que en éste caso nos ofrece los pre-compilados de Yosys, nextpnr, compiladores y otras herramientas útiles, que en otros casos deberían ser compiladas desde las fuentes, lo que requiere más recursos en tiempo y capacidad de máquina.

Los comandos a usar son sencillos tanto para instalar, actualizar y ejecutar entornos. A continuación se explica los pasos de instalación de las herramientas.

Herramientas a instalar Yosis Nextpnr gcc riscv32 conda install antmicro::gcc-riscv64-elf-newlib

```
mkdir -p $HOME/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O \
$HOME/miniconda3/miniconda.sh
bash $HOME/miniconda3/miniconda.sh -b -u -p $HOME/miniconda3
rm -rf $HOME/miniconda3/miniconda.sh
$HOME/miniconda3/bin/conda init bash
$HOME/miniconda3/bin/conda init zsh
conda config --set auto_activate_base false
conda update -n base -c defaults conda
conda create --name fpga
conda activate fpga
conda install -c "litex-hub" nextpnr-ecp5
conda install -c "litex-hub/label/ci-master-1188059080" yosys
conda install -c litex-hub gcc-riscv32-elf-newlib
conda install -c litex-hub iceprog
```

### **Instalación de la herramienta de configuración openFPGALoader**

```
conda activate fpga
wget https://github.com/trabucayre/openFPGALoader/releases/download/v0.10.0/\
ubtuntu18.04-openFPGALoader.tgz
cd /
sudo tar zxvf $HOME/Downloads/ubtuntu18.04-openFPGALoader.tgz
```

### **Instalación del entorno Litex**

```
mkdir -p $HOME/litex
cd $HOME/litex
wget https://raw.githubusercontent.com/enjoy-digital/litex/master/\
litex_setup.py
chmod +x litex_setup.py
./litex_setup.py --init --install --user --config=standard
./litex_setup.py --update
pip3 install meson ninja
./litex_setup.py --gcc=riscv
sudo apt install libevent-dev libjson-c-dev verilator
```

```
litex_sim --cpu-type=vexriscv
```

Se desplegará el mensaje de inicio de la BIOS:

```
/ / ( ) /_____| |/_/
/ /__| / __/ -_)> <
/____/_/\_\_\_/_/|_|
Build your hardware, easily!
```

```
(c) Copyright 2012-2024 Enjoy-Digital
(c) Copyright 2007-2015 M-Labs
```

```
BIOS built on Aug 9 2024 15:12:07
BIOS CRC passed (0cef488c)
```

```
LiteX git sha1: c0517cd1
```

```
===== SoC =====
CPU: VexRiscv @ 1MHz
BUS: wishbone 32-bit @ 4GiB
CSR: 32-bit data
ROM: 128.0KiB
SRAM: 8.0KiB
```

```
===== Boot =====
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
Timeout
No boot medium found
```

```
===== Console =====
litex>
```

## Instalación de las herramientas de síntesis de Efinix

Registrarse y descargar las herramientas de Efinix

```
https://www.efinixinc.com/support/efinity.php
```

crear la variable LITEX\_ENV\_EFINITY en el archivo /.bashrc

```
export LITEX_ENV_EFINITY=/home/carlos/Embedded/efinity/2023.2/
```

### 4.1.1. Plataformas en litex

La plataforma para Litex es el dispositivo donde se realizará la implementación de la funcionalidad requerida, esto es, una placa de circuito impreso en la que una FPGA se conecta a una serie de recursos (dispositivos I2C, SPI, Interfaces de RED, dispositivos de audio o video, etc).

Por lo tanto, al describir una plataforma se debe indicar los pines conectados a los diferentes recursos, los cuales se deben agrupar según su funcionalidad, y se debe indicar la FPGA utilizada especificando la familia y el encapsulado. En el listado 4.1 se muestra un ejemplo de la definición de una placa (*board*), aunque *litex* tiene un banco de estas definiciones en la carpeta *litex-boards/litex.boards/platforms*, es posible crear una nueva y colocarla en cualquier lugar. Una declaración de placa consiste en la definición de la clase de Python *Platform*. En este caso se describe la placa de desarrollo openhardware *ecb\_t8\_t113*, en las líneas 1 - 3, se importan las funciones relacionadas con la declaración de plataformas, las cuales permiten utilizar objetos tales como Subsignal, Pins, y IOStandard; a continuación (líneas 6 - 17) se nombran las señales de entrada/salida que serán utilizadas, en este caso la señal de reloj *clk33*, un led *user\_led*, un pulsador *user\_btn\_n* y las señales de transmisión y recepción del puerto serial *tx* y *rx*.

```

48 from litex.build.generic_platform import *
49 from litex.build.efinix.platform import EfinixPlatform
50 from litex.build.efinix import EfinixProgrammer
51 # IOs
52
53 _io = [
54     # Clk
55     ("clk33", 0, Pins("75"), IOStandard("3.3_V_LVTTL/_LVC MOS")),
56     # Leds
57     ("user_led", 0, Pins("20"), IOStandard("3.3_V_LVTTL/_LVC MOS"), Misc("DRIVE_STRENGTH=3")),
58     # Buttons
59     ("user_btn_n", 0, Pins("74"), IOStandard("3.3_V_LVTTL/_LVC MOS"), Misc("WEAK_PULLUP")),
60     # Serial
61     ("serial", 0,
62         Subsignal("tx", Pins("142")), # 27 on H4 Must be changed in next HW version to 144
63         Subsignal("rx", Pins("144")), # 28 on H4 Must be changed in next HW version to 142
64         IOStandard("3.3_V_LVTTL/_LVC MOS"), Misc("WEAK_PULLUP"))
65     ),
66 ]
67 class Platform(EfinixPlatform):
68     default_clk_name = "clk33"
69     default_clk_period = 1e9/33.333e6
70     def __init__(self, toolchain="efinity"):
71         EfinixPlatform.__init__(self, "T20Q144C3", _io, _connectors, toolchain=toolchain)
72     def create_programmer(self):
73         return EfinixProgrammer()
74     def do_finalize(self, fragment):
75         EfinixPlatform.do_finalize(self, fragment)
76         self.add_period_constraint(self.lookup_request("clk33", loose=True), 1e9/33.333e6)

```

**Listing 4.1** Ejemplo de plataforma en *litex ecb\_t8\_t113*

Las señales en *litex* se declaran utilizando listas de estructuras de datos con el siguiente formato:

```
io_name, id, Pins("pin_name", "pin_name") or Subsignal(...), IOStandard("std_name"), Misc("misc"))
```

**Listing 4.2** Estructura de datos para declarar conectores en *litex*

Donde *io\_name* es el nombre de la señal, *id* es un identificador que distingue múltiples copias de periféricos idénticos, como por ejemplo LEDs. *Pins* es una clase que puede contener una cadena de caracteres correspondiente al pin de la FPGA con el que se conecta. Estas estructuras de datos son utilizadas para crear los nombres de las entradas y salidas en los archivos verilog generados y proporciona el nombre de la variable para asignación de pines en los archivos de restricciones.

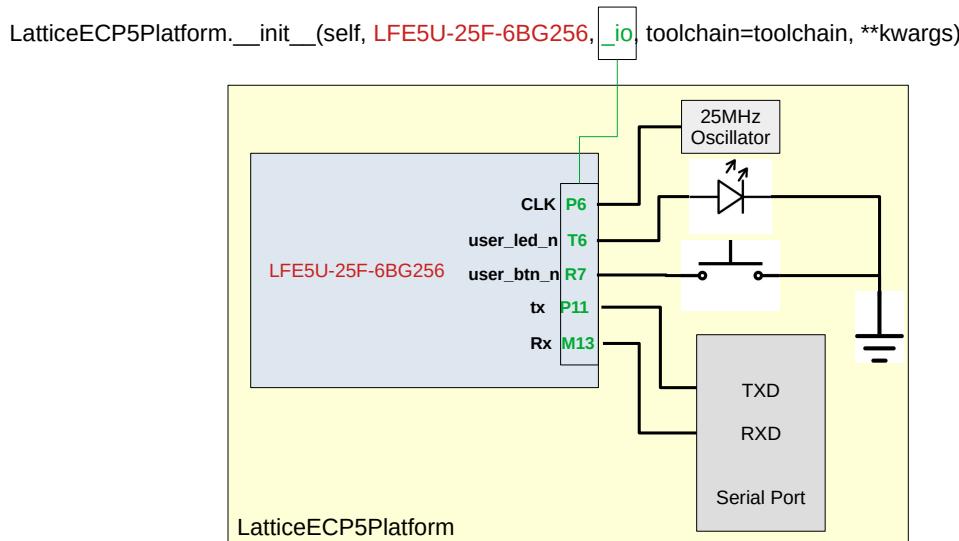
*Subsignals* es una clase que utiliza pines de la FPGA que pueden ser separados para un propósito determinado. Las entradas al constructor *Subsignal* son idénticos a los de las entradas I/O, excepto en que se omite el *id*. Desde el punto de vista del usuario, un recurso es encapsulado en una clase cuyas *Signals* se acceden a través de los miembros de la clase, por ejemplo: *comb += [resource1.sig1.eq(12)]*. Esto recibe el nombre de *record* en *Migen*. Los *Records* poseen una gran variedad de métodos útiles para construir sentencias *Migen* de forma rápida, de forma similar a las estructuras de C.

Como puede verse los demás miembros del arreglo 4.2 pueden omitirse. IOStandard es otra clase que contiene una cadena de caracteres específica de la cadena de herramientas y fija los niveles de voltaje de la lógica utilizada. Finalmente, la clase *Misc* contiene cadenas de caracteres separadas por espacios con información que puede colocarse en los archivos de restricciones junto con IOStandard. Información como: slew rate, pullups, etc. Es posible definir los conectores que los fabricantes de placas de desarrollo utilizan para permitir el acceso a los pines de la FPGA, el listado 4.3 muestra un ejemplo de esta declaración.

```
_connectors = [
    ("j1", "C4 D4 E4 - D3 E3 F4 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j2", "F3 F5 G3 - G4 H3 H4 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j3", "G5 H5 J5 - J4 B1 C2 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j4", "C1 D1 E2 - E1 F2 F1 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j5", "G2 G1 H2 - K5 K4 L3 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j6", "L4 L5 P2 - R2 T2 R3 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j7", "T3 R4 M5 - P5 N6 N7 N4 N5 N3 P3 P4 M3 N1 M4 -"),
    ("j8", "P7 M7 P8 - R8 M8 M9 N4 N5 N3 P3 P4 M3 N1 M4 -"),
]
```

**Listing 4.3** Declaración de conectores en *litex*

En la línea 67 se hace la declaración de la clase *Platform*, que en este caso hereda las propiedades de la clase *EfinixPlatform* (que a su vez hereda de *GenericPlatform*), la cual, contiene la lógica para construir nuestra plataforma; las siguientes líneas declaran las propiedades de la señal de reloj, se define la cadena de herramientas 70 en este caso *efinity* y se declara la referencia de la FPGA (T20Q144C3), las señales de entrada/salida definidos previamente y la cadena de herramientas para definir la nueva plataforma (71). La figura 4.2 muestra el diagrama de bloques de la plataforma definida.



**Figura 4.2** Plataforma generada a partir de la lista 4.1

La función *create\_programmer* (línea 72) retorna un programador específico de la FPGA. Esta función puede omitirse si no puede utilizarse uno que pueda ser invocado de esta forma. En este caso se utilizará la aplicación *openFPGALoader* para configuración.

## 4.2. *migen* como herramienta de diseño

En las siguientes subsecciones se mostraran ejemplos del uso de *migen* y se utilizará una plataforma que usa una FPGA de Efinix. Se implementará inicialmente el "Hello World" del hardware un blink, después dos circuitos más

complejos, un PWM y una UART, con esto se ilustrará la forma de implementar bloques secuenciales, combinatorios y máquinas de estado, después se mostrará como acceder a los registros internos de los módulos para verificar su correcto funcionamiento, para finalmente crear un SoC con uno de estos periféricos y crear una aplicación ejecutándose en un procesador que lo controle. En estos ejemplos se utilizará la plataforma ecb\_t8\_t113 con una FPGA T20 de efinix.

#### 4.2.1. Blink en migen

El equivalente en hardware al programa básico "Hola Mundo" es un Led que se enciende y se apaga de forma permanente, es decir un blink. El blink es básicamente la salida de un divisor de frecuencia asignada a un LED. Este divisor de frecuencia en últimas es el bit más significativo de contador binario.

En la figura 4.3 se muestra la descripción del módulo *Blink*, indicando sus componentes. Para este módulo solo es necesario declarar un componente secuencial (**sync**).

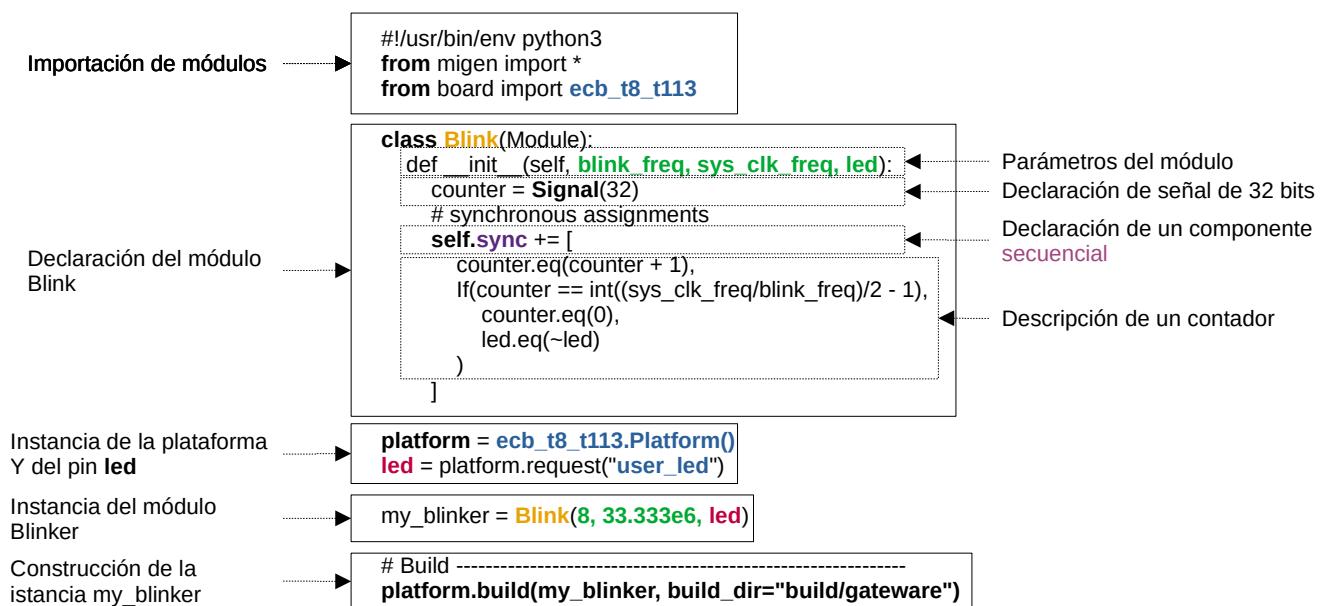


Figura 4.3 Descripción en migen del módulo Blink

#### 4.2.2. Fading LED en migen

En esta ocasión implementaremos un circuito que varíe la intensidad de un LED de 0 al máximo e inmediatamente después la disminuya hasta ser cero. Para esto, es necesario implementar un PWM para controlar el LED.

En la figura 4.4 se muestra el diagrama de bloques. Un divisor de frecuencia (ClockDiv) divide en potencias de 2 la frecuencia del reloj de la FPGA, proporcionando una salida de duración de un ciclo de reloj (*tick*). El módulo *TickUpdownCounter*, proporciona el conteo ascendente y descendente que fija el ciclo útil del PWM.

En este ejemplo podemos observar que se utilizan componentes secuenciales (**sync**) y combinatorios (**comb**).

```

#!/usr/bin/env python3
from migen import *
from board import ecb_t8_t113

class ClockDiv(Module):
    def __init__(self, divbitwidth, divout, divtick):
        divcounter = Signal(divbitwidth+1)
        self.sync += divcounter.eq(divcounter + 1)
        self.comb += divout.eq(divcounter[divbitwidth])
        divcounter_inv = Signal(divbitwidth)
        self.comb += divcounter_inv.eq(~divcounter[0:divbitwidth])
        self.comb += divtick.eq(divcounter_inv == 0)

class PWM(Module):
    def __init__(self, pwm, bitwidth, value):
        pwm_counter = Signal(bitwidth)
        self.comb += pwm.eq(pwm_counter < value)
        self.sync += pwm_counter.eq(pwm_counter + 1)

class TickUpdownCounter(Module):
    def __init__(self, counter, tick, bitwidth):
        icrounter = Signal(bitwidth+1)
        direction = Signal()
        self.comb += direction.eq(icrounter[bitwidth])
        self.comb += If(direction,
                        counter.eq(~icrounter[0:bitwidth]),
                        ).Else(
                            counter.eq( icrounter[0:bitwidth]))
        icrounter_inv = Signal(bitwidth)
        self.comb += icrounter_inv.eq(~icrounter[0:bitwidth])
        self.sync += icrounter_inv.eq(0,
                                    icrounter.eq(icrounter + 2),
                                    ).Else(
                                        icrounter.eq(icrounter + 1))

class PWMFade(Module):
    def __init__(self, pwm_signal, dbg, width, div):
        pwm_value = Signal(width)
        self.submodules.pwm = PWM(pwm_signal, width, pwm_value)

        updown_clock = Signal()
        updown_clock_strobe = Signal()
        self.submodules.updown_clk_div = \
ClockDiv(div, updown_clock, updown_clock_strobe)

        self.submodules.updown = \
TickUpdownCounter(pwm_value, updown_clock_strobe, width)
        self.comb += dbg.eq(updown_clock)

platform = ecb_t8_t113.Platform()
led = platform.request("user_led")

pwm_fade = PWMFade(led, 16, 9)

# Build -----
platform.build(pwm_fade, build_dir="build/gateware")

```

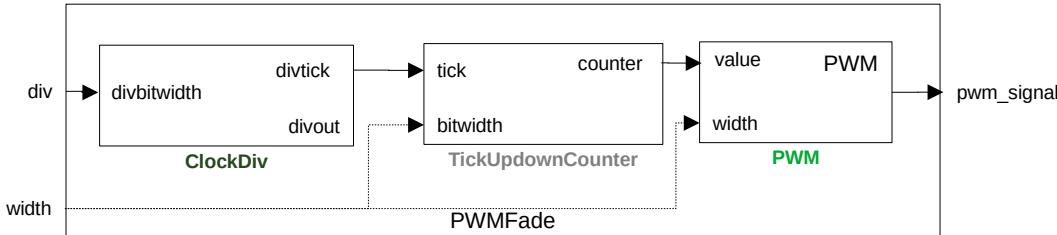


Figura 4.4 Descripción en migen del módulo Fade

#### 4.2.3. UART en migen

En la figura 4.5 se muestra la implementación en migen del receptor de una UART; toda la funcionalidad se puede representar mediante una máquina de estados algorítmicos cuyo diagrama de estados se muestra en la misma figura.

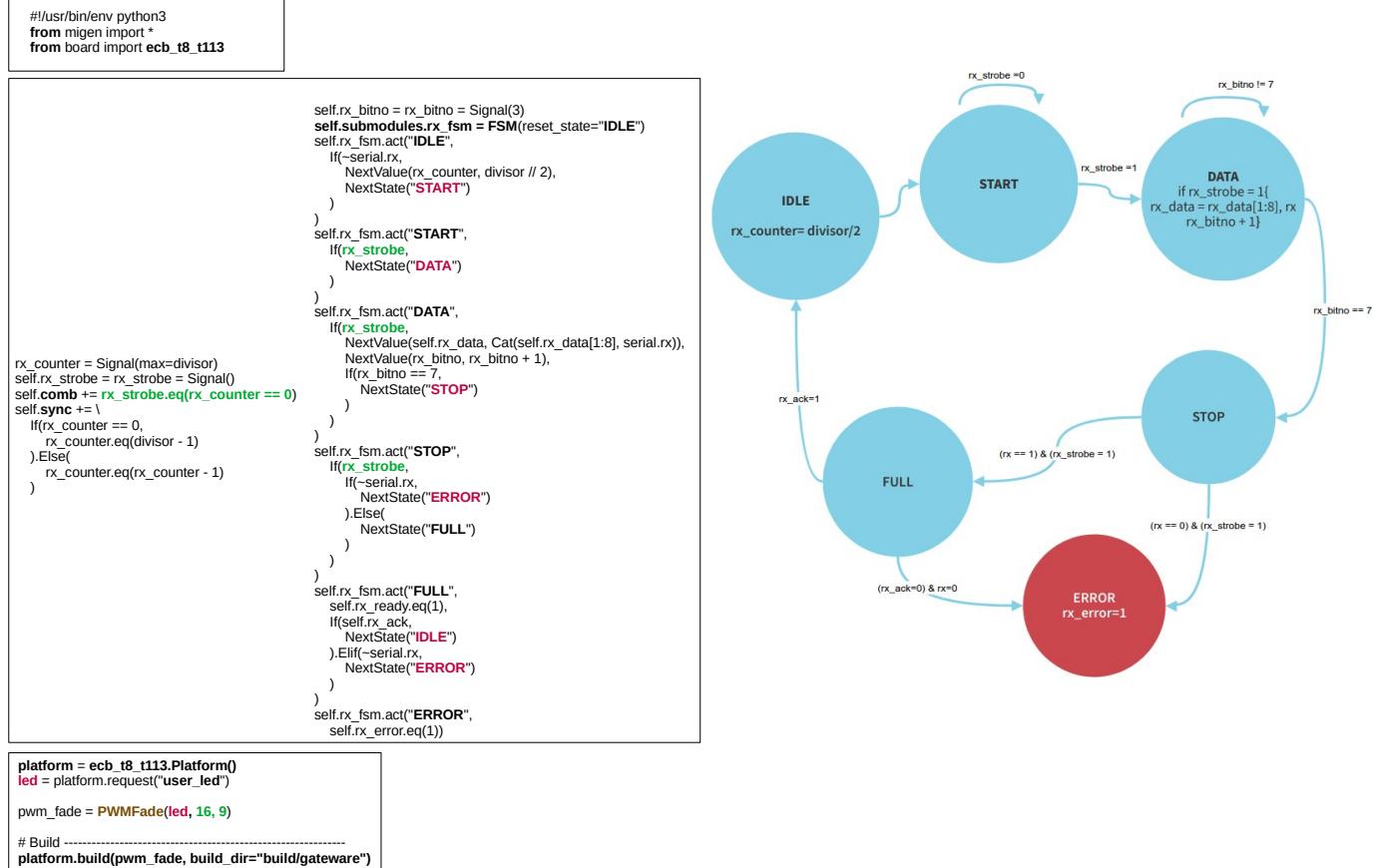


Figura 4.5 Descripción en migen del receptor de una UART

#### 4.2.4. Prueba de periféricos mediante Uart\_Bridge

Litex proporciona un camino para probar los periféricos sin la necesidad de instanciarlos en el SoC y sin tener que escribir el código para controlarlos. Para esto, permite la instancia de un puerto de comunicaciones maestro (que puede ser una UART o un puerto Ethernet) y comunicarse con la aplicación *litex\_server* la que a su vez utiliza un script en python para leer y escribir registros internos del periférico bajo prueba. En la figura 4.6 se muestra el diagrama conceptual de *Litex Server*.

En la figura 4.7 podemos observar los diferentes componentes de un ejemplo de uso de *litex\_server*. En el archivo *base.py* se instancia el módulo *UARTWishboneBridge* y se declara como maestro y el módulo *pwm.py* declarado como esclavo, el cual ha sido descrito para ser utilizado como un periférico, en migen solo basta con declarar las señales de entrada/salida y registros de lectura escritura del periférico, las señales de control las genera litex de forma automática. Por el lado del HOST, se describe el script en python que controlará los registros del periférico, para esto se crea una instancia de la clase *RemoteClient* esta instancia se comunica con *litex\_server*, para controlar los registros de la unidad bajoprueba usando las funciones *wb.regs.register.write* y *wb.regs.register.read*.

#### 4.2.5. Definición de SoC en litex

Una vez definida la plataforma, se procede a la creación del SoC, esto se hace, creando un archivo en python el cual puede tener cualquier nombre (en este ejemplo *base.py*) y se describe el SoC utilizando las librerías, métodos y submétodos de *litex*, indicando que procesador y periféricos vamos a utilizar. En la figura 4.4 se muestra un ejemplo de definición de *litex* basado en la plataforma del listado 4.1 a la que llamamos *ecb\_t8\_t113*.

## 4.2 migen como herramienta de diseño

107

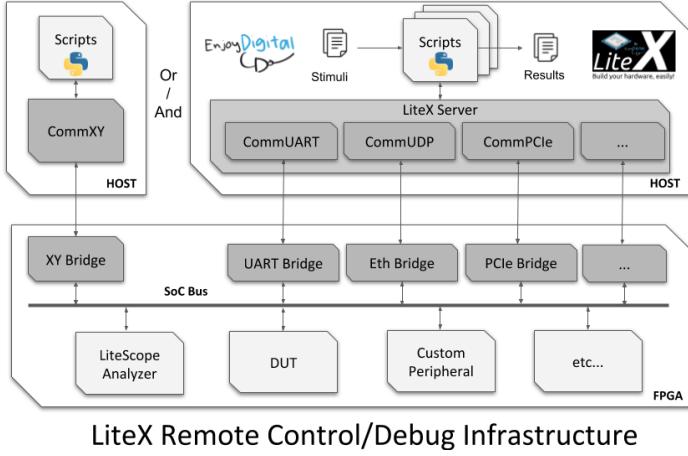
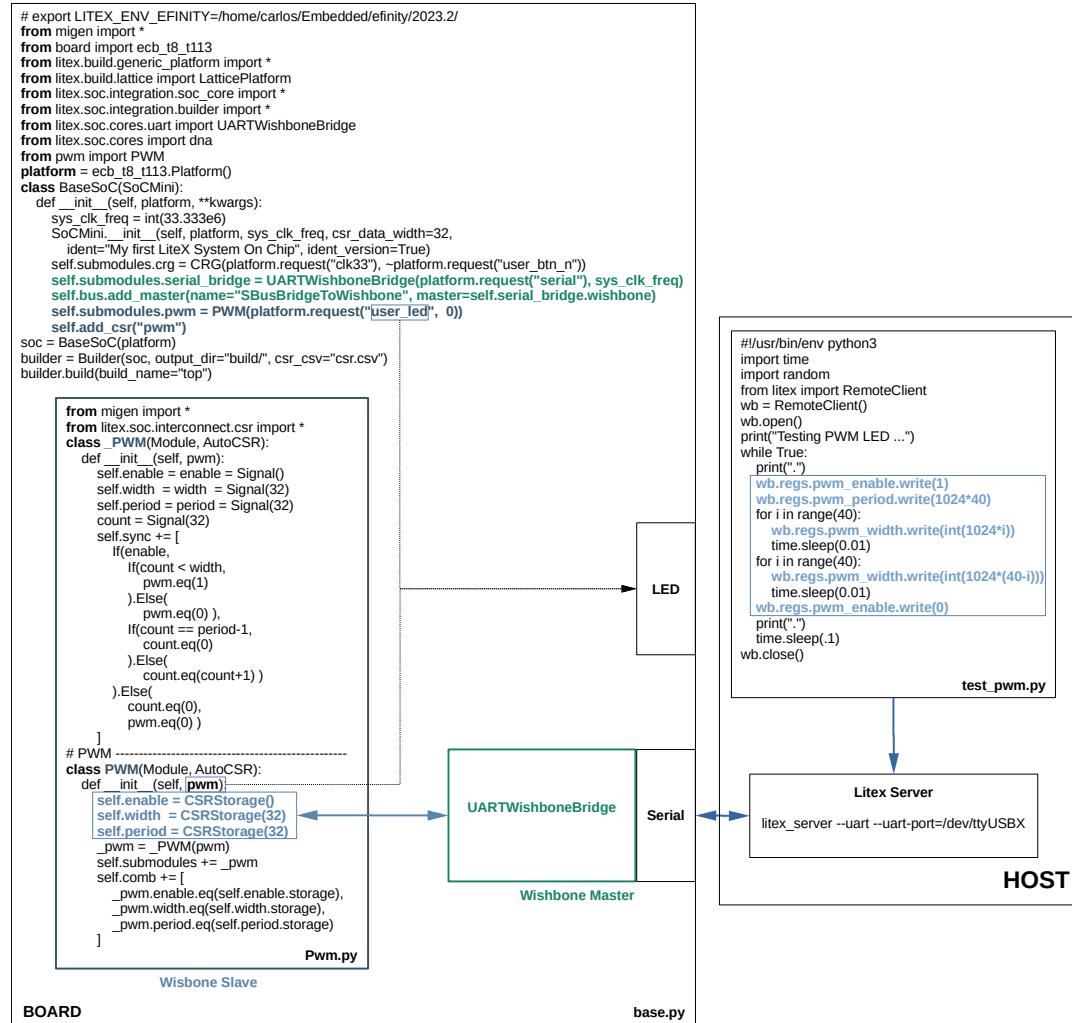
Figura 4.6 Debug de periféricos con litex\_server fuente:<https://github.com/enjoy-digital/litex/wiki/Use-GDB-with-VexRiscv-CPU>

Figura 4.7 Depuración de periféricos usando Litex Server y UartBridge

```

48 #!/usr/bin/env python3
49 from migen import *
50 from board import ecb_t8_t113
51 from litex.build.generic_platform import *
52 from litex.build.lattice import LatticePlatform
53 from litex.soc.integration.soc_core import *
54 from litex.soc.integration.builder import *
55 from litex.soc.cores import dna
56
57 platform = ecb_t8_t113.Platform()
58 # BaseSoC -----
59 class BaseSoC(SoCCore):
60     def __init__(self, platform, **kwargs):
61         sys_clk_freq = int(33.333e6)
62
63         # SoC with CPU
64         SoCCore.__init__(self, platform,
65                         cpu_type = "vexriscv",
66                         clk_freq = 33.333e6,
67                         ident = "LiteX CPU Test on ecb_t8_t113", ident_version=True,
68                         integrated_rom_size = 0x8000,
69                         integrated_main_ram_size = 0x4000)
70
71         # Clock Reset Generation
72         self.submodules.crg = CRG(platform.request("clk33"), ~platform.request("user_btn_n"))
73
74         # Led
75         user_leds = Cat(*[platform.request("user_led", i) for i in range(1)])
76         self.submodules.leds = Led(user_leds)
77         self.add_csr("leds")
78 # Build -----
79 soc = BaseSoC(platform)
80 builder = Builder(soc, output_dir="build", csr_csv="csr.csv")
80 builder.build(build_name="top")

```

**Listing 4.4** Declaración de SoC en *litex* (archivo *base.py*)

En las líneas iniciales (49 - 55), se hace la declaración de las librerías que se utilizarán en la declaración del SoC, litex posee los subpaquetes: litex.gen - proporciona módulos específicos para generar código en lenguaje de descripción de hardware que no esté integrado en migen, litex.build - Provee herramientas para construir los bitstreams de configuración de las FPGAs y para simular código HDL o SoCs, litex.soc - proporciona definiciones/módulos para construir cores y herramientas para construir SoCs a partir de estos cores. Se define la utilización de la plataforma *ecb\_t8\_t113* para implementar este SoC (línea 57), (declarado previamente 4.1).

El siguiente paso consiste en declarar las propiedades básicas del SoC (línea 64). Aquí se define la frecuencia del reloj, se declara el procesador, (en este caso el vexriscv línea 65), que funcionará a 33.33 MHz y tendrá una memoria ROM interna de 32kB 68 y una RAM interna de 16kB.

Se procede ahora a la instanciación del submódulo de generación de reloj y reset(CRG), para esto, se debe asignar estas señales a dos pines definidos en la plataforma utilizando el método *request*, con lo que se asigna *clk33* y *user\_btn\_n* (negada) al reloj y reset respectivamente. En la línea 74 se instancia el pin del Led declarado en la plataforma y en la línea 76 se le indica a la herramienta que instancie el periférico *leds* (el cual se encuentra en la librería de *cores* litex/soc/cores) y le asigne este pin.

Finalmente en la línea 78 se le indica al entorno que genere los archivos necesarios para construir el SoC y coloque los archivos temporales en la carpeta *build*, que el proyecto revibirá el nombre de **top**, así mismo, que cree un archivo llamado *csr.csv* donde aparecerá el mapa de memoria del SoC.

## Síntesis y compilación del SoC

*litex* genera los archivos necesarios para sintetizar el SoC en la FPGA y compilar el software necesario para facilitar el desarrollo de aplicaciones. Primero vamos a examinar el árbol de directorios que genera litex cuando se construye el

SoC. Para iniciar el proceso de construcción basta con ejecutar el programa descrito anteriormente *base.py*. Al finalizar este proceso, tendremos el árbol de directorios que se muestra en la figura 4.8 (la carpeta *firmware* no es generada por *litex*, contiene un ejemplo de cómo utilizar el SoC para desarrollar aplicaciones propias.

```

build/
└── software
    ├── bios
    └── include
        └── generated
            ├── csr.h
            ├── git.h
            ├── mem.h
            ├── output_format.ld
            ├── regions.ld
            ├── soc.h
            └── variables.mak
    ├── libbase
    ├── libc
    ├── libcompiler_rt
    ├── libfatfs
    ├── liblitedram
    ├── libliteeth
    ├── liblitesata
    ├── liblitesdcard
    └── liblitespi
└── gateware
    ├── iface.py
    ├── top_main_ram_grain0.init
    ├── top_main_ram_grain1.init
    ├── top_main_ram_grain2.init
    ├── top_main_ram_grain3.init
    ├── top_mem.init
    ├── top_rom_grain0.init
    ├── top_rom_grain1.init
    ├── top_rom_grain2.init
    ├── top_rom_grain3.init
    ├── top_sdc
    ├── top_sram_grain0.init
    ├── top_sram_grain1.init
    ├── top_sram_grain2.init
    ├── top_sram_grain3.init
    ├── top.v
    ├── top.xml
    ├── top.peri.xml
    ├── top.bit
    ├── top.hex
    ├── top.pgm.out
    ├── top.bin
    ├── outflow
    ├── work_pnr
    ├── work_pt
    └── work_syn

```

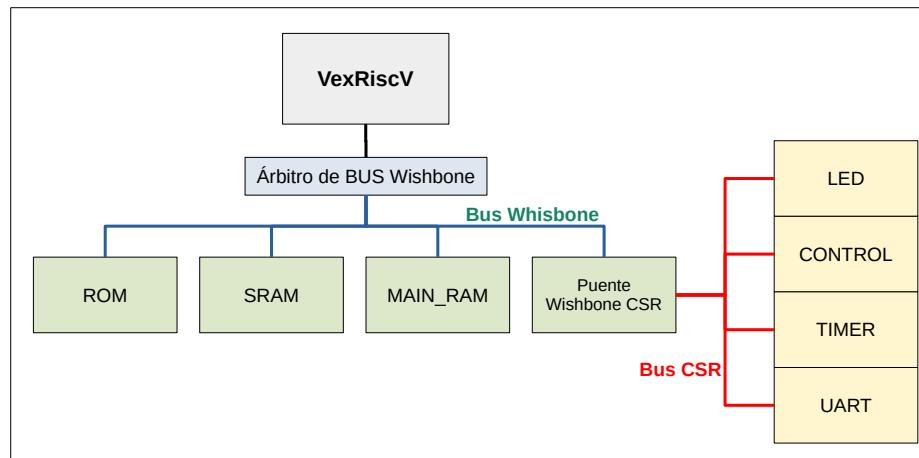
**Figura 4.8** Contenido de los archivos generados por *litex*

### Síntesis - gateware

Como puede verse existen dos sub-carpetas: *software* - que contiene los objetos compilados de la aplicación *bios* , la cual, proporciona herramientas básicas de manejo de periféricos, adicionalmente, permite cargar aplicaciones en las memorias disponibles (memoria RAM interna o SDRAM externa)utilizando los periféricos de comunicaciones como puerto serie o interfaz de red. En la segunda carpeta *gateware* se alojan los archivos necesarios para sintetizar y configurar el SoC en la FPGA, el contenido del dierctorio gareware varía dependiendo del fabricante de la FPGA

y de las herramientas de síntesis; sin embargo, siempre se generará el archivo **top.v** (o equivalente) con el código en verilog generado por *litex*.

En la figura 4.9 se muestra el diagrama de bloques del SoC generado (extraído del archivo *top.v*). En él podemos observar que se utiliza un árbitro de bus wishbone para manejar las memorias ROM, SRAM y RAM. Los periféricos son conectados a un puente Wishbone - CSR (Control and Status Register) en el que se conectan los 4 periféricos control (reset, scratch, bus\_error), Led, Timer, UART (puerto serie), y una memoria ROM (no mostrada en la figura) con la identificación del SoC "LiteX CPU Test on ecb\_t8\_t113".



**Figura 4.9** Diagrama de bloques del SoC generado por LiteX

### Compilación - software

Al cargar el archivo *build/top.bit* a la FPGA mediante el comando "sudo openFPGALoader -b trion\_t120\_bga576 build/gateware/top.bit" se configurará la FPGA con el procesador y las memorias inicializadas, de tal forma que se ejecutará la bios de LiteX, la que desplegará los mensajes que se muestran en la figura 4.10. Allí podemos ver que se despliega un mensaje de créditos, la información de la CPU utilizada, el tamaño de las memorias y se realiza una prueba de la memoria RAM, finalmente se despliega un prompt donde se pueden ejecutar comandos que proporciona la *bios* de acuerdo a los periféricos utilizados.

En el directorio *software* se incluye en directorio *include*, que es de vital importancia al momento de escribir aplicaciones propias (figura 4.8).

1. *bios*: Proporciona facilidades para manejo básico de los periféricos, en este caso el puerto serial y el LED, al tiempo que proporciona soporte para cargar aplicaciones por los medios disponibles (ROM, Serial, Ethernet, Flash, SD Card, SATA)
2. *libbase*: Funciones para manejo de los protocolos serial, i2c, escrituras de memorias SPI y las rutinas de atención a las interrupciones.
3. *libc*: Funciones que proporcionan las funciones *litex\_putc* y *litex\_getc*.
4. *libcompiler\_rt*: Definición de la función *\_mulsr3* entre muchas otras.
5. *libfatfs*: Funciones para trabajar con el sistema de archivos FAT, el que se utiliza para leer archivos de los diferentes periféricos donde la bios carga aplicaciones.
6. *liblitedram*: Funciones de lectura / escritura para memorias SDRAM.
7. *libliteeth*: Funciones para implementar los protocolos UDP, TFTP y MDIO.
8. *liblitesata*: Funciones de lectura y escritura para el bus SATA.
9. *liblitesdcard*: Funciones de bajo nivel para manejo de memorias SD.
10. *liblitespi*: Función de inicialización de memorias flash.
11. *include*

```

litex>

      _ _ _ / _ | _ / /
     / _ / _ / - ) > <
    / _ / _ / _ / | _ |
Build your hardware, easily!

(c) Copyright 2012-2022 Enjoy-Digital
(c) Copyright 2007-2015 M-Labs

BIOS built on Nov 26 2022 20:11:05
BIOS CRC passed (87e761ce)

LiteX git sha1: 310bc777

----- Soc -----
CPU:          VexRiscv @ 25MHz
BUS:          WISHBONE 32-bit @ 4GiB
CSR:          32-bit data
ROM:          32KiB
SRAM:         8KiB
MAIN-RAM:    16KiB

===== Initialization =====
Memtest at 0x40000000 (16.0KiB)...
  Write: 0x40000000-0x40004000 16.0KiB
  Read: 0x40000000-0x40004000 16.0KiB
Memtest OK
Memspeed at 0x40000000 (Sequential, 16.0KiB)...
  Write speed: 39.6MiB/s
  Read speed: 20.6MiB/s

----- Boot -----
Booting from serial...
Press Q or ESC to abort boot completely.
SL5DdSMmkekro
        Timeout
No boot medium found

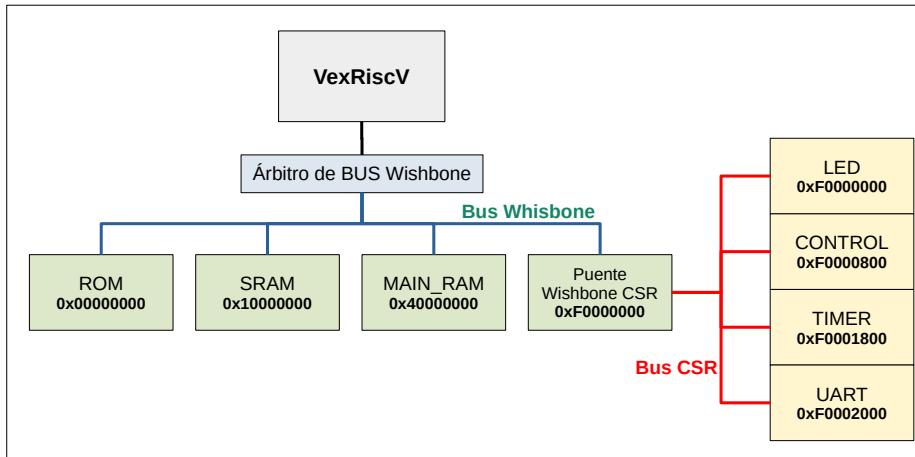
----- Console -----

```

**Figura 4.10** Mensajes de boot del SoC generado

- regions.ld, mem.h:** Estos dos archivos proporcionan información de la dirección de las memorias ROM (0x00000000), SRAM (0x10000000), MAIN\_RAM (0x40000000), CSR (0xF0000000).
- linker.ld:** Archivo de enlazado de la cadena de herramientas GNU, le indica al *linker* en que memoria colocar las diferentes secciones del archivo ELF, en este caso la memoria de programa (sección .text), las variables inicializadas (sección .data) y las cadenas de caracteres y constantes (sección .rodata) se almacenarán en la memoria MAIN\_RAM y las variables sin inicializar se almacenarán en la memoria SRAM.
- soc.h:** Funciones que retornan las especificaciones del SoC como frecuencia, CPU, ancho del bus de datos, bus utilizado, etc.
- csr.h:** Definiciones de la dirección de los periféricos y de sus respectivos registros, así como la definición de las funciones para controlar cada periférico.
- git.h, output\_format.ld:** Representan la versión del git de litex utilizada para crear el SoC y el formato de salida de los ejecutables compilados (elf32-littleriscv).
- variables.mk:** Declaración de los directorios necesarios para compilar aplicaciones y debe ser incluido en el *Makefile* de las aplicaciones.

Con lo anterior podemos asignar las direcciones de memoria a los componentes del SoC como se muestra en la figura 4.11.



**Figura 4.11** Direcciones de los componentes del SoC

### Archivos Makefile

Como se ha dicho en capítulos anteriores la herramienta *make* permite automatizar procesos de síntesis y compilación reduciendo la necesidad de ejecutar comandos de forma repetitiva, *make* utiliza como entrada archivos con el nombre *Makefile* o *makefile*. En el listado 4.5 se muestra un archivo típico para la síntesis HW-SW en LiteX.

```

48 TARGET=top
49 TOP=top
50 GATE_DIR=build/gateware
51 SOFT_DIR=build/software
52 LITEX_DIR=/home/carlos/Embedded/litex/
53 SERIAL?=/dev/ttyUSB0
54
55 all: gateware firmware
56
57 ${GATE_DIR}/${TARGET}.bit:
58     ./base.py
59
60 gateware: ${GATE_DIR}/${TARGET}.bit
61
62 ${SOFT_DIR}/common.mak: gateware
63
64 firmware: ${SOFT_DIR}/common.mak
65     $(MAKE) -C firmware/ -f Makefile all
66
67 litex_term: firmware
68     litex_term ${SERIAL} --kernel firmware/firmware.bin
69
70 configure: ${GATE_DIR}/${TARGET}.bit
71     sudo openFPGALoader -b colorlight-ft232rl -m ${GATE_DIR}/${TARGET}.bit
  
```

**Listing 4.5** Archivo Makefile

Este archivo permite: sintetizar el SoC junto con la *bios* (*make gateware* línea 60), la aplicación del usuario (*make firmware* línea 64), configurar la FPGA con el SoC y *bios* (*make configure* línea 70).

Para cargar la aplicación del usuario a la memoria RAM interna se utiliza la aplicación *litex\_term*, la cual, permite enviar un archivo para ser cargado en la dirección de la memoria (vía puerto serial) *MAIN\_RAM* (línea 67), esto es

posible gracias a que la bios (como puede verse en la figura 4.10) realiza una inspección del puerto serial enviando la cadena de caracteres *sL5DdSMmkekro*, si el SoC no recibe como respuesta la cadena *z6IHG7cYDID6o* antes que se cumpla un tiempo de espera determinado continua y despliega el prompt de *litex* (situación que muestra la figura 4.10). Si se recibe la cadena de caracteres *z6IHG7cYDID6o* antes de dicho tiempo, se establece una comunicación entre el computador donde se ejecuta *litex.term* y el SoC, y se transfiere el archivo que contiene la memoria de programa de la aplicación a la memoria *MAIN\_RAM*. Es importante mencionar que esta operación se realiza durante el boot de la *bios*, por lo que una vez ejecutado el comando *litex.term ..* (línea 67) debe hacerse un reset en el SoC, con lo que obtendremos los mensajes que aparecen en la figura 4.12.

```
--===== Boot ======
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[LITEX-TERM] Received firmware download request from the device.
[LITEX-TERM] Uploading firmware/firmware.bin to 0x40000000 (6136 bytes)...
[LITEX-TERM] Upload calibration... (inter-frame: 10.00us, length: 64)
[LITEX-TERM] Upload complete (9.9KB/s).
[LITEX-TERM] Booting the device.
[LITEX-TERM] Done.
Executing booted program at 0x40000000

--===== Liftoff! ======
Lab004 - CPU testing software built Nov 27 2022 18:26:32

Available commands:
help           - this command
reboot         - reboot CPU
leds-on        - leds test
leds-off       - leds test
RUNTIME>■
```

**Figura 4.12** Mensajes al cargar y ejecutar una aplicación a la memoria RAM del SoC

### Aplicaciones utilizando el SoC y la bios

El SoC obtenido junto con la *bios*, proporciona el software necesario para construir nuestra aplicación, como vimos anteriormente, *litex* en su proceso de compilación y síntesis crea los archivos necesarios para facilitar este proceso. En la figura 4.6 se muestra un archivo típico para automatizar el proceso de compilación.

```

48 BUILD.DIR=../build/
49 SERIAL?=/dev/ttyUSB1
50 include $(BUILD_DIR)/software/include/generated/variables.mak
51 include $(SOC_DIRECTORY)/software/common.mak
52 OBJECTS= crt0.o isr.o main.o
53 all: firmware.bin
54 # pull in dependency info for *existing* .o files
55 -include $(OBJECTS:.o=.d)
56 %.bin: %.elf
57     $(OBJCOPY) -O binary $< $@
58     chmod -x $@
59 firmware.elf: $(OBJECTS)
60     $(CC) $(LDFLAGS) \
61         -T linker.ld \
62         -N -o $@ \
63         $(OBJECTS) \
64         $(PACKAGES:%%=-L$(BUILD_DIR)/software/%) \
65         $(LIBS:lib%=-l%)
66     chmod -x $@
67 main.o: main.c
68     $(compile)
69 crt0.o: $(CPU_DIRECTORY)/crt0.S
70     $(assemble)
71 %.o: %.c
72     $(compile)
73 %.o: %.S
74     $(assemble)
75 litex_term: firmware.bin
76     @ls ${SERIAL} (echo "\n\nNo se encuentra ${SERIAL} conectado, verifique conexión o
77         cambie el valor de SERIAL=${SERIAL} por un puerto serial existente\n\n"; exit
78         123;)
77     litex_term ${SERIAL} --kernel firmware.bin
78 clean:
79     $(RM) $(OBJECTS) $(OBJECTS:.o=.d) firmware.elf firmware.bin .*~ *~
80 .PHONY: all main.o clean load

```

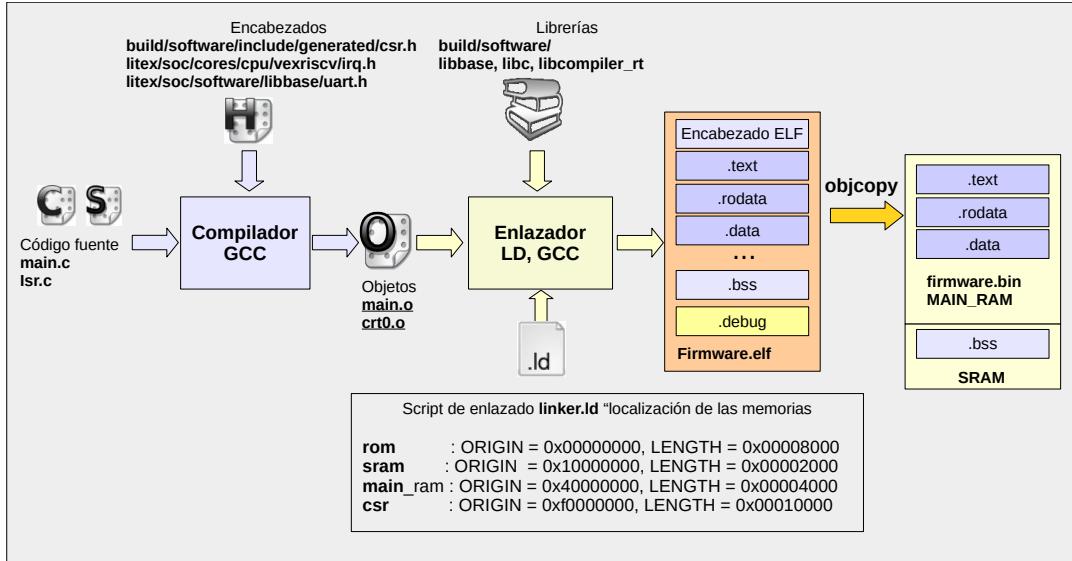
**Listing 4.6** Archivo Makefile para la aplicación del usuario

En este archivo se deben especificar los archivos que hacen parte del proyecto (línea 52), de los tres archivos listados (*crt0.o* *isr.o* *main.o*) solo *crt0.o* (que es el resultado de compilar el archivo *crt0.S* C runtime initialization) lo proporciona *litex*, ya que este es fuertemente dependiente del procesador utilizado, y como se dijo anteriormente, es posible utilizar múltiples CPUs.

Al momento de realizar el proceso de enlazado (línea 59), se utiliza el archivo *linker.ld*, el que le indica al compilador que la memoria de programa (sección *.text*), las constantes y cadenas de caracteres (sección *.rodata*), los valores de inicialización de las variables (sección *.data*) deben estar en la memoria MAIN\_RAM y las variables utilizadas en el programa (sección *.bss*) deben estar en la memoria SRAM (ver figura 4.13). El archivo *variables.mak* incluye las librerías generadas para compilar la *bios* (*libbase*, *libcompiler\_rt*, *libc*, etc) y las declara en la variable **LIBS**, la cual se utiliza en la etapa de enlace (línea 65), con lo que tenemos disponibles múltiples funciones para construir nuestra aplicación.

En el listado 4.7 se muestra un archivo *main.c* que implementa la funcionalidad de la aplicación de usuario. Este es un archivo típico del lenguaje C, donde se declaran al comienzo las librerías que se utilizarán, en este caso se incluyen los archivos de encabezado *stdio.h* (operaciones de entrada salida de la biblioteca standard de C), *stdlib.h* (gestión de memoria dinámica, control de procesos entre otros) y *string.h* (funciones de manipulación de memoria); encabezados relacionados con el hardware *irq.h* (localizada en *litex/soc/cores/cpu/vexriscv/irq.h*) la que declara las funciones para manejo de interrupciones y *uart.h* localizado en (*litex/soc/software/libbase/uart.h*) que declara las funciones para controlar la uart.

El archivo *csr.h* (localizado en *build/software/include/generated/csr.h*) define las direcciones de memoria de los periféricos y de sus registros, al tiempo que declara las funciones para leer y escribir en estos registros y de esta forma controlar los periféricos (En la figura 4.14 se resume el contenido del archivo *csr.h*)

Figura 4.13 Flujo de diseño software para la aplicación *firmware*

```

48 #include <stdio.h>
49 #include <stdlib.h>
50 #include <string.h>
51 #include <irq.h>
52 #include <uart.h>
53 #include <generated/csr.h>
54 void my_busy_wait(unsigned int ms)
55 {
56     timer0_en_write(0);
57     timer0_reload_write(0);
58     timer0_load_write(CONFIG_CLOCK_FREQUENCY/1000*ms);
59     timer0_en_write(1);
60     timer0_update_value_write(1);
61     while(timer0_value_read()) timer0_update_value_write(1);
62 }
63 int main(void)
64 {
65     irq_setmask(0);
66     irq_setie(0);
67     uart_init();
68     puts("\nCPU testing cain-test SoC\n");
69     printf("Hola Mundo \n");
70     while(1) {
71         leds_out_write(0);
72         my_busy_wait(250);
73         leds_out_write(1);
74         my_busy_wait(250);
75     }
76     return 0;
77 }

```

Listing 4.7 Archivo Makefile

En la línea 54 del listado 4.7 se hace la declaración de la función `my_busy_wait`, que hace uso del timer para realizar un retardo controlado del orden de los milisegundos. Para entender el funcionamiento de los periféricos proporcionados por litex se debe ver su implementación en el directorio `litex/litex/soc/cores`, en el archivo `timer.py` se explican los diferentes tipos de funcionamiento y define el modo *One-Shot* en el que el registro que almacena el conteo del timer se carga con un valor predefinido, cada vez que se reciba un pulso de reloj este valor disminuirá hasta que se haga cero, los pasos que se deben seguir para activar este modo son:

1. Deshabilitar el timer escribiendo un 0 en el registro *timer0\_en*.
2. Escribir en el registro *load* el valor del conteo que se desea realizar (*timer0\_load\_write*). En este caso se usa la constante *CONFIG\_CLOCK\_FREQUENCY* (declarada en el archivo *csr.csv*) que tiene la frecuencia del reloj del sistema, en este caso 33.333.333 al dividirlo por 1000 se obtiene el número de ciclos de reloj que necesitamos para contar un milisecondo, al multiplicarlo por *ms* se obtendrá el número de ciclos de reloj para cumplir este tiempo.
3. Habilitar el timer escribiendo un 1 en el registro *timer0\_en*.

Con lo anterior el valor del contador del timer disminuye en 1 cada ciclo de reloj, para leer este valor se debe actualizar el valor del registro de lectura, esto se hace al escribir un 1 en el registro *timer0\_update\_value*, si el valor del registro es diferente de 0 se refresca constantemente el valor de este registro hasta que sea igual a cero con lo que la función terminará.

<b>LED</b>	<b>0xF0000000</b>	<i>leds_out_read</i> <i>leds_out_write</i>
<b>CONTROL</b>	<b>0xF0000800</b>	<i>ctrl_reset_soc_rst_extract</i> <i>ctrl_reset_soc_rst_read</i> <i>ctrl_reset_soc_rst_replace</i> <i>ctrl_reset_soc_rst_write etc.</i>
<b>TIMER</b>	<b>0xF0001800</b>	<i>timer0_load_read/write</i> <i>timer0_reload_read/write</i> <i>timer0_en_read/write</i> <i>timer0_update_value_read/write</i> <i>timer0_value_read/write</i> <i>timer0_ev_status_read/write</i> <i>timer0_ev_status_zero_extract/read</i> <i>timer0_ev_pending_read/write</i> <i>timer0_ev_enable_read/write</i>
<b>UART</b>	<b>0xF0002000</b>	<i>uart_rxtx_read/write</i> <i>uart_txfull_read/write</i> <i>uart_rxempty_read/write</i> <i>uart_ev_status_read/tx_read/rx_read</i> <i>uart_ev_pending_read/tx_read/rx_read</i> <i>uart_ev_enable_read/write</i> <i>uart_txempty_read</i> <i>uart_rxfull_read</i>

Figura 4.14 Contenido del archivo *csr.h*

En la línea 63 se declara la función *main*, en ella se deshabilitan las interrupciones globales y se inicializa la uart, se imprime un mensaje por el puerto serial y se ingresa a un ciclo infinito donde se enciende y se apaga un led esperando 250ms. Acá se utiliza la función *leds\_out\_write* para fijar el valor del único led que tenemos.

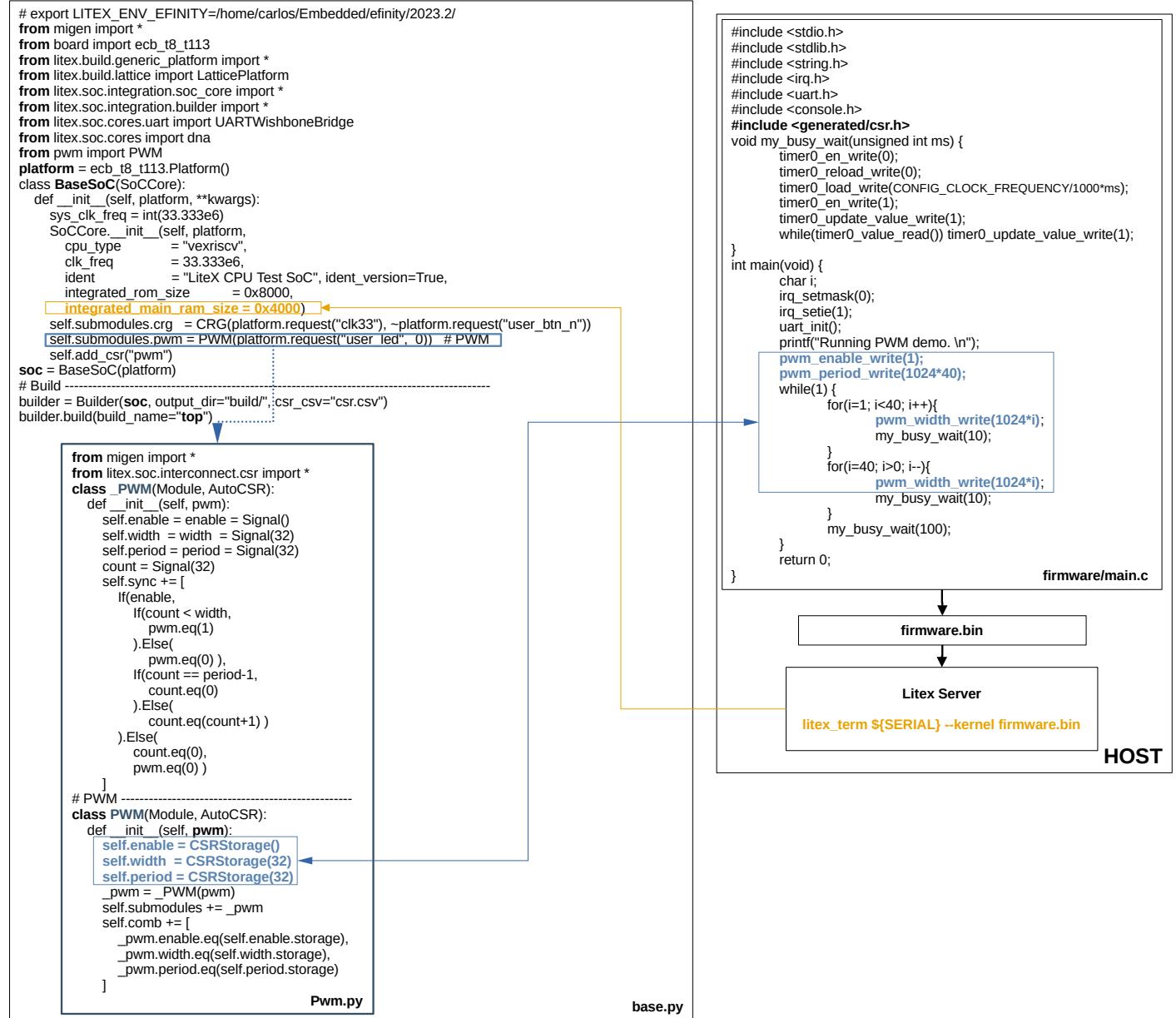
Para cargar este programa se debe cargar el archivo *firmware.bin* a la memoria MAIN\_RAM (0x40000000), como se dijo anteriormente se debe hacer utilizando la aplicación *liteX\_term*, para esto se ejecuta el comando:

```
make liteX_term
```

#### 4.2.6. Creación de periféricos en migen

En este apartado incluiremos el periférico que depuramos con la herramienta *liteX\_server* (pwm) en el SoC creado en la sección anterior, y relizaremos una aplicación para controlarlo, esta aplicación se cargará a la memoria *MAIN\_RAM*. En la figura 4.15 se muestra el código necesario para instanciar el periférico pwm (archivos *base.py* y *pwm.py*).

En el archivo *base.py* se instancia el módulo *pwm* y se agregan los registros de status y control al mapa de memoria del procesador mediante la función *add\_csr*.



**Figura 4.15** Instanciación del periférico pwm en un SoC con un VexriscV

El programa que controla este periférico utiliza las funciones `pwm.enable_write`, `pwm.period_write` y `pwm_width_write` declaradas en el archivo `generated/csr.h`. Una vez se genera el archivo `firmware.bin` que contiene la memoria de programa debe transmitirse vía puerto serie a la placa en la que se ejecuta la aplicación `bios`, para esto se utiliza la aplicación `litex_term`.

#### 4.2.7. Creación de periféricos en verilog

En la sección anterior se introdujo la herramienta `litex` y se mostró la forma en que se pueden declarar SoC utilizando los cores que vienen por defecto, en esta sección se mostrará cómo se trabaja con periféricos creados por el diseñador para personalizar las tareas que realiza el SoC. para esto es necesario recordar la arquitectura mostrada en la figura

4.11, y recordar que el bus *wishbone* (explicado con detalle en el capítulo anterior) es utilizado como medio de comunicación entre los periféricos y el procesador. Sin embargo, en la arquitectura del SoC definida por *liteX* existe un puente entre el bus *wishbone* y el bus CSR utilizado por los periféricos, para comprender cómo está implementado este puente y cómo se comunica con los periféricos se realizará un análisis detallado en la siguiente subsección.

### Puente wishbone - CSR

Como se indicó en la figura 4.9 la arquitectura del SoC de *LiteX* posee un bus *wishbone* que permite la conexión con cuatro periféricos: memorias ROM, SRAM, MAIN\_RAM y CSR. Debido a que no es posible implementar señales bidireccionales dentro de la FPGA, el procesador posee dos buses de datos, uno para escritura (*ibus\_dat\_w*) y otro para lectura (*ibus\_dat\_r*), el bus de escritura se conecta directamente a la entrada de los cuatro periféricos y el bus de lectura se conecta a un multiplexor cuyas entradas son las salidas de datos de los periféricos (ver figura 4.16), únicamente el periférico activado envía la información al procesador, esta activación se realiza mediante una señal de selección.

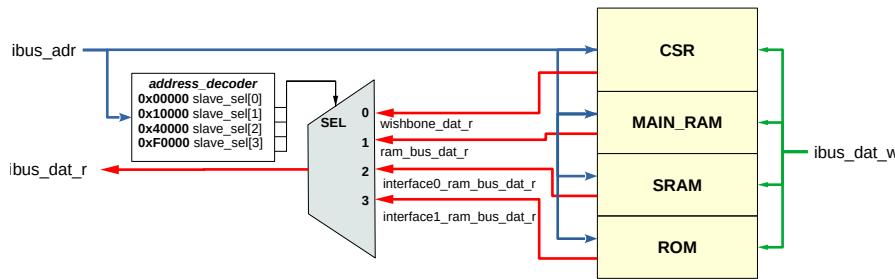


Figura 4.16 Buses de datos y direcciones en LiteX

Estas señales de activación se controlan con el bus de direcciones del procesador (*ibus.adr*) y se asigna a un rango de direcciones determinadas, las cuales no pueden traslaparse; de esta forma se implementa el mapa de memoria que se muestra en la figura 4.10.

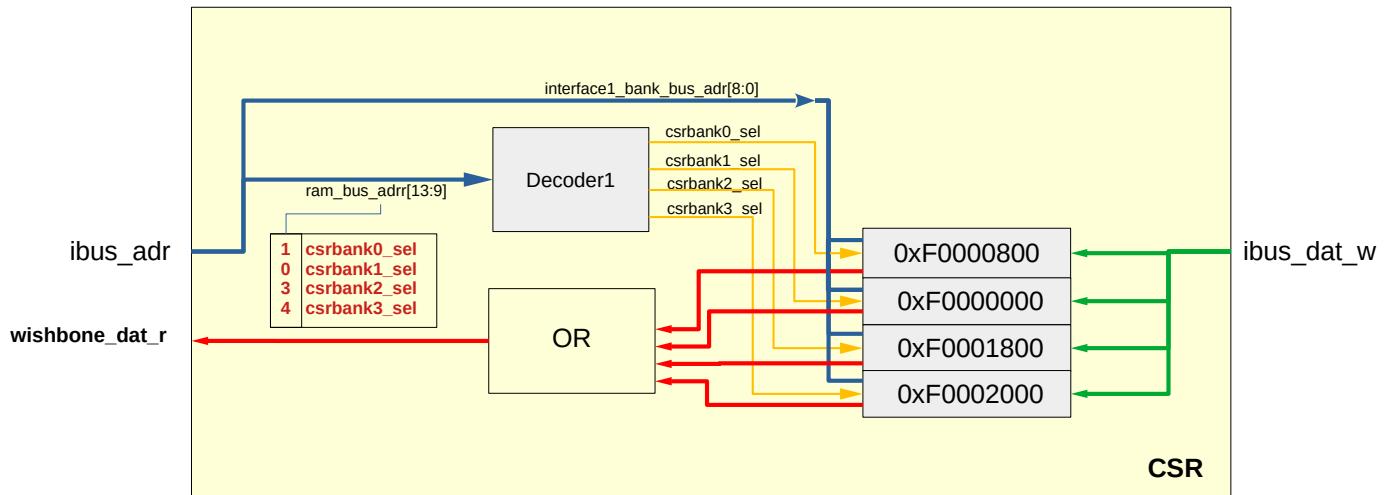


Figura 4.17 Buses de datos y direcciones dentro del puente Wishbone - CSR

La figura 4.17 muestra el diagrama interno del bloque puente *wishbone - CSR*, donde se ve la existencia de un decodificador de direcciones que habilita (usando las señales *csrbankX\_sel*) los diferentes periféricos implementando el mapa de memoria del CSR, el bus de datos de escritura del procesador ingresa a todos los periféricos - CSR, y para la salida de los datos de escritura de los periféricos se hace mediante una compuerta OR, cuando un periférico es seleccionado

coloca el dato en su bus de datos de salida, (los periféricos - CSR no seleccionados colocan su bus de datos en 0) la señal del bus de direcciones del procesador ingresa a todos los periféricos.

Finalmente, en la figura 4.18 se muestran los buses de datos y dirección que hacen parte del periférico y se muestra como se almacena o se lee la información de los registros internos, las operaciones de lectura y escritura de/hacia los registros solo se realizan si la señal `csrbank0_sel` se encuentra activa, indicando que el procesador realiza una operación en el rango de memoria del periférico - CSR. El decodificador implementa el mapa de memoria del periférico indicando el rango de memoria y la dirección base de cada registro.

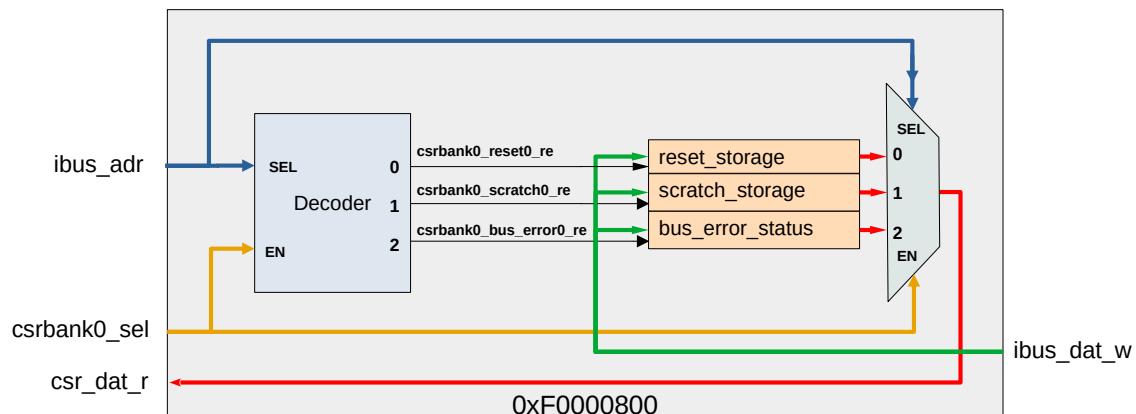


Figura 4.18 Buses de datos y direcciones de un periférico CSR

### Periférico escrito en Verilog

Para entender el proceso de re-utilizar periféricos escritos en verilog utilizaremos el código de la uart que utilizamos en el capítulo anterior como ejemplo para entender cómo se adapta un periférico al bus wishbone; en la figura 4.19 se muestra el diagrama de entradas y salidas de este periférico indicando el número de bits de cada una de ellas.

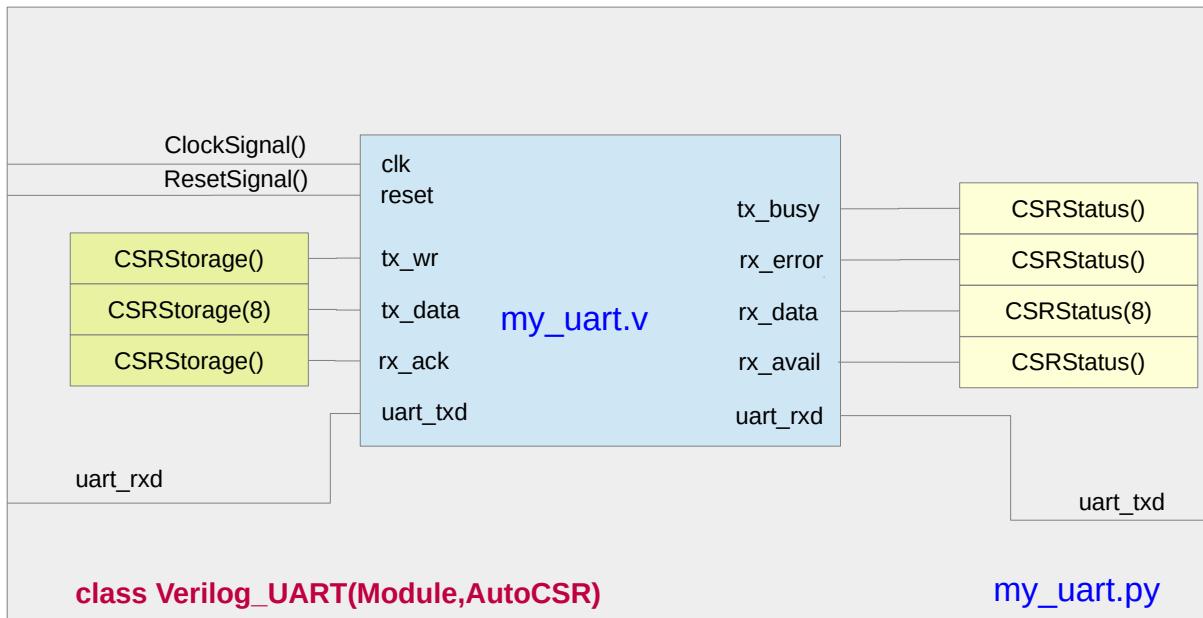
clk	
reset	tx_busy
tx_wr	rx_error
tx_data	my_uart.v
rx_ack	rx_data
uart_txd	rx_avail
	uart_rxd

Figura 4.19 UART escrita en verilog

Para incluir un código verilog en *LiteX* como un core externo es necesario: 1 - La configuración del core (usando parámetros) y la descripción de la interfaz para la integración en el diseño y 2 - La lista de archivos que describen el core para que las herramientas de síntesis la tenga en cuenta. *LiteX* ve el core como una caja negra de la que conoce su nombre y su interfaz y su contenido será tenido en cuenta solo al momento de la síntesis.

Para la instanciación se debe crear un archivo en python que encapsule el módulo verilog y lo presente al entorno de desarrollo. La figura 4.21 ilustra este concepto. La interfaz está compuesta por las señales que se acceden de forma directa, en este caso las señales de reloj, reset y las señales de recepción y transmisión de la uart; las otras señales

serán leídas o escritas a través de registros; las señales de entrada serán registros tipo almacenamiento (CSRStorage) y las señales de salida del módulo serán tipo (CSRStatus).



**Figura 4.20** Instanciación del módulo HDL para ser presentado como core

En el listado 4.8 se muestra la clase Verilog\_UART que realiza la instanciación mostrada en la figura 4.21. En ella se declara el nombre de la clase (línea 52) la cual, al ser derivada de *AutoCSR* le agregará a CSRStatus y CSRStorage métodos de acceso al bus CSR automáticamente y conecta las señales de la uart a las salidas del bloque (*data*, *uart\_rxd*, *uart\_txd*). Las señales de reloj y de reset son comunes a todos los módulos y se declaran como tipo *ClockSignal* y *ResetSignal* respectivamente (línea 55), se declaran las señales *uart\_txd* y *uart\_rxd* como componentes de la variable *data*, los registros de solo lectura y lectura-escritura se declaran a continuación (línea 60) y finalmente *self.specials* es una característica especial de la herramienta LiteX que permite agregar elementos específicos a nivel de diseño RTL. La *Instance* es una clase que representa una instancia de un componente RTL, es decir, un módulo de hardware que será sintetizado en la FPGA, (línea 69), el primer parámetro es el nombre del módulo del archivo en verilog (*uart\_tranceiver*) seguido de los parámetros y puertos del Módulo, se deben utilizar los siguientes prefijos para especificar el tipo de interfaz:

- *p\_* para un parámetro ( str o int de Python o una constante de Migen).
- *i\_* para un puerto de entrada (int de Python o Signal, Cat, Slice de Migen).
- *o\_* para un puerto de salida (int de Python o Signal, Cat, Slice de Migen).
- *io\_* para un puerto bidireccional (Signal, Cat, Slice de Migen).

```

48 from migen import *
49 from migen.genlib.cdc import MultiReg
50 from litex.soc.interconnect.csr import *
51 from litex.soc.interconnect.csr_eventmanager import *
52 class Verilog_UART(Module, AutoCSR):
53     def __init__(self, data):
54         # Interfaz
55         self.clk      = ClockSignal()
56         self.rst      = ResetSignal()
57         self uart_txd = data.uart_txd
58         self uart_rxd = data.uart_rxd
59         # registros solo lectura
60         self.rx_data  = CSRStatus(8)
61         self.rx_avail = CSRStatus()
62         self.rx_error = CSRStatus()
63         self.tx_busy  = CSRStatus()
64         # Registros solo escritura
65         self.rx_ack   = CSRStorage()
66         self.tx_data  = CSRStorage(8)
67         self.tx_wr    = CSRStorage()
68         # Instanciación del módulo verilog
69         self.specials += Instance("uart_transceiver",
70             i_clk      = self.clk,
71             i_reset    = self.rst,
72             o_uart_txd = self.uart_txd,
73             i_uart_rxd = self.uart_rxd,
74             o_rx_data  = self.rx_data.status,
75             o_rx_avail = self.rx_avail.status,
76             o_rx_error = self.rx_error.status,
77             o_tx_busy  = self.tx_busy.status,
78             i_rx_ack   = self.rx_ack.storage,
79             i_tx_data  = self.tx_data.storage,
80             i_tx_wr    = DSD= self.tx_wr.storage,
81         )
82         self.submodules.ev = EventManager()
83         self.ev.ok = EventSourceProcess()
84         self.ev.finalize()

```

**Listing 4.8** Clase custom\_uart\_rtl escrita en python (archivo module/my\_uart.py)

Una vez instanciado el módulo se indica que se deben asignar señales de manejo de eventos a este módulo (línea 82, con esto se reservan señales para atención de interrupciones.

El siguiente paso consiste en llamar la clase recién creada desde el archivo de definición del SoC, en el listado 4.9 se toma como base el SoC de la sección anterior y se resaltan las líneas que deben adicionarse para instanciar el módulo en verilog.

```
#!/usr/bin/env python3
from migen import *
from board import ecb_t8_t113
from litex.build.generic_platform import *
from litex.build.lattice import EfinixPlatform
from litex.soc.integration.soc_core import *
from litex.soc.integration.builder import *
from litex.soc.cores import dna
from pwm import PWM
from module import my_uart

_custom_serial = [
    ("custom_serial", 1,
        Subsignal("uart_txd", Pins("141")), # J1.3
        Subsignal("uart_rxd", Pins("139")), # J1.5
        IOStandard("LVCMOS33")
    ),
]
platform = ecb_t8_t113.Platform()
class BaseSoC(SoCCore):
    def __init__(self, platform, **kwargs):
        sys_clk_freq = int(33.333e6)
        platform.add_extension(_custom_serial)
        platform.add_source("module/my_uart.v")
        SoCCore.__init__(self, platform,
            cpu_type = "vexriscv",
            clk_freq = 33.333e6,
            ident = "LiteX CPU Test SoC", ident_version=True,
            integrated_rom_size = 0x8000,
            integrated_main_ram_size = 0x4000)
        self.submodules.crg = CRG(platform.request("clk33"), ~platform.request("user_btn_n"))
        self.submodules.pwm = PWM(platform.request("user_led", 0)) # PWM
        self.add_csr("pwm")
        SoCCore.add_csr(self,"uart_ver")
        self.submodules.uart_ver = custom_uart.custom_uart_rtl(platform.request("custom_serial",1))
soc = BaseSoC(platform)
# Build -----
builder = Builder(soc, output_dir="build/", csr_csv="csr.csv")
builder.build(build_name="top")
```

**Listing 4.9** SoC que utiliza un core escrito en verilog (archivo base.py)

La línea amarilla le indica a la herramienta que debe importar el archivo *my\_uart.py* localizado en la carpeta *module*; en las líneas rosadas se declaran los pines externos que utilizará el periférico *uart\_txd* y *uart\_rxd*, en las líneas café se indica que se adicionen estos pines al SoC; en la línea azul se le indica al sintetizador que adicione el archivo *module/my\_uart.v*, en las líneas grises se declara un nuevo objeto tipo *CSR* y se le asigna la clase *Verilog-UART* localizada en el archivo *my\_uart.py* (en el directorio *module*). Con esto al finalizar el proceso de creación del SoC se obtienen los nuevos registros (archivo *csr.csv*) y periféricos que aparecen en el listado 4.10, acá se observa que aparece el periférico declarado junto con sus registros.

```

csr_base ,uart_ver ,          0xf0000000 ,,
csr_base ,pwm,                0xf0000800 ,,
csr_base ,ctrl ,              0xf0001000 ,,
csr_base ,identifier_mem ,   0xf0001800 ,,
csr_base ,timer0 ,            0xf0002000 ,,
csr_base ,uart ,               0xf0002800 ,,
csr_register ,uart_ver_rx_data , 0xf0000000 ,1 ,ro
csr_register ,uart_ver_rx_avail , 0xf0000004 ,1 ,ro
csr_register ,uart_ver_rx_error , 0xf0000008 ,1 ,ro
csr_register ,uart_ver_tx_busy , 0xf000000c ,1 ,ro
csr_register ,uart_ver_rx_ack , 0xf0000010 ,1 ,rw
csr_register ,uart_ver_tx_data , 0xf0000014 ,1 ,rw
csr_register ,uart_ver_tx_wr , 0xf0000018 ,1 ,rw
csr_register ,uart_ver_ev_status , 0xf000001c ,1 ,ro
csr_register ,uart_ver_ev_pending , 0xf0000020 ,1 ,rw
csr_register ,uart_ver_ev_enable , 0xf0000024 ,1 ,rw

```

**Listing 4.10** Registros para comunicarse con el core que utiliza verilog

### Control del periférico escrito en Verilog

Para hacer uso del periférico - CSR my\_uart se deben escribir funciones para controlar la recepción y la transmisión:

- Recepción de un byte (El puerto serial transmite y recibe datos de 8 bits únicamente): La función *my\_uart\_getchar* retorna el valor del carácter recibido por el puerto uart\_rxd, para esto, se examina el valor de la señal rx\_avail, la cual, tiene un estado lógico alto cuando recibe un carácter; litex crea la función *uart\_ver\_rx\_avail\_read* que retorna el valor de este registro.,
- Transmisión de u'n byte: *Litex* proporciona la función *uart\_ver\_tx\_data\_write()* para escribir el dato a ser transmitido, esto se debe hacer cuando my\_uart no esté ocupada transmitiendo otro carácter por lo que se debe revisar el estado de la señal *tx\_busy* con la función *uart\_ver\_tx\_busy\_read*. Para que se active la transmisión es necesario que la señal *tx\_wr* esté en 1 (*uart\_ver\_tx\_wr\_write(1)*), pero debe pasarse a 0 inmediatamente (*uart\_ver\_tx\_wr\_write(0)*), de lo contrario transmitirá de forma infinita el dato contenido en el registro de transmisión.
- Transmisión de una cadena de caracteres: La función *my\_uart\_putstr* recibe una cadena de caracteres y los envía uno a uno usando la función *my\_uart\_putchar*.

En la figura 4.21 se muestra el contenido del archivo main.c donde se implementan estas funciones.

En la figura 4.21 podemos observar como se relacionan los diferentes archivos que hacen parte de este ejemplo.

## 4.3. Herramientas de depuración

*Litex* proporciona herramientas que permiten depurar de forma fácil periféricos o sistemas completos. La mayoría de estas herramientas son una combinación de un *maestro wishbone* y un servidor de comunicaciones, el maestro wishbone toma el control del bus wishbone enviando y recibiendo información, que activa un o un grupo de periférico/s; adicionalmente, establece una comunicación con un servidor que se ejecuta en el PC, esta comunicación puede ser via UART, ETH o PCI.

### 4.3.1. UartBone

De forma similar a la que se empleo para depurar el periférico PWM usando el core UartBridge (sección 4.2.4, figura 4.6), es posible depurar un SoC utilizado una interfaz serial y *litex\_server*. En la figura 4.22 se resaltan en morado las líneas que deben ser agregadas para realizar la depuración de periféricos utilizando el core *uartbone*. Lo primero que se debe hacer es asignar los pines para el puerto serial; en la asignación *\_serial\_debug* se usa el mismo nombre que la interfaz serial de la plataforma pero con el ID 1.

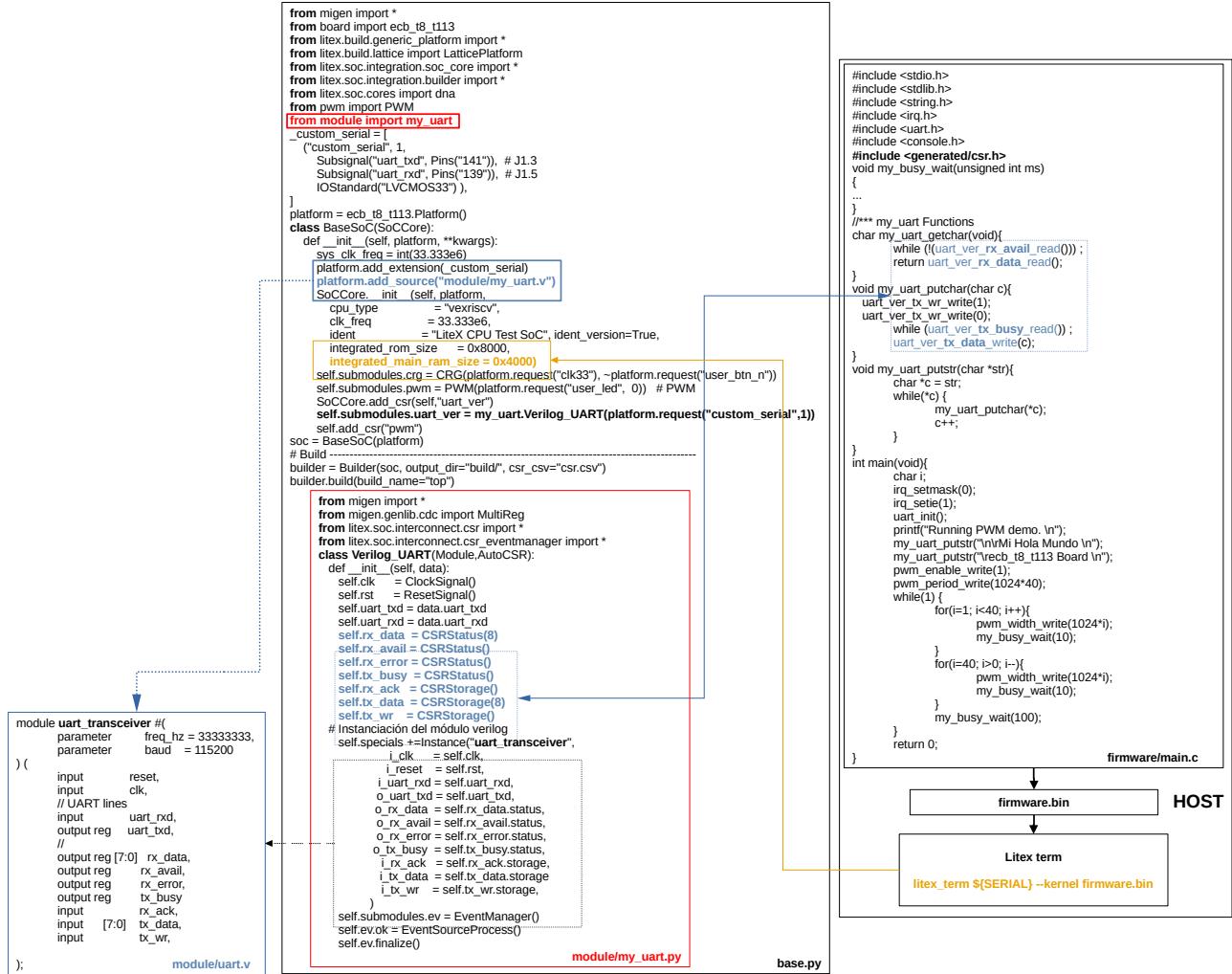


Figura 4.21 Instanciación del módulo HDL para ser presentado como core

El archivo de prueba `test_pwm.py` utilizado anteriormente se puede re-utilizar sin realizar ninguna modificación. *LiteX* genera una arquitectura adicionando la `uartbone` como se muestra en la figura 4.23, aquí la `uartbone` se conecta como si fuera un maestro wishbone y puede controlar los buses de dirección(`xxx_addr`) y de datos de escritura (`xxx_dat_w`) de la misma forma que lo hace el procesador, esto se realiza con los multiplexores que se muestran en verde (datos de escritura) y azul(direcciones), el selector de estos multiplexores está formado por las señales `uartbone_cyc`, `dbus_cyc` e `ibus_cyc`, recordemos que la señal `cyc` del bus wishbone se activa siempre que se realiza una operación de lectura o escritura, cuando se activa alguna de ellas, se espera a que termine la operación actual y se permite el acceso a los buses por parte del solicitante.

### 4.3.2. Etherbone

Como se dijo anteriormente, se pueden utilizar diferentes medios para comunicarse con el SoC, en la figura 4.24 se muestran los archivos necesarios para establecer la comunicación via udp.

## 4.3 Herramientas de depuración

125

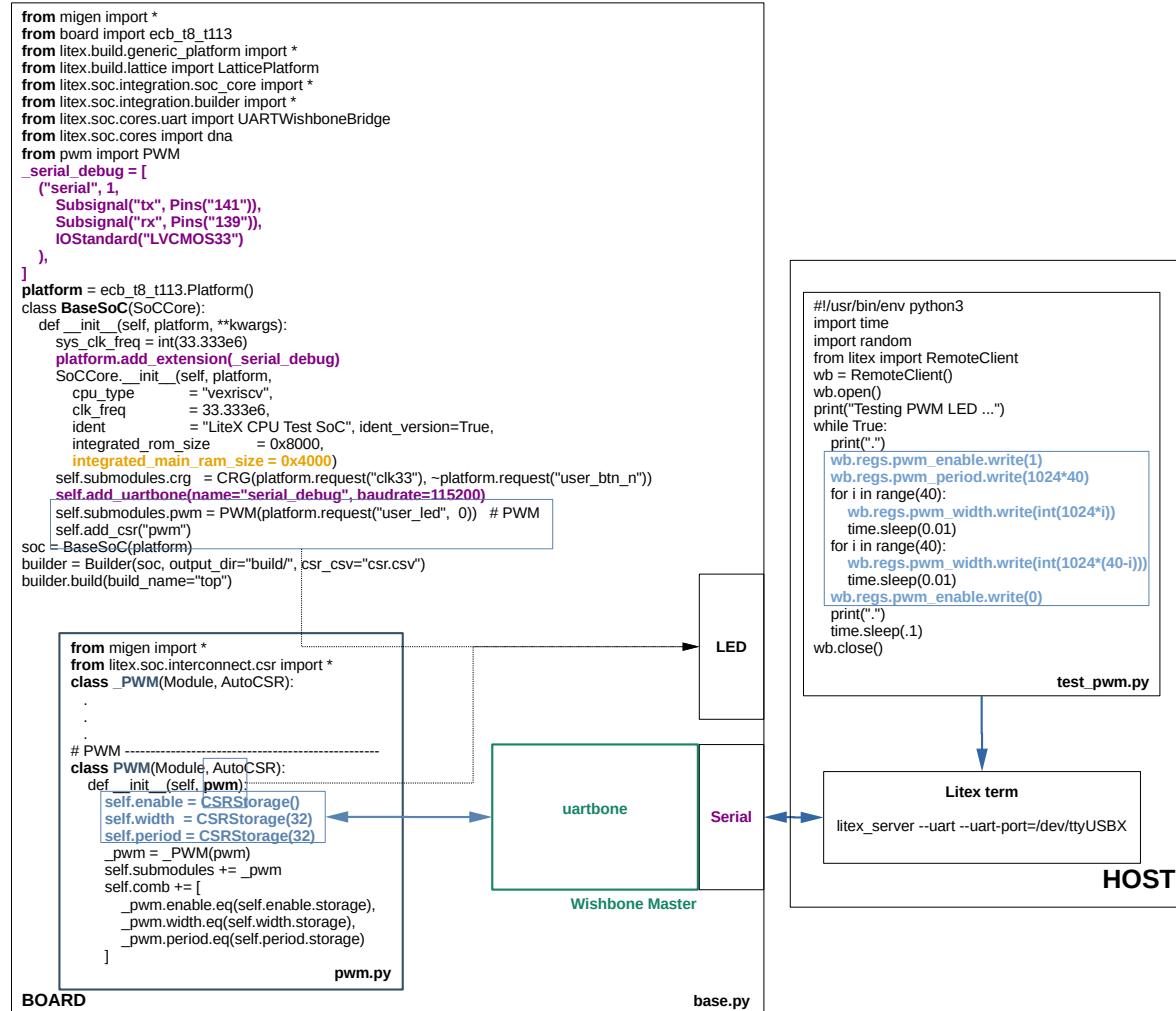


Figura 4.22 Depuración de periféricos instanciados en un core

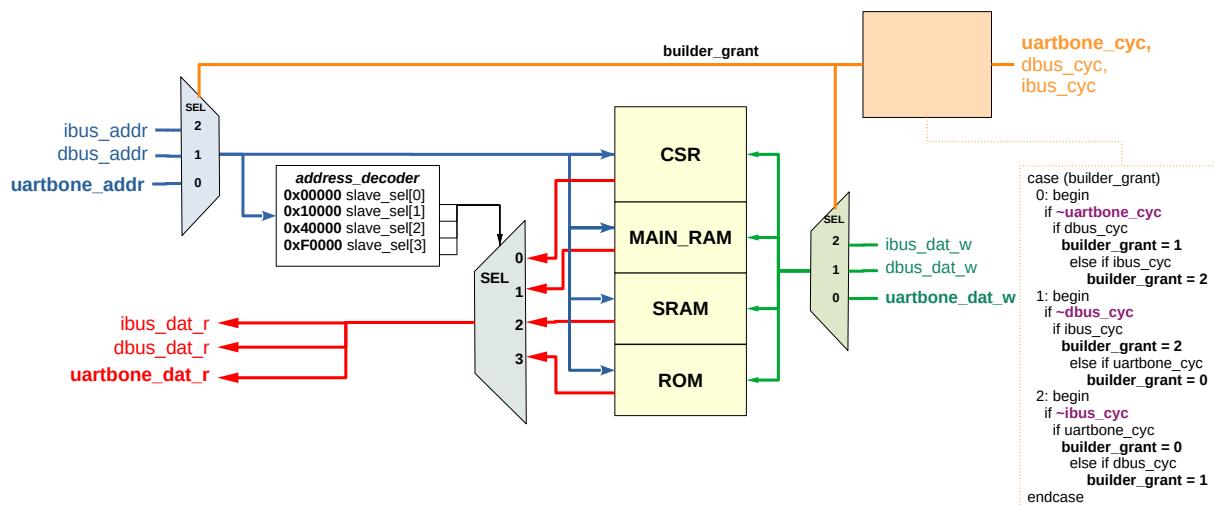


Figura 4.23 Diagrama de bloques de los buses al instanciar un puente entre el SoC y el servidor LiteX

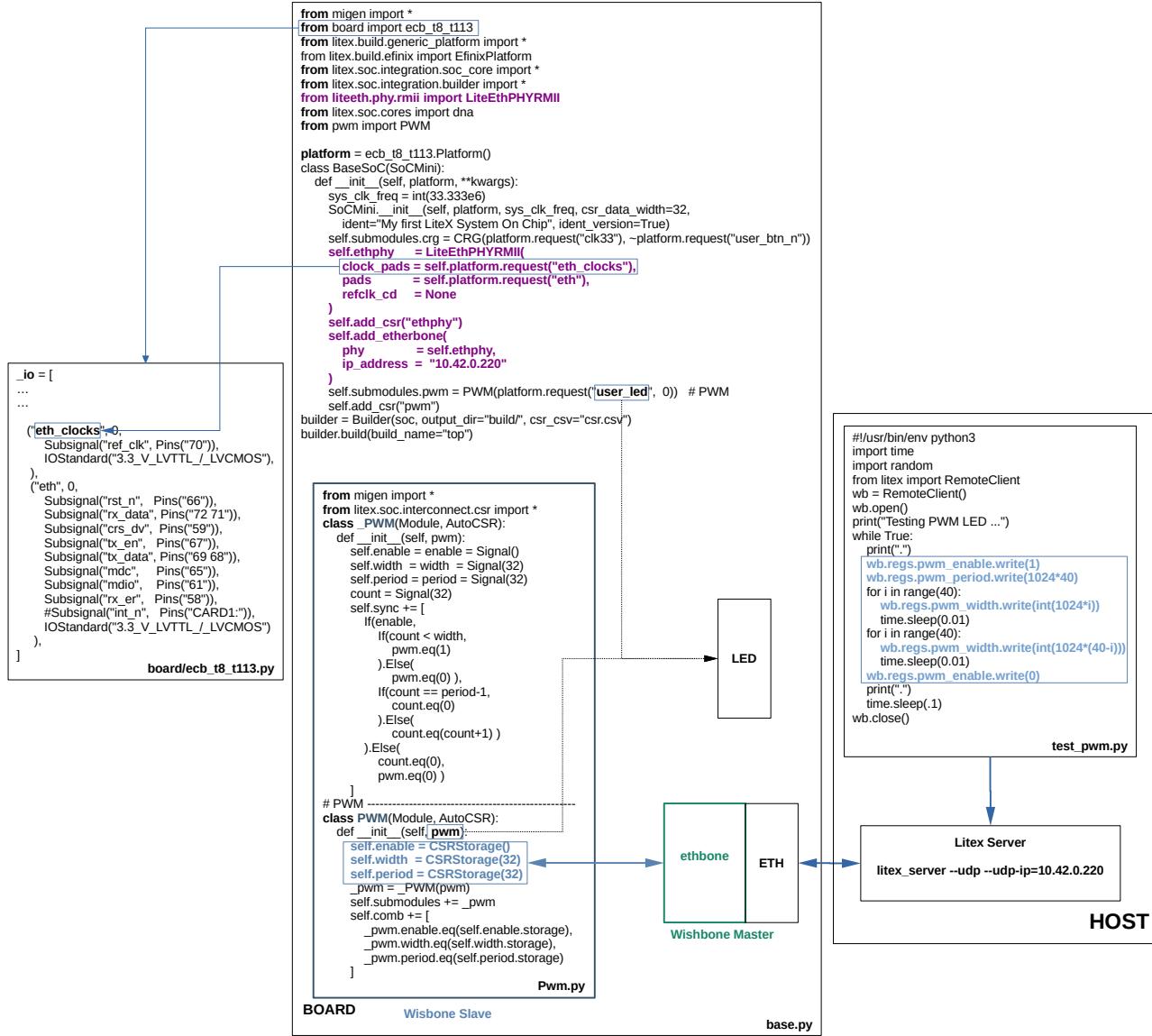


Figura 4.24 Depuración utilizando ethbone

### 4.3.3. LiteX cli

La herramienta *liteX\_cli* proporciona una interfaz gráfica que permite interactuar con los registros del SoC, está desarrollada en el framework *dearpygui* (se debe instalar **por fuera de conda** con el comando: `pip3 install dearpygui`), al ejecutarse (`liteX_cli --gui`) se establece una conexión vía tcp con el servidor que está conectado al puerto serial del SoC y despliega una venta como la de la figura 4.25. En esta aplicación es posible escribir el valor deseado en cualquier registro; en nuestro caso si cambiamos el valor del registro *pwm\_width* podemos observar un cambio en la intensidad del LED.

```
liteX_server --uart --uart-port=/dev/ttyUSB1 &
liteX_cli --gui
```

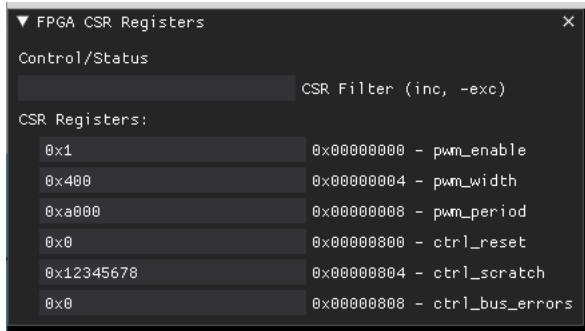


Figura 4.25 Interfaz gráfica de litex\_gui

#### 4.3.4. *litescope*

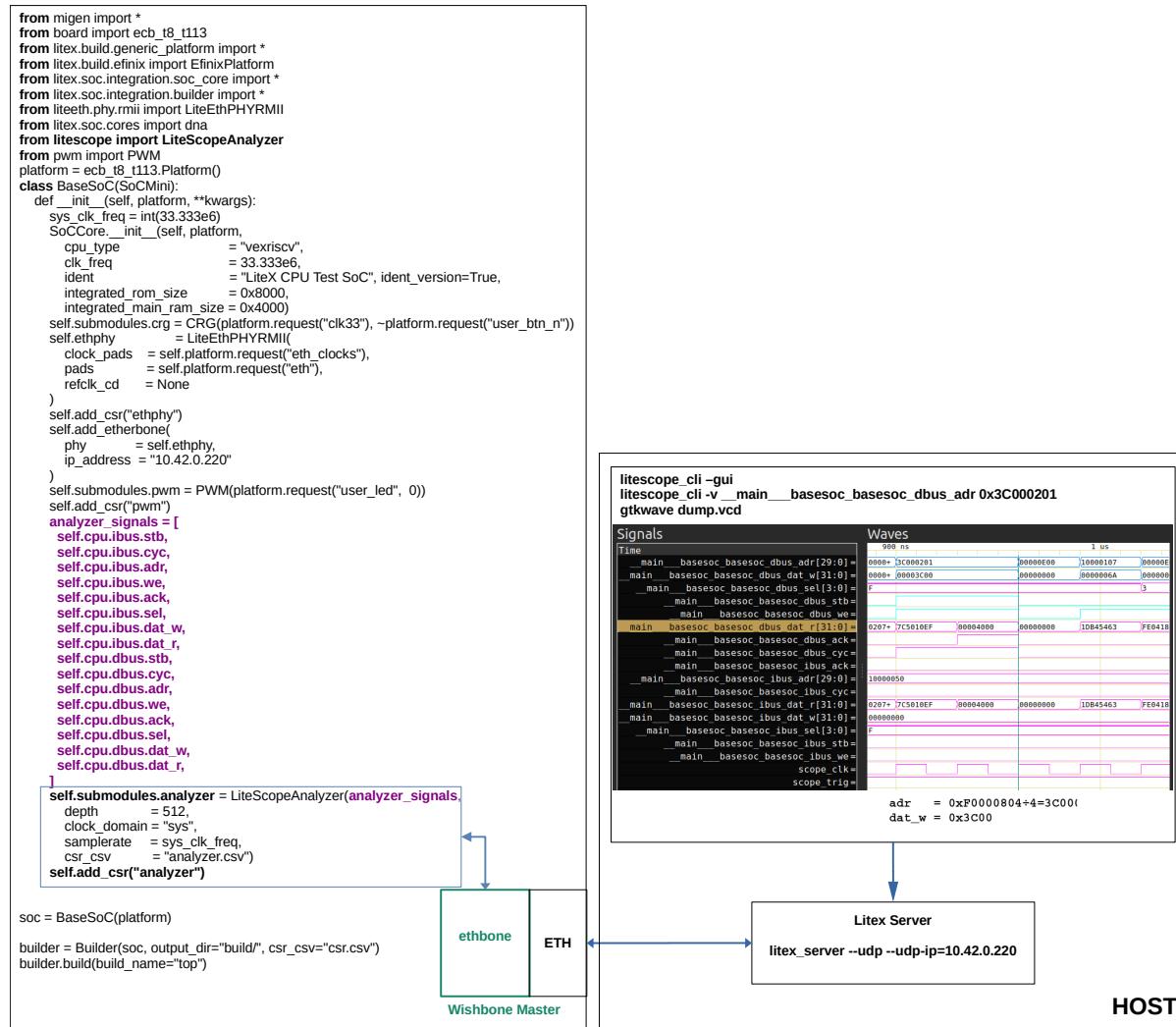
Litescope es un analizador lógico embebido de tamaño pequeño que se puede utilizar en la FPGA, hace parte de las librerías de Litex, dentro de sus características tenemos:

- IO peek and poke with LiteScopeIO.
- Analizador lógico LiteScopeAnalyzer:
- Subsampling.
- Almacenamiento de datos en Block RAM.
- Triggers configurables.
- puentes:
  - UART → Wishbone
  - Ethernet → Wishbone ("Etherbone")
  - PCIe → Wishbone
- Formats: .vcd, .sr(sigrok), .csv, .py, etc..

En la figura 4.26 se muestra como incluir este core en el SoC y como visualizar los resultados de la ejecución del comando:

```
litescope_cli -v __main__basesoc_basesoc_dbus_adr 0x3C000201
```

Donde *0x3C000201* es la dirección del registro width del pwm (aparece en el archivo *build/software/include/generated/csr.h*) *0xF0000804 / 4*.

**Figura 4.26** Uso de LiteScope

# Capítulo 5

## PLATAFORMAS DE DESARROLLO ECB

### 5.1. Introducción

En esta sección se realizará una explicación detallada del proceso de adaptación de Linux a la familia de plataformas ECB\_AT91, y ECB\_T8\_T113, este proceso es aplicable a otros dispositivos que trabajen con procesadores soportados por la distribución de Linux. Adicionalmente, se explicará de forma detallada el funcionamiento de este sistema operativo, el proceso de arranque y su puesta en marcha, así como las principales distribuciones, aplicaciones y librerías disponibles para el desarrollo de aplicaciones. Esta y otra información recolectada durante más de tres años permite que la industria y la academia desarrollen aplicaciones comerciales utilizando herramientas de diseño modernas<sup>1</sup>.

### 5.2. Herramientas abiertas para diseño de sistemas embebidos

#### 5.2.1. Herramientas de Desarrollo

En este apartado se describirán las herramientas abiertas necesarias para el desarrollo de aplicaciones software en sistemas sistemas digitales. Todas las aplicaciones mencionadas a continuación hacen parte de la cadena de herramientas GNU, que son parte de los recursos suministrados por la comunidad de software libre.

#### GNU binutils

Colección de utilidades para archivos binarios y están 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 ejecutable para determinar qué nombre de archivo y número de línea 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.
- **as** Utilidad para compilar código fuente en lenguaje ensamblador.
- **c++filt** Este programa 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 es el llamado a ld.
- **nm** Realiza un listado de símbolos de archivos tipo objeto.

---

<sup>1</sup> A lo largo de este capítulo se utilizaran cuadros de texto donde se listará código fuente en color gris, instrucciones ejecutadas en el computador personal en color verde y operaciones que se ejecutan en el sistema embebido en color amarillo

- **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 escribir 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 al menos 4 caracteres de longitud.
- **strip** Elimina todos los símbolos de un archivo tipo objeto.

## Compilador

El *GNU Compiler Collection* normalmente llamado GCC, es un grupo de compiladores de lenguajes de programación producido por el proyecto GNU. Es el compilador estándar para el software libre, de los sistemas operativos basados en Unix. Soporta los lenguajes ADA, C, C++, Fortran, Java, Objective-C, Objective-C++ para las arquitecturas Alpha, ARM, RISCV, Atmel AVR, Blackfin, H8/300, System/370, System/390, IA-32 (x86), x86-64, IA-64 i.e. the "Itanium", Motorola 68000, Motorola 88000, MIPS, PA-RISC, PDP-11, PowerPC, SuperH, SPARC, VAX, Renesas R8C/M16C/M32C y MorphoSys. Gracias a esto puede considerarse como 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 procesador (reutilización de código), lo que no es posible cuando se utiliza el lenguaje ensamblador.

## GNU Debugger

El depurador oficial de GNU (GDB) al igual que GCC, soporta múltiples lenguajes y plataformas; permite 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 es una aplicación que se ejecuta en consola de comandos, se han desarrollado varios front-ends como DDD o GDB/Insight.

## Librerías C

Es necesario contar con las librerías standard de C: stdio, stdlib, math, etc; las más utilizadas en sistemas embebidos son:

- **glibc** Es la librería C oficial del proyecto GNU; el principal inconveniente 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** Es una librería diseñada especialmente para sistemas embebidos, es mucho más pequeña que **glibc**.
- **newlib** 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.
- **diet libc** 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.

Aunque este grupo de herramientas pueden ejecutarse en los sistemas operativos más populares (Linux, Mac OS y Windows) se prefiere el uso de Linux ya que es un sistema operativo gratuito y no es necesario pagar ningún tipo de licencia, lo que reduce aún más la inversión en software.

### 5.3. Métodos de arranque

Como se vio anteriormente, la mayoría de los SoC utilizados en aplicaciones modernas no poseen memorias programables en su interior. Todo SoC debe ser programado para que pueda ejecutar una determinada tarea; este programa

debe estar almacenado en una memoria no volátil externa y debe estar en el formato requerido por el procesador. Normalmente los SoCs proporcionan varios caminos (habilitando diferentes periféricos) para hacer esto. Un programa de inicialización (*boot program*) contenido en una pequeña ROM del SoC se encarga de configurar, y revisar ciertos periféricos en búsqueda de un ejecutable válido, una vez lo encuentra, lo copia a la memoria RAM interna y lo ejecuta desde allí. No sobra mencionar que este ejecutable debe estar enlazado de tal forma que todas sus secciones se encuentren en el espacio de la memoria RAM interna.

### 5.3.1. Arranque del procesador AT91RM9200

A manera de ejemplo, tomemos el procesador de Atmel AT91RM9200, este procesador posee una memoria interna SRAM de 16 kbytes. Después del *reset* esta memoria está disponible en la posición 0x200000; después del remap esta memoria se puede acceder en la posición 0x0. Algunos fabricantes, en la etapa de producción graban en las memorias no volátiles las aplicaciones definitivas, y sueldan en la placa de circuito impreso los dispositivos programados, esto es muy conveniente cuando se trabaja con grandes cantidades ya que ahorra tiempo en el montaje de los dispositivos. El programa de inicialización del AT91RM9200 (se ejecuta si el pin BMS se encuentra en un valor lógico alto) busca una secuencia de 8 vectores de excepción válidos en la DataFlash conectada al puerto SPI, en una EEPROM conectada a la interfaz I2C o en una memoria de 8 bits conectada a la interfaz de bus externo (EBI). Estos vectores son instrucciones LDR o Bbranch, a excepción de la sexta instrucción (posición 14 a 17) que contiene información sobre el tamaño de la imagen (en bytes) a descargar y el tipo de dispositivo *DataFlash*. Si la secuencia es encontrada, el código es almacenado en la memoria SRAM interna y se realiza un *remap* (con lo que la memoria interna SRAM es accesible en la posición 0x0 ver Figura 5.1).

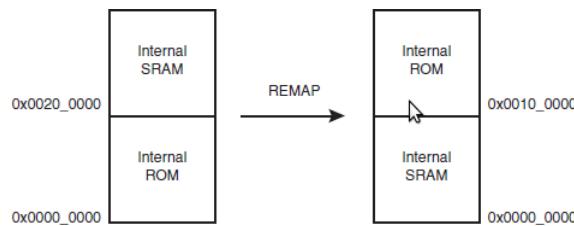


Figura 5.1 Diagrama de flujo del programa de inicialización del SoC AT91RM9200

Si no se encuentra esta secuencia de vectores, se inicializa un programa que configura el puerto serial de depuración (DBGU) y el puerto USB Device. Quedando en espera de la descarga de una aplicación a través del protocolo DFU (Digital Firmware Upgrade) por el puerto USB o con el protocolo XMODEM en el puerto serial DBGU (115200,N81). La figura 5.2 muestra el diagrama de flujo del programa de inicialización del SoC AT91RM9200.

El programa descargado a la memoria SRAM interna debe: programar una memoria no volátil (DataFlash SPI para la familia de plataformas ECB\_AT91); proporcionar un canal de comunicación que permita descargar ejecutables más grandes, e inicializar el controlador de memoria SDRAM para que almacene temporalmente el ejecutable a grabar, esto debido a que la memoria interna del AT91RM9200 es de 16kBytes. Más adelante hablaremos detalladamente de la aplicación que realiza estas funciones.

### 5.3.2. Interfaz JTAG

A mediados de los 70s, la estructura de pruebas para tarjetas de circuito impreso (PCB, Printed Circuit Boards) se basaba en el uso de la técnica “bed-of-nails”. Este método utilizaba un dispositivo que contenía una gran cantidad de puntos de prueba, que permitían el acceso a dispositivos en la tarjeta a través de puntos colocados en la capa de cobre para dicho fin. Las pruebas se realizaban en dos fases: con el circuito apagado y con el circuito funcionando. Con la aparición de los dispositivos de montaje superficial se empezó a colocar dispositivos en las dos caras de la tarjeta,

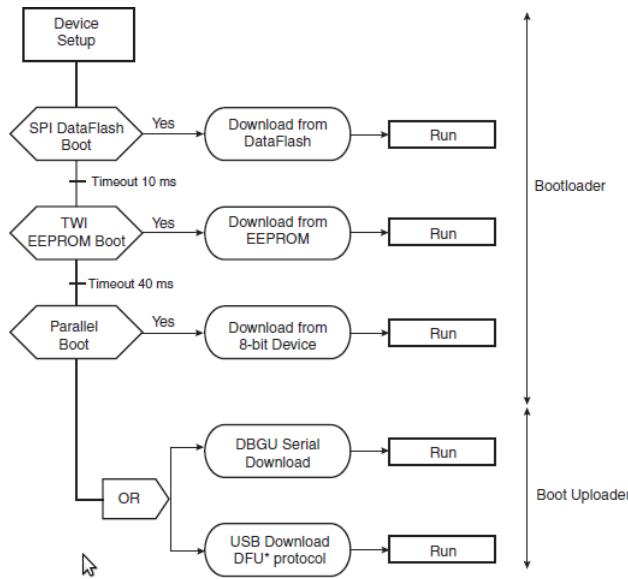


Figura 5.2 Diagrama de flujo del programa de inicialización del SoC AT91RM9200

y se redujeron de forma considerable las dimensiones de los dispositivos, disminuyendo la distancia física entre las interconexiones (0.4 - 1mm), dificultando el proceso de pruebas tradicional.

A mediados de los 80s un grupo de ingenieros de pruebas miembros de compañías europeas se reunieron para examinar el problema y buscar posibles soluciones. Este grupo se autodenominó JETAG (Joint European Test Action Group). El método de solución propuesto por ellos estaba basado en el concepto de un registro de corrimiento serial colocado alrededor de la frontera dispositivo, de aquí el nombre “Boundary Scan”. Después el grupo se asoció a compañías norteamericanas y la “E” de “European” desapareció del nombre de la organización convirtiéndose en JTAG (Join Test Action Group).

### Arquitectura BOUNDARY SCAN

A cada señal de entrada o salida se le adiciona un elemento de memoria multi-propósito llamado “Boundary Scan Cell” (BSC). Las celdas conectadas a los pines de entrada reciben el nombre de “Celdas de entrada”, y las que están conectadas a los pines de salida “Celdas de salida”. En la Figura 5.3 se muestra esta arquitectura.

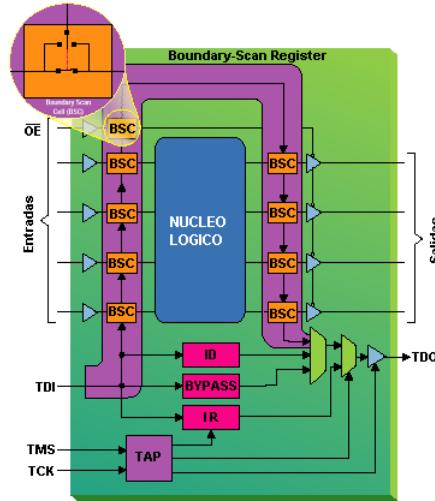
Las BSC se configuran en un registro de corrimiento de entrada y salida paralela. Una carga paralela de los registros (captura) ocasiona que los valores de las señales aplicadas a los pines del dispositivo pasen a las celdas de entrada y que opcionalmente los valores de las señales internas del dispositivo pasen a las celdas de salida. Una descarga paralela (Actualización) ocasiona que los valores presentes en las celdas de salida pasen a los pines del dispositivo, y opcionalmente los valores almacenados en las celdas de entrada pasen al interior del dispositivo.

Los datos pueden ser corridos a través del registro de corrimiento de forma serial, empezando por un pin dedicado TDI (Test Data In) y terminando en un pin de salida dedicado llamado TDO (Test Data Out). La señal de reloj se proporciona por un pin externo TCLK (Test Clock) y el modo de operación se controla por la señal TMS (Test Mode Select). Los elementos del Boundary Scan no afectan el funcionamiento del dispositivo. Y son independientes del núcleo lógico del mismo.

### Instrucciones JTAG

El Standard IEEE 1149.1 describe tres instrucciones obligatorias: Bypass, Sample/Preload, y Extest [4].

- **BYPASS** Esta instrucción permite que el chip permanezca en un modo funcional, hace que el registro de Bypass se coloque entre TDI y TDO; permitiendo la transferencia serial de datos a través del circuito integrado desde TDI



**Figura 5.3** Arquitectura Boundary Scan

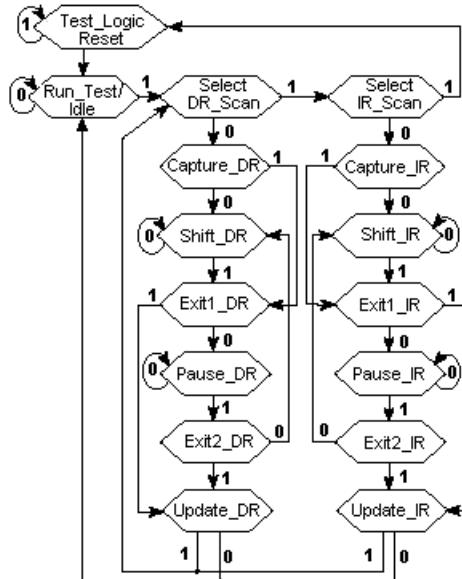
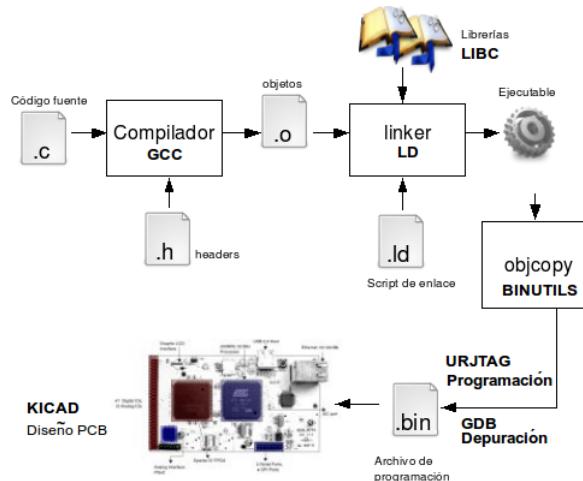
hacia TDO sin afectar la operación. La codificación en binario para esta instrucción debe ser con todos sus bits en uno.

- **SAMPLE/PRELOAD** Esta instrucción selecciona coloca el registro Boundary-Scan entre los terminales TDI y TDO. Durante esta instrucción, se puede acceder al registro Boundary-Scan y obtener una muestra de los datos de entrada y salida del chip a través de la operación *Data Scan*. Esta instrucción también se utiliza para precargar los datos de prueba en el registro Boundary-Scan, antes de ejecutar la instrucción EXTEST. La codificación de esta instrucción la define el fabricante.
- **EXTEST** Esta instrucción coloca al circuito integrado en modo de test externo (pruebas de interconexión) y conecta el registro Boundary-Scan entre TDI y TDO. Las señales que salen del circuito son cargadas en el registro boundary-scan en el flanco de bajada de TCK del estado Capture-DR; las señales de entrada al dispositivo son cargadas al registro boundary-scan durante el flanco de bajada de TCK dl estado Update-DR (ver Figura 5.4). La codificación para esta instrucción está definida con todos sus bits en cero.
- **INTEST** La instrucción INTEST (opcional) selecciona el registro boundary-scan, pero es utilizado para capturar las señales que salen del núcleo lógico del dispositivo, y para aplicar valores conocidos a las señales de entrada del núcleo. La codificación para esta señal es asignada por el diseñador.

## 5.4. Flujo de diseño software

En la figura 5.5 se ilustra la secuencia de pasos que se realizan desde la creación de un archivo de texto que posee el código fuente de una aplicación hasta su implementación en la tarjeta de desarrollo. Los pasos necesarios para generar un ejecutable para un sistema embebido son:

1. **Escritura del código fuente:** Creación del código fuente en cualquier editor de archivos de texto.
2. **Compilación:** Utilizando GCC se compila el código fuente; el compilador busca en los encabezados (*headers* .h) de las librerías la definición de una determinada función, pero no busca el segmento de código donde está implementada. (por ejemplo el *printf* en el archivo *stdio.h*). Como resultado de este paso se obtiene un archivo binario tipo objeto que incluye el código necesario para realizar la funcionalidad deseada utilizando en conjunto de instrucciones del procesador.
3. **Enlazado:** En esta etapa se realizan dos tareas:

**Figura 5.4** Arquitectura Boundary Scan**Figura 5.5** Flujo de diseño SW utilizando la cadena de herramientas GNU

- a) Se enlazan los archivos tipo objeto del proyecto junto con las librerías, si una determinada función no es definida por ninguna de las librerías pasadas como parámetro al enlazador (*linker*), este generará un error y no se generará el ejecutable.
  - b) Se definen las posiciones físicas de las secciones del ejecutable (tipo ELF), esto se realiza a través de un *script de enlace* que define de forma explícita su localización.
4. **Extracción del archivo de programación** En algunas aplicaciones es necesario extraer únicamente las secciones que residen en los medios de almacenamiento no volátil y eliminar las demás secciones del ejecutable. Esto se realiza con la herramienta *objcopy*, la cual, permite generar archivos en la mayoría de los formatos soportados por los programadores de memorias y procesadores, como por ejemplo S19 e Intel Hex. Adicionalmente se puede generar un archivo binario que contiene las instrucciones en lenguaje del procesador, y pueden ser descargadas directamente a la memoria de la plataforma.
5. **Descarga del programa.** Dependiendo de la plataforma, existen varios métodos para descargar el archivo de programación:

- 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 o una interfaz de red a una memoria no volátil externa.
  - 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 posición de memoria determinada.
6. **Depuración** Una vez se descarga la aplicación a la plataforma es necesario someterla a una serie de pruebas, para verificar 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.

## Make

Como vimos anteriormente, 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 ejecutar una serie de comandos cada vez que se realiza un cambio en el código fuente, lo que resulta poco práctico durante la etapa de desarrollo. Para realizar este proceso de forma automática, se creó la herramienta *make*, la cual recibe como entrada un archivo con una serie de instrucciones que normalmente lleva el nombre de *Makefile* o *makefile*. La herramienta *make* ejecuta los comandos necesarios para realizar la compilación, depuración, o programación, indicados en el archivo *Makefile* o *makefile*. Un ejemplo de este tipo de archivo se muestra a continuación:

```

48 SHELL = /bin/sh
49
50 basetoolsdir = /home/cain/Embedded/ARM/iMX233/toolchain/arm-2008q3
51 bindir = ${basetoolsdir}/bin
52 libdir = ${basetoolsdir}/lib/gcc/arm-none-linux-gnueabi/4.3.2
53
54 CC = arm-none-linux-gnueabi-gcc
55 AS = arm-none-linux-gnueabi-as
56 LD = arm-none-linux-gnueabi-ld
57 OBJCOPY = arm-none-linux-gnueabi-objcopy
58
59 CFLAGS = -mcpu=arm920t -I . -Wall
60 LDFLAGS =-L${libdir} -lgcc
61
62 OBJS = \
63     main.o \
64     debug.io.o \
65     at91rm9200_lowlevel.o \
66     p_string.o
67
68 ASFILES = arm_init.o
69
70 LIBS=${libdir}/libgcc.a
71
72 all: hello_world
73
74 hello_world: ${OBJS} ${ASFILES}
75     ${LD} -e 0 -o hello_world.elf -T linker.cfg ${ASFILES} ${OBJS} ${LDFLAGS}
76     ${OBJCOPY} -O binary hello_world.elf hello_world.bin
77
78 clean:
79     rm -f *.o *~ hello_world.*
80
81 .c.o:
82     $(CC) $(CFLAGS) -c $< -o $@
83 .S.o:
84     $(CC) $(AFLAGS) -c $< -o $@

```

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: los compiladores de C (CC), de assembler (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 lenguaje ensamblador, para este ejemplo *arm\_init.o*. Las líneas 12 y 13 definen dos variables especiales que el compilador de C (CFLAGS) y el enlazador (LDFLAGS) utilizarán como parámetros .

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

*make* ejecutará:

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

Observemos los comandos asociados a la etiqueta *hello\_world*: En la misma línea aparecen *\$OBJS \$ASFILES*, estos parámetros reciben el nombre de dependencias y le indican a la herramienta *make* que antes de ejecutar los coman-

dos asociados a esta etiqueta, debe realizar las acciones necesarias para generar las dependencias; es decir: *main.o*, *debug.io.o*, *at91rm9200\_lowlevel.o*, *p\_string.o* (\$OBJS) y *arm.init.o* (\$ASFILES).

En las líneas 34 y 35 se le indica a la herramienta *make* la regla para generar un archivo *.o* a partir de un archivo *.c*; *make* aplicará esta regla a cada uno de los elementos que conforman la variable \$OBJS. \$< Es el nombre del primer pre-requisito (*.c*) y \$@ es el nombre del destino (*.o*).

En las líneas 36 y 37 se le indica a la herramienta *make* la regla para generar un archivo *.o* a partir de un archivo *.s*; *make* aplicará esta regla a cada uno de los elementos que conforman la variable \$OBJS.

En la línea 28 se realiza el proceso de enlazado; al *linker* (ld) se le pasan los parámetros:

- **-e 0:** Punto de entrada , utilice 0 como símbolo para el inicio de ejecución.
- **-o hello\_world.elf:** Nombre del archivo de salida *hello\_world*
- **-T linker.cfg:** Utilice el archivo de enlace *linker.cfg* para definir las direcciones de las secciones del ejecutable.
- **\$ASFILES \$OBJS \$LDFLAGS:** Lista de objetos y libreras para crear el ejecutable.

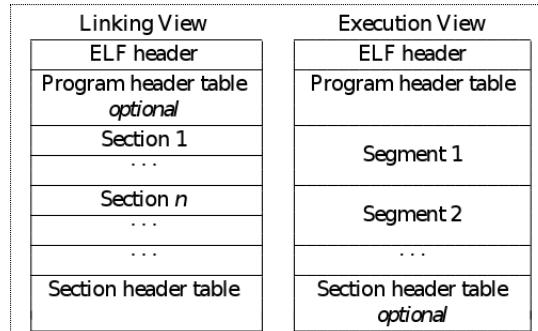
En la línea 29 se utiliza la herramienta *objcopy* para generar un archivo binario (-O binary) con la información necesaria para cargar en una memoria no volátil. Esto se explicará con mayor detalle más adelante.

Al ejecutar el comando: *make hello\_world*, *make* realizará las siguientes operaciones:

```
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c main.c -o main.o
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c debug.io.c -o debug.io.o
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c at91rm9200_lowlevel.c
-o at91rm9200_lowlevel.o
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c p_string.c -o p_string.o
arm-none-linux-gnueabi-as -o arm.init.o arm.init.s
arm-none-linux-gnueabi-ld -e 0 -o hello_world.elf -T linker.cfg arm.init.o main.o
debug.io.o at91rm9200_lowlevel.o p_string.o -lgcc
arm-none-linux-gnueabi-objcopy -O binary hello_world.elf hello_world.bin
```

## El formato ELF

El formato ELF (*Executable and Linkable Format*) Es un estándar para objetos, libreras y ejecutables y es el formato que generan las herramientas GNU. Como puede verse en la figura 5.6 un ejecutable ELF está compuesto por las 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 [5].



**Figura 5.6** Formato ELF

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

- **.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 enlace dinámico
- **.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 .rodata1** Datos de solo-lectura (ROM)
- **.shstrtab** Nombres de secciones.
- **.syntab** Tabla de símbolos.
- **.text** Instrucciones ejecutables (ROM)

Para aclarar el contenido de cada una de estas secciones, consideremos la siguiente 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-linux-gnueabi-gcc -c hello.c*  
Examinemos que tipo de secciones tiene este ejecutable *arm-none-linux-gnueabi-readelf -S hello.o*

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00	A	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]	.syntab	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 ejecutando:

*arm-none-linux-gnueabi-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: e51b2000  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-linux-gnueabi-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 información 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 inicializadas. En Linux todas las variables no inicializadas, se inicializaran en cero:

*arm-none-linux-gnueabi-objdump -d -j .bss hello*

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

La sección *.rodata* contiene los datos que no cambian durante la ejecución del programa, es decir, los 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 47 43 43 3a 20 28 43 6f 64 65 53 |.....GCC: (CodeS|
```

Observamos que en el archivo se almacena la cadena de caracteres *Printing %d* n la cual no se modifica durante la ejecución del programa.

## Linker Script

Como vimos anteriormente, el *linker* es el encargado de agrupar todos los archivos objeto *.o*, y las librerías necesarias para crear el ejecutable, este *linker* permite definir donde serán ubicados los diferentes segmentos del archivo ELF, por medio de un archivo de enlace *linker script*. De esta forma podemos ajustar el ejecutable a plataformas con diferentes configuraciones de memoria. Esto brinda un grado mayor de flexibilidad de la cadena de herramientas GNU. Cuando se dispone de un sistema operativo como Linux no es necesario definir este archivo para los ejecutables, ya que el sistema operativo se encarga de guardar las secciones en el lugar indicado; 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) */
        .text = .;           /* define a global symbol _text 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)
    .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)
} >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 continuación 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.

En algunos SoCs no se dispone de una memoria no volátil, por lo que es necesario que la aplicación sea cargada por completo en la RAM. Algunos desarrolladores prefieren almacenar y ejecutar sus aplicaciones en las memorias volátiles durante la etapa de desarrollo, debido a que la programación de las memorias no volátiles toman mucho más tiempo. Obviamente una vez finalizada la etapa de desarrollo las aplicaciones deben ser almacenadas en memorias no volátiles.

## 5.5. Dispositivos semiconductores

En esta sección se realizará una breve descripción de los dispositivos semiconductores más utilizados para la implementación de dispositivos digitales.

### 5.5.1. SoC

La Figura 5.7 muestra la arquitectura de un SoC actual, específicamente del T113 de Allwinner. En este diagrama podemos observar que posee dos núcleos ARM Xortex-A7 de 1GHz 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 posee una memoria DDR3 interna de 128M, lo que permite diseñar un sistema completo que ejecuta el sistema operativo Linux con solo 1 chip.

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. A continuación se realizará una descripción de los diferentes periféricos que se encuentran disponibles en este SoC, indicando los que fueron utilizados en la plataforma de desarrollo.

### 5.5.2. Memorias Volátiles

Como se mencionó anteriormente existen secciones del 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 acceso y al ilimitado número de ciclos de lectura/escritura.

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 5.8).

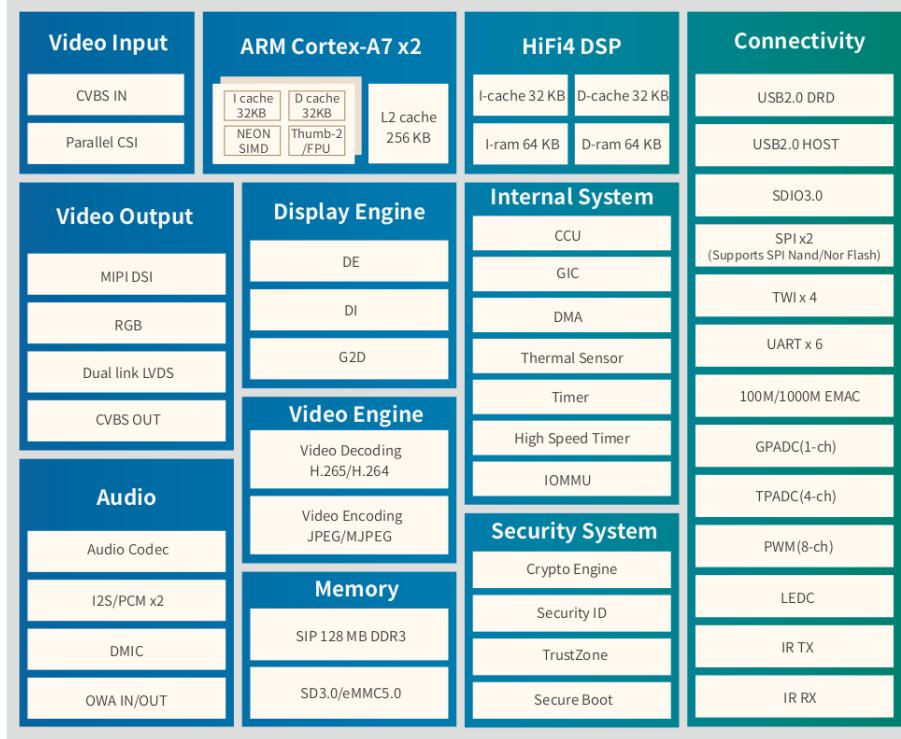


Figura 5.7 Diagrama de Bloques del SoC T113 fuente: Hoja de Especificaciones Allwinner T113, Allwinner

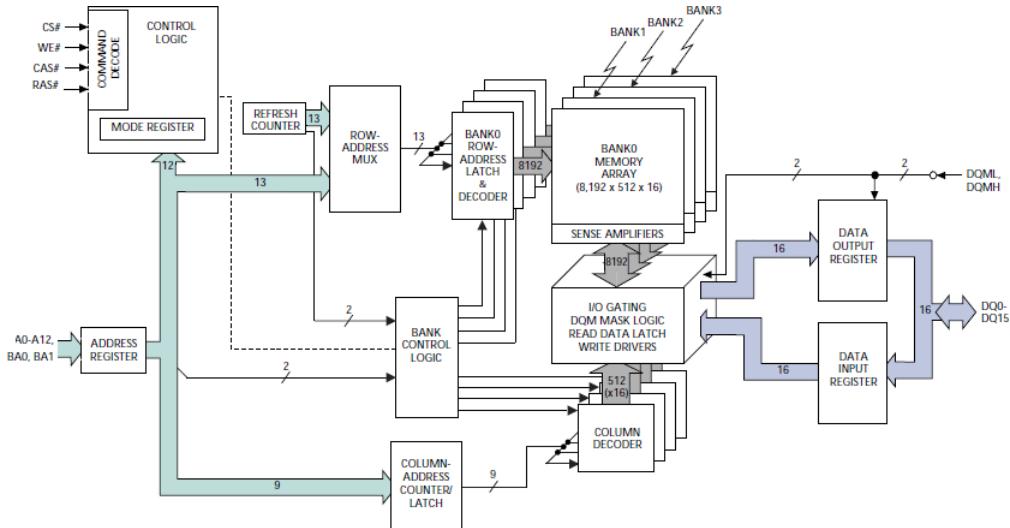


Figura 5.8 Diagrama de Bloques de una memoria SDRAM fuente: Hoja de Especificaciones MT48LC16M16, Micron Technology

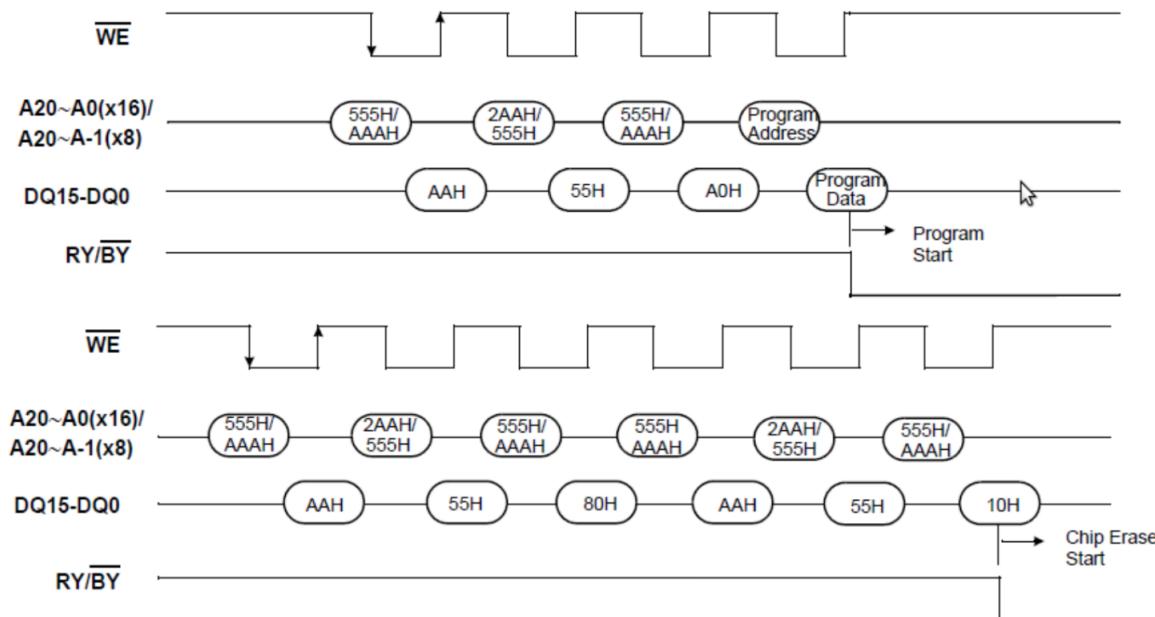
Un ejemplo simplificado de una operación de lectura es el siguiente: Una posición de memoria se determina colocando la dirección de la fila y la de la columna en las líneas de dirección de fila y columna respectivamente, un tiempo después el dato almacenado aparecerá en el bus de datos. El procesador 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). Una celda de memoria SDRAM está 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. El periférico que controla la SDRAM está encargado de garantizar los ciclos de refresco de acuerdo con los requerimientos de la SDRAM [6].

### 5.5.3. Memorias No Volátiles

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

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 datos pueden ser cambiados de 0 a 1 por 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 permite reducir 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 5.9) no es necesario contar con un periférico especializado para su manejo.



**Figura 5.9** Ciclos de escritura y borrado de una memoria flash NOR

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. 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 NOR, por esta razón este tipo de memorias son utilizadas en casi todos los dispositivos de almacenamiento modernos como las memorias SD y USB, las que 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 encarguen de almacenar tablas de mapeo para manejar los bloques defectuosos; o puede hacerse un chequeo en cada inicialización del sistema de toda la memoria para actualizar esta lista de sectores defectuosos. El algoritmo de ECC (Error-Correcting Code) debe ser capaz de corregir errores tan pequeños como un bit de cada 2048 bits, hasta 22 bits de cada 2048. Este algoritmo es capaz de detectar bloques defectuosos en la fase de programación, 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 defectos para almacenar la información.

La tabla 5.1 resume las principales características de los diferentes tipos de memoria flash.

	<b>SLC NAND</b>	<b>MLC NAND</b>	<b>MLC NOR</b>
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 5.1** Cuadro de comparación de las memorias flash NAND y NOR

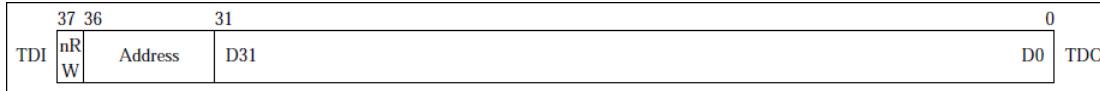
Adicionalmente, se encuentran disponibles las memorias *DATAFLASH*, estos dispositivos son básicamente una memoria flash tipo NOR con una interfaz SPI, permite una velocidad de lectura de hasta 66MHz utilizando solamente 4 pines para la comunicación con el procesador.

## 5.6. Depuración del core ARM [1]

Todos los núcleos ARM7 y ARM9 poseen soporte de depuración en modo *halt*, lo que permite detener por completo el núcleo. Durante este estado es posible modificar y capturar las señales del núcleo, permitiendo cambiar y examinar el estado del sistema. En este estado, la fuente de reloj del núcleo es el reloj de depuración (DCLK) que es generado por la lógica de depuración. Los núcleos ARM7 y ARM9 implementan un controlador compatible con JTAG, con dos cadenas boundary-scan alrededor de las señales del núcleo; una con las señales del núcleo, para pruebas del dispositivo y la otra es un sub-set de la primera con señales importantes para la depuración. La Figura 5.10 muestra el orden de las señales en las cadenas para los núcleos ARM7TDMI y ARM9TDMI. Para propósitos de depuración es suficiente la cadena 1. Esta cadena puede ser utilizada en modo *INTEST*, el que permite capturar las señales del núcleo y aplicar vectores de prueba al mismo, o en modo *EXTTEST*, permitiendo la salida y entrada de información hacia y desde exterior del núcleo respectivamente.

Las señales *ID[0:31]* del ARM7TDMI están conectadas al bus de datos del núcleo y se utiliza para capturar instrucciones o lectura/escritura de información; la señal *BREAKPT* se utiliza para indicar que la instrucción debe ejecutarse a la velocidad del sistema, esto es, utilizando el reloj MCLK en lugar de DCLK.

Las señales *ID[0:31]* del ARM9TDMI están conectadas al bus de instrucciones y se utilizan para capturar instrucciones, las señales *DD[31:0]* están conectadas al bus de datos bi-direccional y se utilizan para leer o escribir información. Los ARM7 y ARM9 poseen un módulo *ICE* (In Circuit Emulator) que reemplaza el microcontrolador con una variación que posee facilidades para la depuración hardware. El emulador es conectado a un computador que ejecuta el software de depuración. Esto permite realizar depuración activa y pasiva, dando un punto de vista no intrusivo del flujo del programa. La Figura 5.11 muestra la cadena scan ICE, que es la misma para los núcleos ARM7 y ARM9, está

**Figura 5.10** Cadena Boundary Scan 1**Figura 5.11** Cadena Boundary Scan 2 (Embedded ICE)

formada por 32 bits de datos, 5 bits de direcciones y una bandera para diferenciar entre lectura (nRW bajo) y escritura. Se puede acceder a las características ICE a través de registros, cuya dirección es colocada en el bus de direcciones.

### 5.6.1. Proyecto OpenOCD

El proyecto *OpenOCD* (Open On-Chip Debugger<sup>2</sup>) permite la programación de la memoria flash interna de algunos procesadores ARM7TDMI, y la depuración de procesadores ARM7 y ARM9 utilizando el módulo ICE. Este proyecto se ha convertido en el más popular dentro del grupo de desarrolladores de sistemas Embebidos. En [1] se puede encontrar el funcionamiento interno de esta herramienta. La figura 5.12 muestra el principio de funcionamiento de la herramienta *OpenOCD*.

La herramienta de depuración *GDB* utiliza la información suministrada por las secciones *debug\_xxx* del ejecutable *ELF*. Esta información es generada al agregar **-g** a los parámetros pasados al compilador (CFLAGS); proporcionando la información necesaria para relacionar símbolos y código lo que permite su depuración. A continuación se muestra una parte de la salida del comando *arm-none-linux-gnueabi-objdump -S -j.debug\_info hello* el cual muestra el contenido de la sección *debug\_info*:

```
int main(void) {
 160: 0001e502      andeq   lr, r1, r2, ls1 #10
 164: 24230200      strtcs r0, [r3], #-512
 168: 00029f08      andeq   r9, r2, r8, ls1 #30

  char      l_char;
  DebugPrint("\n\rEntry :_main\n\r");
 16c: 025e0200      subseq  r0, lr, #0      ; 0x0
 170: 00000025      andeq   r0, r0, r5, ls1 #32
```

Con esta información, *GDB* utiliza el protocolo serial (Remote Serial Protocol - RSP) para ejecutar una operación determinada; estas instrucciones son enviadas a un puerto TCP (2000 en este ejemplo). Por ejemplo, para leer dos bytes desde la posición de memoria *0x4015bc*, *GDB* enviará el paquete: **\$m4015bc,2#5a** por el puerto 2000; *m* indica que se debe leer la memoria, todo paquete *RSP* comienza con el signo \$ y termina con el signo # seguido por dos bytes que representan el *checksum*.

<sup>2</sup> <http://openocd.berlios.de/web/>

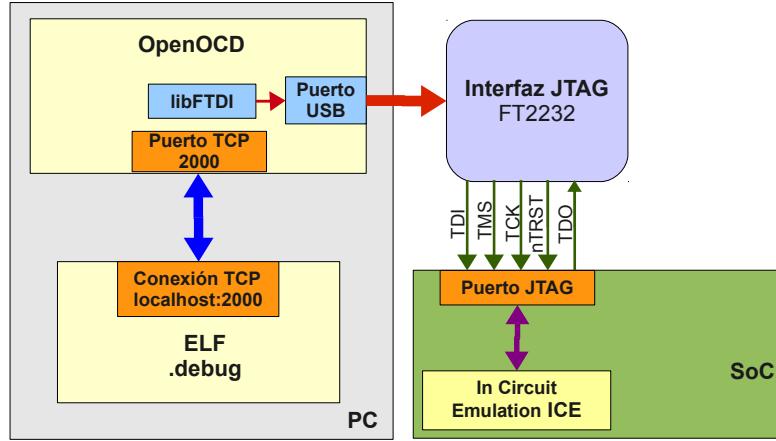


Figura 5.12 Principio de funcionamiento del proyecto OpenOCD

Estos paquetes enviados por *GDB* son recibidos por un programa que se encuentra en ejecución, esperando por una comunicación RSP en el mismo puerto. *OpenOCD* establece la comunicación con *GDB* y convierte los comandos *RSP* a una serie de instrucciones JTAG que implementan la operación requerida. Estas instrucciones JTAG son enviadas a una interfaz especial basada en un dispositivo de la compañía *FTDI* que permite controlar el puerto JTAG a través de una interfaz USB. *OpenOCD* utiliza la librería abierta *libftdi* para controlar esta interfaz.

### 5.6.2. Programación de memorias Flash

Algunos SoC no poseen programas de inicialización que permitan la descarga de aplicaciones ya sea a las memorias externas o a la interna. *OpenOCD* puede ser utilizado en estos casos para cargar las aplicaciones en la RAM interna; sin embargo, existen procesadores en los que no se conocen las especificaciones del ICE y en otros este módulo no existe. En estos casos se utiliza la interfaz JTAG en modo *EXTEST* para controlar directamente los pines a los que se encuentran conectadas las memorias a los SoCs. El proyecto UrJTAG,<sup>3</sup> y el proyecto *OpenOCD* permiten la programación de diversas memorias flash, utilizando los pines del dispositivo. Adicionalmente, *UrJTAG* permite la creación de diferentes interfaces para conectarse con las memorias, estas interfaces reciben el nombre de buses y pueden ser creadas e incluidas en el código original de forma fácil.

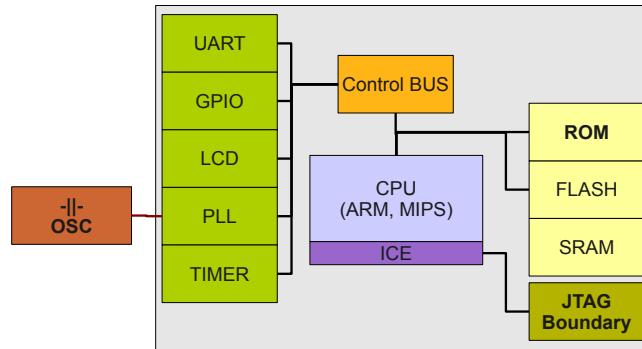
## 5.7. Arquitectura: SoC, Memorias, periféricos

Los SoCs comerciales se pueden dividir en dos grandes grupos dependiendo de la existencia o no de memoria no volátil para el almacenamiento del programa (memoria de instrucciones) dentro del SoC. Los que poseen memoria no volátil (hasta 512 Kbytes) normalmente incorporan una memoria RAM (hasta 32 kbytes) junto con una serie de periféricos (timers, I2C, SPI, USARTs, ADCs, Watchdog, USB device, canales para acceso directo a memoria - DMA); están diseñados para no utilizar componentes externos; normalmente este tipo de dispositivos utilizan procesadores que no tienen unidad de manejo de memoria<sup>4</sup> (MMU) como la familia ARM7, cuyas velocidades de ejecución varían entre los 50 y 70MHz. En la figura 5.13 se muestra la arquitectura típica de un sistema basado en estos dispositivos.

Los procesadores que no poseen memoria no volátil interna se dividen en dos grupos: los que poseen o no unidad de manejo de memoria; en ambos casos, se cuenta con una memoria RAM (del orden de cientos de Kbytes) y adicionalmente a los periféricos mencionados anteriormente se suministran controladores para USB host, puertos SSI,

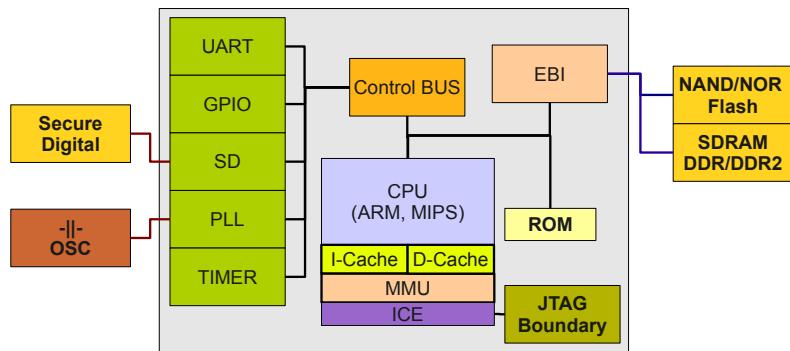
<sup>3</sup> <http://urjtag.sourceforge.net/>

<sup>4</sup> La MMU permite el manejo de memoria, dentro de sus funciones se encuentra el traslado de la memoria física a virtual, protección de la memoria, control de cache, control de buses



**Figura 5.13** Arquitectura típica de un sistema embebido que utiliza SoC con memoria volátil interna

controlador de LCD, codecs de audio, controlador de touch screen; debido a la ausencia de memoria no volátil interna, estos dispositivos poseen periféricos dedicados al manejo de memorias no volátiles NAND flash, NOR flash, SPI, I2C y SD; y memorias volátiles SDRAM y DDR; su velocidad de operación varía entre los 75MHz y 800MHz. En la figura 5.14 se muestra la arquitectura típica de un sistema basado en estos procesadores.



**Figura 5.14** Arquitectura típica de un sistema embebido que utiliza SoC sin memoria volátil interna

Debido a la falta de memoria volátil, las aplicaciones de este tipo de dispositivos requieren una memoria externa de este tipo, en la actualidad las más populares son las memorias NAND flash, NOR flash, SPI, EEPROM y SD. Normalmente, este tipo de procesadores son utilizados en aplicaciones que utilizan sistemas operativos, como veremos más adelante, para que ciertos sistemas operativos (Linux, Windows CE) puedan ejecutarse se requiere una mínima cantidad de memoria RAM (del orden de los Mbytes), por esta razón es necesario incluir una memoria RAM externa, en la actualidad las más utilizadas son las SDRAM, DDR y DDR2.

Como conclusión, podemos decir que en el mercado existen diferentes arquitecturas de SoCs que nos permiten realizar proyectos con diferentes grados de complejidad y que se ajustan a las opciones más utilizadas por los desarrolladores; la opción más económica es la utilización de un SoC que incluya las memorias no volátiles y RAM internamente; sin embargo, hasta el momento no existen dispositivos con grandes capacidades de memoria Flash y RAM internas, por lo que no es recomendado su uso en ciertas aplicaciones. Utilizar un SoC que no integren las memorias no volátiles proporciona una mayor flexibilidad, ya que estos dispositivos proporcionan periféricos que pueden controlar varios tipos de memorias, y se puede elegir la más económica, algo similar ocurre con la memoria RAM; sin embargo, el costo total de las memorias externas, SoC y área de circuito impreso es mayor que en el caso anterior.

Aunque estos procesadores operan a velocidades entre los 75 y 800 MHz, no todos los componentes del SoC operan a esta frecuencia, el componente externo que requiere la mayor velocidad de operación es la memoria RAM y puede estar entre los 50 y 130 MHz, los demás periféricos funcionan a frecuencias del orden de las decenas de MHz o KHz; por esta razón estos SoC suministran un circuito PLL que permite generar la frecuencia de operación a partir de cristales de frecuencias del orden de las decenas de MHz, lo que facilita el diseño de la placa de circuito impreso.

Cada periférico requiere una conexión específica con el dispositivo que controla, los SoC modernos incluyen la mayor parte del circuito internamente con el objetivo de minimizar las conexiones y dispositivos adicionales. Existen tendencias de los fabricantes a agrupar periféricos teniendo en mente dos aplicaciones: Multimedia, e industriales; para aplicaciones multimedia se proporciona controladores de LCDs, mouse, teclado, touch screen, CODECs de audio, control de potencia, relojes de tiempo real, control de carga de baterías entre otros; para aplicaciones industriales se proporcionan controladores de red cableada, puertos CAN, I2C, SPI.

### 5.7.1. Programación

Como se mencionó anteriormente, para este estudio se utilizarán herramientas abiertas para la creación de aplicaciones. La ventaja de utilizar estas herramientas (adicional a la económica) es el soporte a diferentes procesadores (24 diferentes CPUs, incluyendo micro-controladores de 8 bits), lo que permite la fácil migración entre CPUs; adicionalmente, su alto grado de configurabilidad permite el cambio de disposición de las memorias volátiles y no volátiles de forma fácil (a través del script de enlazado). El proceso de generación de el archivo binario que debe ser grabado en la memoria no volátil de la plataforma puede ser realizado en su totalidad por la cadena de herramientas GNU.

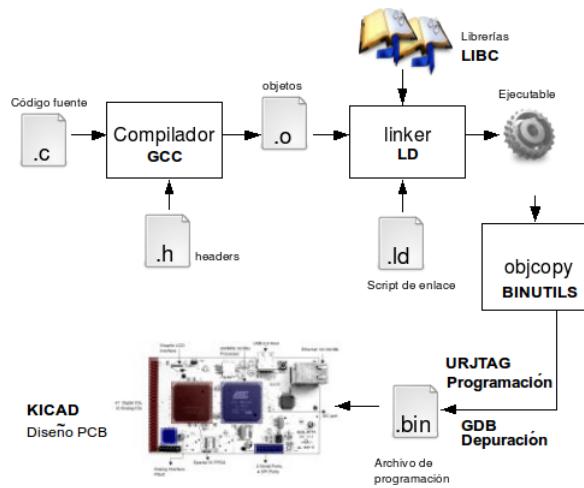
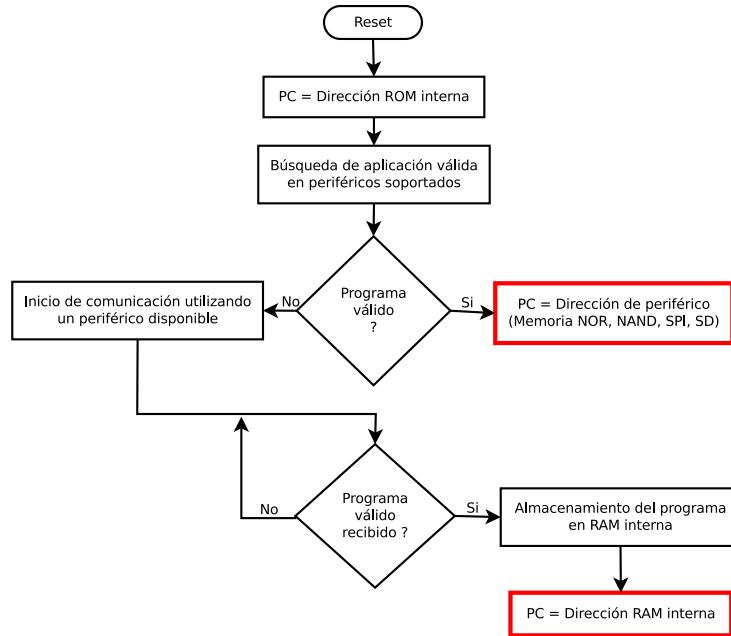


Figura 5.15 Flujo de diseño software para creación de aplicaciones.

Los SoC poseen la capacidad de *iniciar* desde diferentes dispositivos; cuando se activa la señal de *reset* a un SoC que no posee memoria volátil interna, el primer programa en ejecutarse es el que reside en una memoria ROM interna, este programa revisa varios periféricos en búsqueda de un programa válido; los periféricos soportados varían según el fabricante, pero por lo general siempre soportan el uso de memorias NOR Flash (paralelas) y en SoCs más recientes memorias NAND Flash, SPI, o SD; sin embargo, la mayoría de SoC soportan memorias que se encuentran soldadas en la placa de circuito impreso, lo que hace necesario buscar métodos de programación de estas memorias que no implique desoldarlas o el uso de costosos conectores. En la mayoría de los SoC cuando el programa residente en la ROM no encuentra ninguna aplicación válida en los periféricos soportados, establece una comunicación por uno de sus puertos seriales o USB y queda en espera del envío de un programa válido, el programa enviado es almacenado en la memoria RAM interna, y una vez finaliza su descarga se ejecuta desde la RAM interna. La figura 5.16 muestra este proceso.

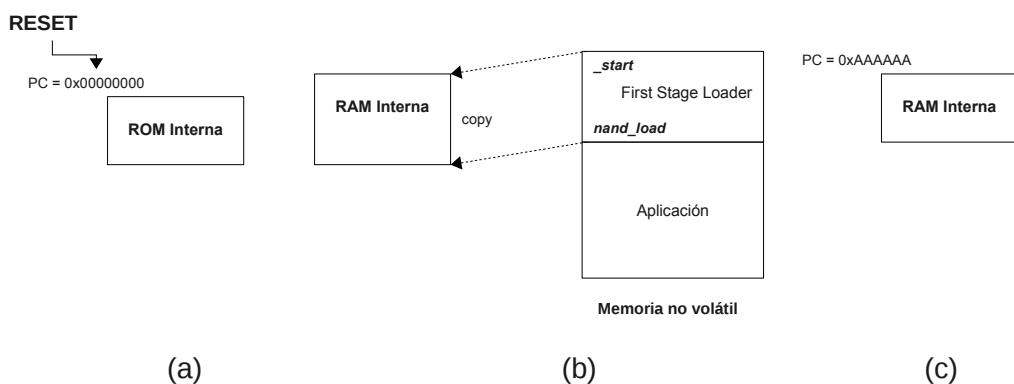
Debido a que la RAM interna normalmente es pequeña (del orden de decenas de Kbytes), no es posible cargar aplicaciones muy grandes en ella, por lo que es necesario realizar el proceso de programación en varias etapas: en la primera etapa se carga una aplicación (*first - stage bootloader*) que se encarga de configurar el procesador (pila, frecuencia de operación), configurar la memoria RAM externa y habilitar un canal de comunicación para descarga de aplicaciones, de esta forma, es posible almacenar aplicaciones tan extensas como la capacidad de la memoria RAM externa (del orden de MBytes). En la segunda etapa se descarga una aplicación a la memoria externa que tiene la capacidad de



**Figura 5.16** Inicialización de un SoC cuando las memorias no volátiles no están programadas.

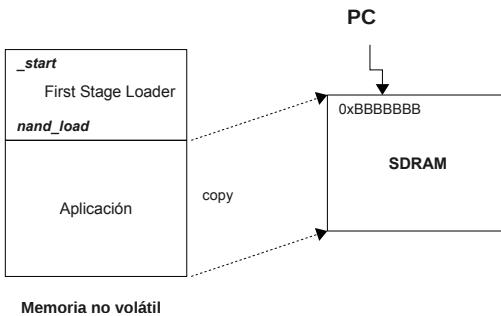
programar las memorias no volátiles externas con la información proveniente de los diferentes periféricos de comunicación del SoC (como puerto serial, memoria SD, USB), este segundo programa recibe el nombre de *bootloader* y se auto-almacena en las primeras posiciones de la memoria no volátil, de tal forma que sea ejecutado después de la activación de la señal de reset y de la búsqueda que realiza el programa interno de la ROM.

Una vez programada la memoria no volátil con una aplicación válida, los SoCs realizan una serie de pasos para ejecutar las aplicaciones almacenadas en ella, esto debido a la poca capacidad de la memoria RAM interna. Como se dijo anteriormente, una vez se activa la señal de *reset* se ejecuta un programa contenido en la memoria ROM interna del SoC (figura 5.17 (a)), esta aplicación configura un periférico que permite la comunicación con los dispositivos de almacenamiento masivo externos, y además copia una determinada cantidad de información desde la memoria no volátil externa a la memoria RAM interna (figura 5.17 (b)), esto se hace porque el programa en la ROM no conoce la configuración de la plataforma y esta puede cambiar según la aplicación; después de esto ejecuta la aplicación copiada a la memoria RAM interna colocando en el contador de programa (PC) el valor correspondiente a la memoria RAM interna (figura 5.17 (c)).



**Figura 5.17** Inicialización de un SoC cuando la memoria no volátil está programada, parte 1.

Este programa (*loader*) está encargado de: configurar la memoria RAM externa (su capacidad varía dependiendo de la aplicación) y de copiar la aplicación propiamente dicha desde la memoria no volátil a la memoria RAM externa, (con lo que es posible cargar aplicaciones de mayor tamaño que la memoria RAM interna); finalmente, el *loader* ejecuta la aplicación almacenada en la memoria RAM haciendo que el contador de programa (PC) sea igual a la dirección donde se almaceno esta aplicación (ver figura 5.18)



**Figura 5.18** Inicialización de un SoC cuando la memoria no volátil está programada, parte 2.

### 5.7.2. Programación utilizando el puerto JTAG

Algunos SoC no suministran un camino para la programación de la memoria RAM interna, para estos casos, se puede utilizar un periférico que la mayoría de los dispositivos proporciona: el puerto JTAG (creado inicialmente como un mecanismo para realizar pruebas en las tarjetas de circuito impreso para verificar la correcta conexión entre componentes, y verificar el correcto funcionamiento de los circuitos integrados) el cual, está formado por un registro de desplazamiento que controla el paso de información desde y hacia cada uno de los pines del circuito integrado, permitiendo realizar varias operaciones. Con el paso del tiempo, se han adicionado funcionalidades a este protocolo y una de ellas es el control de circuitos especializados dentro de los SoCs para realizar emulación en circuito (ICE), suministrando un canal para la programación de la memoria RAM interna.

Algunos SoC antiguos no poseen una unidad de emulación en circuito por lo que no es posible acceder a la memoria RAM interna, en estos casos es posible utilizar el protocolo JTAG para controlar directamente los pines del SoC conectados a las memorias no volátiles y ejecutar los protocolos de programación de las mismas; debido a que es necesario programar todos los registros de la cadena Boundary Scan, el tiempo de programación suele ser más largo que cuando se utiliza el ICE.

## 5.8. El sistema Operativo Linux

*Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones.*

Con este correo enviado al foro de discusión comp.os.minix, Linus Torvalds, un estudiante de la Universidad de Helsinki en Finlandia introduce Linux el 25 de Agosto de 1991. A principios de los 90, el sistema operativo Unix (desarrollado en *The Bell Labs* a principios de los 60s), tenía una sólida posición en el mercado de servidores, y estaba muy bien posicionado en las Universidades, por lo que un gran número de estudiantes trabajaban a diario con él y muchos de ellos deseaban poder utilizarlo en sus computadores personales; sin embargo, Unix era un producto comercial muy caro y solo podía ejecutarse en costosos servidores. La figura 5.19 muestra el desarrollo previo a la creación de la primera versión de Linux hasta el día de hoy.

Una de las características más importantes de Linux es que desde su creación, fue pensado como un sistema operativo gratuito y de libre distribución. Esta característica ha permitido que programadores a lo largo del mundo puedan

manipular el código fuente para eliminar errores y para aumentar sus capacidades. Sin embargo, el crédito de las distribuciones que conocemos hoy (Debian, Ubuntu, Suse, etc) no solo se debe a Torvalds, ya que Linux es solo el kernel del sistema operativo. En 1983, Richard Stallman funda el proyecto GNU, el cual proporciona una parte esencial de los sistemas Linux. A principios de los 90s, GNU había producido una serie de herramientas como librerías, compiladores, editores de texto, Shells, etc. Estas herramientas fueron utilizadas por Torvalds para escribir el elemento que le hacía falta al proyecto GNU para completar su sistema operativo: el kernel.

Desde el lanzamiento de la primera versión de Linux, cientos, miles, cientos de miles y millones de programadores se han puesto en la tarea de convertirlo en un sistema operativo robusto, amigable y actualizado; tan pronto como se desarrolla una nueva pieza de hardware existen cientos de programadores trabajando en crear el soporte para Linux.

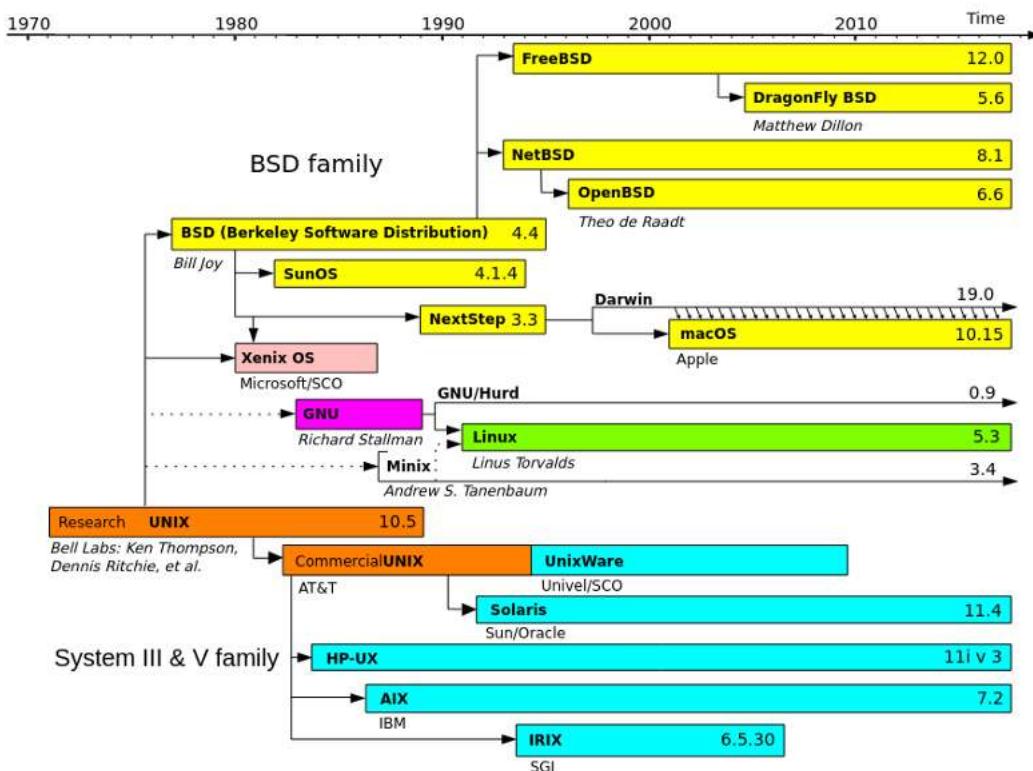


Figura 5.19 Linux: Historia Fuente Wikipedia.

## Porqué Linux

Existen varias motivaciones para escoger Linux frente a un sistema operativo tradicional para sistemas embebidos [7]:

- **Calidad Y confiabilidad del código:** Aunque estas medidas son subjetivas, miden el nivel de confianza en el código, que compromete software como el kernel y las aplicaciones proporcionadas por las distribuciones.
- **Disponibilidad de Código:** Todo el código fuente de las aplicaciones, del sistema operativo y de las herramientas de compilación se encuentran disponibles sin ninguna restricción. Existen varios tipos de licencias, siendo las más populares la GNU General Public License (GPL) y la BSD (Berkeley Software Distribution), esta última permite la distribución de binarios sin el código fuente.
- **Soporte de hardware:** Linux soporta una gran variedad de dispositivos y plataformas, a pesar de que muchos fabricantes no proporcionan soporte para Linux, la comunidad trabaja arduamente en incluir el nuevo hardware en

las nuevas distribuciones de Linux. Linux en la actualidad se puede ejecutar en docenas de diferentes arquitecturas hardware, lo cual lo convierte en el Sistema Operativo más portable.

- **Protocolos de comunicación y estándares de software:** Linux proporciona muchos protocolos de comunicación, lo que permite su fácil integración a plataformas de trabajo existentes.
- **Disponibilidad de herramientas:** La variedad de herramientas disponibles para Linux lo hacen muy versátil. Existen grandes comunidades como SourceForge<sup>5</sup> o Freshmeat<sup>6</sup> que permiten a miles de desarrolladores compartir sus trabajos y es posible que en ellas los usuarios encuentren una aplicación que cumpla con sus necesidades.
- **Soporte de la comunidad:** Esta es la principal fortaleza de Linux. Millones de usuarios alrededor del mundo pueden encontrar errores en alguna aplicación y desarrolladores miembros de la comunidad (en algunos casos los creadores de las aplicaciones) arreglarán este problema y difundirán su solución. El mejor sitio para intercambio de información son las listas de correo de soporte y desarrollo.
- **Licencia:** Al crear una aplicación bajo alguna de las licencias usuales en Linux, no implica perder la propiedad intelectual ni los derechos de autor de la misma.
- **Independencia del vendedor:** No existe solo un distribuidor del sistema operativo GNU-Linux, ya que las licencias de Linux garantizan igualdad a los distribuidores. Algunos vendedores proporcionan aplicaciones adicionales que no son libres y por lo tanto no se encuentran disponibles con otros vendedores. Esto debe tenerse en cuenta en el momento de elegir la distribución a utilizar.
- **Costo:** Muchas de las herramientas de desarrollo y componentes de Linux son gratuitos, y no requieren el pago por ser incluidos en productos comerciales.

### **5.8.1. Arquitectura de Linux [2] [3]**

Linux (Linus' Minix) es un clon del sistema operativo Unix para PC con procesador Intel 386. Linux tomó dos características muy importantes de Unix: sistema multitarea y multiusuario, lo cual fue implementado posteriormente por los sistemas operativos Windows y MacOS. Con estas características es posible ejecutar tareas de forma independiente, y transparente para él/los usuarios.

En la figura 5.20 se muestra la interacción entre el kernel de linux y las aplicaciones de espacio de usuario. El kernel de Linux es el encargado de la comunicación a bajo nivel con el hardware (procesador, memorias, periféricos) y proporcionar un camino para que una aplicación creada con una función específica pueda utilizar de forma adecuada estos recursos.

En estos dos espacios se identifican sus creadores, como ya se mencionó Richard Stallman proporcionó todas las herramientas que permiten la creación de aplicaciones, mientras Linus Torvalds creó el kernel que proporciona los recursos para poder ejecutarlas.

Linux está compuesto por cinco submódulos [3]: El programador (Scheduler), el manejador de memoria, el sistema de archivos virtual, la interfaz de red y la comunicación entre procesos (IPC) (figura 5.20).

Como puede observarse en la figura 5.20 el kernel de Linux posee sub-módulos que son independientes de la arquitectura y otros que deben ser escritos para el procesador utilizado, esto hace que Linux sea un sistema operativo portable. En la actualidad existe una gran variedad de arquitecturas soportadas por Linux, entre las que se encuentran: alpha arm26 frv i386 m32r m68knommu parisc ppc64 sh sparc um x86\_64 arm cris h8300 ia64 m68k mips ppc s390 sh64 sparc64 v850. Las funciones de estos componentes se describen a continuación [2]:

#### **Programador de procesos (Scheduler)**

Es el corazón del sistema operativo Linux; está encargado de realizar las siguientes tareas:

- Permitir a los procesos hacer copias de sí mismos.
- Determinar que procesos pueden acceder a la CPU y efectuar la transferencia entre los procesos en ejecución.
- Recibir interrupciones y llevarlas al subsistema del kernel adecuado.
- Enviar señales a los procesos de usuario.

<sup>5</sup> <http://www.sourceforge.net>

<sup>6</sup> <http://www.freshmeat.net>

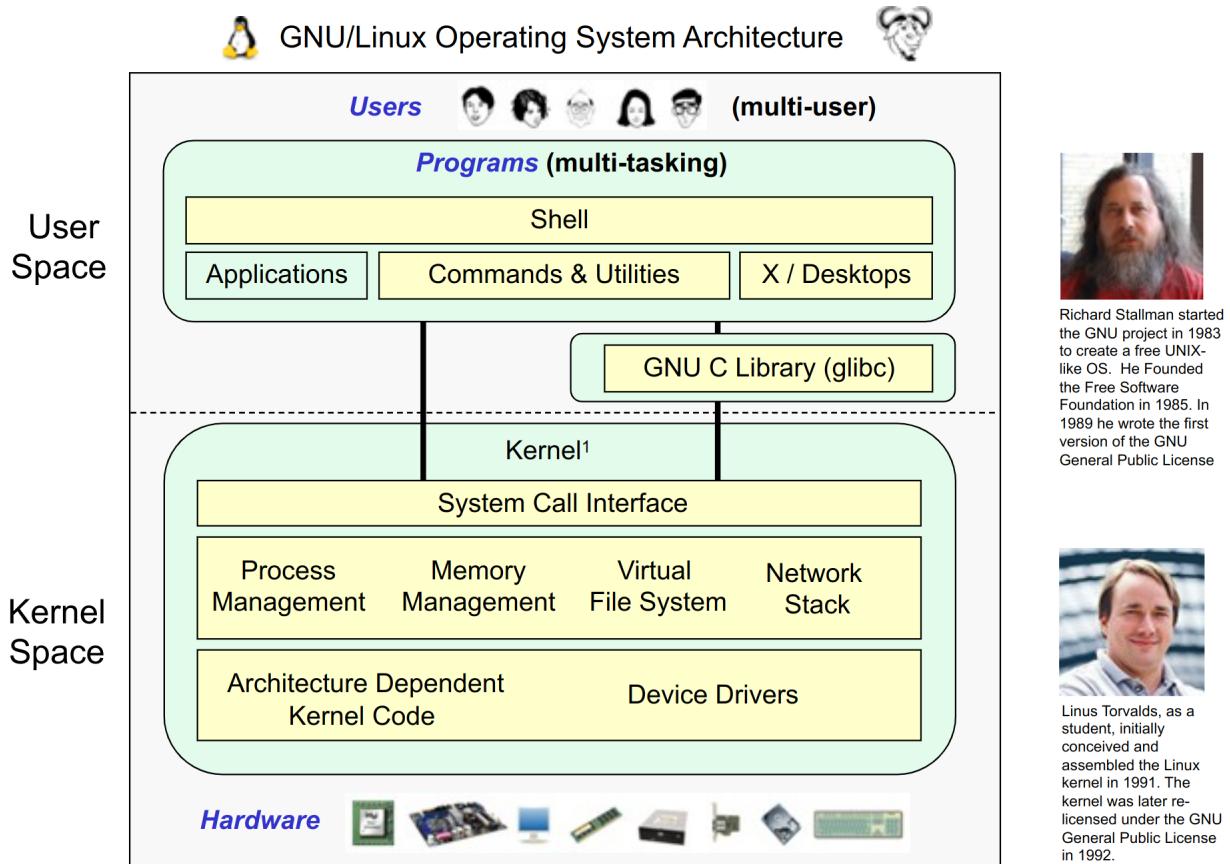


Figura 5.20 Estructura del sistema operativo GNU/Linux. Fuente:Cabrillo College

- Manejar el timer físico.
- Liberar los recursos de los procesos cuando estos finalizan su ejecución.

El programador de procesos proporciona dos interfaces: Una con recursos limitados para los procesos de usuario y una interfaz amplia para el resto del kernel. Debido a que el programador utiliza una interrupción del timer que se presenta cada 10 ms, el cambio de estado del programador<sup>7</sup> se realiza cada 10 ms. Esto debe ser tenido en cuenta a la hora de realizar procesos que manejen dispositivos hardware veloces.

### Manejador de Memoria

En el momento en que se energiza la CPU, está solo ve 1 MByte de memoria física (incluyendo las ROMs). El código de inicialización del sistema operativo (OS) debe activar el modo protegido del procesador, de tal forma que se pueda acceder a la memoria extendida (incluyendo la memoria de los dispositivos). Finalmente, el OS habilita la memoria virtual para dar la ilusión de un espacio de memoria de 4 GB. El manejador de memoria proporciona los siguientes servicios (ver figura 5.22):

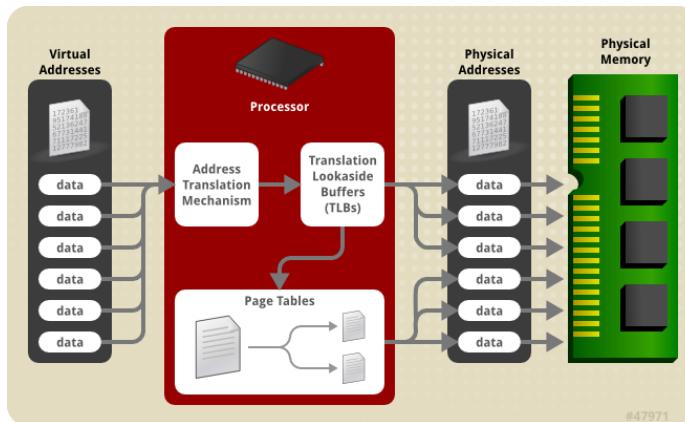
- **Gran espacio de memoria.** Los procesos usuario, pueden referenciar más memoria que la existente físicamente.
- **Protección** La memoria de un proceso es privada y no puede ser leída o modificada por otro proceso. Adicionalmente evita que los procesos sobreescrbían datos de solo lectura.

<sup>7</sup> Un proceso puede estar en uno de los siguientes estados: ejecución, retornando de un llamado de sistema, procesando una rutina de interrupción, procesando un llamado del sistema, listo, en espera

- **Mapeo de memoria** Los usuarios pueden mapear un archivo en un área de memoria virtual y acceder a él como si fuera una memoria.
- **Acceso transparente a la memoria física** lo que asegura un buen desempeño del sistema.
- **Memoria compartida**

Al igual que el programador, el manejador de memoria proporciona dos niveles de acceso a memoria diferentes a nivel de usuario y a nivel de kernel.

- Nivel de Usuario
  - *malloc() / free()*. Asigna o libera memoria para que sea utilizada por un proceso.
  - *mmap() / munmap() / msync() / mremap()* Mapea archivos en regiones de memoria virtual.
  - *mprotect* Cambia la protección sobre una región de una memoria virtual.
  - *mlock() / mlockall() / munlock() / munlockall()* Rutinas que permiten al super-usuario prevenir el intercambio de memoria.
  - *swapon() / swapoff()* Rutinas que le permiten al super-usuario agregar o eliminar archivos *swap* en el sistema.
- Nivel de kernel
  - *kmalloc() / kfree()* Asigna o libera memoria para que sea utilizada por estructuras de datos del kernel.
  - *verify\_area()* Verifica que una región de la memoria de usuario ha sido mapeada con los permisos necesarios.
  - *get\_free\_page() / free\_page()* Asigna y libera páginas de memoria física.



**Figura 5.21** Memoria virtual en Linux fuente:<https://docs.redhat.com/>

## Comunicación Entre Procesos (IPC)

El mecanismo IPC de Linux posibilita la ejecución *concurrente* de procesos, permitiendo compartir recursos y la sincronización e intercambio de datos entre ellos. Linux proporciona los siguientes mecanismos:

- **Señales** Mensajes asíncronos enviados a los procesos.
- **Listas de espera** Proporciona mecanismos para colocar a dormir a los procesos mientras esperan que una operación se complete, o un recurso se libere.
- **Bloqueo de archivos** Permite a un proceso declarar una región de un archivo como solo lectura para los demás procesos.
- **Conductos (pipe)** Permite transferencias bi-direccionales entre dos procesos.
- **System V**
  - **Semáforos** Una implementación del modelo clásico del semáforo.

- **Lista de Mensajes** Secuencia de bytes, con un tipo asociado, los mensajes son escritos a una lista de mensajes y pueden obtenerse leyendo esta lista.
- **Memoria Compartida** Mecanismo por medio del cual varios procesos tienen acceso a la misma región de memoria física.
- **Sockets del dominio Unix** Mecanismo de transferencia de datos orientada a la conexión.

## Interfaces de Red

Este sistema proporciona conectividad entre máquinas, y un modelo de comunicación por sockets. Se proporcionan dos modelos de implementación de sockets: BSD e INET. Además, proporciona dos protocolos de transporte con diferentes modelos de comunicación y calidad de servicio: El poco confiable protocolo UDP (*User Datagram Protocol*) y el confiable TCP (*Transmission Control Protocol*), este último garantiza el envío de los datos y que los paquetes serán entregados en el mismo orden en que fueron enviados. Se proporcionan tres tipos diferentes de conexión: SLIP (Serial), PLIP (paralela) y ethernet.

## Sistema de archivo virtual

El sistema de archivos de Linux cumple con las siguientes tareas:

- **Controlar múltiples dispositivos hardware**
- **Manejar sistemas de archivos lógicos**
- **Soporta diferentes formatos ejecutables** Por ejemplo a.out, ELF, java)
- **Homogeneidad** Proporciona una interfaz común a todos los dispositivos lógicos o físicos.
- **Desempeño**
- **Seguridad**
- **Confiabilidad**

## Drivers de Dispositivos

La capa que controla los dispositivos es responsable de presentar una interfaz común a todos los dispositivos físicos. El kernel de Linux tiene 3 tipos de controladores de dispositivo: Carácter (acceso secuencial), bloque (acceso en múltiplos de tamaño de bloque) y red. Ejemplos de controladores secuenciales son el modem, el mouse; ejemplos de controladores tipo bloque son los dispositivos de almacenamiento masivo como discos duros, memorias SD. Los controladores de dispositivos soportan las operaciones de archivo, y pueden ser tratados como tal.

## Sistema de Archivos lógico

Aunque es posible acceder a dispositivos físicos a través de un archivo de dispositivo, es común acceder a dispositivos tipo bloque utilizando un sistema de archivos lógico, el que puede ser montado en un punto del sistema de archivos virtual.

Para dar soporte al sistema de archivos virtual, Linux utiliza el concepto de *inodes*. Un inode representa un archivo sobre un dispositivo tipo bloque; el inode es virtual en el sentido que contiene operaciones que son implementadas de forma diferente dependiendo del sistema lógico y del sistema físico donde reside el archivo. La interfaz Inode hace que todos los archivos se vean igual a otros subsistemas Linux. El inode se utiliza como una posición de almacenamiento para la información relacionada con un archivo abierto en el disco. El inode almacena los buffers asociados, la longitud total del archivo en bloques, y el mapeo entre el offset del archivo y los bloques del dispositivo.

## Módulos

La mayor funcionalidad del sistema de archivos virtual se encuentra disponible en forma de módulos cargados dinámicamente. Esta configuración dinámica permite a los usuarios de Linux compilar un kernel tan pequeño como sea posible, mientras permite cargar el controlador del dispositivo y módulos del sistema de archivos solo si son necesarios durante una sesión. Esto es útil en el caso de los dispositivos que se pueden conectar en caliente.

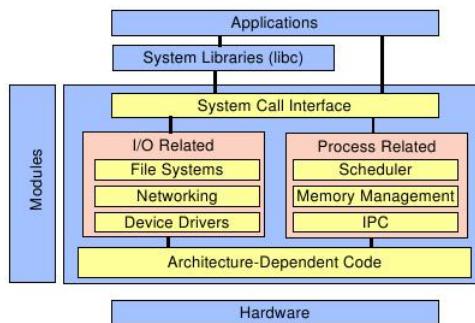


Figura 5.22 Mapeo de memoria del Kernel.

### 5.8.2. Árbol de dispositivos

El *Open Firmware Device Tree*, o Devicetree (DT), es una estructura de datos y un lenguaje para describir hardware. Específicamente, es una descripción de hardware que se puede leer por un sistema operativo, de tal forma que no es necesario codificar detalles de la máquina. Estructuralmente, el DT es un árbol, o un grafo con nodos, donde cada nodo puede tener un número arbitrario de propiedades que encapsulan datos arbitrarios.

En la figura 5.23 se muestra el ciclo de vida del *dts*, el cual tiene como propósito proporcionar información a Linux en el arranque. Esto se realiza a través de los siguientes pasos:

1. Los archivos Device Tree source (DTS) son procesados por un compilador para generar una estructura tipo árbol.
2. Esta estructura es serializada (*flattened*) para crear un archivo DTB (Device Tree Blob). Es en este punto donde se resuelven las referencias simbólicas realizadas en los nodos.
3. Este archivo se coloca en un dispositivo de almacenamiento al que tiene acceso el *bootloader*.
4. El bootloader extrae el archivo DTB y lo pasa al sistema operativo. A este nivel puede realizar pequeñas modificaciones, deshabilitando o habilitando dispositivos.
5. El sistema operativo procesa el DTB que le ha pasado el bootloader y crea una estructura en memoria la cual es escaneada para inicializar y configurar el hardware descrito en él.

Conceptualmente, se define un grupo de convenciones llamados vínculos para describir como se deben declarar los datos en el árbol para describir características típicas de hardware como buses de datos, líneas de interrupción, GPIOs y periféricos. En nuestro caso la placa *ECB.T3.T113* se basa en los DT escritos por ALLWINNER para sus plataformas de desarrollo, cuyos archivos dts se encuentran en el directorio:

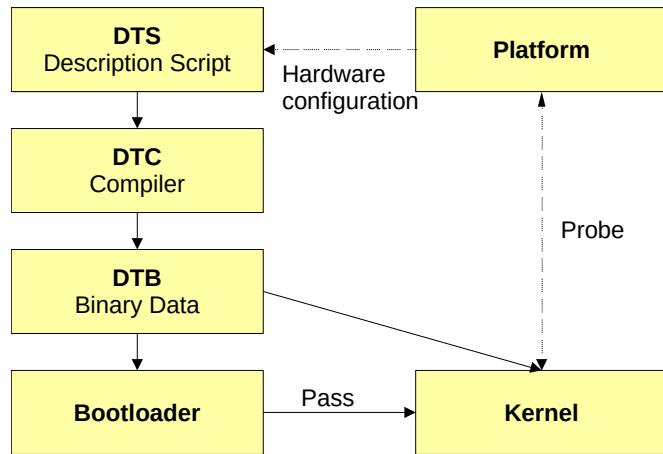
```
/kernel_source/arch/arm64/boot/dts/
```

Donde *arm64* es la familia de la CPU y debe ajustarse dependiendo de la CPU utilizada.

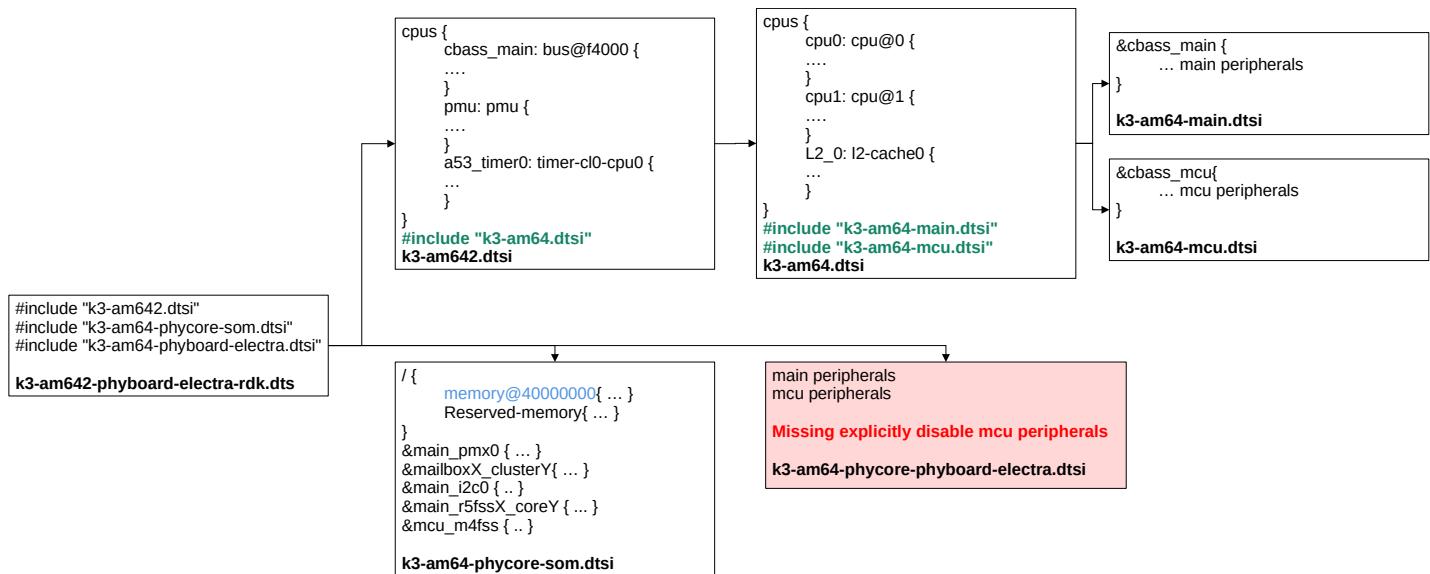
A manera de ejemplo, la figura 5.27 muestra la dependencia del archivo donde se aloja el DT de la placa Electra *k3-am642-phyboard-electra-rdk.dts* de Phytec que utiliza el procesador AM64.

Como puede verse en la figura 5.27 el DT de la placa Electra depende de:

- Descripción del procesador *k3-am642* donde se declaran todos los componentes del SoC AM642: procesadores A53, memorias reservadas, mailboxes

**Figura 5.23** Ciclo de Vida del Device Tree

- Descripción de los periféricos asociados al dominio de los procesadores A53 y a los R5F y M4( *k3-am64-main.dtsi* y *k3-am64-mcu.dtsi* ).

**Figura 5.24** Ejemplo de árbol de directorios en Linux

### 5.8.3. Distribuciones Linux

Todas las distribuciones de Linux se basan en el estándar *Filesystem Hierarchy Standard* utilizado en los sistemas operativos UNIX. Este estándar permite que los programas y los usuarios conozcan de antemano la localización de los archivos instalados. Los siguientes directorios o links simbólicos son de uso obligatorio:

1. / Directorio raíz.
2. **bin** Ejecutables esenciales.
3. **boot** Archivos estáticos del boot loader.

4. **dev** Archivos de dispositivos.
5. **etc** Configuración específica del host.
6. **proc** Sistema de archivos para manejar procesos y sistemas de información.
7. **lib** Librerías esenciales y módulos de kernel.
8. **media** Punto de montaje para dispositivos removibles.
9. **mnt** Punto de montaje temporal.
10. **opt**
11. **sbin** Ejecutables esenciales del sistema, comandos relacionados con el mantenimiento del sistema de archivos y el bootloader.
12. **srv** Datos de servicios suministrados por el sistema.
13. **tmp** Archivos temporales.
14. **sys** Se almacena información sobre dispositivos, drivers, y algunos atributos expuestos del kernel.
15. **usr** Segunda jerarquía.
  - **/usr/include/** Encabezador de las librerías de desarrollo.
  - **/usr/src/** Código fuente.
  - **/usr/**
16. **var** Datos variables.

Aunque es posible construir el sistema de archivos desde cero, no es nada práctico ya que es una tarea tediosa que requiere cierto nivel de experiencia. En la actualidad, existen varias distribuciones que realizan estas tareas por nosotros, dentro de las más utilizadas se encuentran:

## Busybox

Diseñado para optimizar el tamaño y rendimiento de aplicaciones embebidas, BusyBox<sup>8</sup> combina en un solo ejecutable más de 70 utilidades estándares UNIX, en sus versiones ligeras. BusyBox es considerada la navaja suiza de los sistemas embebidos, dado que permite sustituir la gran mayoría de utilidades que se suelen localizar en los paquetes GNU fileutils, shellutils, findutils, textutils, modutils, grep, gzip, tar, etc.

Busybox hace parte de la mayoría de distribuciones de Linux para sistemas embebidos y en la actualidad proporciona las siguientes funciones:

*addgroup, adduser, adjtimex, ar, arping, ash, awk, basename, bunzip2, busybox, bzcat, cal, cat, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, crond, crontab, cut, date, dc, dd, deallocvt, delgroup, deluser, df, dirname, dmesg, dos2unix, dpkg, dpkg-deb, du, dumpkmap, dumpleases, echo, egrep, env, expr, false, fbset, fdflush, fdisk, fgrep, find, fold, free, freeramdisk, fsck.minix, ftpget, ftpput, getopt, getty, grep, gunzip, gzip, halt, head, hexdump, hostid, hostname, httpd, hwclock, id, ifconfig, ifdown, ifup, init, ip, ipaddr, ipcalc, iplink, iproute, iptunnel, kill, killall, klogd, last, length, linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread, losetup, ls, makedevs, md5sum, mesg, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktemp, more, mount, mt, mv, nameif, nc, netstat, nslookup, od, openvt, passwd, patch, pidof, ping, ping6, pivot\_root, poweroff, printf, ps, pwd, rdate, readlink, realpath, reboot, renice, reset, rm, rmdir, route, rpm, rpm2cpio, run-parts, sed, setkeycodes, sh, sha1sum, sleep, sort, start-stop-daemon, strings, stty, su, sulogin, swapoff, swapon, sync, syslogd, tail, tar, tee, telnet, telnetd, test, tftp, time, top, touch, tr, traceroute, true, tty, udhcpc, udhcpd, umount, uname, uncompress, uniq, unix2dos, unzip, uptime, usleep, uudecode, uuencode, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat.*

## Buildroot

Buildroot<sup>9</sup> Es un grupo de *Makefiles* y *patches* que facilita la generación de la cadena de herramientas y el sistema de archivos para un sistema embebido que usa Linux. Posee una interfaz que permite realizar de forma fácil la configura-

---

<sup>8</sup> <http://www.busybox.net/>

<sup>9</sup> <http://buildroot.uclibc.org/>

ción; utiliza busybox para generar la utilidades básicas de Linux y permite adaptar software adicional de forma fácil<sup>10</sup>.

### **Openembedded**

Al igual que Buildroot, el proyecto *openembedded* proporciona un entorno que permite generar la cadena de herramientas y el sistema de archivos para un sistema embebido, utiliza busybox y permite la creación de archivos que permiten compilar aplicaciones que no se incluyen en la distribución original. Adicionalmente *openembedded* crea archivos instaladores con un formato derivado del proyecto *handhelds*<sup>11</sup> *ipk*, lo que permite la instalación de paquetes de forma similar a la distribución debian.<sup>12</sup>

#### **5.8.4. Yocto**

El Proyecto Yocto es un proyecto colaborativo de código abierto que ayuda a los desarrolladores a crear sistemas a la medida basados en Linux. Este proyecto proporciona un grupo de herramientas y un entorno donde se pueden compartir tecnologías, configuraciones y mejores prácticas que pueden ser utilizadas para crear imágenes de Linux para dispositivos IoT a la medida. Al igual que openwrt y buildroot posee una gran variedad de recetas que permiten compilar e incluir aplicaciones en imágenes.

#### **5.8.5. Debian**

El proyecto Debian es una asociación de individuos que comparten una meta común: crear un sistema operativo abierto, de libre distribución<sup>13</sup>. A diferencia de las anteriores distribuciones, Debian solo proporciona el sistema de archivos, es decir, no proporciona imágenes del kernel para una determinada plataforma. Debien posee una ventaja sustancial sobre los demás y es la facilidad de instalar nuevas aplicaciones ya que posee servidores con paquetes precompilados que hacen de este proceso algo fácil y rápido.

## **5.9. Adaptando Linux a la plataforma ECB\_T8\_T113**

En esta sección se realizará el proceso de adaptar el kernel de Linux y el sistema de archivos a la plataforma ECB\_T8\_T113, como vimos anteriormente, es necesario crear dos componentes: El kernel de Linux y el sistema de archivos. Utilizaremos dos métodos para hacer este proceso.

- Utilizando buildroot generaremos una imagen que se puede grabar directamente en una memoria SD, este método es muy sencillo de implementar ya que básicamente es una lista de comandos que se deben ejecutar, su desventaja es que no se sabe como se creó la imagen y no se tiene conocimiento del proceso, conceptos que deben tenerse muy claros para poder hacer modificaciones.
- Imagen construida parte por parte partiendo desde el bootloader, siguiendo por la imagen del kernel y finalizando con la adaptación de los periféricos utilizados.

---

<sup>10</sup> [http://buildroot.uclibc.org/buildroot.html#add\\_software](http://buildroot.uclibc.org/buildroot.html#add_software)

<sup>11</sup> <http://handhelds.org>

<sup>12</sup> <https://www.openembedded.org/>

<sup>13</sup> Tomado de <http://debian.org>

### 5.9.1. Creación de la imagen utilizando buildroot

Este método automatizado permite la creación de una imagen que puede ser descargada directamente a la tarjeta SD. La versión de buildroot a utilizar es una adaptación del utilizado en la tarjeta YuzukiSBC.

```
git clone https://github.com/yuzukihd/Buildroot-YuzukiSBC # Clonar el repositorio Buildroot
cd Buildroot-YuzukiSBC
source envsetup.sh                                         # Cargar las variables de entorno
lunch                                                       # Ejecutar el entorno buildroot
make mangopi_mq_dual_defconfig                         # Placa basada en el T113.
make -j4
```

Es normal que se presenten problemas debido a la distribución utilizada en el computador donde se crea el sistema de archivos ya sea por incompatibilidad de librerías, versión de aplicaciones, o ausencia de aplicaciones que se deben instalar, parte del proceso de trabajar con estas herramientas es el de saber corregir estos errores y finalizar el proceso de compilación.

Al finalizar dicho proceso se obtienen los siguientes directorios y archivos:

```
output
|-- build
|   |-- alsa-lib-1.2.6.1
|   |-- alsa-utils-1.2.6
...
|   |-- linux-custom
...
|   |-- uboot-custom
...
|   '-- zlib
|-- host
|   |-- arm-buildroot-linux-gnueabi
|   |-- bin
|   |-- etc
|   |-- include
|   |-- lib
|   |-- lib64 -> lib
|   |-- opt
|   |-- sbin
|   |-- share
|   '-- usr -> .
|-- images
|   |-- boot0_nand.fex
|   |-- boot0_sdcard.fex
|   |-- boot0_sdcard_sdc2.fex
|   |-- boot0_spinor.fex
|   |-- boot.img
|   |-- boot_package.cfg
|   |-- boot_package.fex
|   |-- boot-resource.fex
|   |-- boot.vfat
|   |-- dragonsecboot
|   |-- env.cfg
|   |-- env.fex
|   |-- mkenvimage
|   |-- optee.fex
```

```

|   |-- ramdisk.img
|   |-- rootfs.ext2
|   |-- rootfs.ext4 -> rootfs.ext2
|   |-- sdcard.img
|   |-- sun8i-mangopi-mq-dual-linux.dtb
|   |-- u-boot-sun8iw20p1.bin
|   '-- zImage
`-- target
    |-- bin
    |-- dev
    |-- etc
    |-- lib
    |-- lib32 -> lib
    |-- linuxrc -> bin/busybox
    |-- media
    |-- mnt
    |-- opt
    |-- proc
    |-- root
    |-- run
    |-- sbin
    |-- sys
    |-- THIS_IS_NOT_YOUR_ROOT_FILESYSTEM
    |-- tmp
    |-- usr
    '-- var

```

En la carpeta build se encuentran los directorios de las aplicaciones necesarias para crear el sistema de archivos, en este listado se encuentran los directorios del kernel (*linux-custom*) y de u-boot(*uboot-custom*), estos directorios son importantes en el proceso de adaptación de drivers ya que es aquí donde se deben realizar los cambios.

El directorio *host* contiene las aplicaciones que requiere el computador donde se realiza la compilación.

El directorio *target* es un directorio donde *buildroot* instala las aplicaciones y sirve como base para la creación de la imagen.

El directorio *images* contiene los archivos necesarios para crear la imagen:

- *dragonseboot, mkenvimage* - Ejecutables para crear archivos necesarios para la imagen.
- *boot0\_nand.fex, boot0\_sdcard.fex, boot0\_sdcard\_sdc2.fex, boot0\_spinor.fex* - Binarios para diferentes medios de almacenamiento de la primera etapa del Bootloader, estos archivos se distribuyen con buildroot en board/allwinner-generic/sun20i-d1s/bin/
- *optee.fex* - Trusted Execution Environment (TEE) provides isolation from the non-secure OS and protects the loaded Trusted Applications (TAs) from each other using underlying hardware support.
- *u-boot-sun8iw20p1.bin* - Binario de u-boot
- *ramdisk.img* - Archivo vacío necesario para crear la imagen.
- *boot\_package.cfg, boot\_package.fex* - Archivo que empaqueta u-boot, optee y el árbol de dispositivos (dtb).
- *boot-resource.fex*
- *env.cfg, env.fex* - Variables de entorno de u-boot.
- *sun8i-mangopi-mq-dual-linux.dtb, zImage*
- *boot.img* - Imagen de boot de Android.
- *boot.vfat* - Imagen que contiene la imagen del kernel comprimida y en formato para android, y el árbol de dispositivos. u-boot solo tiene en cuenta el archivo *boot.img*.
- *rootfs.ext2* - Imagen del sistema de archivos en formato ext2/4.
- *sdcard.img* - Imagen para crear la SD

Es bueno conocer el proceso de creación de la imagen para poder realizar cambios en la imagen del kernel y dar soporte a otros periféricos. Adicionalmente, aunque buildroot permite crear archivos tipo *ipk* para instalar nuevas aplicaciones a

un sistema de archivos existentes, en algunas ocasiones este proceso es tedioso debido a problemas de compatibilidad de versiones de librerías.

La figura 5.25 muestra la estructura de la imagen generada. En la posición 0x2000 aparece el archivo boot0\_sdcard.fex, el cual corresponde a una imagen eGON, del que hablaremos más adelante. En la posición 0x1004000 aparecen los archivos u-boot-sun8iw20p1.bin, optee.fex y sun8i-mangopi-mq-dual-linux.dtb. A continuación aparece la primera partición visible desde el ordenador, una partición DOS que contiene archivos que no utilizamos por ahora. A continuación vienen dos particiones en donde se copia el entorno de u-boot, es decir, variables que ayudan a que el arranque de u-boot sea de forma automática, estas son:

```

boot_check=sunxi_card0_probe;mmcinfo;mmc part
boot_mmc=fatload mmc ${mmc_dev}:${mmc_boot_part} 45000000 ${kernel}; bootm 45000000
boot_partition=boot
bootargs=earlyprintk=sunxi-uart,0x02500000 clk_ignore_unused initcall_debug=0 console=tty
bootcmd=run boot_check boot_mmc
bootdelay=2
console=ttyS0,115200
fdtcontroladdr=4487fe70
force_normal_boot=1
kernel=boot.img
keybox_list=widenv,ec_key,ec_cert1,ec_cert2,ec_cert3,rsa_key,rsa_cert1,rsa_cert2,rsa_ce
mmc_boot_part=4
mmc_dev=0
partitions=boot-resource@mmcblk0p1:env@mmcblk0p2:env-redund@mmcblk0p3:boot@mmcblk0p4:@mm
root_partition=rootfs

```

Seguidamente aparece la segunda partición DOS visible, la cual contiene los archivos boot.img, Zimage y sun8i-mangopi-mq-dual-linux.dtb. De estos únicamente se utiliza boot.img, que como se puede ver en las variables de entorno, es la imagen del kernel que se utiliza en el arranque (kernel=boot.img), si queremos cambiar la imagen del kernel basta con reemplazar este archivo por uno nuevo, sin embargo el árbol de dispositivos no puede ser reemplazado de esta forma ya que está en la segunda partición y no se puede acceder con un file-manager, más adelante explicaremos como se puede modificar esa partición.

Finalmente, tenemos una partición EXT4 con el sistema de archivos.

### 5.9.2. Arranque del procesador Allwinnert T113

En la figura 5.26 se muestra de forma detallada los pasos que se realizan en el proceso de arranque del procesador T113.

De forma similar a la mayoría de procesadores, se ejecuta un programa en la ROM interna (bootrom), la cual lee el contenido del registro *BUSBOOT\_MODE* (0x03006210) y selecciona entre dos formas de seleccionar el dispositivo en la primera el dispositivo es "grabado" en un medio interno llamado eFuse y la segunda, el periférico se elige por el estado de dos pines externos, para la placa se seleccionará el segundo caso, y específicamente se fija la memoria SD. A continuación, se lee el estado del *FEL pin* (bit 8 del registro 0x03000024) para seleccionar entre dos formas de arranque, el *Mandatory upgrade* obliga a que se realice un proceso obligatorio de upgrade, esto se utiliza en casos en donde el dispositivo detecta que el fabricante liberó una nueva versión de software. Y el *fastboot* que permite cargar una imagen siguiendo el siguiente proceso:

El bootrom activa el periférico seleccionado, dando un soporte básico, y busca una imagen eGON.BTO válida en la posición 0x2000 (8kB), esto es, la imagen tiene la firma, el tamaño adecuado (menor que el tamaño de la memoria SRAM interna) y el checksum válido. Una vez verificada la imagen, se copia el contenido a la memoria SRAM interna (posición 0x0) y se ejecuta este programa.

Este primer programa que se ejecuta en la SRAM interna recibe el nombre de *First Stage Bootloader* y realiza las tareas necesarias para dar soporte total al medio de boot y configurar una memoria RAM con mayor capacidad para

<b>Offset: 8k (0x2000)</b>	<b>boot0_sdcard.fex</b> First stage bootloader
<b>Offset: 16400k (0x1004000)</b>	<b>boot_package.fex</b> u-boot-sun8iw20p1.bin optee.fex sun8i-mangopi-mq-dual-linux.dtb
<b>Size: 512k</b>	<b>boot-resource.fex</b> bootlogo.bmp magic.bin
<b>Size: 128k</b>	<b>env.fex</b> U-boot environment
<b>Size: 128k</b>	<b>env.fex</b> U-boot environment
<b>FAT</b>	<b>boot.vfat</b> boot.img Zimage sun8i-mangopi-mq-dual-linux.dtb
<b>EXT4</b>	<b>rootfs.ext4</b>

**Figura 5.25** Estructura de la imagen generada por buildroot

poder cargar aplicaciones más grandes, en el caso del procesador T113, se habilita la memoria de 128MB DDR interna al procesador; de tal forma que se puedan copiar desde la SD hasta la DDR.

En la distribución *Buildroot-YuzukiSBC* se utiliza un binario precompilado (boot0\_sdcard.fex) localizado en la carpeta *buildroot/board/allwinner-generic/sun8i-t113/bin*. Si se desea conocer más sobre la forma de arranque y como inicializar los diferentes periféricos del chip se incluye en el repositorio el código fuente para generarla y los fuentes de awboot (awboot-main.zip).

En este archivo se define:

UBOOT\_START\_SECTOR\_IN\_SDMMC (32800)

Ya que cada sector de 512bytes la dirección donde se debe grabar la imagen de u-boot es la  $(32800 \times 512) = 0x1004000$ , tal como se muestra en la imagen.

## Ejecución de u-boot

En este momento se inicia la ejecución de la segunda etapa del bootloader, que en nuestro caso es u-boot, el cual permite automatizar el proceso de carga de la imagen del kernel y el árbol de dispositivos.

Tal como está configurado u-boot no es posible cargar el árbol de dispositivos compilado (archivo **dtb**) desde la tarjeta SD ya que no fué habilitado el comando *bootz* o *booti*, por lo que para hacerlo es necesario crear el archivo *boot\_package.fex* con el nuevo archivo **dtb**. Para esto debemos utilizar la aplicación *dragonsecboot* proporcionada por buildroot y por nuestro repositorio en *Develop/Linux/src/images*. *textitdragonsecboot* requiere un archivo de configuración donde se le especifique el orden en el que se almacenarán los archivos, para esto se crea un archivo con el nombre *boot\_package.cfg* con el siguiente contenido

```
[package]
;item=Item_TOC_name,           Item_filename,
item=u-boot, u-boot-sun8iw20p1.bin
item=optee, optee.fex

item=dtb, sun8i-mangopi-mq-dual-linux.dtb
```

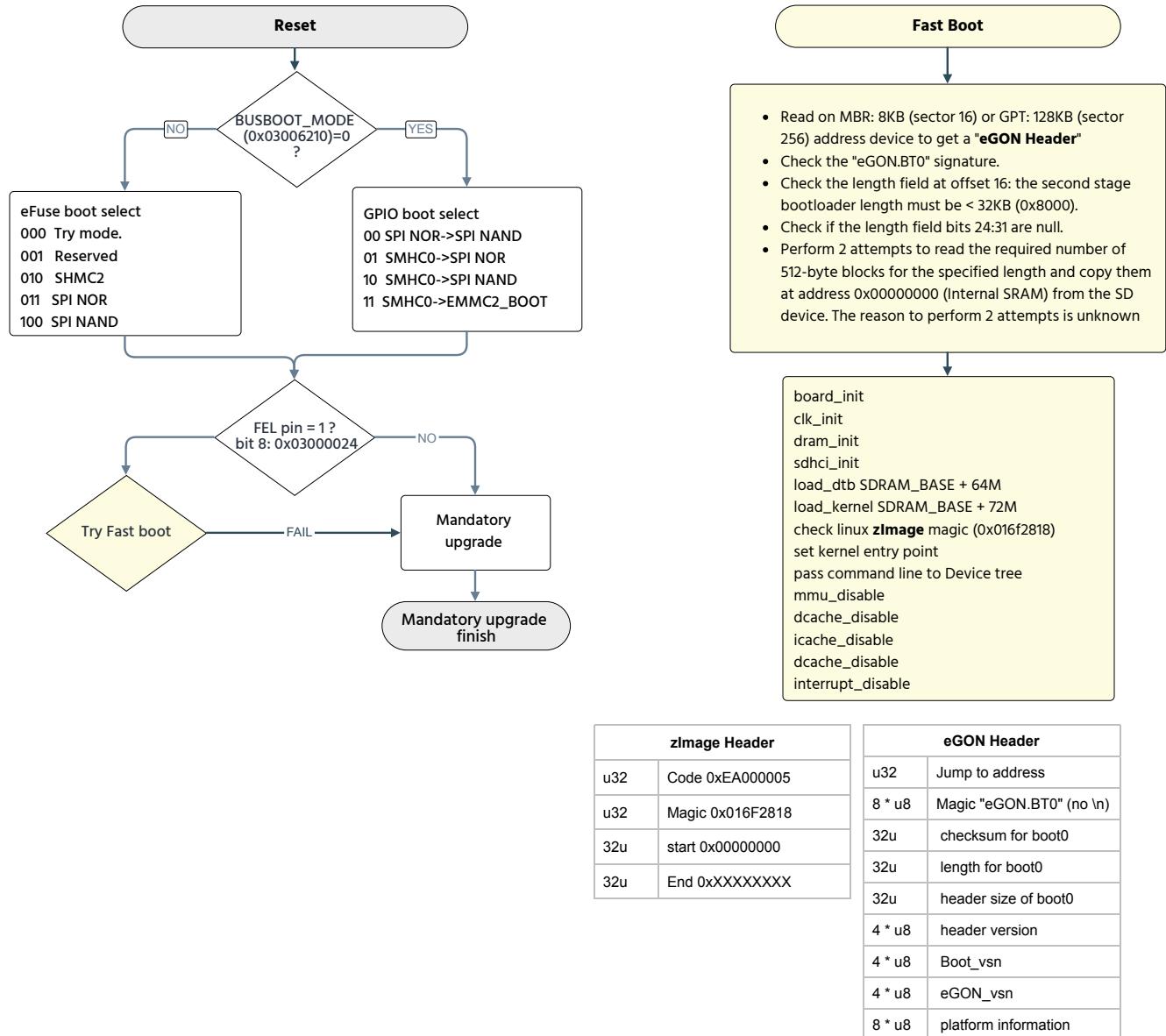


Figura 5.26 Diagrama de flujo del programa de inicialización del SoC T113

Al ejecutar el comando: `./dragonsecboot -pack boot_package.cfg` obtenemos el nuevo archivo `boot_package.fex`, el cual podemos transferir a la tarjeta usando el comando:

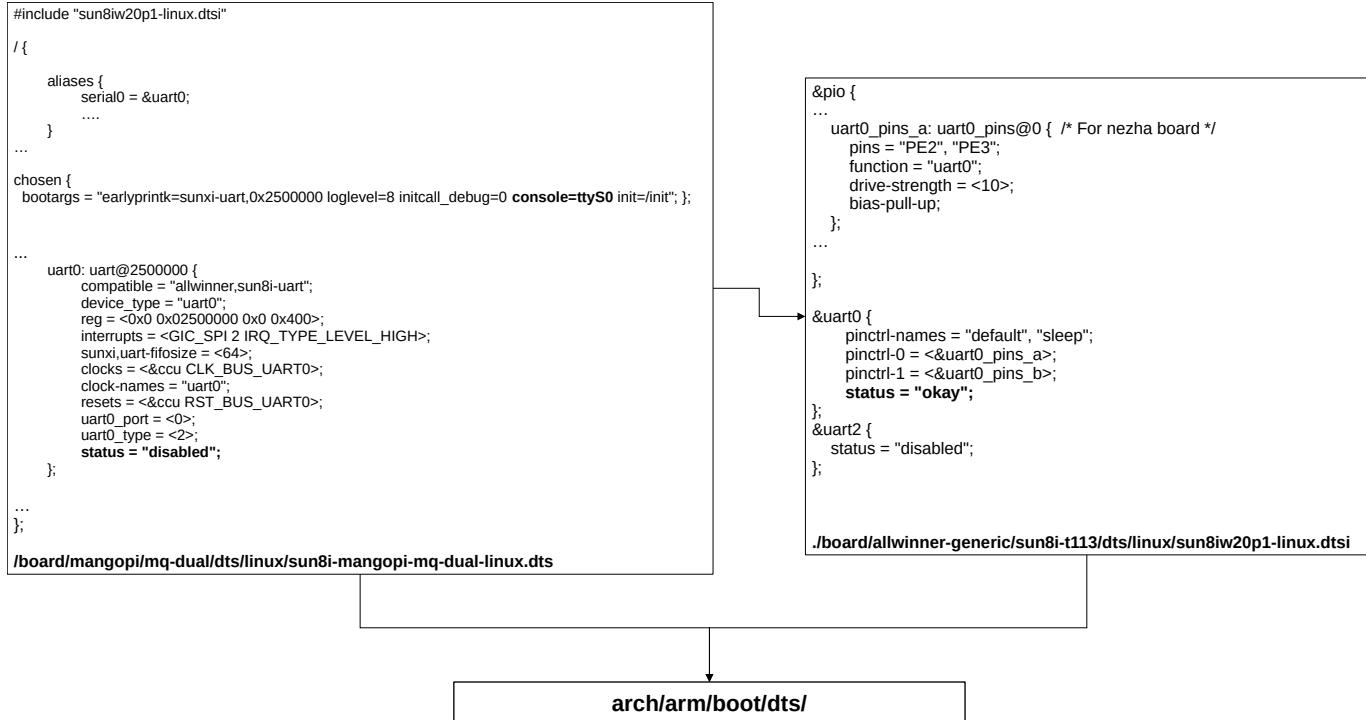
```
sudo dd if=boot_package.fex of=/dev/sdX(cambiar) bs=1k seek=16400
```

Con esto ya podemos cambiar el árbol de dispositivos.

### 5.9.3. Configuración del kernel de Linux y de u-boot desde Buildroot

Para modificar la configuración del kernel de Linux desde buildroot debemos ejecutar el siguiente comando:

```
make linux-menuconfig
```

**Figura 5.27** Definición del puerto serial 0

Para dar soporte a la tarjeta de red, al manejo de GPIOs desde sysfs y permitir el boot a debian debemos modificar:

```
Device Drivers
  GPIO Support  --->
    <*> /sys/class/gpio/
    <*> Debug GPIO calls

Device Drivers
  Network device support  --->
    PHY Device support and infrastructure  --->
      < > RealTek RTL-8363NB Ethernet Adapter support

File systems
  <*> Old Kconfig name for Kernel automounter support
  <*> Kernel automounter support (supports v3, v4 and v5)
  < > SquashFS 4.0 - Squashed file system support

General setup
  <*> Control Group support
  [*] Support for paging of anonymous memory (swap)
```

Para que se genere la nueva imagen con estos cambios debemos ejecutar los comandos:

```
make linux-rebuild
make
```

### 5.9.4. Creación de la tarjeta SD

Finalmente para crear la SD apartir de buildroot debemos copiar el contenido del archivo *sdcard.img* a la sd, para esto ejecutamos el comando:

```
cd output/images
sudo dd if=sdcard.img of=/dev/sdX (cambiar por la letra correspondiente)
```

### 5.9.5. Configuración del puerto serial

La plataforma ECB\_T8\_T113 cuenta con tres puertos USB tipo C: OTG, HOS, y DBG. El puerto con el label DBG (más cercano al bore de la placa) es utilizado para configurar la FPGA y como medio de depuración. Las señales del puerto 2 del FTDI están disponible en el centro del conector J13 tal como se muestra en la figura 5.28.

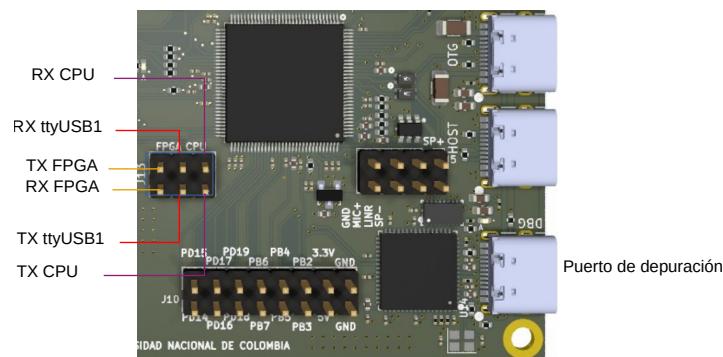


Figura 5.28 Puerto de depuración de la tarjeta ECB\_T8\_T113

Para tener acceso a los mensajes del proceso de arranque del procesador debemos hacer un puente entre las señales *RX ttyUSB1 - TX CPU* y *TX ttyUSB1 - RX CPU*. Se debe configurar una terminal serial ya sea *minicom* o similar a una velocidad de 115200 BPS y **sin control de flujo SW ni HW**.

Cuando se introduzca la SD en la placa y la conectemos por el puerto de depuración aparecerán los mensajes en el terminal serial.

## 5.10. Creación y adaptación de Debian

Como se dijo anteriormente, y como se dieron cuenta durante el proceso de compilación de buildroot, y el proceso de creación de nuevas aplicaciones es un proceso largo y en algunas ocasiones tedioso. Debian facilita la instalación de paquetes, y la creación del sistema de archivos, por esta razón indicaremos los pasos necesarios para usar Debian en esta placa.

### 5.10.1. Creación/aumento de tamaño de la partición en la SD

El primer paso consiste en aumenta el tamaño de la 5ta partición donde se aloja el sistema de archivos, ya que buildroot le deja un tamaño de 250MB. Para esto debemos borrar la 5ta partición, volverla a crear con un tamaño mayor, grabar la tabla de particiones y formatearla con el formato ext4. AL finalizar debemos tener la siguiente tabla de particiones:

```
sudo fdisk /dev/sdX (editar)
```

```
Command (m for help): p
```

Device	Start	End	Sectors	Size	Type
/dev/sdc1	35360	36383	1024	512K	Linux filesystem
/dev/sdc2	36384	36639	256	128K	Linux filesystem
/dev/sdc3	36640	36895	256	128K	Linux filesystem
/dev/sdc4	36896	102431	65536	32M	Linux filesystem
/dev/sdc5	104448	62333918	62229471	29,7G	Linux filesystem

Borrar partición: d

Crear nueva partición n

Gardar cambios: w

```
sudo mkfs.ext4 /dev/sdX5 (editar)
```

### 5.10.2. Descarga del sistema de archivos

Debian utiliza dos etapas para crear el sistema de archivos en la primera se crea el árbol de directorios y se descargan los paquetes. En el siguiente comando se le indica a la herramienta *debootstrap* que el procesador es un **armhf**, se listan los paquetes que debe contener, se fija la distribución a **bookworm**,

```
mkdir debian_fs
cd debian_fs
# (re-)generate only if rootfs doesn't exist or runme script has changed
# bootstrap a first-stage rootfs
sudo rm -rf debian_bookworm

# Ejecutar la primera etapa de bootstrap
fakeroot debootstrap --variant=minbase \
--arch=armhf --components=main,contrib,non-free \
--foreign \
--include=apt-transport-https,busybox,ca-certificates,can-utils, \
command-not-found,chrony,curl,e2fsprogs,ethtool,fdisk,gpiod,haveged, \
i2c-tools,ifupdown,iputils-ping,isc-dhcp-client,initramfs-tools, \
libibio-utils,lm-sensors,locales,nano,net-tools,wpa_supplicant,ntpdate, \
openssh-server,psmisc,rfkill,sudo,systemd-sysv,tftp,tftp-hpa,tio,usbutils, \
wget,xterm,xz-utils,vim,fim,build-essential,libftdi1-dev,libftdi1,chuck,chuck-data, \
alsa-utils,libasound2-dev,libsndfile1-dev,bison,flex,jackd2,libasound2,libsndfile1, \
libpulse-dev,pulseaudio,libjack-jackd2-dev,faust,apache2, \
libmicrohttpd12,libmicrohttpd-dev \
bookworm \
debian_bookworm \
https://deb.debian.org/debian
```

#### Segunda etapa

```
touch debian_bookworm/stage2.sh
```

El comando *chroot* permite cargar el sistema de archivos recién creado y realizar procesos de configuración sin la necesidad de cargarlo en la placa.

```
sudo apt-get install qemu-user-static
```

```
sudo chroot debian_bookworm
/debootstrap/debootstrap --second-stage
mount -vt proc proc /proc
mount -vt sysfs sysfs /sys
passwd -d root
```

Modificar o crear el archivo */etc/resolv.conf* con el contenido:

```
nameserver 10.42.0.1
```

Modificar o crear el archivo */etc/network/interfaces*

```
auto eth0
iface eth0 inet static
    address 10.42.0.220
    netmask 255.255.255.0
    gateway 10.42.0.1
```

Se debe modificar el archivo */etc/apt/sources.list* a:

```
deb https://deb.debian.org/debian bookworm main contrib non-free-firmware
```

Después se debe ajustar el reloj a una hora actual, ya que la fecha y hora por defecto de Linux es muy vieja y los servidores de debian no permiten la conexión.

```
date -s "19 JUN 2024 20:33:00"
apt-file update
printf "/dev/root / ext4 defaults 0 1\\n" > /etc/fstab
rm -f /stage2.sh
sync
umount /proc/
umount /sys
exit
```

Finalmente debemos copiar este sistema de archivos a la tarjeta SD:

```
sudo cp -avr * /media/**user**/mount_sdx5_partition/
```

### **5.10.3. Modificación del kernel para Debian**

Es necesario realizar una serie de modificaciones para que el sistema de archivos corra en la placa. Si esto no se hace obtendremos el siguiente mensaje:

```
[] systemd[1]: Failed to mount tmpfs at /sys/fs/cgroup: No such file or directory
[] systemd[1]: Failed to mount cgroup at /sys/fs/cgroup/systemd: No such file or directory
[!!!!!!] Failed to mount API filesystems.
```

Para solucionar esto y dar soporte a la interfaz de red, al manejo de GPIOs utilizando la interfaz sysfs, debemos modificar la configuración del kernel ejecutando el siguiente comando en Buildroot-YuzukiSBC/buildroot:

```
make linux-menuconfig
```

Debemos configurar los siguientes campos:

```

Device Drivers
  GPIO Support --->
    <*> /sys/class/gpio/
    <*> Debug GPIO calls

Device Drivers
  Network device support --->
    PHY Device support and infrastructure --->
      < > RealTek RTL-8363NB Ethernet Adapter support

File systems
  <*> Old Kconfig name for Kernel automounter support
  <*> Kernel automounter support (supports v3, v4 and v5)
  < > SquashFS 4.0 - Squashed file system support

General setup
  <*> Control Group support
  [*] Support for paging of anonymous memory (swap)

```

Ahora debemos recomilar el kernel:

```
make linux-rebuild
make
```

### Creación de la imagen de ANDROID

Por defecto *buildroot* solo permite cargar la imagen del kernel en el formato que acepta ANDROID, para esto debemos ejecutar los comandos:

```
cd output/images
mkbootimg --kernel zImage --output boot.img
```

Esta imagen se debe grabar en la partición DOS donde se encuentra el archivo del mismo nombre.

Al energizar la tarjeta debe cargar la imagen del kernel, el sistema de archivos y al finalizar mostrar el mensaje:

```
Debian GNU/Linux 12 verivolt ttyS0
verivolt login:
```

## 5.11. Compilación de u-boot

### 5.11.1. Descarga de la cadena de Herramientas

Es necesario contar con una cadena de herramientas que permita generar la imagen del kernel y de u-boot para la arquitectura arm, aunque podemos usar la que utiliza buildroot, es mejor descargarla de forma separada para eliminar la dependencia.

```
mkdir ~TOOLCHAIN
cd ~TOOLCHAIN
wget https://github.com/YuzukiHD/YuzukiSBC-Toolchains/releases/download/\
linaro-7.2.1/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi.tar.xz
tar xvf gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi.tar.xz
```

Debemos agregar el directorio donde se encuentran los ejecutables a la variable de entorno **PATH**, que es donde linux busca los ejecutables. Para esto debemos editar el archivo **\$HOME/.bashrc** agregando al final del archivo:

```
export PATH=$HOME/TOOLCHAIN/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-\n\gnueabi/bin:$PATH
```

**Para que esta variable se actualice debemos cerrar la terminal y abrir una nueva.**

### 5.11.2. Compilación de U-BOOT 2018

Para realizar cambios en u-boot y "sacarlo" del entorno de buildroot, procedemos con las siguientes instrucciones (la carpeta patch\_uboot se encuentra en nuestro repositorio en Develop/Linux/src/patch\_uboot)

```
cd Develop/Linux/src/patch_uboot
wget https://github.com/Tina-Linux/u-boot-2018/archive/refs/tags/v1.0.1.tar.gz

tar zxvf v1.0.1.tar.gz
cd u-boot-2018-1.0.1

patch -p1 < ../patch_uboot/0001-add-support-for-buildroot.patch
patch -p1 < ../patch_uboot/0002-fix-uboot-disable-dtc-selfbuilt.patch
patch -p1 < ../patch_uboot/0003-fix-uboot-support-for-buildroot-dts-file.patch
patch -p1 < ../patch_uboot/0004-fix-No-rule-to-make-target-sunxi_challenge.patch
patch -p1 < ../patch_uboot/0005-fix-yylloc.patch

cp ../patch_uboot/uboot_mangopi_mq_dual_defconfig configs/
cp ../patch_uboot/sun8i-mangopi-mq-dual-uboot.dts ./arch/arm/dts/
cp ../patch_uboot/sun8iw20p1-soc-system.dtsi arch/arm/dts/

make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm uboot_mangopi_mq_dual_defconfig
make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm menuconfig

select head
Command line interface --->
  Boot commands --->
    [*] bootz
    [*] go

Device Tree Control --->
  Provider of DTB for DT control --->
    [*] Provided by the board at runtime

make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm
```

Al finalizar la compilación obtenemos el archivo u-boot-sun8iw20p1.bin, que debe ser utilizado para crear el archivo *boot\_package.fex* ejecutando el comando:

```
\textit{./dragonsecboot -pack boot_package.cfg}
```

Enviarlo a la tarjeta:  
`scp boot_package.fex root@10.42.0.220:`

Con lo que obtenemos el archivo *boot\_package.fex*, el cual debemos transferir a la tarjeta usando el comando:

### Configuración de u-boot

Como se dijo anteriormente la configuración por defecto de u-boot solo permite cargar imágenes tipo ANDROID, para cargar la imagen junto con el árbol de dispositivos debemos modificar las siguientes variables de entorno de u-boot. Para acceder a la consola de u-boot es necesario oprimir una tecla cuando aparece el mensaje:

```
Hit any key to stop autoboot: 2

setenv kernel zImage
setenv dts sun8i-mangopi-mq-dual-linux.dtb
setenv boot_mmc 'fatload mmc ${mmc_dev}:${mmc_boot_part} 45000000 \
${kernel}; fatload mmc ${mmc_dev}:${mmc_boot_part} 40000000 ${dts}; \
bootz 45000000-40000000'
saveenv
```

El comando que se ejecuta de forma automática es el que se encuentra en:

```
bootcmd=run boot_check boot_mmc
```

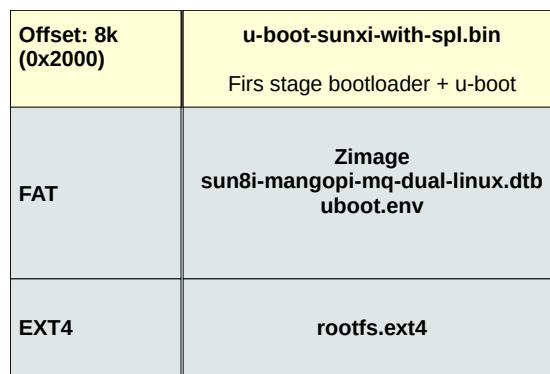
El comando *boot\_check* inicializa la memoria SD imprimiendo información de la tarjeta y su tabla de particiones

```
boot_check=sunxi_card0_probe;
```

Con los comandos *setenv* se declaran los nombres de los archivos del **kernel** y el **dts**. Los que se utilizan en el comando **boot\_mmc**, el cual lee los archivos **zImage** y **dtb** y los carga a las posiciones de memoria 45000000 y 40000000, respectivamente, y finalmente los ejecuta mediante el comando **bootz**.

### 5.11.3. U-BOOT mainline

La versión oficial de *u-boot* ya posee soporte para el procesador *T113S3*. Una ventaja adicional es que incorpora el SPL, lo que cambia de forma considerable la estructura de la imagen mostrada en la figura 5.25.



**Figura 5.29** Estructura requerida para u-boot main-line

Como podemos observar en la figura 5.29 las 5 primeras particiones se reducen a una, donde encontramos la imagen de u-boot + el bootloader de primera etapa. La segunda partición (la primera visible para Linux) se utiliza para almacenar la imagen del kernel, el archivo dtb y el entorno de u-boot. La tercera partición corresponde a una partición ext4 donde se aloja el sistema de archivos.

## Descarga de u-boot

Para descargar el código oficial de u-boot ejecutamos el siguiente comando:

```
git clone https://github.com/u-boot/u-boot.git
```

## Cambios para adaptarse a la plataformas ECB

### 5.11.3.0.1 Para plataformas con el procesador T113S4

El procesador T113S4 posee 256 MB de memoria RAM, mientras que el T113S3 posee 128M, por esto es necesario realizar la siguiente modificación:

```
En el archivo u-boot/drivers/ram/sunxi/dram_sun20i_d1.c cambiar a:  
case 10: cfg = ac_remapping_tables[0]; break;
```

### 5.11.3.0.2 Cambiar el puerto de depuración a UART0

Para fijar el puerto de depuración y la consola de u-boot a la UART0 debemos hacer los siguientes cambios:

```
En u-boot/arch/arm/dts/sunxi-d1s-t113-mangopi-mq-r.dtsi agregar:  
&uart0 {  
    uart-has-rtscts;  
    pinctrl-0 = <&uart0_pe2_pins>;  
    pinctrl-names = "default";  
    status = "okay";  
};
```

```
En u-boot/arch/riscv/dts/sunxi-d1s-t113.dtsi agregar:  
uart0_pe2_pins: uart0-pe2-pins {  
    pins = "PE2", "PE3";  
    function = "uart0";  
};
```

```
En u-boot/configs/mangopi_mq_r_defconfig cambiar a:  
CONFIG_DRAM_CLK=936  
CONFIG_SUNXI_MINIMUM_DRAM_MB=256  
CONFIG_CONS_INDEX=1
```

## Cambios necesarios para iniciar el sistema de archivos

Se deben realizar las siguientes modificaciones para que u-boot cargue la imagen del kernel, el árbol de dispositivos e inicie el sistema de archivos:

```
Modificar u-boot/include/configs/sunxi-common.h a:
```

```
#define CFG_EXTRA_ENV_SETTINGS \  
CONSOLE_ENV_SETTINGS \  
MEM_LAYOUT_ENV_SETTINGS \  
MEM_LAYOUT_ENV_EXTRA_SETTINGS \  
DFU_ALT_INFO_RAM \  
"bootcmd=run boot_mmc" "\0" \  
"mmc_bootpart=1" "\0" \  
"console=ttyS0,115200\0" \  
"dtbfile=sun8i-mangopi-mq-dual-linux.dtb" "\0" \  
"
```

```
"bootargs=mem=256M cma=72M root=/dev/mmcblk0p5 init=/sbin/init
rootwait console=tty0 console=ttyS0,115200 \0" \
"boot_mmc=fatload mmc 0:${mmc_bootpart} ${kernel_addr_r} zImage;
fatload mmc 0:${mmc_bootpart} ${fdt_addr_r} ${dtbfile};
bootz ${kernel_addr_r} - ${fdt_addr_r}\0"
```

Con lo anterior u-boot creará la variables de entorno *boot\_mmc* necesaria para cargar (desde la primera partición de la memoria SD) en memoria la imagen del kernel (zImage) el árbol de dispositivos (sun8i-mangopi-mq-dual-linux.dtb) e inicie el kernel con los parámetros definidos en *bootargs*, esto es, consola en la *UART0* y el sistema de archivos en la segunda partición de la memoria SD.

### Compilación y carga de u-boot

Una vez realizados estos cambios procedemos a compilar u-boot y a cargar el binario en la memoria SD:

```
export PATH=$PATH:/home/carlos/TOOLCHAIN/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi/bin
make CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm mangopi_mq_r_defconfig
make CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm menuconfig
make CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm

sudo dd if=u-boot-sunxi-with-spl.bin of=/dev/sdX bs=1024 seek=8
```

La memoria SD debe tener la siguiente tabla de particiones:

Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/sda1		4096	106495	102400	50M	b	W95 FAT32
/dev/sda2		106496	31563775	31457280	15G	83	Linux
/dev/sda3		31563776	62333951	30770176	14,7G	83	Linux

## 5.12. Compilación del kernel sin buildroot

Para dejar de utilizar el entorno *buildroot*, es necesario compilar el kernel desde los fuentes, para ello debemos ejecutar los siguientes comandos:

```
cd
wget https://github.com/Tina-Linux/linux-5.4/archive/refs/tags/1.0.0.tar.gz
tar zxvf 1.0.0.tar.gz
cd linux-5.4-1.0.0
patch -p1 < ../../dts_files/kernel5/0001-rename-linux-dtb-build-file.patch
```

```
cp ../../dts_files/kernel5/sun8i-mangopi-mq-dual-linux.dts \
arch/arm/boot/dts/sun8i-mangopi-mq-dual-linux.dts
cp ../../dts_files/kernel5/sun8iw20p1-linux.dtsi \
arch/arm/boot/dts/sun8iw20p1-linux.dtsi
cp ../../dts_files/kernel5/config_eth_lcd_kernel_5.4 .config

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- oldconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

## 5.13. Aplicaciones gráficas

El procesador T113 posee un periférico dedicado para manejo de LCDs, sin embargo, estos LCDs resultan poco prácticos para aplicaciones con limitación de espacio. En la actualidad existe una gran gama de pantallas de cristal líquido que trabajan con una interfaz SPI, lo que simplifica su conexión, este tipo de LCDs utilizan el driver FBTFT de Linux. El tamaño de estos LEDs varía de 2 a 3 pulgadas y la resolución es del orden de 320x240.

### 5.13.1. Configuración del driver FBTFT para el LCD ILI9340

Para dar soporte a esta familia de LCDs debemos seleccionar los siguientes items en la herramienta de configuración de Linux:

```
Device Drivers
[*] Staging drivers --->
    <*> Support for small TFT LCD display modules --->
        <*> FB driver for the ST7789V LCD Controller
        <*> FB driver for the ILI9340 LCD Controller

Device Drivers
Graphics support --->
Frame buffer Devices --->
    <*> Support for frame buffer devices --->
Video support for sunxi --->
    [*] Framebuffer Console Support(sunxi)
    < > DISP Driver Support(sunxi-disp2)
Console display driver support --->
    [*] Framebuffer Console support
    [*] Map the console to the primary display device
```

#### Cambios en el código fuente del driver fbtft

El driver de la distribución utilizada no puede importar los pines declarados en el árbol de dispositivos y genera una advertencia indicando que estos pines no existen, estos pines son la señal de *reset* y el pind *dc*, utilizado para indicarle al display si la información enviada es un dato o un comando. Para solucionar esto se debe realizar el siguiente cambio en el archivo *drivers/staging/fbtft/fbtft-core.c*, esto es reemplazar el contenido de la función *fbtft\_request\_one\_gpio* e incluir el encabezado *linux/of\_gpio.h*.

```
#include <linux/of_gpio.h>

static int fbtft_request_one_gpio(struct fbtft_par *par,
    const char *name, int index,
    struct gpio_desc **gpiop)
{
    struct device *dev = par->info->device;
    struct device_node *node = dev->of_node;
    int gpio, flags, ret = 0;
    enum of_gpio_flags of_flags;
    if (of_find_property(node, name, NULL)) {
        gpio = of_get_named_gpio_flags(node, name, index, &of_flags);
        if (gpio == -ENOENT)
            return 0;
```

```

    if (gpio == -EPROBE_DEFER)
        return gpio;
    if (gpio < 0) {
        dev_err(dev,
                "failed to get '%s' from DT\n", name);
        return gpio;
    }
    //active low translates to initially low
    flags = (of_flags & OF_GPIO_ACTIVE_LOW) ? GPIOF_OUT_INIT_LOW :
                                                GPIOF_OUT_INIT_HIGH;
    ret = devm_gpio_request_one(dev, gpio, flags,
                                dev->driver->name);
    if (ret) {
        dev_err(dev,
                "gpio_request_one('%s'=%d) failed with %d\n",
                name, gpio, ret);
        return ret;
    }

    *gpiop = gpio_to_desc(gpio);
    fbtft_par_dbg(DEBUG_REQUEST_GPIOS, par, "%s: '%s' = GPIO%d\n",
                   __func__, name, gpio);
}

return ret;
}

```

### 5.13.2. Adición del LCD al árbol de dispositivos

En la figura 5.31 se muestra la forma de instanciar el periférico **spi0**. En la declaración del periférico **spi0** se deben definir los pines que se utilizarán para el protocolo, es este caso se definen las estructuras *spi0\_pins\_lcd* y *spi0\_pins\_lcd.cs*, las cuales se deben declarar en la sección **&pio**.

A continuación se declara el dispositivo que se conectará a este puerto, en este caso, un LCD *ilitek ili9340* compatible, acá definimos dos pines adicionales para el control del lcd (**reset**, **dc**), en esta caso se asignan a los pines PC6 y PG9

#### Cargar la imagen del kernel y el árbol de dispositivos a la SD

Como se ve en la figura 5.25 U-boot lee el archivo sun8i-mangopi-mq-dual-linux.dtb desde el archivo *boot\_package.fex*. Por lo tanto para incorporar cambios en el DTB debemos crear este archivo y copiarlo en la posición (0x1004000: 16400k) de la tarjeta. Para esto construiremos este archivo y lo enviaremos a la placa vía red cableada con **scp**.

```

cd ecb_T8_T113/Develop/Linux/src/images
cp xxxx/linux-5.4-1.0.0/arch/arm/boot/dts/sun8i-mangopi-mq-dual-linux.dtb .
cp xxxx/linux-5.4-1.0.0/arch/arm/boot/zImage .
sescp zImage boot_package.fex root@10.42.0.220:

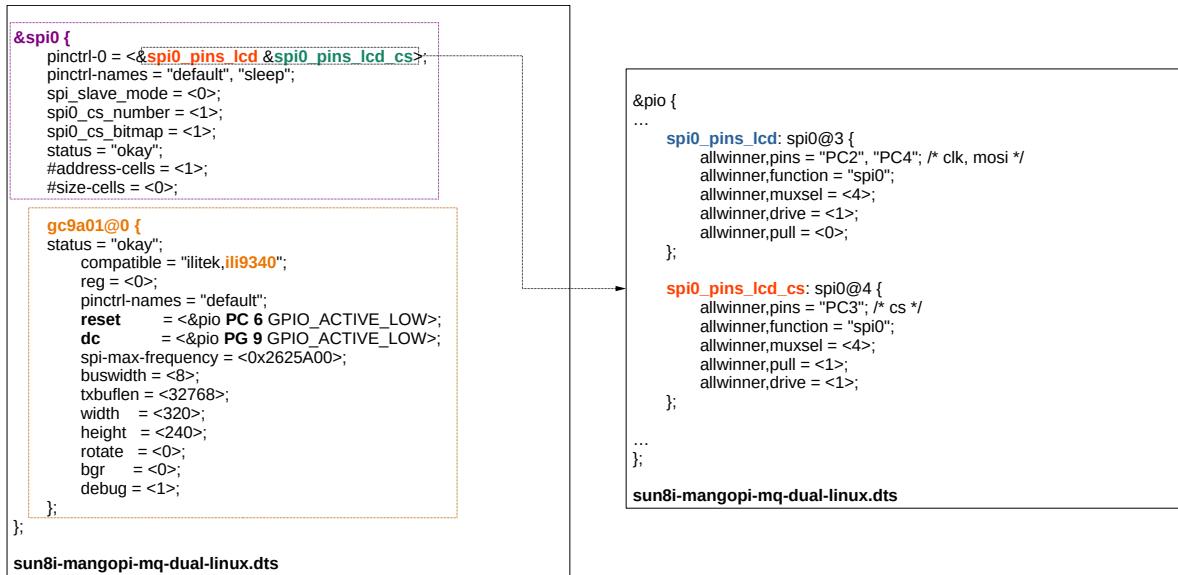
```

Lo anterior copia el archivo *boot\_package.fex* en */root/*. Para grabarlo en la SD debemos ejecutar:

```

dd if=boot_package.fex of=/dev/mmcblk0 bs=1k seek=16400
mount /dev/mmcblk0p4 /mnt/
cp zImage /mnt/
halt

```



**Figura 5.30** Instanciación del periférico spi0 con el driver ili9340

Después de oprimir el botón de reset, se cargarán los nuevos archivos y durante el proceso de boot se mostrarán los siguientes mensajes:

```

[] fbtft_of_value: width = 320
[] fbtft_of_value: height = 240
[] fbtft_of_value: buswidth = 8
[] fbtft_of_value: debug = 1
[] fbtft_of_value: rotate = 0
[] fbtft_of_value: txbuflen = 32768
[] fb_ili9340 spi0.0: fbtft_request_one_gpio: 'reset' = GPIO70
[] fb_ili9340 spi0.0: fbtft_request_one_gpio: 'dc' = GPIO201
[] graphics fb0: fb_ili9340 frame buffer, 320x240, 150 KiB video memory,
  32 KiB buffer memory, fps=20, spi0.0 at 40 MHz

```

En el LCD se debe desplegar dos veces Tux (se tienen dos procesadores) y un tiempo después se mostrará el mensaje

```

Debian GNU/Linux 12 verivolt tty1
verivolt login:

```

## 5.14. configuración de la red en Debian

Para permitir la conexión desde el computador a la tarjeta es necesario configurar el PC y la placa.

### 5.14.1. Configuración de ssh en el PC

Debemos crear un archivo de configuración (.ssh/config) en el host (x86) con el siguiente contenido (la dirección IP puede variar si no se aplica una regla de configuración al router):

```

Host 10.42.0.220
HostName 10.42.0.220

```

```
User root
Port 22
IdentityFile ~/.ssh/id_rsa
HostKeyAlgorithms +ssh-rsa
PubkeyAcceptedKeyTypes +ssh-rsa
```

### **5.14.2. Configuración de ssh en ECB\_T8\_T113**

Debemos conectarnos por el puerto serial a la tarjeta y modificar los siguientes archivos:

```
/etc/ssh/sshd_config
PasswordAuthentication yes
ChallengeResponseAuthentication no

/etc/ssh/sshd_config:
PermitRootLogin yes
PubkeyAuthentication no
```

Finalmente debemos reiniciar el daemon ssh:

```
/etc/init.d/ssh restart
```

### **5.14.3. Aplicación con LVGL**

LVGL es una librería gráfica abierta para crear interfaces de usuario para diferentes procesadores. Existe en Internet una gran cantidad de tutoriales y ejemplos e uso de esta librería, en esta sección mostraremos como crear una interfaz utilizando la aplicación SquareLine Studio (<https://squareline.io/>).

SquareLine permite crear de forma rápida la interfaz y genera el código necesario para ser utilizado en el sistema Embebido. En el directorio *ecb\_T8\_T113/lvgl* encontramos los siguientes directorios:

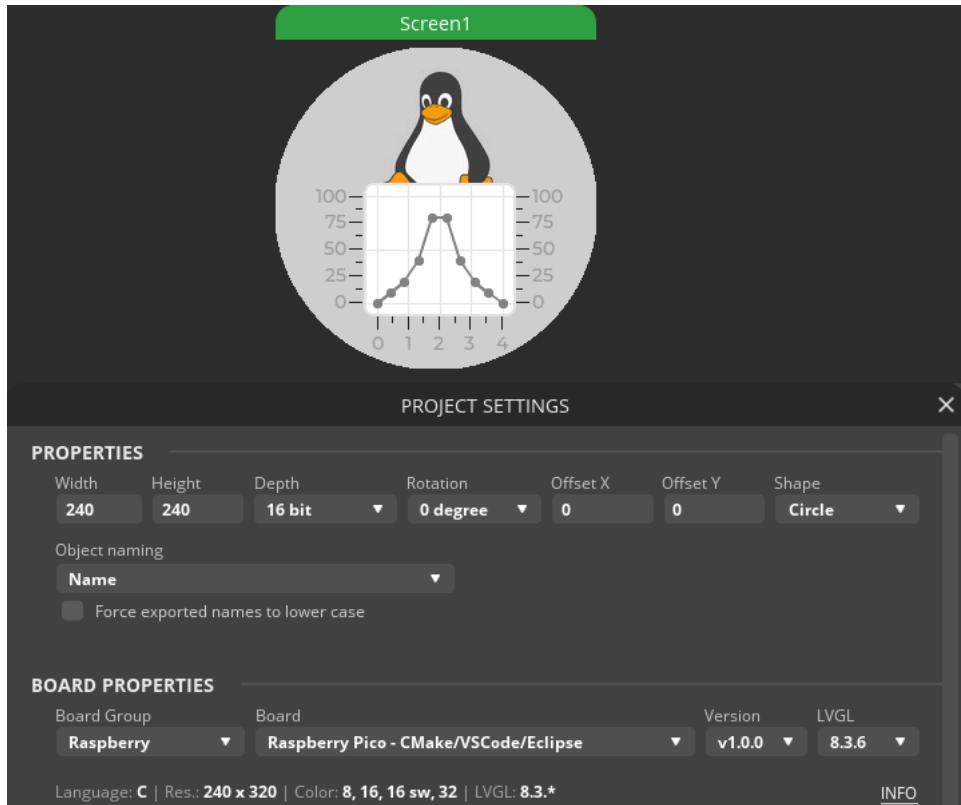
```
|-- demo      : Ejemplo basado en SquareLine
|-- gui       : Ejemplo de lvgl adaptado a Linux
|-- Gui       : Ejemplo de lvgl adaptado a Linux
|-- littlevg1-8 : Código fuente de la librería lvgl adaptado a Linux FB
|-- lvgl_image_converter :
```

## **5.15. Configurando el sonido con ALSA**

```
amixer set "Headphone" unmute; amixer set "Headphone volume" 80%
```

```
DAC volumen 63
digital 100
```

```
Simple mixer control 'digital volume',0
    Capabilities: volume volume-joined
    Playback channels: Mono
```



**Figura 5.31** Ejemplo de configuración para SquareLine Studio

Capture channels: Mono

Limits: 0 – 63

Mono: 63 [100%]

```
Simple mixer control 'DAC volume',0
Capabilities: volume
Playback channels: Front Left - Front Right
Capture channels: Front Left - Front Right
Limits: 0 - 255
Front Left: 171 [67%]
Front Right: 171 [67%]
```

```
ffmpeg -i i-miss-the-rage-wav-194890.mp3 -acodec pcm_s16le -ac 1 -ar 16000 output.wav
```

explicar los ejemplo, los archivos de configuración, lv\_drv\_conf.h

```
%https://tina.100ask.net/SdkModule/Linux_AudioFrequency_DevelopmentGuide-02/#274-audiocoo
```

## 5.16. Módulos del kernel

Los módulos son pequeñas piezas de código que pueden ser cargadas y descargadas en el kernel en el momento que sea necesario. Ellos extienden la funcionalidad del kernel, sin la necesidad de reiniciar el sistema y recompilar el kernel. Por ejemplo, un tipo de módulo es el controlador de dispositivo, el cual permite al kernel acceder a un dispositivo hardware conectado al sistema. Este tipo de módulos serán estudiados en esta sección.

Existen tres tipos de dispositivos en Linux [8]:

- Carácter: Puede accederse de forma similar a un archivo; este tipo de dispositivos permite por lo menos las operaciones *open*, *close*, *read*. Ejemplos de este tipo de dispositivo son el puerto serie (*/dev/ttyS0*) y la consola (*/dev/console*)
- Bloque: Este tipo de dispositivo puede hospedar un sistema de archivos; normalmente realiza operaciones de bloques de datos únicamente; un ejemplo de este tipo de dispositivo es el disco duro (*/dev/hda*).
- Red: Toda transacción de red se realiza a través de una interfaz, esto es, un dispositivo hardware (*/dev/eth0*) o software (*loopback*) capaz de intercambiar datos con otros hosts.

### 5.16.1. Ejemplo de un driver tipo caracter

Recuerde que una aplicación en el área de usuario, no puede acceder directamente al área del kernel; los dispositivos se acceden a través de archivos de dispositivos, localizados en */dev* (ver figura 5.32). A continuación se muestra la salida del comando *ls -l /dev*

```
brw-rw---- 1 root disk      3,   0 Nov 27 hda
brw-rw---- 1 root disk      3,   1 Nov 27 hda1
brw-rw---- 1 root disk      3,   2 Nov 27 hda2
crw-rw---- 1 root uucp     4,  64 Nov 27 ttyS0
crw-rw---- 1 root uucp     4,  65 Nov 27 ttyS1
```

Los archivos tipo carácter están identificados por una “*c*” en la primera columna, mientras que los dispositivos tipo bloque por una “*b*”. Podemos observar que existen dos números (5ta y 6ta columna) que identifican al driver, el número de la 5ta columna recibe el nombre de *major number* y el de la sexta *minor number*; estos números son utilizados por el sistema operativo para determinar el dispositivo y el driver que deben ser utilizados ante una solicitud a nivel de usuario.

El *major number* identifica la clase o grupo del dispositivo, mientras que el *minor number* se utiliza para identificar sub-dispositivos (Ver Figura 5.32).

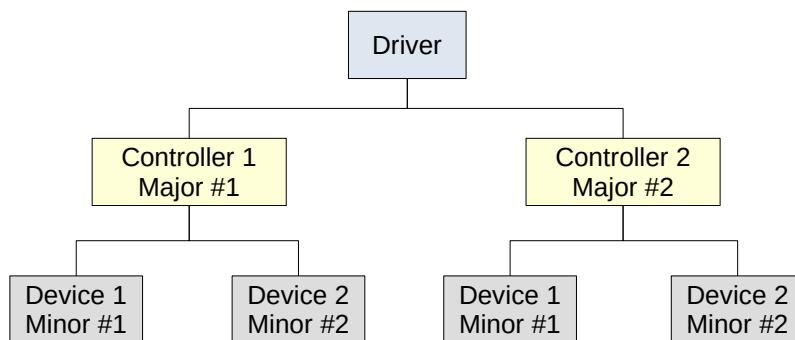


Figura 5.32 Números *major* y *minor* de un driver

El kernel de Linux permite que los drivers comparten el número mayor, como el caso del disco duro, *hda* posee dos particiones *hda1* y *hda2*, las que son manejadas por el mismo driver, pero se asigna un número menor único a cada una; lo mismo sucede con el puerto serie.

### Implementación del driver de un LED

A continuación se realizará la descripción de un driver tipo carácter para un dispositivo muy sencillo, un LED. Este ejemplo realiza las siguientes operaciones:

- *init*: Se ejecuta cuando se carga el módulo, el LED se encenderá.
- *open*: Se ejecuta cuando se abre el archivo que realiza la interfaz con el driver en una operación de lectura o escritura. El LED parpadeará 5 veces y quedará encendido.
- *release*: Se ejecuta cuando se cierra el archivo que hace la interfaz con el módulo después de una operación de lectura o escritura. El LED se apagará.
- *exit*: Se ejecuta cuando se descarga el módulo, el LED se apagará.

Existen dos funciones que deben estar presentes en todo tipo de módulo, estas son: *module\_init* y *module\_exit* las cuales se ejecutan cuando se carga y descarga el módulo respectivamente (ver figura).

```
#define PINID_GPMI_D07 MXS_PIN_ENCODE(0, 7)
#define LED_PIN 202 // PG10 6*32+10 202
#define SUCCESS 0
#define DEVICE_NAME "blink" /* Dev name as it appears in /proc/devices */
#define BUF_LEN 80 /* Max length of the message from the device */
static int is_device_open = 0; /* Used to prevent multiple access to device */
static int Major;

struct file_operations fops = {
    .open = device_write,
    .write = device_open,
    .release = device_release,
};

static int __init blink_init(void)
{
    int ret;
    printk(KERN_INFO "BLINK module is Up.\n");
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
        return Major;
    }
    printk(KERN_ALERT "I was assigned major number %d. To talk to\n", Major);
    printk(KERN_ALERT "the driver, create a dev file with\n");
    printk(KERN_ALERT "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
    ret = gpio_request(LED_PIN, "led");
    if (ret) {
        pr_err("fail request LED pin\n");
        return -1;
    }
    gpio_direction_output(LED_PIN, 0);
    gpio_set_value(LED_PIN, 1);
    return 0;
}

static void __exit blink_exit(void)
{
    gpio_set_value(LED_PIN, 0);
    gpio_free(LED_PIN);
    unregister_chrdev(Major, DEVICE_NAME);
    printk(KERN_INFO "BLINK driver is down...\n");
}

module_init(blink_init);
module_exit(blink_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Carlos Camargo <cicamargoba@gmail.com>");
MODULE_DESCRIPTION("BLINKER LED driver");
MODULE_VERSION("1:0.1");
```

```
static ssize_t
device_write(struct file *filp, const char *buff, size_t count, loff_t * off)
{
    char cmd;
    if (copy_from_user( &cmd, buff, 1 )) {
        return -EFAULT;
    }
    if(cmd=='Q')
    {
        printk(KERN_INFO "Q...\n");
        gpio_set_value(LED_PIN, 1);
    }
    else
        if(cmd=='S'){
            printk(KERN_INFO "S...\n");
            gpio_set_value(LED_PIN, 0);
        }
    return 1;
}
```

```
static int device_open(struct inode *inode, struct file *file)
{
    unsigned int i;
    printk( KERN_INFO "Open BLINKER\n" );
    if (is_device_open)
        return -EBUSY;
    is_device_open = 1;
    for( i=0; i<5; i++ ){
        gpio_set_value(LED_PIN, 0);
        mdelay(0x0040);
        gpio_set_value(LED_PIN, 1);
        mdelay(0x0040);
    }
    try_module_get(THIS_MODULE);
    return SUCCESS;
}
```

```
static int device_release(struct inode *inode, struct file *file)
{
    is_device_open = 0;
    module_put(THIS_MODULE);
    printk( KERN_INFO "Close BLINKER\n" );
    return 0;
}
```

**Figura 5.33** Módulo blink

Las funciones *module\_init* *module\_exit* deben ser declaradas como *static* ya que no serán visibles fuera del archivo. Como puede observarse se hace la definición de las funciones que deben ejecutarse al cargar y descargar el módulo en nuestro caso *blink\_init* y *blink\_exit* respectivamente. La información sobre el módulo aparece al final.

Como se mencionó anteriormente, todos los módulos del kernel tienen asociado un dispositivo y un archivo en el directorio `/dev`. Por lo tanto, es necesario definir una estructura de operaciones de archivo del módulo (`struct file_operations fops`); cada campo de la estructura corresponde a una función definida por el driver para manejar una solicitud determinada. En nuestro caso existen las funciones `open` y `release`.

Como se puede ver en la figura 5.32 es necesario que el kernel sepa que driver está encargado del dispositivo, esto es, el *major number* del driver que lo maneja; por esto, la primera acción de la función `blink_init` es obtener este número. La función `register_chrdev` retorna el número mayor asignado de forma dinámica, esto es recomendable ya que si se fijara un número de forma arbitraria, podría causar conflictos con otros dispositivos. De esta forma, al cargar el módulo con el comando: `insmod blinker.ko`, el LED se encenderá y aparecerá el siguiente mensaje en la consola:

```
BLINK module is Up.
I was assigned major number 253. To talk to
the driver, create a dev file with
'mknod /dev/blink c 253 0'.
```

Con lo anterior, nuestro dispositivo es registrado y se le asigna el número 253 (o el que asigne el kernel) como *major number*. En el archivo `/proc/devices` aparecen los dispositivos que están siendo utilizados por el kernel, este archivo debe contener una entrada para blink de la forma: `253 blink`.

En la función `blink_exit` se realiza la liberación del dispositivo (la función `unregister_chrdev`), (la que se ejecuta cuando se lanza el comando: `rmmmod blinker.ko`), se apaga el LED y se imprime en la consola el mensaje:

```
BLINK driver is down...
```

Como se mencionó anteriormente es posible manejar un archivo tipo carácter como si fuera un archivo de texto, por lo tanto, es posible adicionar funciones a las acciones de abrir, cerrar, escribir en o leer del dispositivo.

La función `device_open` se ejecuta cada vez que el archivo de dispositivo `/dev/blink` es abierto, esto sucede en operaciones de lectura o escritura. Es decir si se utilizan los siguientes comandos:

```
more /dev/blink
cat file > /dev/blink
cp file /dev/blink
```

La función `device_release` se ejecuta cada vez que se cierre el archivo de dispositivo `/dev/blink`, después de una operación de lectura o escritura.

Con cada acceso de lectura o escritura el LED se parpadeará 5 veces y en la consola se despliega el mensaje:

```
Open BLINKER
Close BLINKER
```

## 5.16.2. Instalación/desinstalación automatizada del driver

### Systemd

`Systemd` es un sistema y administrador de servicios, escrito por Lennart Poettering diseñado para el kernel de Linux, con el fin de sustituir el sistema de inicio de los sistemas operativos UNIX para unificar los comportamientos de servicios y configuraciones básicas en todas las distribuciones.

Al ser el primer proceso en ejecutarse en el espacio de usuario en el inicio de Linux, es el proceso padre de todos los procesos de espacio de usuario. `Systemd` ofrece las siguientes características:

- Utiliza socket para ofrecer una paralelización agresiva y poder acelerar el arranque iniciando más procesos en paralelo, para ello crea todos los sockets para todos los demonios en un solo paso en el sistema de inicio (`init`) y en un segundo paso ejecuta a la vez todos los demonios.
- Mantiene puntos de montaje y auto montaje.
- Ofrece inicio de demonios bajo demanda, realiza el seguimiento y rastreo de procesos utilizando cgroups, el cual es una característica de Linux que aísla, limita y representa los recursos usados de una colección de procesos.
- Soporta copia instantánea y restauración de volúmenes (snapshot) y la restauración de estado de sistemas.
- Implementa un elaborado servicio lógico de control transaccional basado en la dependencia.

`Systemd` se basa en la noción de unidades, cada una de ellas compuestas de un nombre, tipo y coincidencia de un archivo de configuración, existen siete tipos de unidades:

1. Service: Demonios que pueden ser iniciados, detenidos, reiniciados o recargados.
2. Mount: Unidad que encapsula un punto de montaje en la jerarquía del sistema de archivos.
3. Automount: Unidad que encapsula un punto de montaje automático en la jerarquía del sistema de archivos, cada unidad automount tiene una unidad mount correspondiente.
4. Target: Se utiliza para agrupación lógica de unidades, no hace nada por sí misma, sino que hace referencia a otras unidades.
5. Snapshot: Similar a las unidades target, su único propósito es hacer referencia a otras unidades.
6. Device: Encapsula un dispositivo en el árbol de dispositivos de Linux. Si un dispositivo está marcado por las reglas de *udev*, se expondrá como una unidad device en Systemd.
7. Socket: Encapsula un socket en el sistema de archivos o en internet, cada unidad de este tipo tiene una unidad de servicio correspondiente, que se inicia si la primera conexión entra en el socket.

### Carga del módulo con un servicio Systemd

El siguiente script controla la carga y descarga de nuestro módulo blink.

```
[Unit]
Description=Carga modulo blink, crea /dev/blink
[Service]
Type=oneshot
ExecStart=/sbin/insmod /root/blink.ko
RemainAfterExit=true
ExecStop=/sbin/rmmod blink

[Install]
WantedBy=multi-user.target
```

Este módulo debe ser copiado en el directorio /lib/systemd/system:

```
cp blink.service /lib/systemd/system
```

Para activar el servicio se debe ejecutar el comando:

```
systemctl enable blink.service
```

Al reiniciar el sistema el servicio se ejecutará automáticamente.

### 5.16.3. Ejemplo Módulo de driver I2C

En esta ocasión mostraremos como escribir un driver que controle un dispositivo I2C que realiza una función específica, el sensor BMP280. En la figura 5.34 se muestra el código de este módulo, al igual que el módulo *blink* debemos declarar las funciones para la inicialización y salida del módulo, *ModuleInit* y *ModuleExit*.

La función *ModuleInit* hace un llamado a *alloc\_chrdev\_region* quien asigna un rango de números de dispositivos tipo carácter. El número mayor se escogerá de forma dinámica, el formato de esta función es:

```
int alloc_chrdev_region(dev_t * dev, unsigned baseminor, unsigned count, const char * name)
donde:
    dev Parámetro de salida para el primer número asignado
    baseminor Primer rango solicitado de números menores
    count Cantidad de números menores requeridos.
    name El nombre del dispositivo asociado.
```

A continuación la función *class\_create* crea una clase para ser utilizada en la creación del dispositivo la función. Las funciones *device\_create()* inicializa la estructura del dispositivo asignando la estructura de clase genérica asignada y el dispositivo recibido como parámetro. Adicionalmente, creará un atributo de la clase, dev, la cual contiene los número mayor y menor del dispositivo. Con esto una aplicación como udev puede crear un nodo en el directorio /dev. Existen dos formas de asignar e inicializar estas estructuras. Si se desea obtener una estructura *cdev* autónoma en tiempo de ejecución, se debe definir de la siguiente forma:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Si se desea embeber la estructura *cdev* dentro de una estructura específica de un dispositivo, se debe inicializar la estructura:

```
cdev_init(&myDevice, &fops);
```

Una vez definida la estructura *cdev*, el paso final consiste en informarle al kernel sobre ella:

```
cdev_add(&myDevice, myDeviceNr, 1)
```

Se deben tener en cuenta un par de consideraciones al usar *cdev\_add*. La primera es que el llamado puede fallar, si esto pasa, retornará un código de error negativo, el dispositivo no será adicionado al sistema. Sin embargo, casi siempre tiene éxito y eso trae a colación el otro punto: tan pronto como *cdev\_add* retorna, el dispositivo está "activo" y el kernel puede llamar sus operaciones. No se debe llamar a *cdev\_add* hasta que su controlador esté completamente listo para manejar las operaciones en el dispositivo.

A continuación y conociendo el controlador I2C que se va a utilizar se llama a *i2c\_get\_adapter()* para obtener el adaptador de la estructura del controlador I2C. Esta es una estructura que almacena la información del adaptador I2C basada en el número del bus. *i2c\_new\_device*, finalmente, *i2c\_add\_driver* registra el módulo i2c.

Una vez registrado el driver se pueden utilizar las funciones *i2c\_smbus\_read\_byte\_data*, *i2c\_smbus\_read\_word\_data*, *i2c\_smbus\_write\_byte\_data* para comunicarse con el dispositivo.

Al insertar este módulo con **insmod bmp280.ko** obtendremos los siguientes mensajes:

```
MyDeviceDriver - Hello Kernel
MyDeviceDriver - Device Nr 255852544 was registered
BMP280 Driver added!
ID: 0x58
```

## 5.17. comunicación con periféricos desde espacio de usuario

En la figura 5.35 se muestra un ejemplo de comunicación desde espacio de usuario a un módulo del kernel. Todo módulo proporciona funciones tipo operaciones de archivo (*file\_operations*), estas operaciones permiten transmitir información desde (*copy\_from\_user*) y hacia (*copy\_to\_user*) el espacio de usuario.

## 5.18. Interrupciones en módulos

A continuación se describirá la forma de manejar las interrupciones utilizando un driver de Linux, en este caso usaremos el GPIO PD14 (110) como fuente de interrupción. A continuación se muestra la inicialización de este módulo:

```
static int __init qem_init(void)
{
    int res, ret;
    printk(KERN_INFO "IRQ_module_is_Up.\n");
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
        printk(KERN_ALERT "Registering _char_device_ failed _with_%d\n", Major);
        return Major;
    }
    printk(KERN_ALERT "I_was_assigned_major_number_%d,_To_talk_to_\n", Major);
    printk(KERN_ALERT "the_driver,_create_a_dev_file_with_\n");
    printk(KERN_ALERT "'mknod_/dev/%s,c,%d'.\n", DEVICE_NAME, Major);
```

```

ret = gpio_request(IRQ_PIN, "led");
if (ret) {
    pr_err("fail_request_LED_pin\n");
    return -1;
}
gpio_direction_input(IRQ_PIN);
irq_number = gpio_to_irq(IRQ_PIN);
res = request_irq(irq_number, irq_handler, IRQF_TRIGGER_RISING, "IRQ", NULL);
return 0;
}

```

Esta rutina es similar a la presentada en el módulo *blink*, solo se agregan un par de instrucciones para definir el pin del procesador como señal *IRQ*, en la línea:

```

irq_number = gpio_to_irq(IRQ_PIN);
res = request_irq(irq_number, irq_handler, IRQF_TRIGGER_RISING, "IRQ", NULL);

```

Se hace un llamado a la función *request\_irq* la cual, asigna recursos a la interrupción, habilita la función que se ejecutará cuando se presente una interrupción (*irq\_handler*) y la fuente de interrupción. En nuestro caso define el pin *IRQ\_PIN* como la línea de interrupción, el flag *IRQF\_TRIGGER\_RISING* define un flanco de subida como trigger, el nombre del dispositivo que realiza la interrupción es *IRQ*.

La función que se ejecuta cuando se presenta la interrupción es:

```

static irqreturn_t irq_handler(int irq, void *dev_id)
{
    if(irq_enabled)
    {
        interrupt_counter++;
        printk(KERN_INFO "interrupt_counter=%d\n", interrupt_counter);
        wake_up_interruptible(&wq);
    }
    return IRQ_HANDLED;
}

```

Cada vez que se produce una interrupción y si la variable global *irq\_enabled* es igual a 1, se aumenta en 1 el valor de *interrupt\_counter*, se imprime su valor.

En este driver utilizaremos la función *device\_read* para enviar información a un programa en espacio de usuario.

```

static ssize_t
device_read(struct file *filp, char *buffer, size_t count, loff_t * offset)
{
    if(irq_enabled){
        wait_event_interruptible(wq, interrupt_counter!=0);
        copy_to_user ( buffer, &interrupt_counter, sizeof(interrupt_counter) );
    }
    return sizeof(interrupt_counter);
}

```

Cuando se realice una operación de lectura desde espacio de usuario, el proceso quedará bloqueado por la función *wait\_event\_interruptible* hasta que la rutina de atención a la interrupción ejecute la función *wake\_up\_interruptible*, pero es necesario que se cumpla la condición evaluada por *wait\_event\_interruptible* para que se ejecute la tarea. Para este ejemplo:

```

wait_event_interruptible(wq, interrupt_counter!=0);

```

Por lo que *irq\_handler* debe hacer:

```

interrupt_counter++;
wake_up_interruptible(&wq);

```

Si no se hace esto el proceso nunca se despertará y el proceso de lectura quedará bloqueado. En la línea:

```

copy_to_user ( buffer, &interrupt_counter, sizeof(interrupt_counter) );

```

Se utiliza la función *copy\_to\_user* para intercambiar información con el programa que se ejecuta en espacio de usuario. En este caso se copia a *char \*buffer*, la variable *interrupt\_counter*.

Inicialmente la variable *irq\_enabled* tiene un valor de 0, para cambiar su valor se utiliza la función *device\_write*, la que permite pasar información al driver desde espacio de usuario. A continuación se muestra esta función:

```

static ssize_t
device_write(struct file *filp, const char *buff, size_t count, loff_t * off)
{
    char cmd ;
    if (copy_from_user( &cmd, buff, 1 )) {
        return -EFAULT;
    }
    if(cmd=='Q')
    {
        irq_enabled = 1;
        printk(KERN_INFO "irq_enabled...\n");
    }
    else
        if(cmd=='S'){
            irq_enabled=0;
            printk(KERN_INFO "irq_disabled.\n");
        }
    return 1;
}

```

La información que el usuario escribe en el driver se encuentra disponible en la variable *buff*, esta función permite escribir cualquier cantidad de información, el tamaño de la información escrita desde espacio de usuario es almacenado en la variable *count*. En este ejemplo, si se escribe el carácter “Q” *irq\_enabled* será igual a **1** y si se escribe “S” *irq\_enabled* será igual a **0**.

En la función *qem\_exit* se liberan los recursos de la interrupción y la el gpio utilizados.

```
static void __exit qem_exit(void)
{
    free_irq(irq_number, NULL);
    gpio_free(IRQ_PIN);
    unregister_chrdev(Major, DEVICE_NAME);
    printk(KERN_INFO "IRQ_driver_is_down...\n");
}
```

En la figura 5.36 se muestra el código fuente del módulo *irq*.

Es posible verificar la correcta definición del módulo con interrupciones si se observa el archivo */proc/interrupts*, en este debe existir una entrada para la interrupción con nombre **IRQ**.

## 5.19. Instrumentos musicales electrónicos

En esta sección realizaremos un ejemplo de aplicación de Linux en la plataforma EC\_SAXO, la cual es una simplificación de la plataforma ecb\_T8\_T113. En la cual se ha eliminado la totalidad del circuito de la FPGA y solo se ha dejado el procesador con un amplificador para la salida de audio (ver figura 5.37)

### 5.19.1. Teclado Matricial

Para permitir simular instrumentos digitales es necesario proporcionar formas de ingresar eventos externos, para esto utilizaremos un teclado matricial de pulsadores y lo configuraremos en Linux para que sea visto como un dispositivo de entrada (*/dev/input*). Para esto debemos desplegar el menú de configuración de Linux y eleccionar *GPIO driven matrix keypad support*:

```
Device Drivers
  Input device support --->
    -*- Matrix keymap support library
    [*] Keyboards --->
      <*> GPIO driven matrix keypad support
```

El kernel 5.4-1.0.0 proporcionado por el proyecto Tina-Linux coloca un pull-up en las filas del teclado, y debe ajutarse a la configuración del teclado utilizado, en este caso utilizaremos pull-down, para esto, se debe realizar el siguiente cambio:

```
En el archivo linux-5.4-1.0.0/drivers/input/keyboard/matrix_keypad.c
cambiar a:
/* pull up row gpios for sunxi platform */
config_set = pinconf_to_config_packed(PIN_CONFIG_BIAS_PULL_UP, config_arg);
```

Por último debemos agregar el teclado matricial en el árbol de dispositivos:

```
En el archivo linux-5.4-1.0.0/arch/arm/boot/dts/sun8iw20p1-linux.dtsi
adicionar:
keyboard {
    compatible = "gpio-matrix-keypad";
    col-scan-delay-us = <500>;
    debounce-delay-ms = <100>;
    status = "okay";
    wakeup-source;
```

```

linux,no-autorepeat;
drive_inactive_cols;
row-gpios = <&pio PD 16 0 &pio PD 14 0 &pio PD 13 0 &pio PD 10 0
              &pio PD 21 0 &pio PD 6 0 &pio PD 4 0 &pio PD 2 0>;
col-gpios = <&pio PD 8 GPIO_ACTIVE_HIGH &pio PD 0 GPIO_ACTIVE_HIGH
              &pio PD 20 GPIO_ACTIVE_HIGH>;
allwinner,pull = <1>;
bias-pull-down;
linux,keymap = <
    MATRIX_KEY(0, 0, KEY_Z) /* COLO, ROW0 */
    MATRIX_KEY(1, 0, KEY_X) /* COLO ROW1 */
    MATRIX_KEY(2, 0, KEY_C) /* COLO ROW2 */
    MATRIX_KEY(3, 0, KEY_V) /* COLO ROW3 */
    MATRIX_KEY(4, 0, KEY_B) /* COLO ROW4 */
    MATRIX_KEY(5, 0, KEY_N) /* COLO ROW5 */
    MATRIX_KEY(6, 0, KEY_M) /* COLO ROW6 */
    MATRIX_KEY(7, 0, KEY_Q) /* COLO ROW7 */
    MATRIX_KEY(0, 1, KEY_W) /* COL1, ROW0 */
    MATRIX_KEY(1, 1, KEY_E) /* COL1, ROW1 */
    MATRIX_KEY(2, 1, KEY_R) /* COL1, ROW2 */
    MATRIX_KEY(3, 1, KEY_T) /* COL1, ROW3 */
    MATRIX_KEY(4, 1, KEY_Y) /* COL1, ROW4 */
    MATRIX_KEY(5, 1, KEY_U) /* COL1, ROW5 */
    MATRIX_KEY(6, 1, KEY_I) /* COL1, ROW6 */
    MATRIX_KEY(7, 1, KEY_O) /* COL1, ROW7 */
    MATRIX_KEY(0, 2, KEY_A) /* COL2, ROW0 */
    MATRIX_KEY(1, 2, KEY_S) /* COL2, ROW1 */
    MATRIX_KEY(2, 2, KEY_D) /* COL2, ROW2 */
    MATRIX_KEY(3, 2, KEY_F) /* COL2, ROW3 */
    MATRIX_KEY(4, 2, KEY_G) /* COL2, ROW4 */
    MATRIX_KEY(5, 2, KEY_H) /* COL2, ROW5 */
    MATRIX_KEY(6, 2, KEY_J) /* COL2, ROW6 */
    MATRIX_KEY(7, 2, KEY_K) /* COL2, ROW7 */
>;
>;
};

```

Para verificar que nuestro kernel tiene soporte a este tipo de entrada debemos instalar la aplicación *evtest*, la cual, al ejecutarse nos mostrará los dispositivos de entrada reconocidos:

```

Available devices:
/dev/input/event0:      keyboard
/dev/input/event1:      sunxi-gpadc0
/dev/input/event2:      sunxi-ir
/dev/input/event3:      audiocodec sunxi Audio Jack
Select the device event number [0-3] :

```

Si seleccionamos el dispositivo */dev/input/event0* y presionamos una tecla debemos ver el siguiente mensaje:

```

Input device name: "keyboard"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
  Event code 16 (KEY_Q)
  Event code 17 (KEY_W)
  Event code 18 (KEY_E)
  Event code 19 (KEY_R)

```

```

Event code 20 (KEY_T)
Event code 21 (KEY_Y)
Event code 22 (KEY_U)
Event code 23 (KEY_I)
Event code 24 (KEY_O)
Event code 30 (KEY_A)
Event code 31 (KEY_S)
Event code 32 (KEY_D)
Event code 33 (KEY_F)
Event code 34 (KEY_G)
Event code 35 (KEY_H)
Event code 36 (KEY_J)
Event code 37 (KEY_K)
Event code 44 (KEY_Z)
Event code 45 (KEY_X)
Event code 46 (KEY_C)
Event code 47 (KEY_V)
Event code 48 (KEY_B)
Event code 49 (KEY_N)
Event code 50 (KEY_M)
Event type 4 (EV_MSC)
    Event code 4 (MSC_SCAN)

Properties:
Testing ... (interrupt to exit)
Event: time 1718551323.593231, type 4 (EV_MSC), code 4 (MSC_SCAN), value 19
Event: time 1718551323.593231, type 1 (EV_KEY), code 23 (KEY_I), value 1
Event: time 1718551323.593231, ----- SYN_REPORT -----
Event: time 1718551323.703216, type 4 (EV_MSC), code 4 (MSC_SCAN), value 19
Event: time 1718551323.703216, type 1 (EV_KEY), code 23 (KEY_I), value 0
Event: time 1718551323.703216, ----- SYN_REPORT -----

```

### 5.19.2. Faust

Faust (Functional Audio Stream) es un lenguaje de programación funcional para síntesis de sonido y procesamiento de audio con especial énfasis en el diseño de sintetizadores, instrumentos musicales, efectos de audio, etc. Faust es una creación de GRAME-CNCM Research Department.

El núcleo de Faust es su compilador, el cual, permite "transladar" cualquier especificación DSP (Digital Signal Processing) a un amplio rango de lenguajes de programación tales como C++, C, LLVM bit code, WebAssembly, Rust, etc.

Gracias a su sistema de "arquitecturas", el código generado por Faust puede ser compilado fácilmente en una gran variedad de objetos desde plug-ins de audio a aplicaciones standalone o aplicaciones web o smartphone.

A manera de ejemplo generemos una señal de ruido en faust *noise.dsp*

```

declare options "[osc:on]";
import ("stdfaust.lib");
process = no.noise*hslider("level",0.02,0,1,0.01);

```

En esta aplicación se define un componente hslider llamado "*level*" que puede variar de 0 a 1 en pasos de 0.01 y su valor inicial es 0.02. *no.noise* genera una señal de ruido la cual es multiplicada por el valor de *level*, lo que implementa un control de volumen para esta señal (ver figura 5.38).

Debido a que el procesador T113 posee soporte alsal para su codec de audio interno utilizaremos el programa fasut2alsa, el cual compila programas Faust para alsal-gtk.

```
fasut2alsa -osc noise.dsp (PC)
```

```
faust2alsaconsole -osc noise.dsp (ec_saxo)
```

### 5.19.3. Open Sound Control

Open Sound Control (OSC) es un protocolo de red para sintetizadores de sonido, computadores y otros dispositivos multimedia, para fines tales como interpretación musical o control de espectáculos. Las ventajas de OSC incluyen interoperabilidad, exactitud, flexibilidad y una buena documentación.

Se utilizará OSC (Open Sound Control) para controlar el nivel de volumen en la aplicación *noise*. Para esto debemos instalar liblo (Lightweight OSC implementation)

```
sudo apt-get install liblo7 liblo-dev liblo-dev
```

El código del programa *set\_level.c* es:

```
#include <stdio.h>
#include <stdlib.h>
#include <lo/lo.h>
#include <unistd.h>
#define OSC_PORT 5513

int main() {
    lo_address t = lo_address_new("localhost", "5513");

    double level;
    level = 0.01 ;
    for (int i = 0; i < 10; i++) {
        lo_send(t, "/noise/level", "f", level);
        printf("Aumentando Nivel de volumen a %f\n", level);
        level += 0.1;
        usleep(500000);
    }
    return 0;
}
```

Se compila con el comando:

```
gcc set_level.c -o set_level -llo
```

Al lanzar la aplicación faust y en seguida ejecutar *set\_level* debemos escuchar como aumenta el nivel del volumen de la señal de ruido.

```
nohup ./noise -port 5513 &
```

Las aplicaciones faust con OSC habilitado tienen las siguientes opciones:

```
-port n set the port number used by the application to receive messages
-outport n set the port number used by the application to transmit messages
-errport n set the port number used by the application to transmit error messages
-desthost h set the destination host for the messages sent by the application
-xmit 0|1|2 turn transmission OFF, ALL or ALIAS (default OFF)
-xmitfilter s
```

### 5.19.4. Notas musicales con entrada desde key\_pad

EC\_SAXO posee un teclado que trata de seguir la disposición de teclas de un saxofón (ver figura 5.39). Esta configuración ya fué implementada en el dispositivo de entrada de la plataforma.

#### Generador de ondas senoidales en Faust

Para mostrar como utilizar dispositivos físicos como entrada a Faust se utilizará un generador de ondas senoidales note.dsp:

```
declare options "[osc:on]";
import ("stdfaust.lib");
process = os.osc(hslider("frequency", 100, 100, 2400, 100.0))*0.5;
```

Para compilar esta aplicación debemos ejecutar el comando:

```
faust2alsa -osc note.dsp (PC)
faust2alsaconsole -osc note.dsp (ec_saxo)
```

#### Control de teclado

El siguiente código (note\_input.c) captura los eventos del key\_pad y envía a la aplicación de Faust la frecuencia que debe generar:

```
#include <stdio.h>
#include <stdlib.h>
#include <lo/lo.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>
#include <math.h>
#define OSC_PORT 5513

int main() {
    int fd;
    double note;
    lo_address t = lo_address_new("localhost", "5513");
    fd = open("/dev/input/event0", O_RDONLY);
    if (fd == -1) {
        perror("Opening /dev/input/event0");
        return EXIT_FAILURE;
    }
    struct input_event ie;
    while (1) {
        // Read input events
        read(fd, &ie, sizeof(struct input_event));
        if (ie.type == EV_KEY) {
            if (ie.value == 1) {
                switch (ie.code) {
                    case KEY_Q: note = 100.0; break;
                    case KEY_W: note = 200.0; break;
                    case KEY_E: note = 300.0; break;
                }
                if (note != 0.0) {
                    os.osc(hslider("frequency", 100, 100, 2400, note));
                }
            }
        }
    }
}
```

```

        case KEY_R: note = 400.0; break;
        case KEY_T: note = 500.0; break;
        case KEY_Y: note = 600.0; break;
        case KEY_U: note = 700.0; break;
        case KEY_I: note = 800.0; break;
        case KEY_O: note = 900.0; break;
        case KEY_A: note = 1000.0; break;
        case KEY_S: note = 1100.0; break;
        case KEY_D: note = 1200.0; break;
        case KEY_F: note = 1300.0; break;
        case KEY_G: note = 1400.0; break;
        case KEY_H: note = 1500.0; break;
        case KEY_J: note = 1600.0; break;
        case KEY_K: note = 1700.0; break;
        case KEY_Z: note = 1800.0; break;
        case KEY_X: note = 1900.0; break;
        case KEY_C: note = 2000.0; break;
        case KEY_V: note = 2100.0; break;
        case KEY_B: note = 2200.0; break;
        case KEY_N: note = 2300.0; break;
        case KEY_M: note = 2400.0; break;
    }
    lo_send(t, "/note/frequency", "f", note);
    printf("Fijando nota a %f\n", note);
} else if (ie.value == 0) {
    lo_send(t, "/note/frequency", "f", 0);
    printf("Enviado: Nota Off=%f\n", note);
}
}
close(fd);
return EXIT_SUCCESS;
}

```

### 5.19.5. Configuración de amidi

```

import("stdfaust.lib");
freq = hslider("freq",300,50,2000,0.01);
gain = hslider("gain",0.8,0,1,0.01);
gate = button("gate");
process = os.osc(freq)*gain*gate;

```

freq is computed by retrieving MIDI note numbers and converting them to frequencies using

Of course, this instrument is not polyphonic. In fact, the same result can be achieved in

```
import("stdfaust.lib");
freq = hslider("freq",300,50,2000,0.01);
gain = hslider("gain",0.8,0,1,0.01);
gate = button("gate");
process = os.osc(freq)*gain*gate;
```

In that case, MIDI note numbers are automatically converted to frequencies by Faust. gain

Try it out in the Faust online editor (and don't forget to activate the polyphony mode)!

Keep in mind that in both cases, continuous control change events (usbMIDI.sendControlChange)

```
oscsend localhost 5513 /Oscillator/volume f 0.2
oscsend localhost 5513 /Oscillator/frequency f 800.0
```

```

#define DRIVER_NAME "bmp280"
#define DRIVER_CLASS "bmp280Class"
static struct i2c_adapter * bmp_i2c_adapter = NULL;
static struct i2c_client * bmp280_i2c_client = NULL;
#define I2C_BUS_AVAILABLE 1
#define SLAVE_DEVICE_NAME "BMP280"
#define BMP280_SLAVE_ADDRESS 0x76

static const struct i2c_device_id bmp_id[] = {
    { SLAVE_DEVICE_NAME, 0 },
    {}
};
static struct i2c_driver bmp_driver = {
    .driver = {
        .name = SLAVE_DEVICE_NAME,
        .owner = THIS_MODULE
    }
};
static struct i2c_board_info bmp_i2c_board_info = {
    I2C_BOARD_INFO(SLAVE_DEVICE_NAME, BMP280_SLAVE_ADDRESS)
};

static dev_t myDeviceNr;
static struct class *myClass;
static struct cdev myDevice;
s32 dig_T1, dig_T2, dig_T3;
static ssize_t driver_read(struct file *file, char *user_buffer, size_t count, loff_t *offs) {
    int to_copy, not_copied, delta;
    char out_string[20];
    int temperature;
    to_copy = min(sizeof(out_string), count);
    temperature = read_temperature();
    snprintf(out_string, sizeof(out_string), "%d.%d\n", temperature/100, temperature%100);
    not_copied = copy_to_user(user_buffer, out_string, to_copy);
    delta = to_copy - not_copied;
    return delta;
}
static int driver_open(struct inode *deviceFile, struct file *instance) {
    printk("MyDeviceDriver - Open was called\n");
    return 0;
}
static int driver_close(struct inode *deviceFile, struct file *instance) {
    printk("MyDeviceDriver - Close was called\n");
    return 0;
}
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = driver_open,
    .release = driver_close,
    .read = driver_read,
};
static int __init ModuleInit(void) {
    int ret = -1;
    u8 id;
    if (alloc_chrdev_region(&myDeviceNr, 0, 1, DRIVER_NAME) < 0) {
        printk("Device Nr. could not be allocated!\n");
    }
    printk("MyDeviceDriver - Device Nr %d was registered!\n", myDeviceNr);
    if ((myClass = class_create(THIS_MODULE, DRIVER_CLASS)) == NULL) {
        goto ClassError;
    }
    if (device_create(myClass, NULL, myDeviceNr, NULL, DRIVER_NAME) == NULL) {
        goto FileError;
    }
    cdev_init(&myDevice, &fops);
    if (cdev_add(&myDevice, myDeviceNr, 1) == -1) {
        goto KernelError;
    }
    bmp_i2c_adapter = i2c_get_adapter(I2C_BUS_AVAILABLE);
    if(bmp_i2c_adapter != NULL) {
        bmp280_i2c_client = i2c_new_device(bmp_i2c_adapter, &bmp_i2c_board_info);
        if(bmp280_i2c_client != NULL) {
            i2c_add_driver(&bmp_driver);
            ret = 0;
        }
        i2c_put_adapter(bmp_i2c_adapter);
    }
    printk("BMP280 Driver added!\n");
    id = i2c_smbus_read_byte_data(bmp280_i2c_client, 0xD0);
    printk("ID: 0x%02x\n", id);
    dig_T1 = i2c_smbus_read_word_data(bmp280_i2c_client, 0x88);
    dig_T2 = i2c_smbus_read_word_data(bmp280_i2c_client, 0x8A);
    dig_T3 = i2c_smbus_read_word_data(bmp280_i2c_client, 0x8C);
    if(dig_T2 > 32767)
        dig_T2 -= 65536;
    if(dig_T3 > 32767)
        dig_T3 -= 65536;
    i2c_smbus_write_byte_data(bmp280_i2c_client, 0xF5, 5<<5);
    i2c_smbus_write_byte_data(bmp280_i2c_client, 0xF4, ((5<<5) | (5<<2) | (3<<0)));
    return ret;
KernelError:
    device_destroy(myClass, myDeviceNr);
FileError:
    class_destroy(myClass);
ClassError:
    unregister_chrdev(myDeviceNr, DRIVER_NAME);
    return (-1);
}
static void __exit ModuleExit(void) {
    printk("MyDeviceDriver - Goodbye, Kernel!\n");
    i2c_unregister_device(bmp280_i2c_client);
    i2c_del_driver(&bmp_driver);
    cdev_del(&myDevice);
    device_destroy(myClass, myDeviceNr);
    class_destroy(myClass);
    unregister_chrdev(myDeviceNr, 1);
}
module_init(ModuleInit);
module_exit(ModuleExit);

```

```

s32 read_temperature(void) {
    int var1, var2;
    s32 raw_temp;
    s32 d1, d2, d3;
    d1 = i2c_smbus_read_byte_data(bmp280_i2c_client, 0xFA);
    d2 = i2c_smbus_read_byte_data(bmp280_i2c_client, 0xFB);
    d3 = i2c_smbus_read_byte_data(bmp280_i2c_client, 0xFC);
    raw_temp = (((d1<<16) | (d2<<8) | d3) >> 4);
    var1 = (((raw_temp >> 3) - (dig_T1 << 1)) * (dig_T2)) >> 11;
    var2 = (((((raw_temp >> 4) - (dig_T1) * ((raw_temp >> 4) - (dig_T1))) >> 12) * (dig_T3)) >> 14;
    return ((var1 + var2) * 5 + 128) >> 8;
}

```

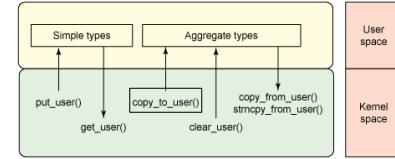


Figura 5.34 Módulo BMP280

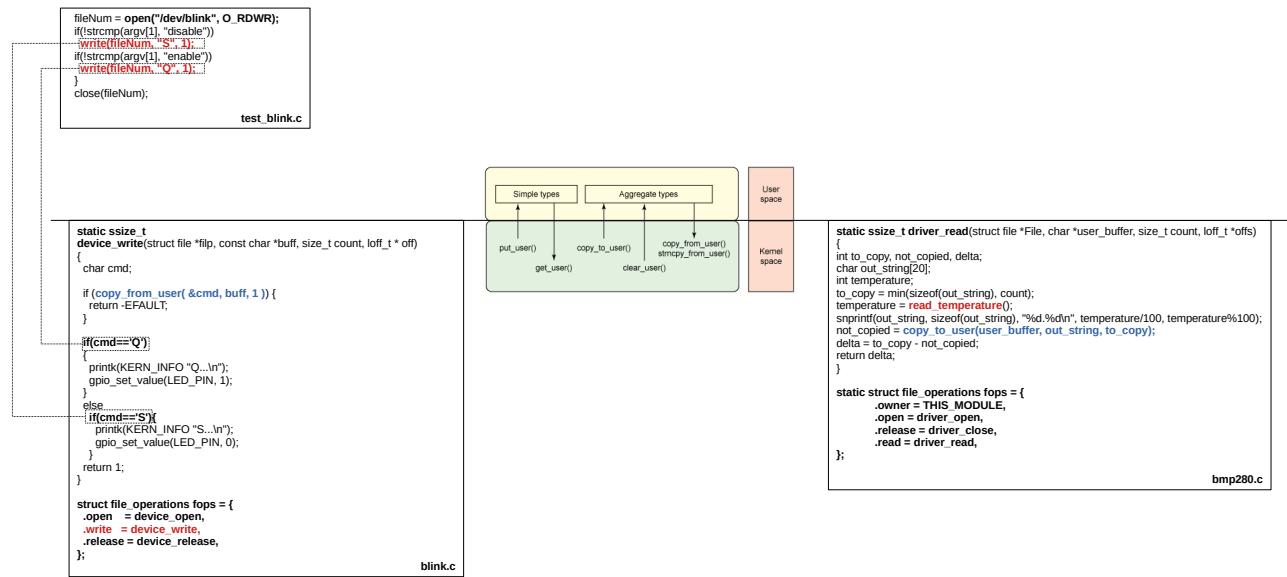


Figura 5.35 Comunicación con módulos desde espacio de usuario

```

#define IRQ_PIN 110 // (position of letter in alphabet - 1) * 32 + pin number PD14 3*32+14 110
#define DEVICE_NAME "irq" /* Dev name as it appears in /proc/devices */
static DECLARE_WAIT_QUEUE_HEAD(wq);
unsigned char irq_enabled;
unsigned int interrupt_counter = 0;
unsigned int irq_number;

static irqreturn_t irq_handler(int irq, void *dev_id)
{
    if(irq_enabled)
    {
        interrupt_counter++;
        printk(KERN_INFO "interrupt_counter=%d\n",interrupt_counter);
        wake_up_interruptible(&wq);
    }
    return IRQ_HANDLED;
}

static ssize_t device_read(struct file *filp, /* see include/linux/fs.h */
                         char *buffer, /* buffer to fill with data */
                         size_t count, /* length of the buffer */
                         loff_t *offset)
{
    if(irq_enabled)
    {
        wait_event_interruptible(wq, interrupt_counter!=0);
        copy_to_user(buffer, &interrupt_counter, sizeof(interrupt_counter));
    }
    return sizeof(interrupt_counter);
}

static ssize_t device_write(struct file *filp, const char *buff, size_t count, loff_t *off)
{
    char cmd;
    if(copy_from_user(&cmd, buff, 1))
        return -EFAULT;
    if(cmd=='Q')
    {
        irq_enabled = 1;
        printk(KERN_INFO "irq_enabled...\n");
    }
    else
    if(cmd=='S')
    {
        irq_enabled=0;
        printk(KERN_INFO "irq disabled.\n");
    }
    return 1;
}

struct file_operations fops = {
    .read = device_read,
    .write = device_write,
};

static int __init qem_init(void)
{
    int res, ret;
    printk(KERN_INFO "IRQ module is Up.\n");
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if(Major < 0)
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
    return Major;
}

printk(KERN_ALERT "I was assigned major number %d. To talk toin", Major);
printk(KERN_ALERT "the driver, create a dev file within");
printk(KERN_ALERT "mknode /dev/51s c %d 0.\n", DEVICE_NAME, Major);
ret = gpio_request(IRQ_PIN, "led");
if (ret) {
    pr_err("fail request LED pin\n");
    return -1;
}
gpio_direction_input(IRQ_PIN);
irq_number = gpio_to_irq(IRQ_PIN);
res = request_irq(irq_number, irq_handler, IRQF_TRIGGER_RISING, "IRQ", NULL);
return 0;
}

static void __exit qem_exit(void)
{
    free_irq(irq_number, NULL);
    gpio_free(IRQ_PIN);
    unregister_chrdev(Major, DEVICE_NAME);
    printk(KERN_INFO "IRQ driver is down...\n");
}

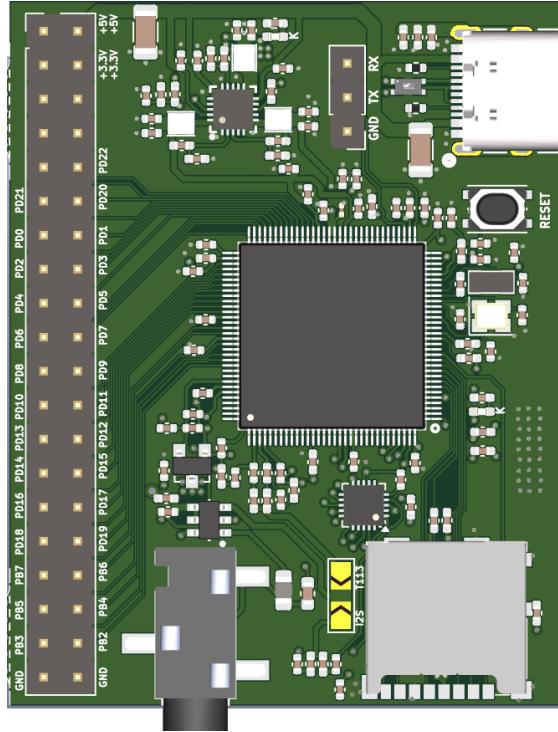
module_init(qem_init);
module_exit(qem_exit);

```

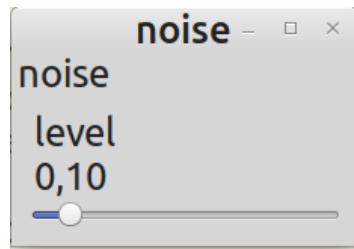
irq.c

143:	68	0	sunxi_pio_edge	78	Edge	IRQ	/proc/interrupts
------	----	---	----------------	----	------	-----	------------------

**Figura 5.36** Interrupciones en módulos



**Figura 5.37** Tarjeta de desarrollo ec\_saxo



**Figura 5.38** Aplicación en Faust

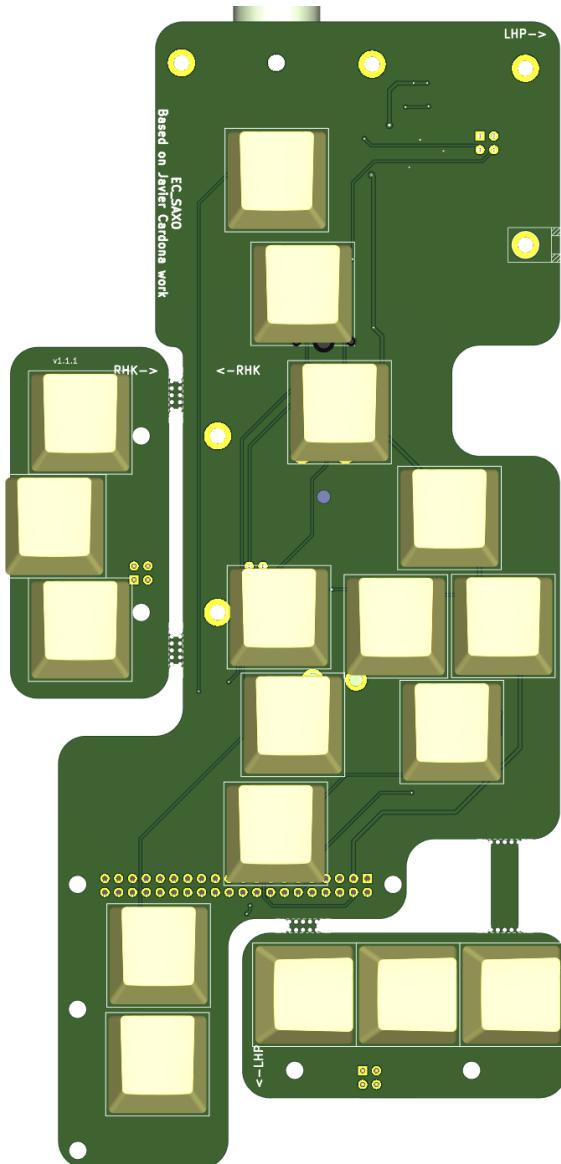


Figura 5.39 Teclado de EC\_SAXO

## Índice general



## Capítulo 6

# Herramientas libres para desarrollar proyectos Hardware-Software

### 6.1. OpenHardware

Reciben la denominación *hardware libre*, hardware de código abierto, OpenHardware a los dispositivos que tienen a disposición de cualquier interesado los archivos que permitan su reproducción, en el caso de las placas de circuito impreso (PCBs), los esquemáticos, el layout y la lista de componentes; con lo que es posible realizar modificaciones al diseño original o fabricarlo y ensamblarlo. Para el primer caso es necesario que las herramientas CAD utilizadas sean de carácter abierto, de lo contrario sería necesario adquirir licencias propietarias para realizar modificaciones. En este anexo se presentará la placa de desarrollo OpenHardware ECB\_T8\_T113, la cual permite la implementación de aplicaciones utilizando tareas SW (ejecutadas en un procesador), tareas HW (ejecutadas en un dispositivo lógico programable) o tareas HW - SW.

### 6.2. Arquitectura de la tarjeta ECB\_T8\_T113

En la figura 6.1 se muestra el diagrama de bloques de la tarjeta de desarrollo ECB\_T8\_T113. Como puede verse está formada por dos componentes: Un procesador Allwinner T113 (ARM Cortex-A7 Dual-Core, 1.2 GHz, 128MB DDR3) y una FPGA Efinix T20 (Logic elements: 19,728, RAM 1044.48kb, RAM, Multipliers (18 x 18): 36).

Algunos pines del T113 se encuentran conectados a la FPGA con el fin de permitir la comunicación entre ellos, entre estos pines se encuentran periféricos como la UART y el SPI. Adicionalmente, tanto la FPGA como el ARM pueden conectarse a una red utilizando un conector RJ45. Se dispone de conectores de expansión con una amplia variedad de periféricos (I2C, PWM, Audio, SPI) y pines de Entrada/Salida de propósito general. Lo que posibilita:

1. Creación de periféricos dedicados en la FPGA que se comunican por el puerto serie o SPI con el procesador ARM.
2. Conexión de la FPGA y del procesador con dispositivos externos.
3. Creación de una red entre la FPGA y el procesador.
4. Depuración de las tareas que se ejecutan en el procesador y en la FPGA.

Con esto se pretende abarcar las diferentes etapas del diseño de sistemas digitales, en los primeros pasos se utiliza la FPGA para realizar tareas hardware que se implementan mediante Máquinas de estado algorítmicas, para esto existe un canal de configuración y depuración utilizando el puerto USB. Como puede verse del diagrama de bloques, el canal de depuración se comparte con el procesador, lo que permite que en pasos posteriores en el aprendizaje se utilice para observar los procesos que ocurren en el procesador como inicialización de bajo nivel (boot), inicialización del sistema operativo, e interacción con la aplicaciones.

La FPGA utilizada permite la implementación de SoC con una variedad de procesadores dentro de los que se encuentra el RiscV (muy popular en estos días), gracias a herramientas como Litex es posible realizar aplicaciones muy rápidamente, lo que posibilita el estudio de arquitectura de procesadores y desarrollo de aplicaciones.

Finalmente, el procesador T113 es capaz de ejecutar el kernel y aplicaciones del sistema Operativo Linux, lo que posibilita el estudio de los diferentes componentes de este sistema, así como la forma de ajustarlo a una determinada arquitectura HW que cumpla con unos requerimientos.

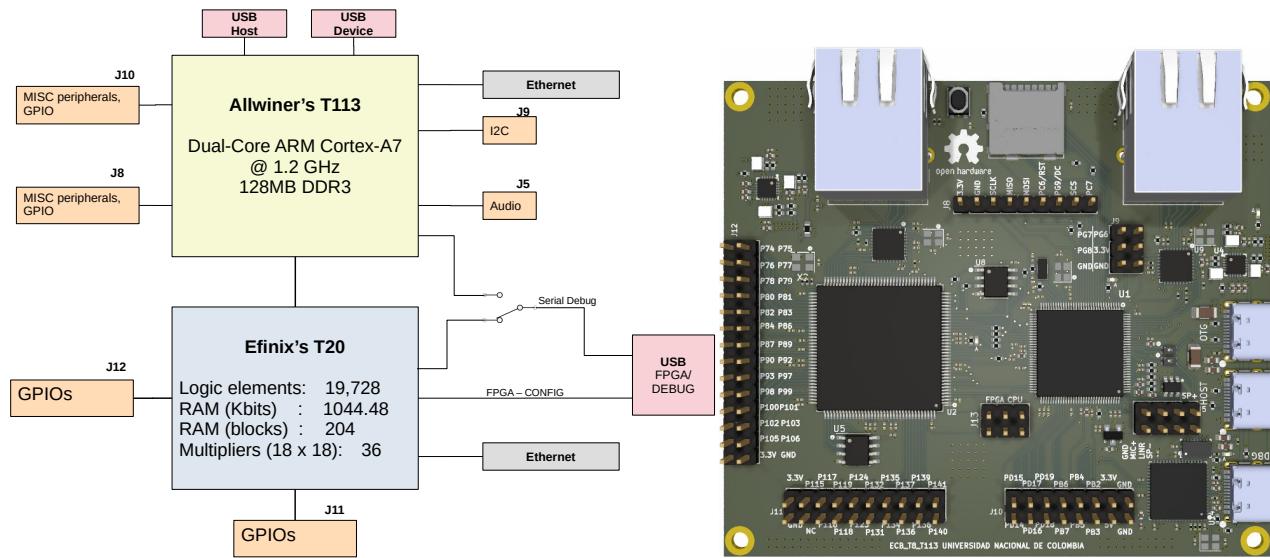


Figura 6.1 Diagrama de Bloques de la tarjeta ECB.T8\_T113

Con lo que se tendrían cubiertos todos las habilidades que requiere un diseñador de sistemas embebidos en la actualidad, sin tener que adquirir varias tarjetas lo que simplifica la implementación de tareas HW-SW  
En la figura 6.2 se muestra de forma detallada las señales disponibles en esta tarjeta.

### 6.3. Software y Aplicaciones Básicas

#### 6.3.1. FPGA

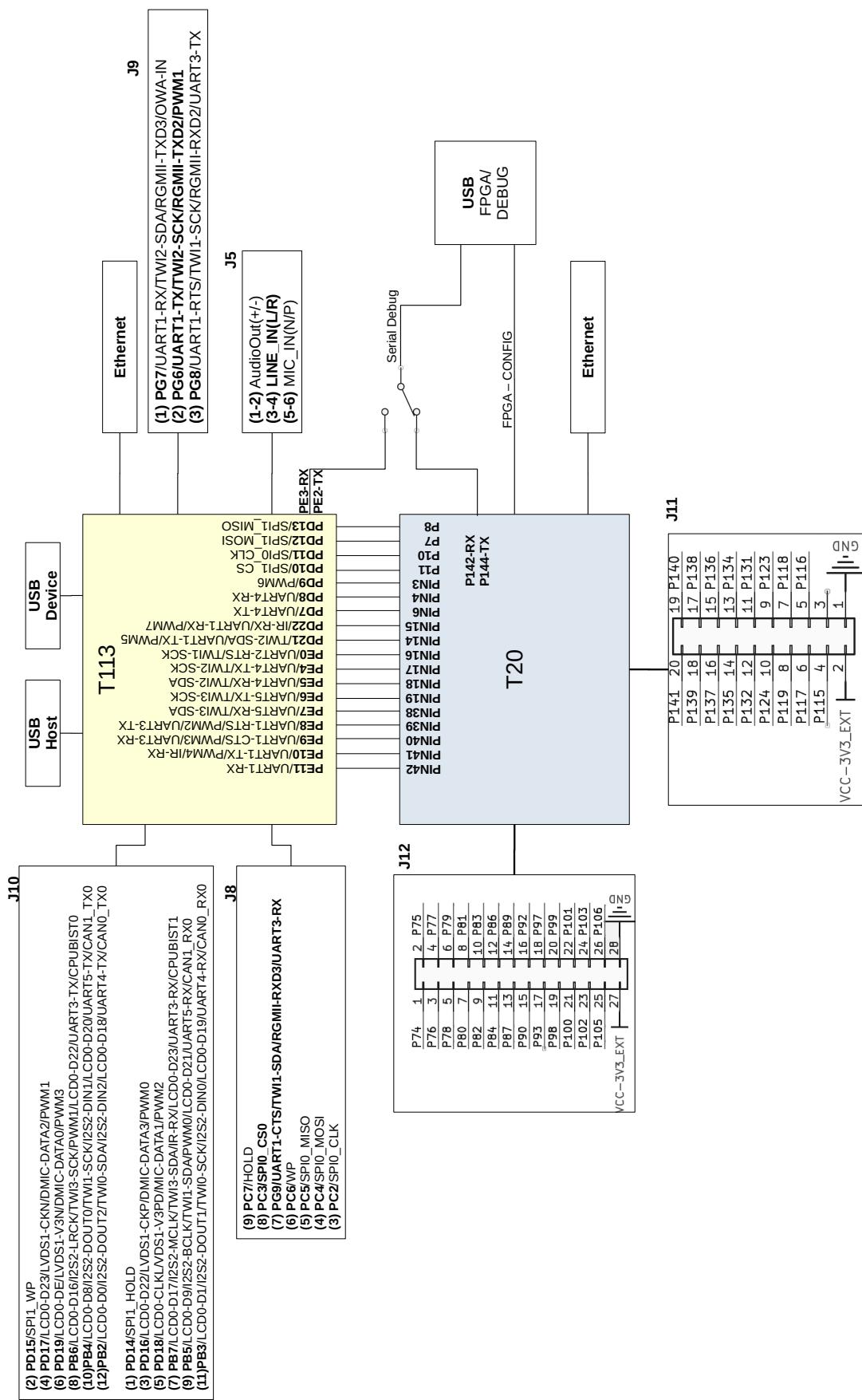


Figura 6.2 Señales de la tarjeta ECB\_T8\_T113

## Referencias

1. Dominic Rath. *Open On-Chip Debugger*. PhD thesis, University of Applied Sciences Augsburg, 2005.
2. I. Bowman, S. Siddiqi, and M. Tanuan. Concrete Architecture of the Linux Kernel. <http://docs.huihoo.com/linux/kernel/a2/index.html>, 12 February 1998.
3. I. Bowman. Conceptual Architecture of the Linux Kernel. <http://docs.huihoo.com/linux/kernel/a1/>, 1998.
4. Texas Instruments. IEEE Std 1149.1 (JTAG) Testability. *1997 Semiconductor Group*, 1996.
5. Michael L. Haungs. The Executable and Linking Format (ELF). <http://www.cs.ucdavis.edu/~haungs/paper/node1.html>, 21 September 1998.
6. C. Hallinan. *Embedded Linux Premiere A Practical Real-World Approach*. Prentice Hall, 18 September 2006.
7. Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems*. O'REILLY, 2008.
8. J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, 2005.