Capítulo 1
Implementación de tareas Software utilizando procesadores Soft Core

1.1. 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 caracter 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

1.2. Arquitectura del procesador RV32I

El RISC-V posee una estructura modular que permite adaptar la arquitectura a requerimientos funcionales y económicos, en la figura 1.1 se muestra el concepto de moduaridad, 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).

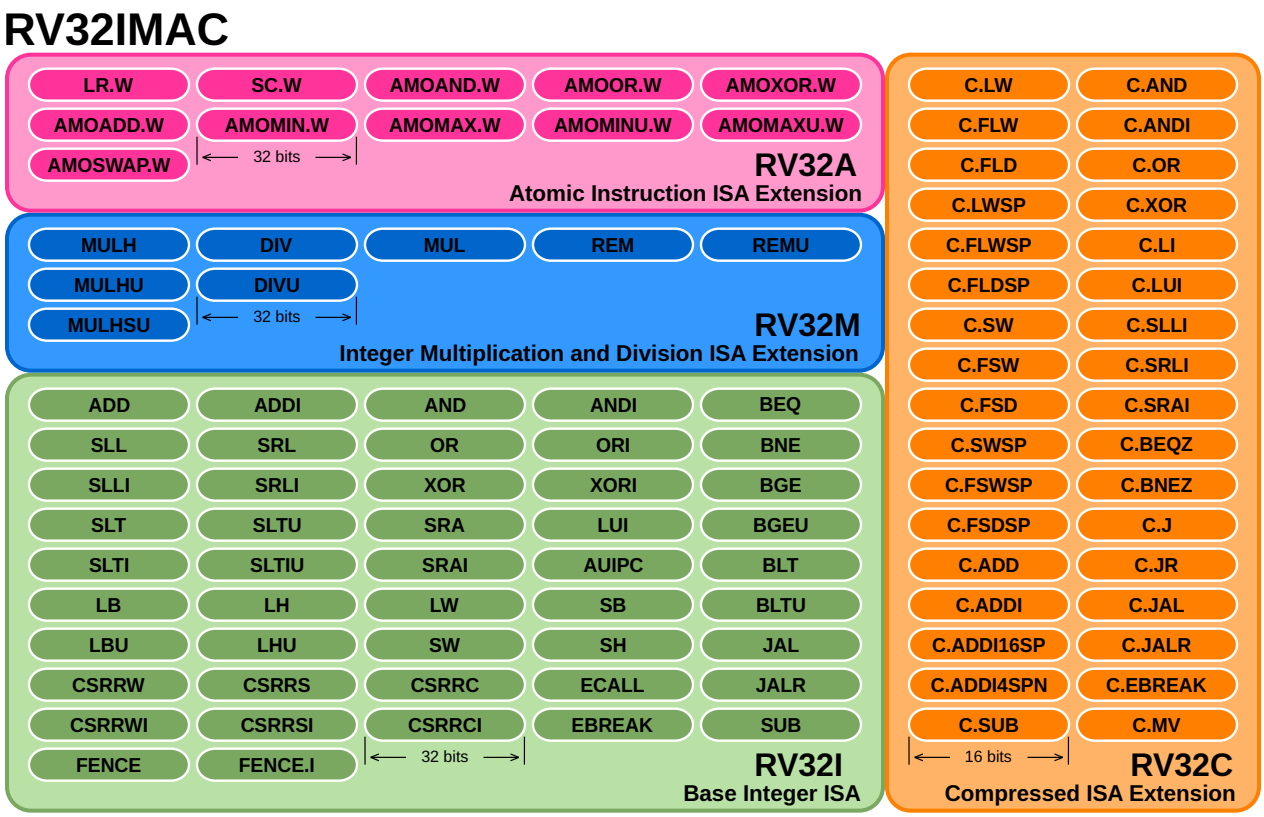

Figura 1.1 Set de instrucciones modular de la variante RV32IMAC. CPU de 32 bits de Base entera (RV32I), con extensión ISA para multiplicación y división entera (RV32M), instrucciones Atómicas (RV32A) e instrucciones Comprimidas (RV32C): fuente: Eduardo Corpeño (https://github.com/kuashio)

1.2.0.1. Registros

El RISC-V posee 32 registros (o 16 en la variante embebida). En la tabla 1.1 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 1.1 Registros del procesador RV32I.

1.2.0.2. Set de Instrucciones

En la tabla 1.2 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 ^ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 — 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 ¡ rs2)?1:0
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 ¡ rs2)?1:0
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm
ori	OR Immediate	I	0010011	0x6		rd = rs1 — 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 ¡ imm)?1:0
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 ¡ 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
bgeu	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 1.2 Set de Instrucciones del RV32I.

1.2.1. Diagrama de Bloques del FemtoRV

Como se mencionó anteriormente, en este capítulo se utilizará el procesador **femtoRV** desarrollado por Bruno Levy [1]. En la figura 1.2 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:

[1] https://github.com/BrunoLevy

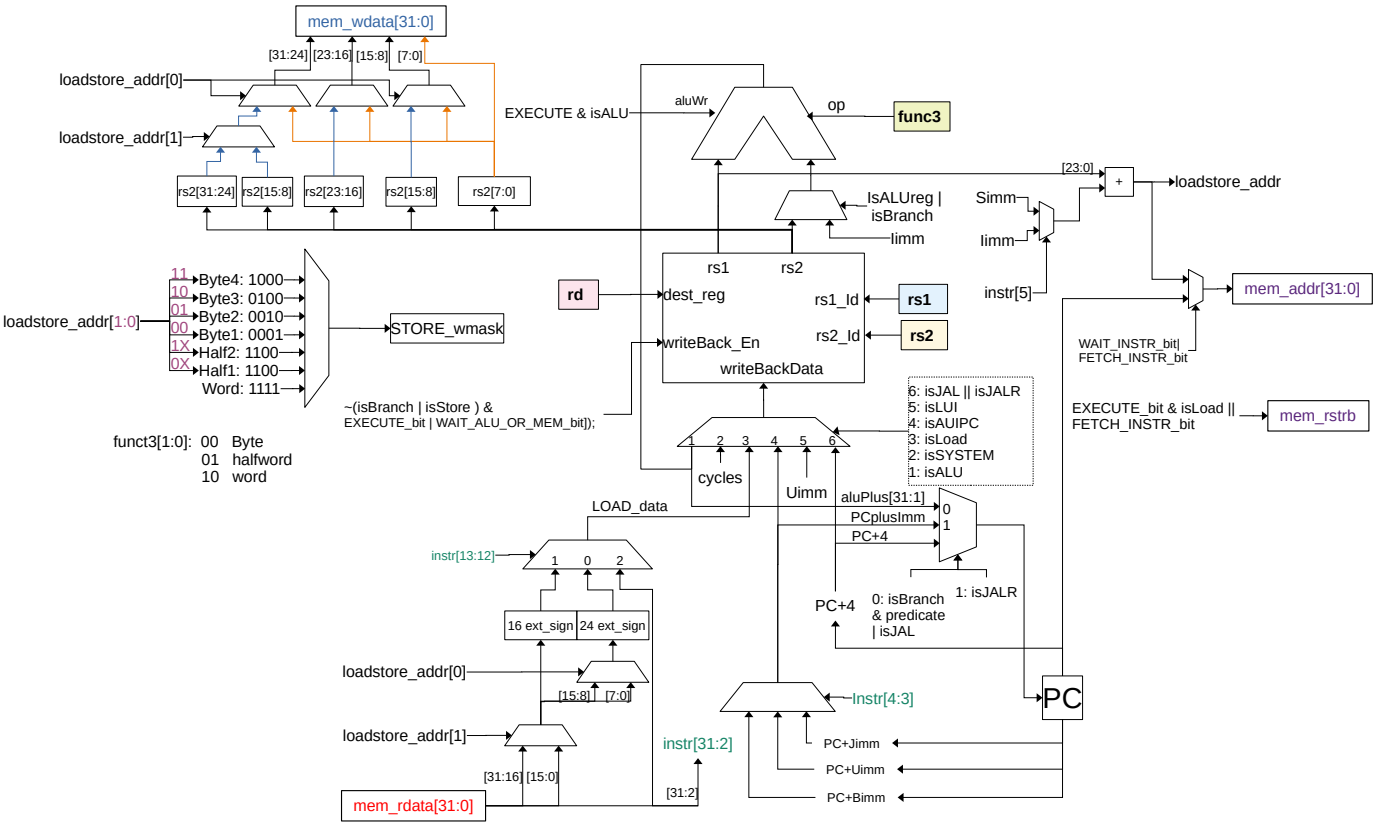

Figura 1.2 Diagrama de bloques del RV32I

- 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 perfiféricos y las instrucciones almacenadas en la memoria de progrma.
- Control mem_rstrb, store_wmask: Señales que indican a los periféricos y a la memoria de programa cuando se pueden realizar las operaciones de lectura y escritura.

1.2.2. 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 como se implementan las diferentes instrucciones en el camino de datos mostrado en la figura 1.2.

1.2.2.1. Instrucciones aritméticas y lógicas

Las instrucciones aritméticas y lógicas que puede implementar la variante RV32I pueden verse en la figura 1.3. 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 1.3 y 1.4 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 1.3 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 a 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.

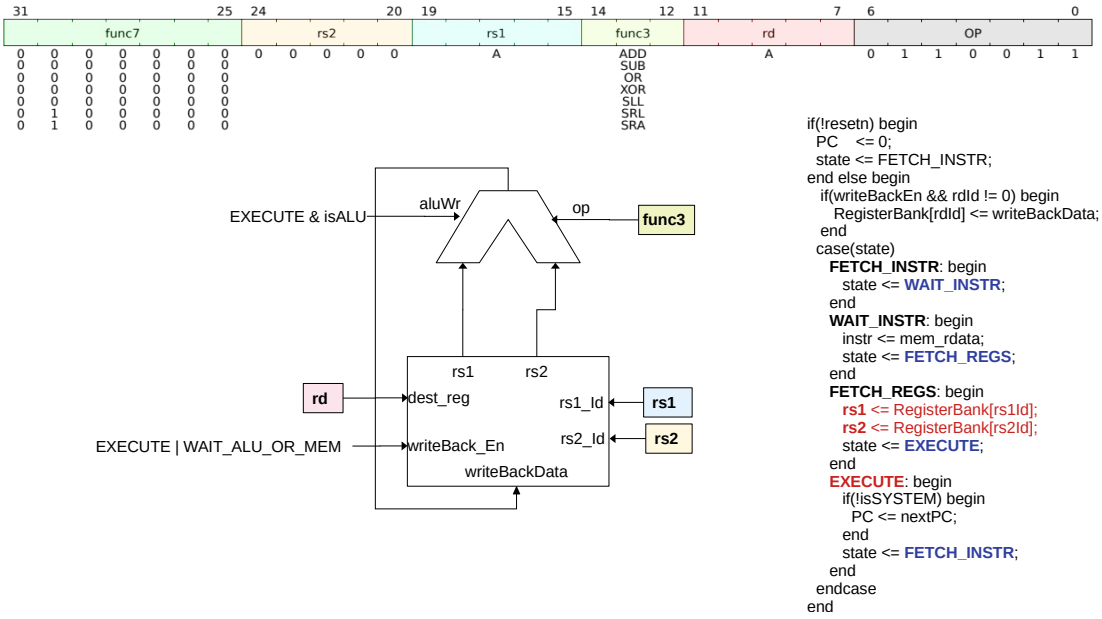

Figura 1.3 Instrucciones aritméticas entre registros del RV32I

Operaciones Aritméticas y Lógicas con un operador inmediato

Las operaciones aritméticas pueden realizarse con operandos que están imersos en la instrucción a este tipo de aoperaciones se les conoce con el nombre de inmediatas; como puede verse en la figura 1.4, 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.

1.2.2.2. Simulación de las instrucciones aritméticas y lógicas.

En la figura 1.8 se muestra la simulación

1.2.3. 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:

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

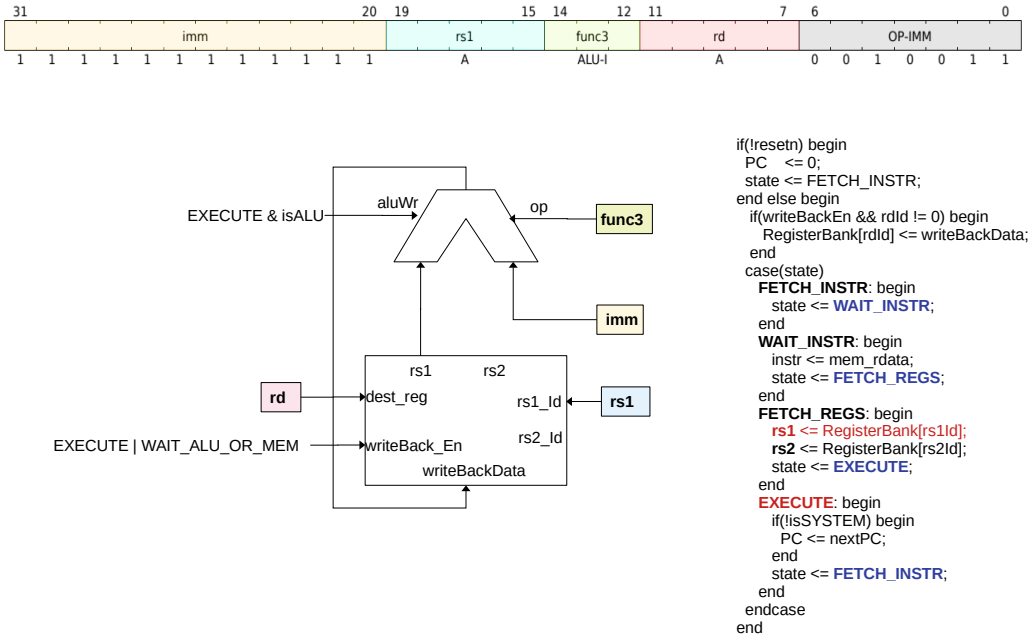

Figura 1.4 Instrucciones aritméticas inmediatas del RV32I

1.2.3.1. Saltos Incondicionales

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

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

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 1.6 se muestra la instrucción **JAL** en ella podemos ver que el contador de programa se modifica a *PC + Jimm*.

En la figura 1.7 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).

1.2.4. Arquitectura del SOC basado en RV32I

1.3. Arquitectura del procesador LM32

La figura 1.12 muestra el diagrama de bloques del soft-core LM32, este procesador utiliza 32 bits y una arquitectura de 6 etapas del pipeline; tambien cuenta con una logica de bypass que se encarga de hacer que el caminos de datos entre operaciones sea mas 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.

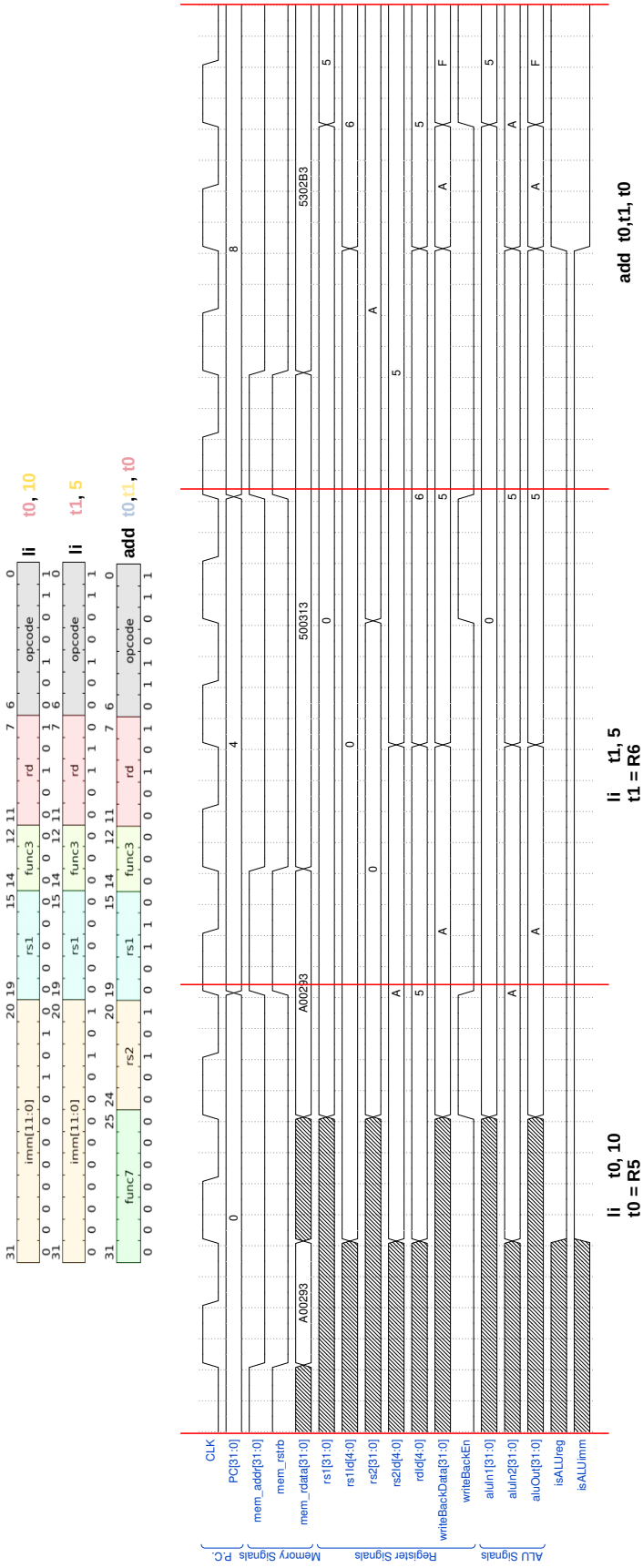

Figura 1.5 Formas de onda de la simulación de las instrucciones aritméticas inmediatas y entre registros del RV32I

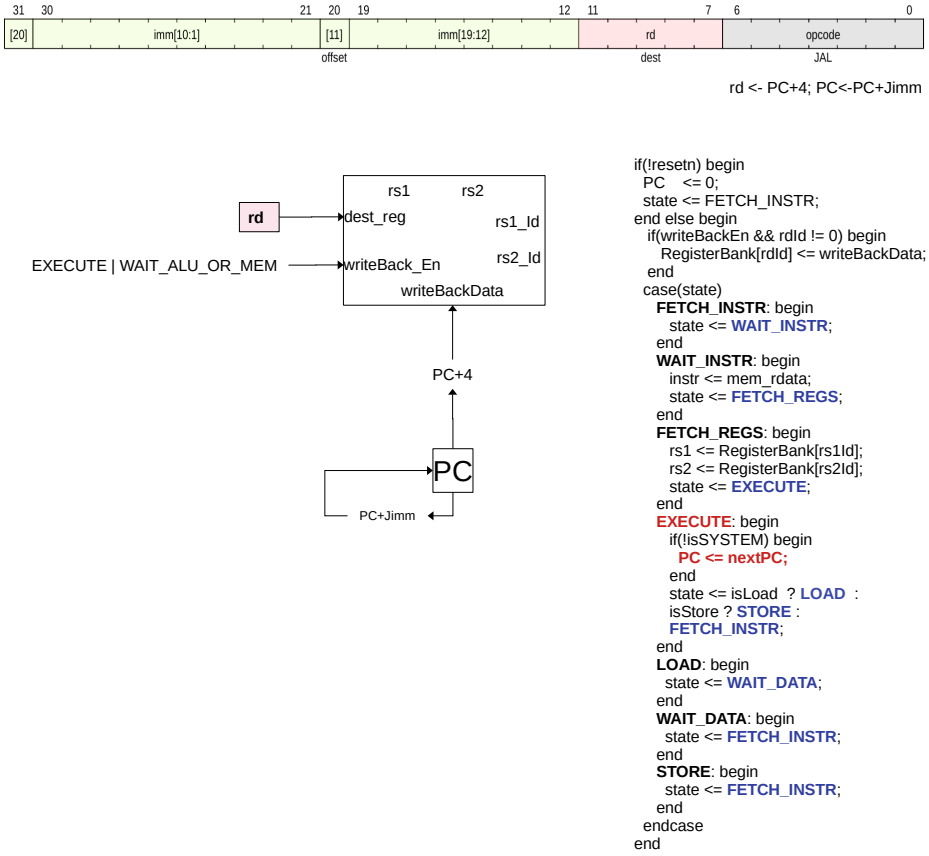

Figura 1.6 Instrucción JAL en el RV32I

D *Decode*: Se decodifica la instrucción y se toman los operandos del banco de registros o tomados del bypass.
X *Execute*: Se realiza la operacion especificada por la instrucción. Para instrucciones simples (sumas y operaciones logicas), 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.

1.3.1. 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)*

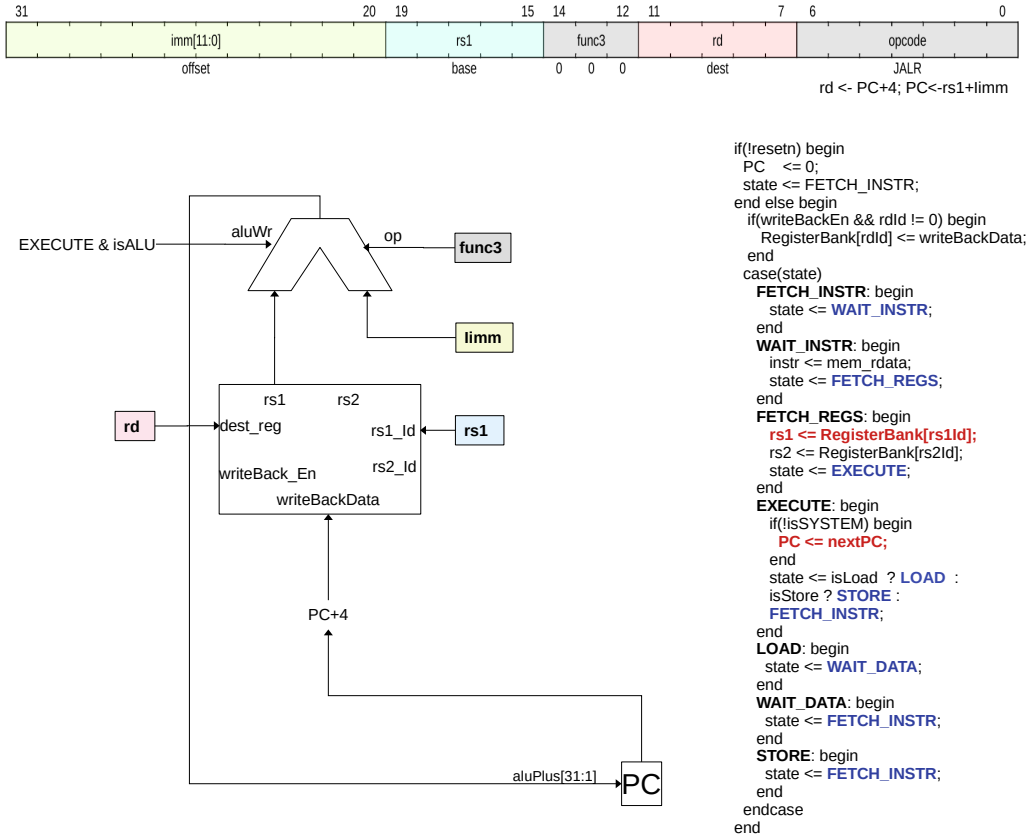

Figura 1.7 Instrucción JALR del RV32I

1.3.2. Registro de estado y control

La tabla 1.3 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 1.3 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

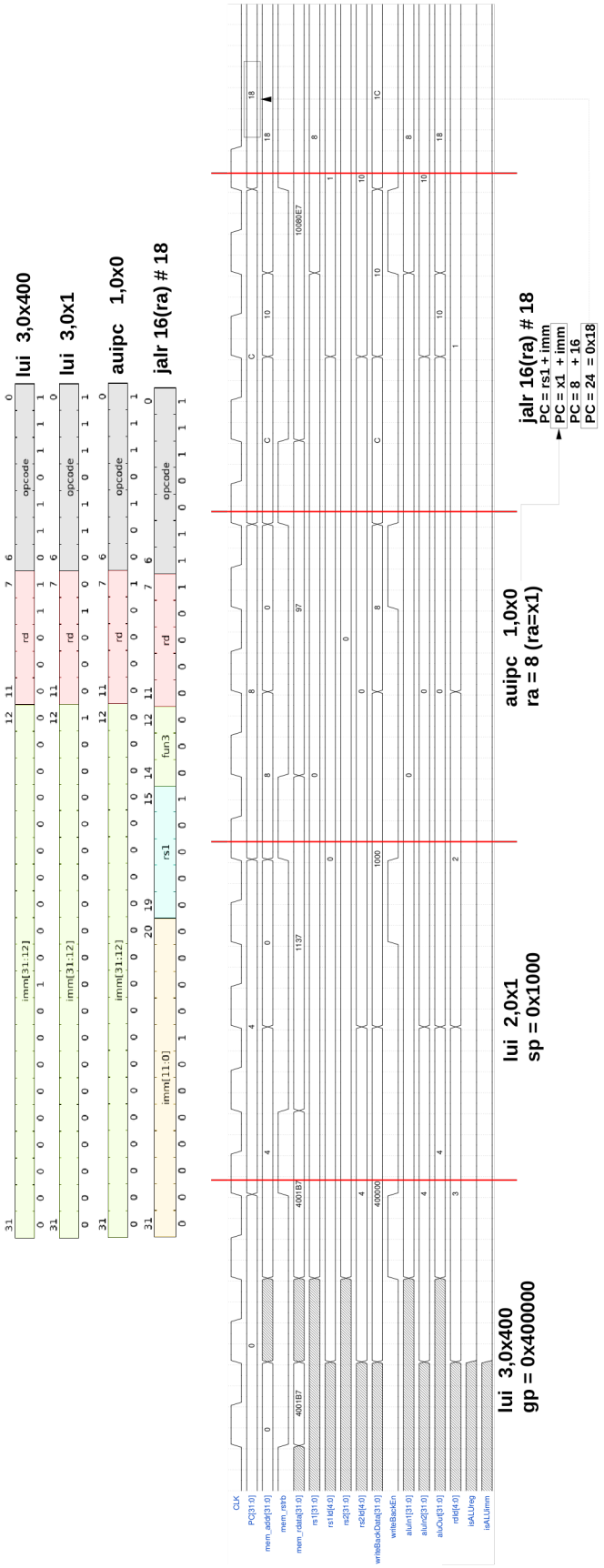

Figura 1.8 Formas de onda de la simulación de la instrucción JALR del RV32I

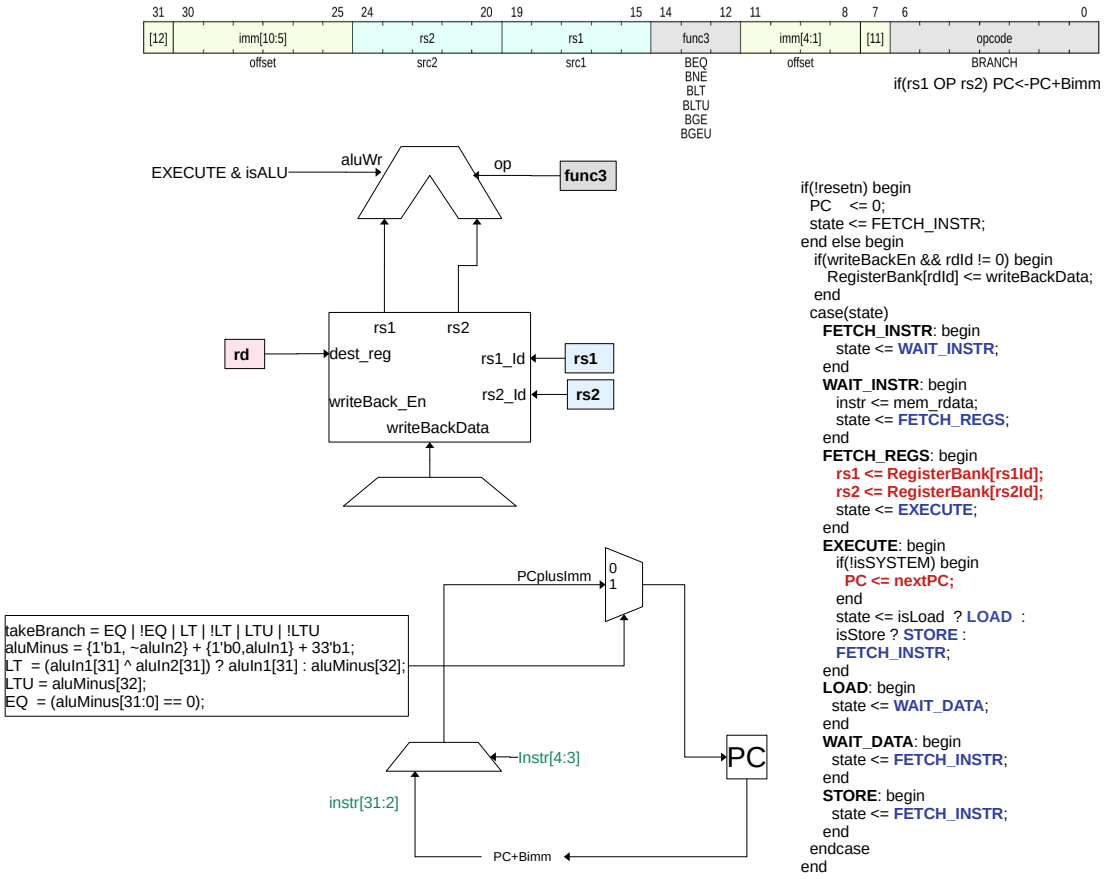

Figura 1.9 Saltos condicionales en el RV32I

1.3.2.1. 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 zero. El valor de este registro después del reset es *h00000000*

1.3.2.2. IE Habilitación de interrupción

l 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.

1.3.2.3. 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.

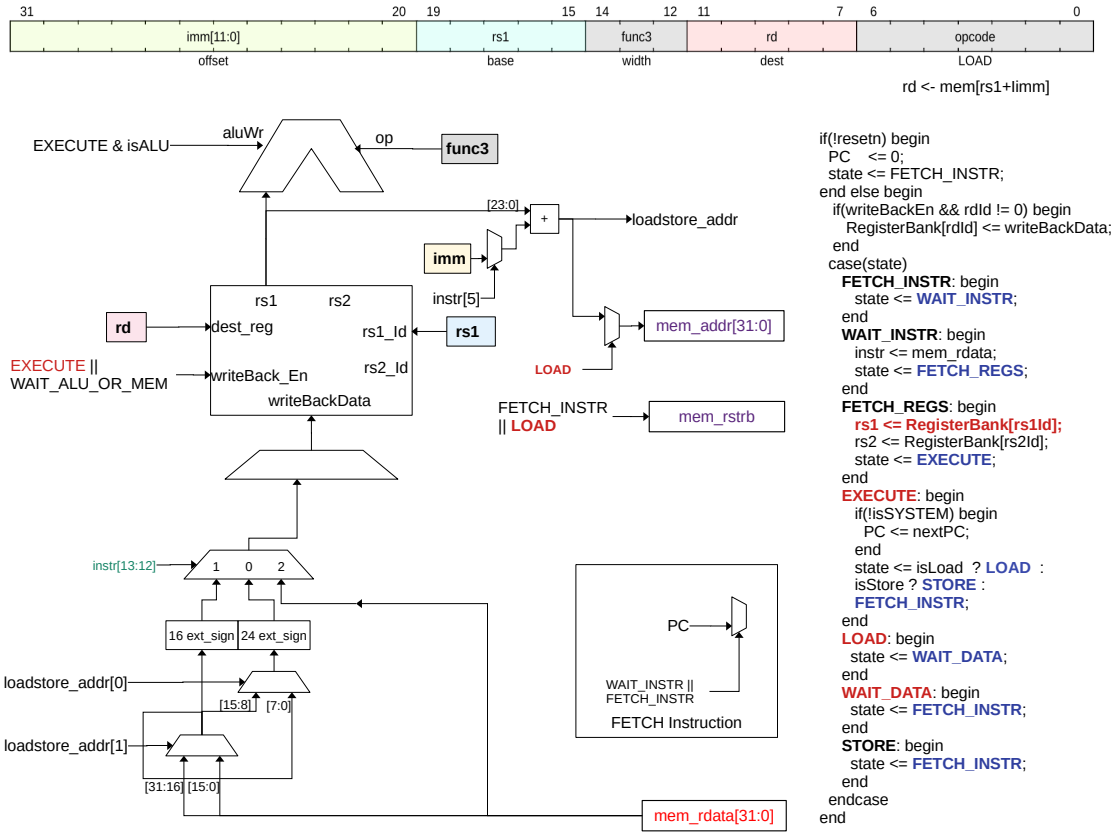

Figura 1.10 Lectura desde memoria externa en el RV32I

- **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, típica mente 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*.

1.3.2.4. 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. Después del reset el valor de IM es *h00000000*

1.3.2.5. IP Interrupción pendiente

El registro IP contine 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.

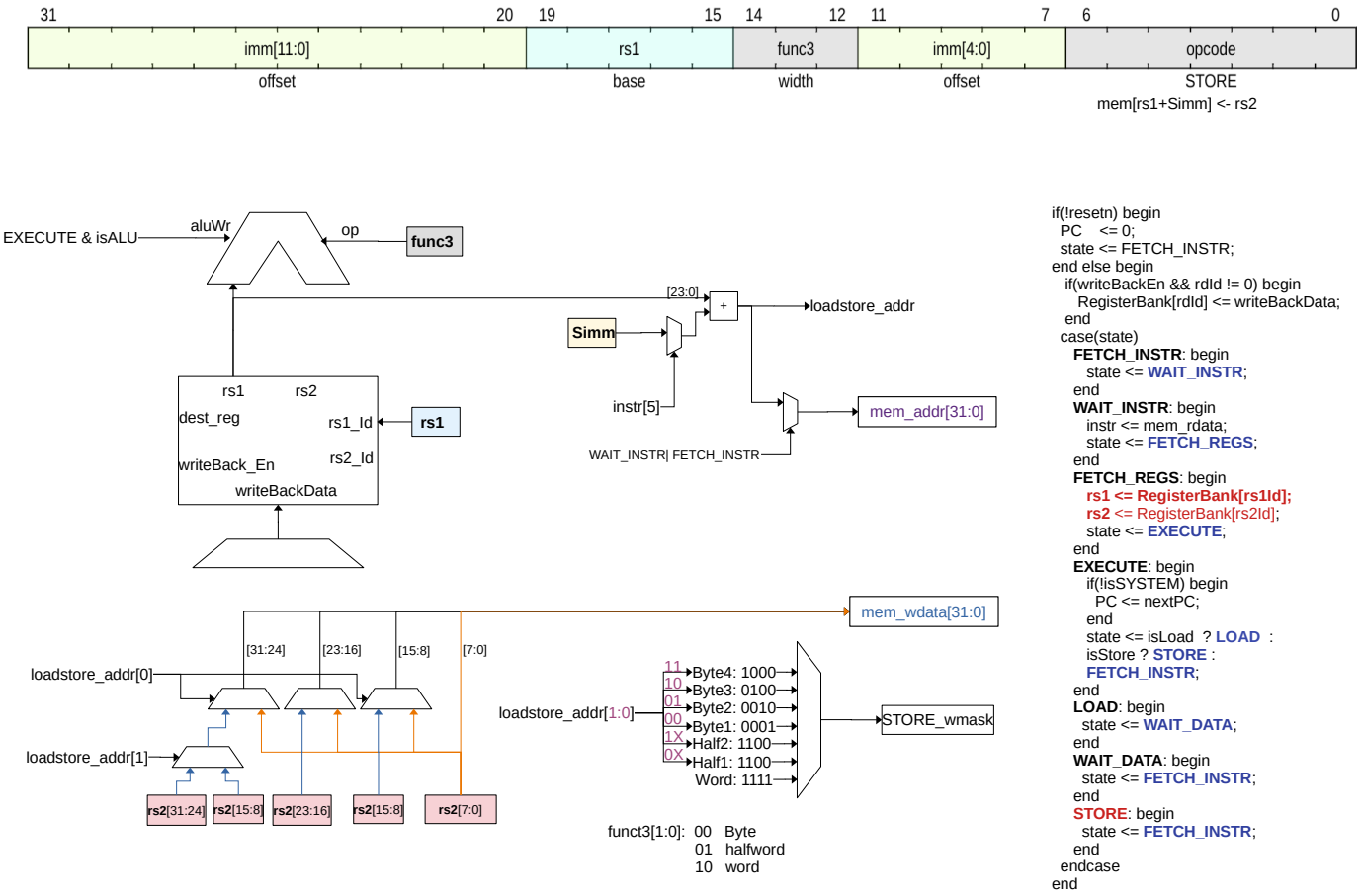

Figura 1.11 Escritura a memoria externa en el RV32I

1.4. 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.

1.4.1. Instrucciones aritméticas

1.4.1.1. Entre registros

En la figura 1.13 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:

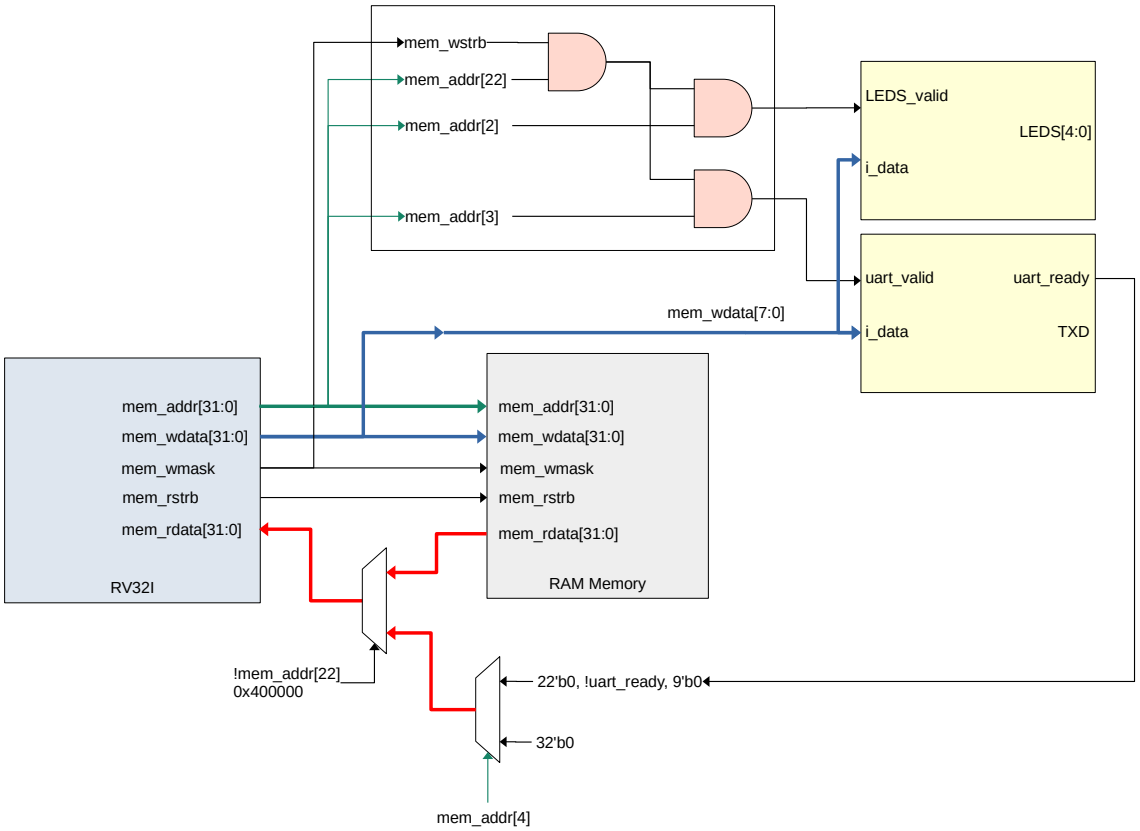

Figura 1.12 Diagrama de bloques del SoC basado en el RV32I

```
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 operacion 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

    if (reg_write_enable_q_w == 'TRUE)

        registers[write_idx_w] <= w_result;

end
```

1.4.1.2. 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 nos son almacenadas previamente en registros, sino que son almacenadas en la memoria de programa. En la figura 1.14 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