METODOLOGÍA DE DISEÑO E IMPLEMENTACIÓN DE SISTEMAS EMBEBIDOS

—Utilizando Herramientas Abiertas—

CARLOS IVÁN CAMARGO BAREÑO

ÍNDICE GENERAL

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.

1.2. Arquitectura del procesador LM32

La figura 1.1 muestra el diagrama de bloques del soft-core LM32, este procesador utiliza 32 bits y una arquitectura de 6 etapas del pipeline; las 6 etapas del pipeline son:

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

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

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

4. *Execute*: Se realiza la operacion especificada por la instrucción. Para instrucciones simples como las lógicas o suma, la ejecución finaliza en esta etapa, y el resultado se hace disponible para el bypass.

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

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

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 el valor no se define el valor de los registros, por lo que la primera acción que debe ejecutar el programa de inicialización en colocar un cero en el registro *r0 (**xor** r0, r0, r0)*

Registro de estado y control

La tabla 1.1 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.

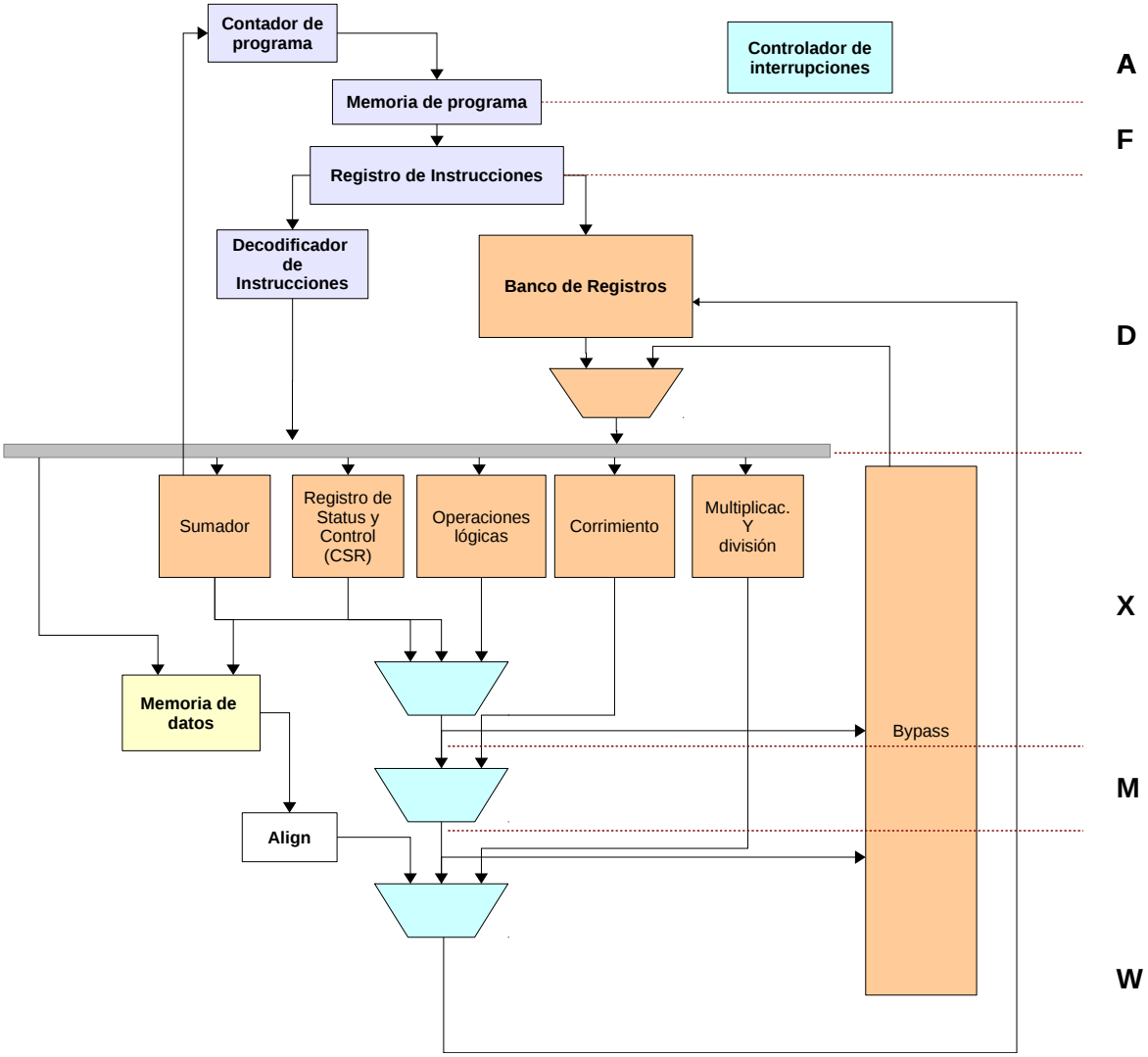

Figura 1.1: Diagrama de bloques del LM32

Cuadro 1.1: Registro de Estado y Control

Nombre	Index	Descripción
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 zero. El valor de este registro después del reset es *h00000000*

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 e una instrucción, típicamente 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*.

IE Habilitación de interrupción

l registro IE contiene el flag 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.

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*

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.

- Compilación de progamas para el LM32, explicar un ejemplo sencillo puede ser el de tipos de datos comentandop todos los archivos, lm32, crt0.s, etc - Set de instrucciones, con ejemplos en donde sea necesario como en: - llamado a funciones: Ejemplo sencillo que muestre como se pasan parámetros a través de r0, r1, r2. - saltos: If, while, forma - interrupciones explicar como se debe modificar el crt0.s para incluir los vectores de excepción y como se atiende la interrupción. - acceso a memoria externa: Explicar como se mapean los registros de los periféricos a C, y tipos de datos. - Acceso a memoria externa: Bus wishbone: Topologías, señales del WB, arquitectura del conbus, explicar uart y timer. - Como se forma el SoC con el LM32. Diagrama de bloques del SoC, explicando donde quedan los diferentes periféricos.

1.3. 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.

Instrucciones aritméticas

En la figura 1.2 se muestra el camino de datos simplificado de las operaciones aritméticas y lógicas cuyos operandos son registros, y el 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 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
```

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 nos son almacenadas previamente en registros, sino que son almacenadas en la memoria de programa. En la figura 1.3 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],*

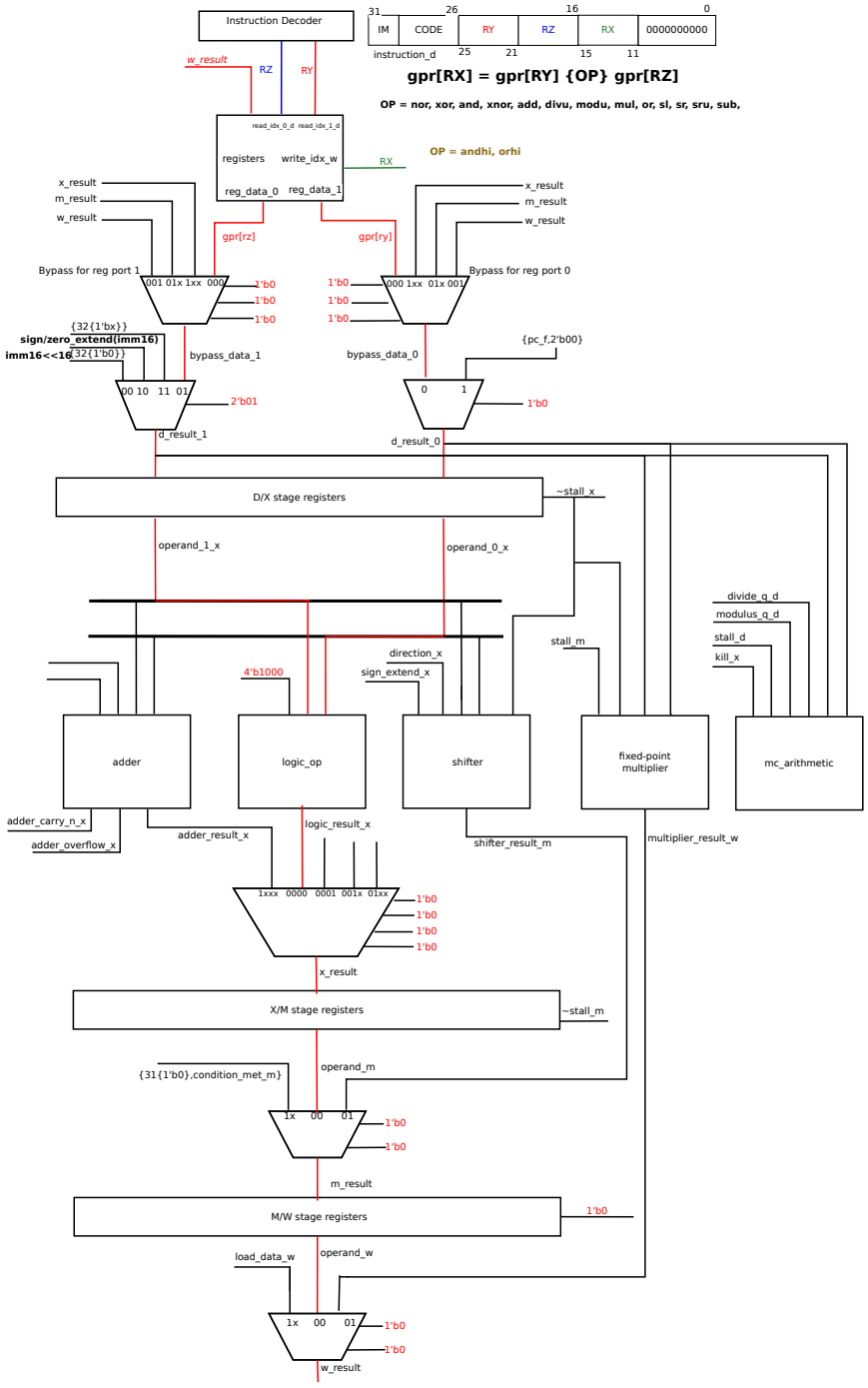

Figura 1.2: Camino de datos de las operaciones aritméticas y lógicas entre registros

16'h0000 respectivamente; el corrimiento de 16 bits a la derecha se hace para poder realizar las operaciones *andhi* y *orhi*, las cuales solo operan sobre la parte alta de los operandos.

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 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 nó la condición (señales rojas en la gráfica); la señal *condition_met_x* se activa si la condición se cumple.

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 amarillas 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 señak *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*; está 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 1.4 se muestra el camino de datos equivalente a las instrucciones relacionadas con condicionales;

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 1.6 se muestra el camino de datos correspondiente a las instrucciones *calli* y *bi*, estas almacenan en la instrucción la dirección y en la figura 1.6 se

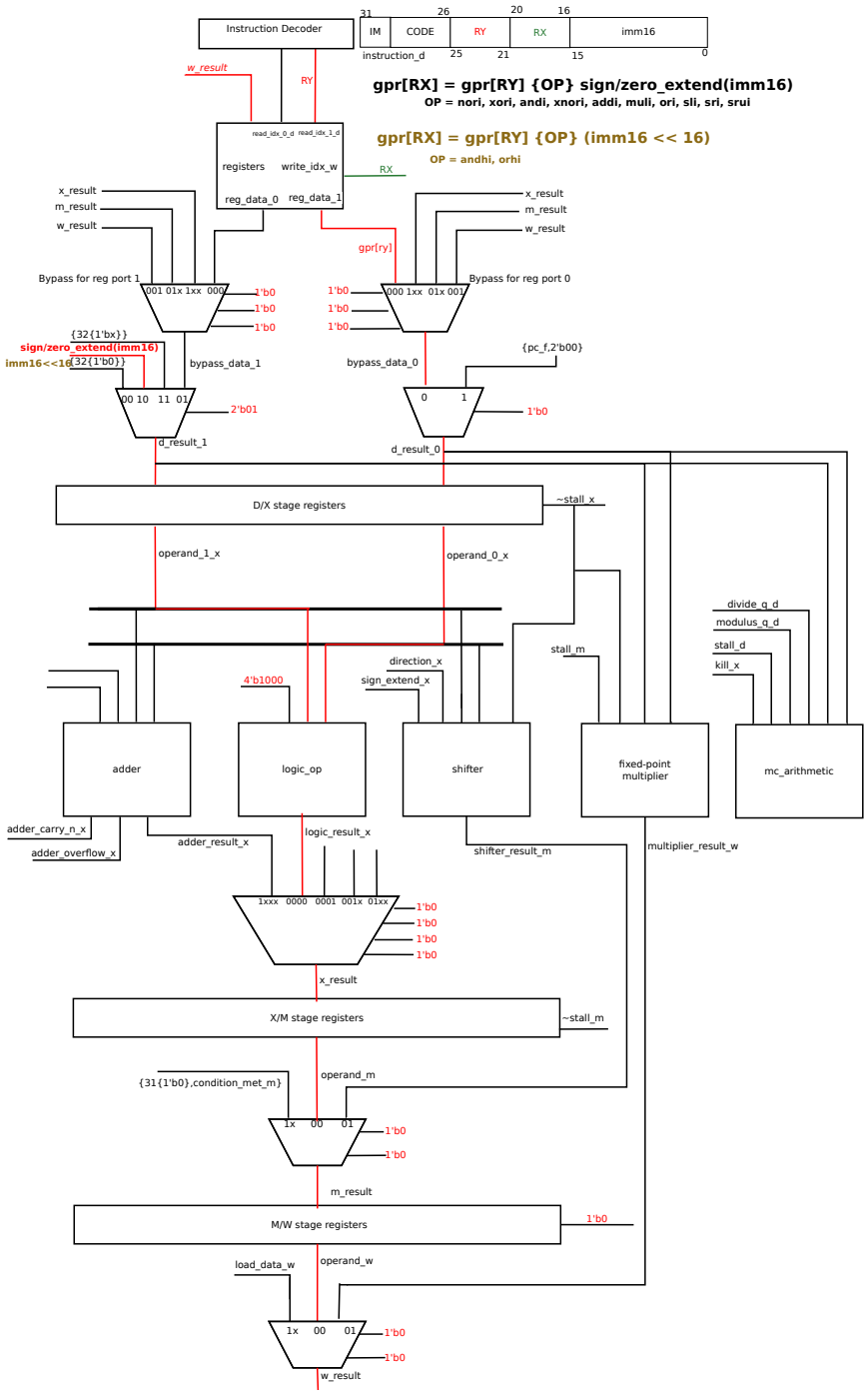

Figura 1.3: Camino de datos de las operaciones aritméticas y lógicas inmediatas

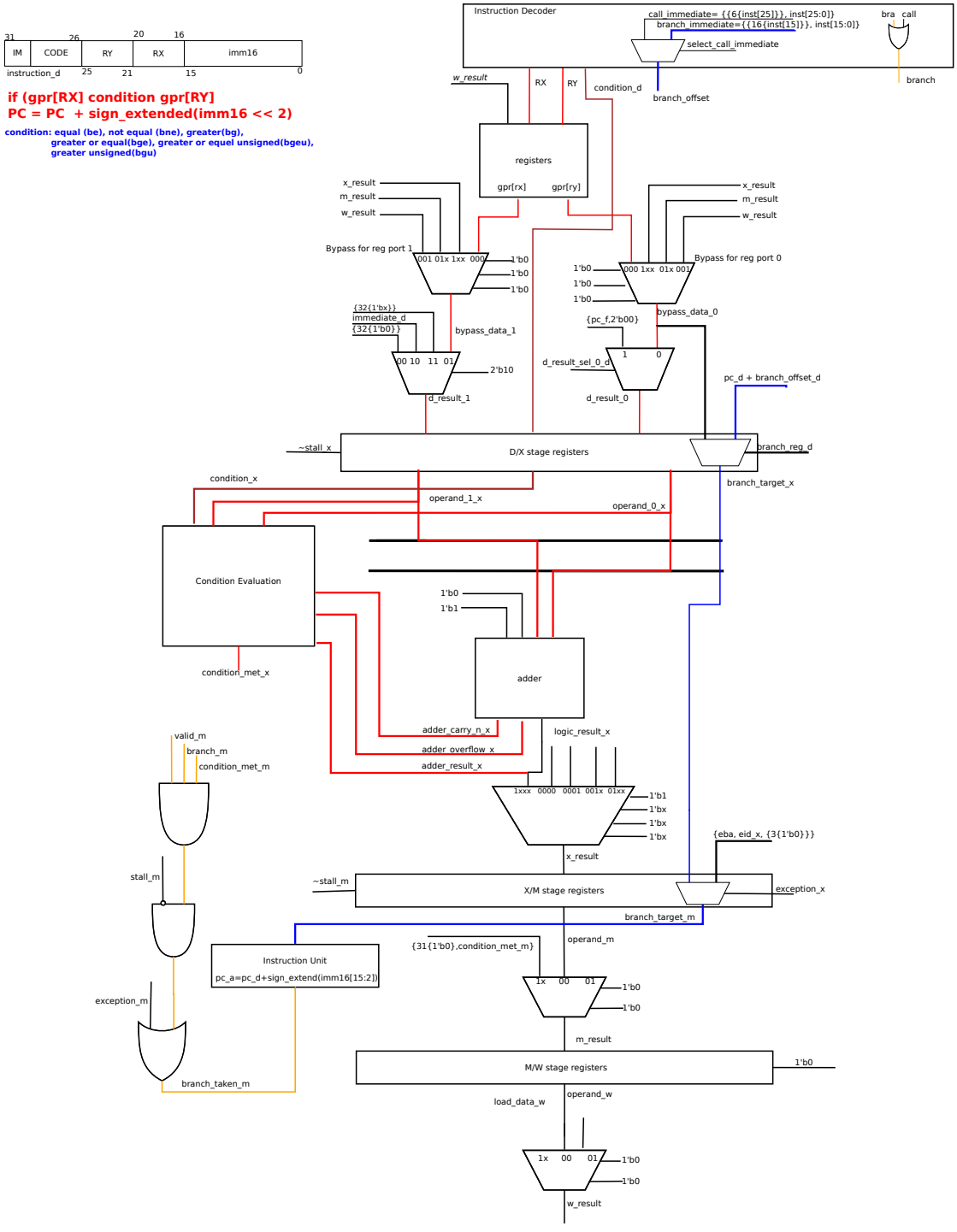

Figura 1.4: Camino de datos de los saltos condicionales

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 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 incondiconales, 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])

    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 las 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 fín 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*.

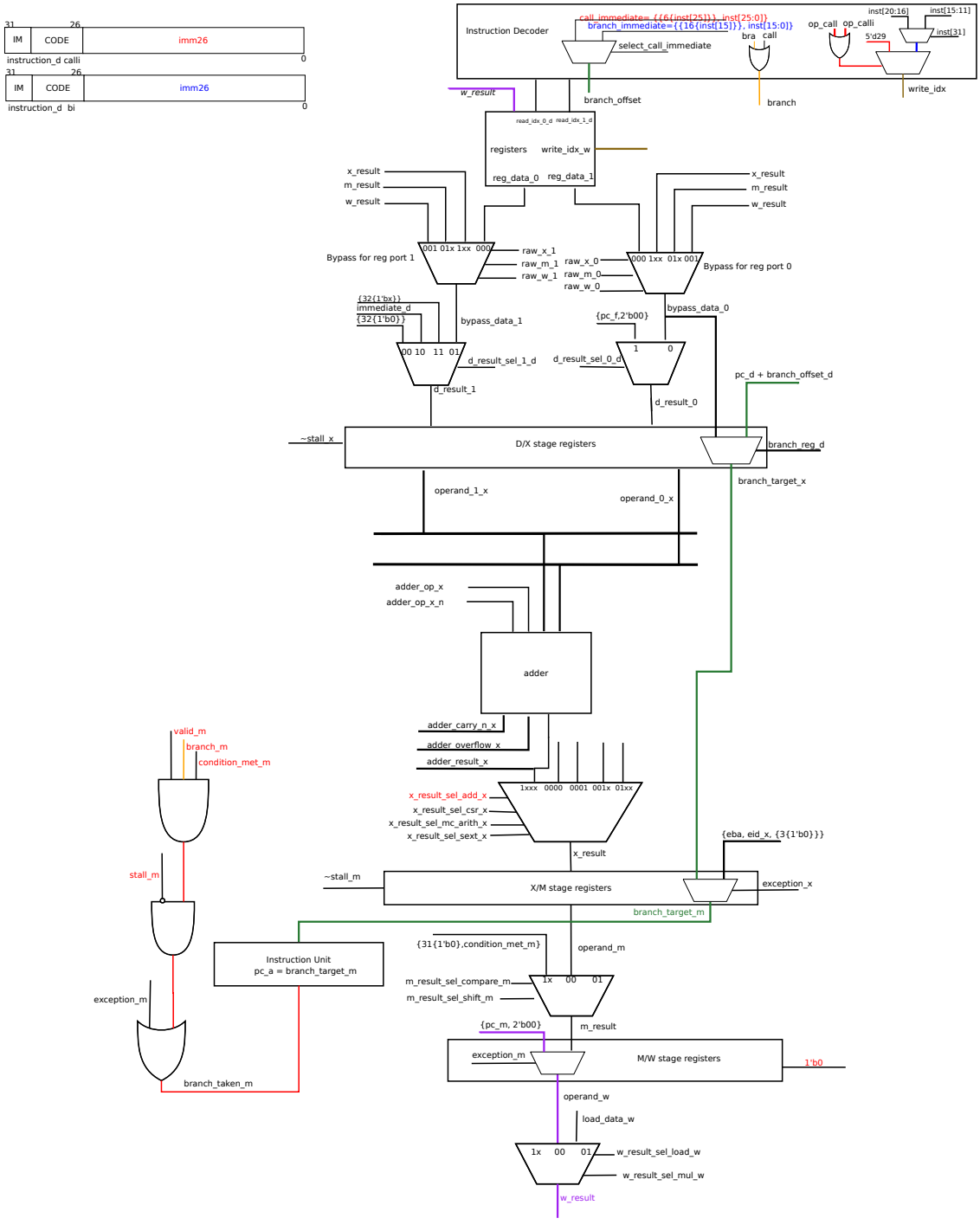

Figura 1.5: Camino de datos de los saltos y llamado a funciones inmediatos

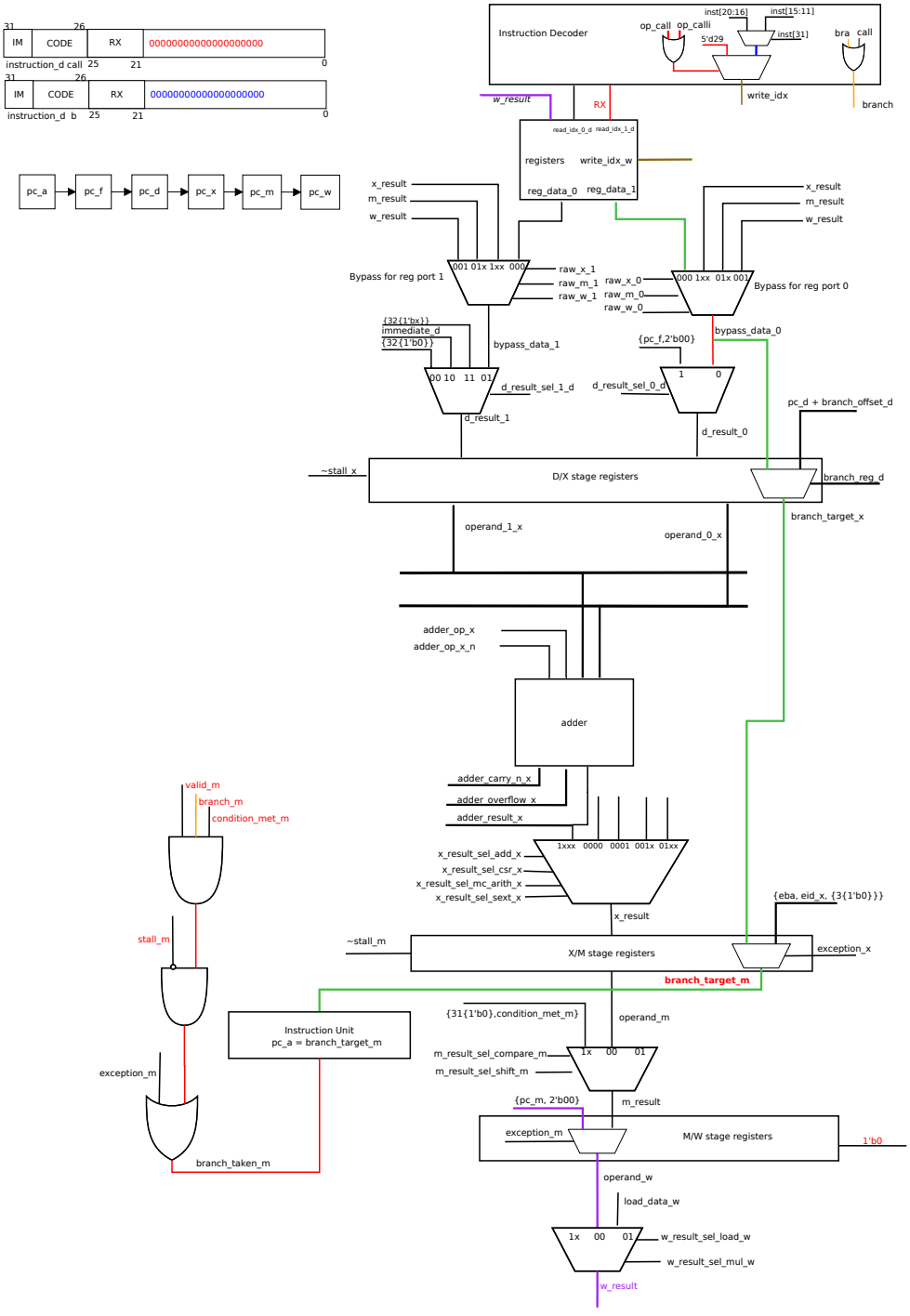

Figura 1.6: Camino de datos de los saltos y llamado a funciones

Comunicación con la memoria de datos

Antes de estudiar el camino de datos correspondiente a este grupo de ibstrucciones, hablaremos de los tipos de datos que soporta el procesador MICO32. En la figura 1.7 se muestran ejemplos de manipulación de diferentes tipos de datos y como estos son tratados en la memoria del procesador.

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 nuevi 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 1.7 ilustra el manejo del tipo de dato *short* el cual es de 8 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 inical 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.

Tipos de datos

En la figura 1.8

Escritura

Lectura

Interrupciones

Explicar el código de atención a la irq

1.4. Arquitectura del SoC LM32

Explicar el componente hardware y software,explicar la estructura sw y hw de los proyectos

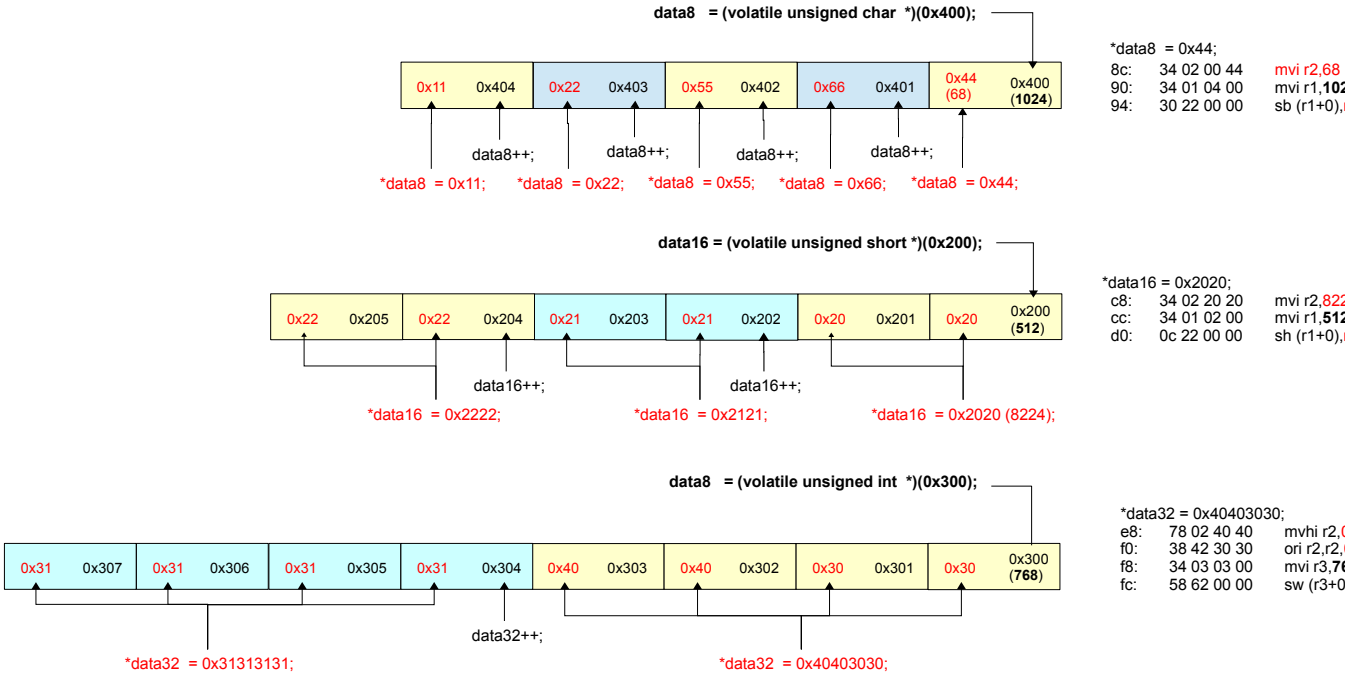

Figura 1.7: Tipos de datos soportados por el procesador Mico32

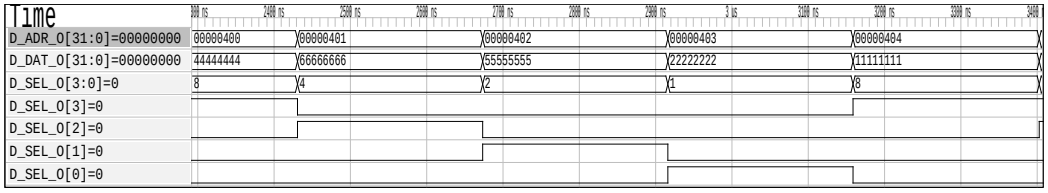

Figura 1.8: Acceso a un data tipo *char*

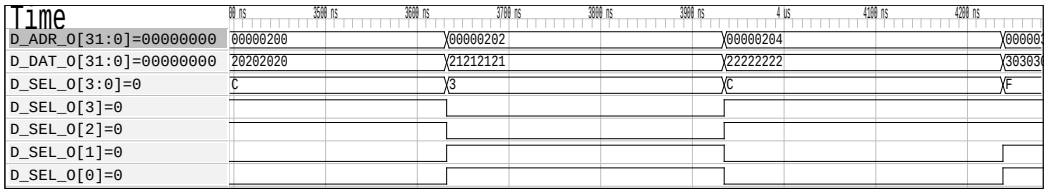

Figura 1.9: Acceso a un data tipo *short*

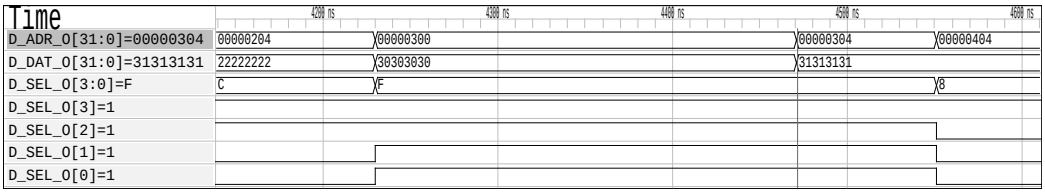

Figura 1.10: Acceso a un data tipo *int*

Bus wishbone

Señales principales

- *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 que 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*: Cuando se activa esta señal se indica al esclavo que ha sido seleccionado. Un esclavo wishbone 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.

Interface del bus wishbone

explicar como funciona el conmax, mostrar diagrama de flujo y simulación

Comunicación con periféricos

EXplicación de la UART y del TIMER globales interface en C.

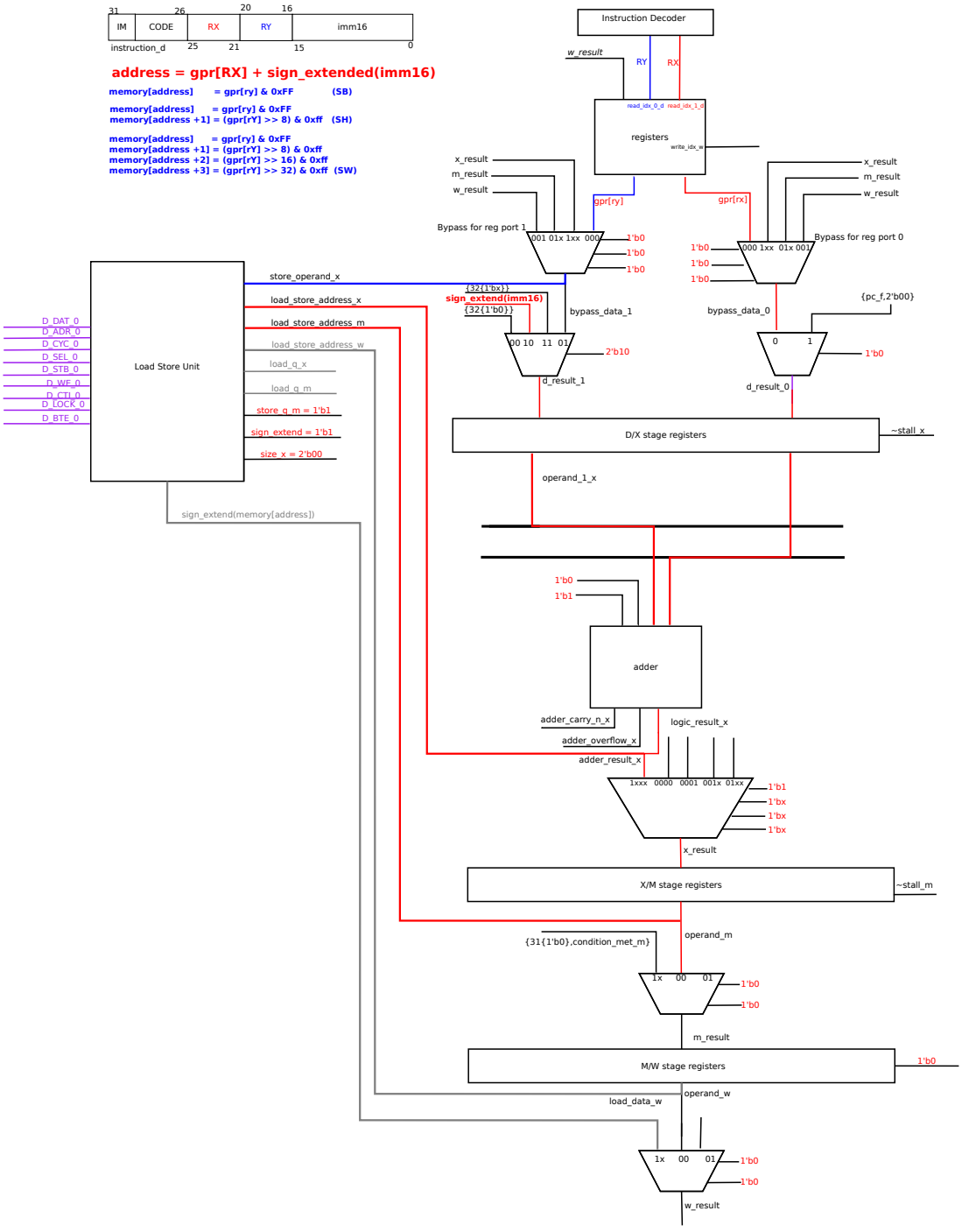

Figura 1.11: Camino de datos de las instrucciones de escritura a memoria

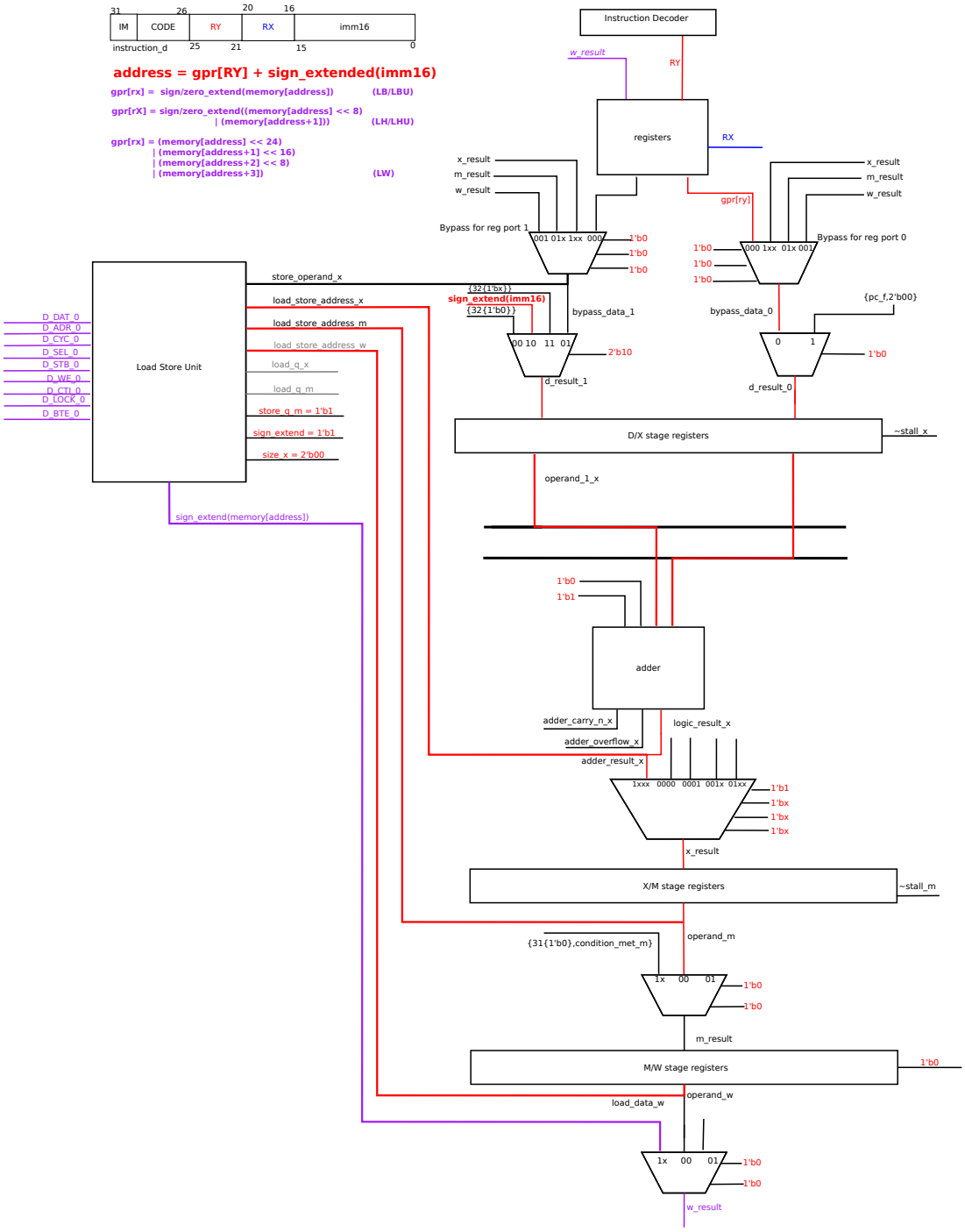

Figura 1.12: Camino de datos de las instrucciones de escritura a memoria

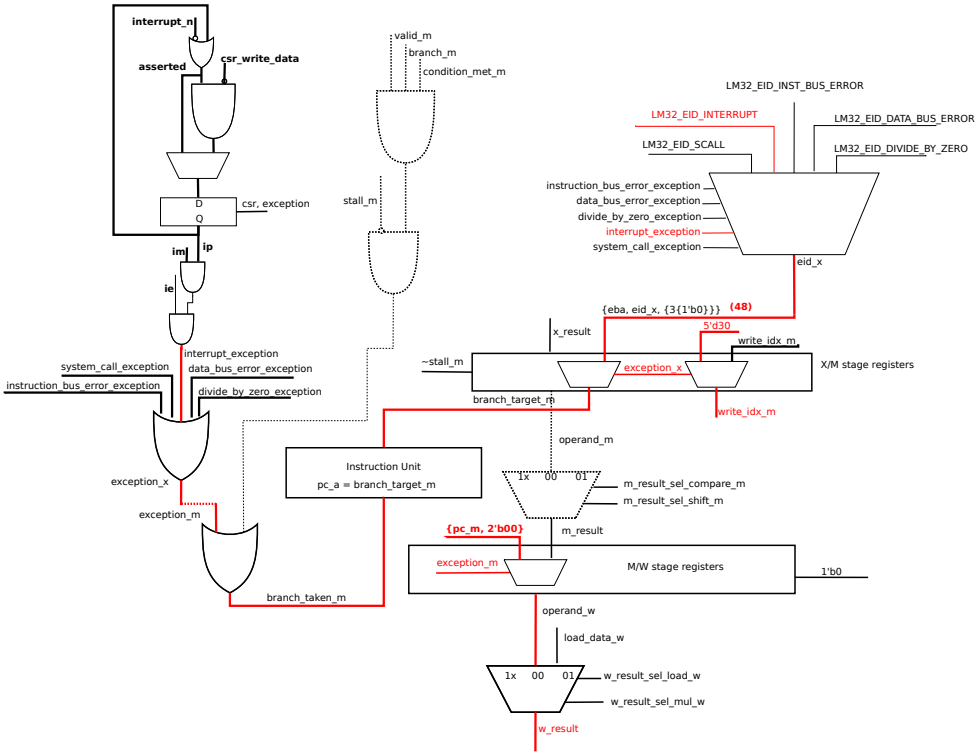

Figura 1.13: Camino de datos correspondiente a las generación de excepciones

Lectura

Escritura

Interfaz Software

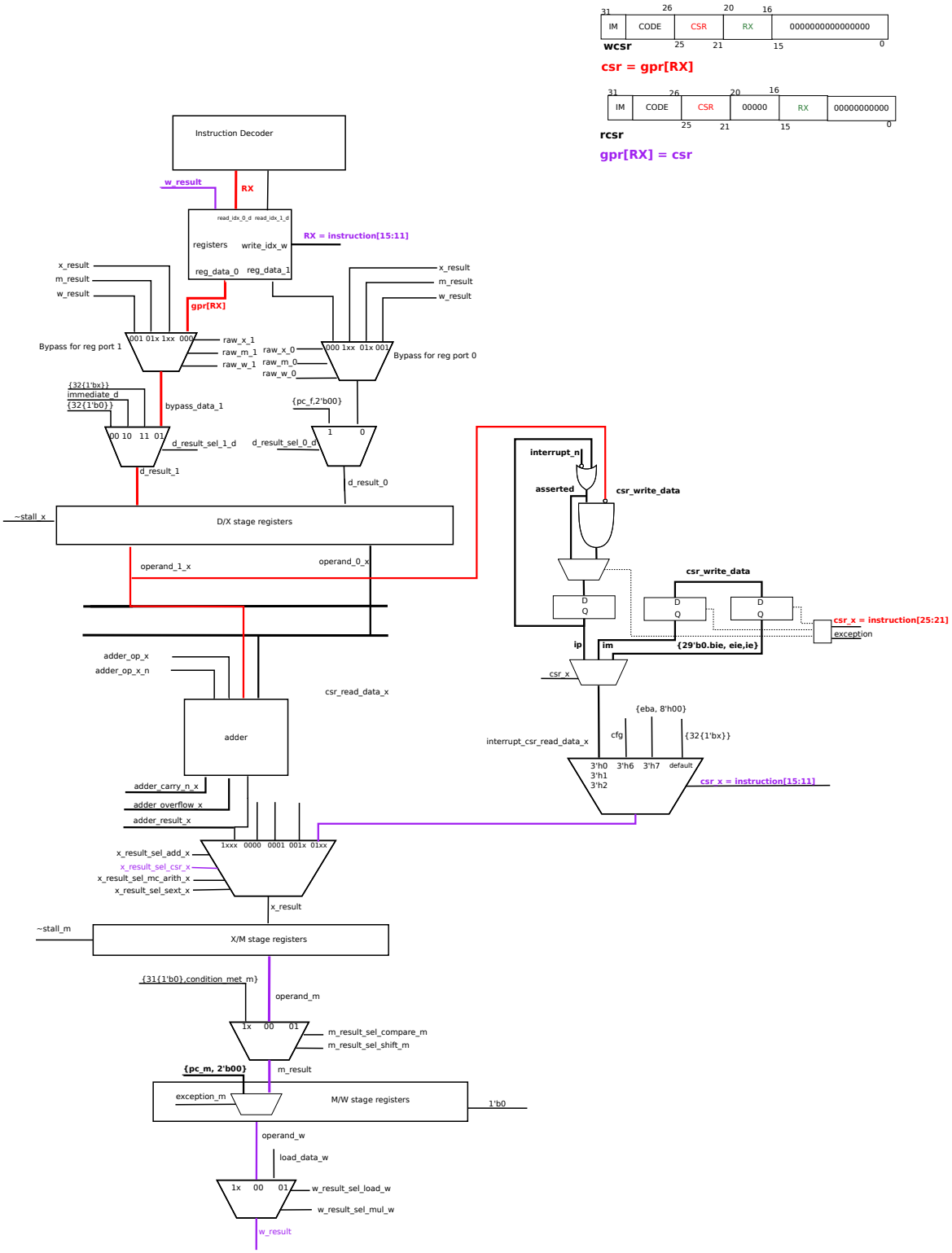

Figura 1.14: Camino de datos correspondiente al acceso de los registros asociados a las excepciones

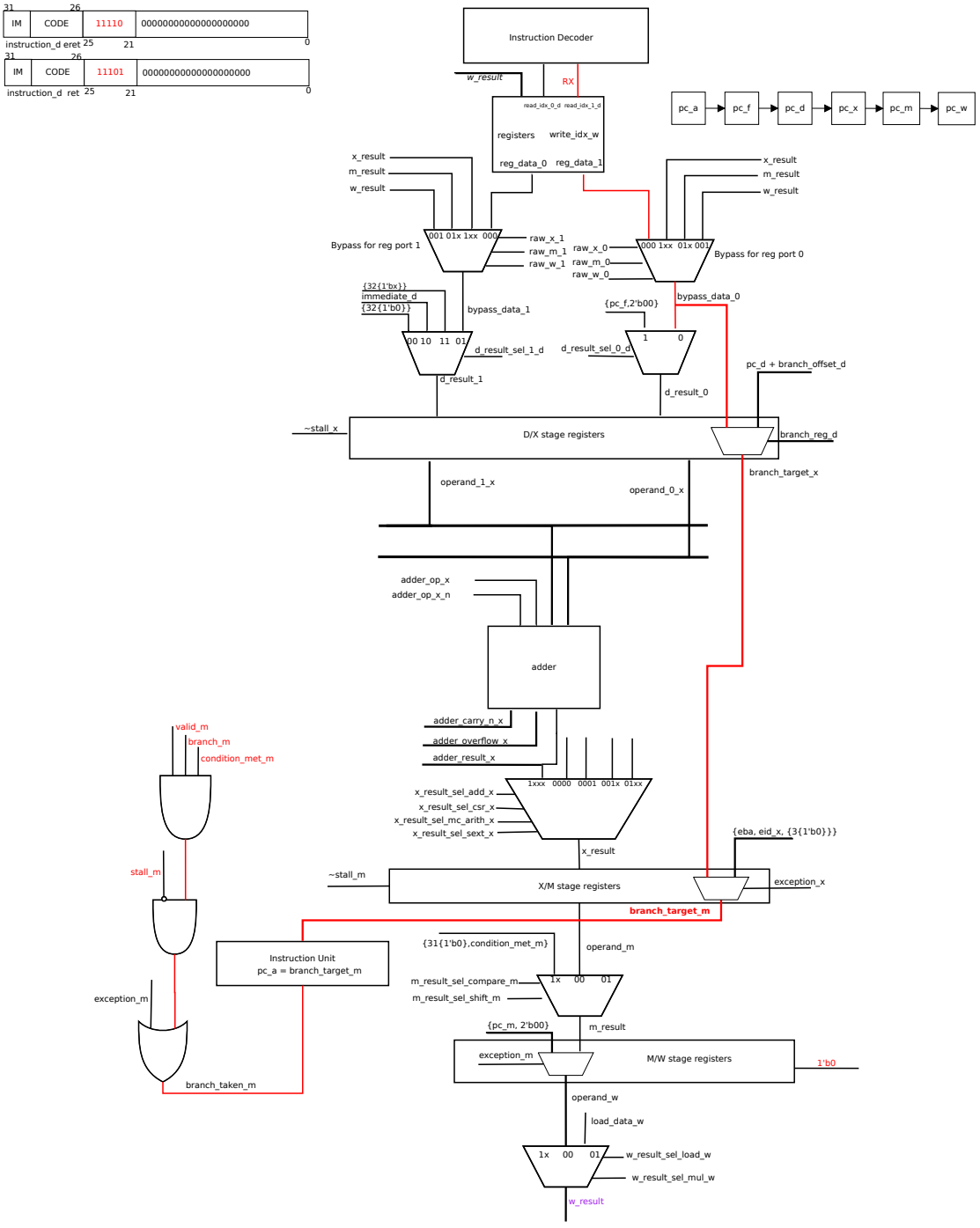

Figura 1.15: Camino de datos asociado al retorno de función y de excepción

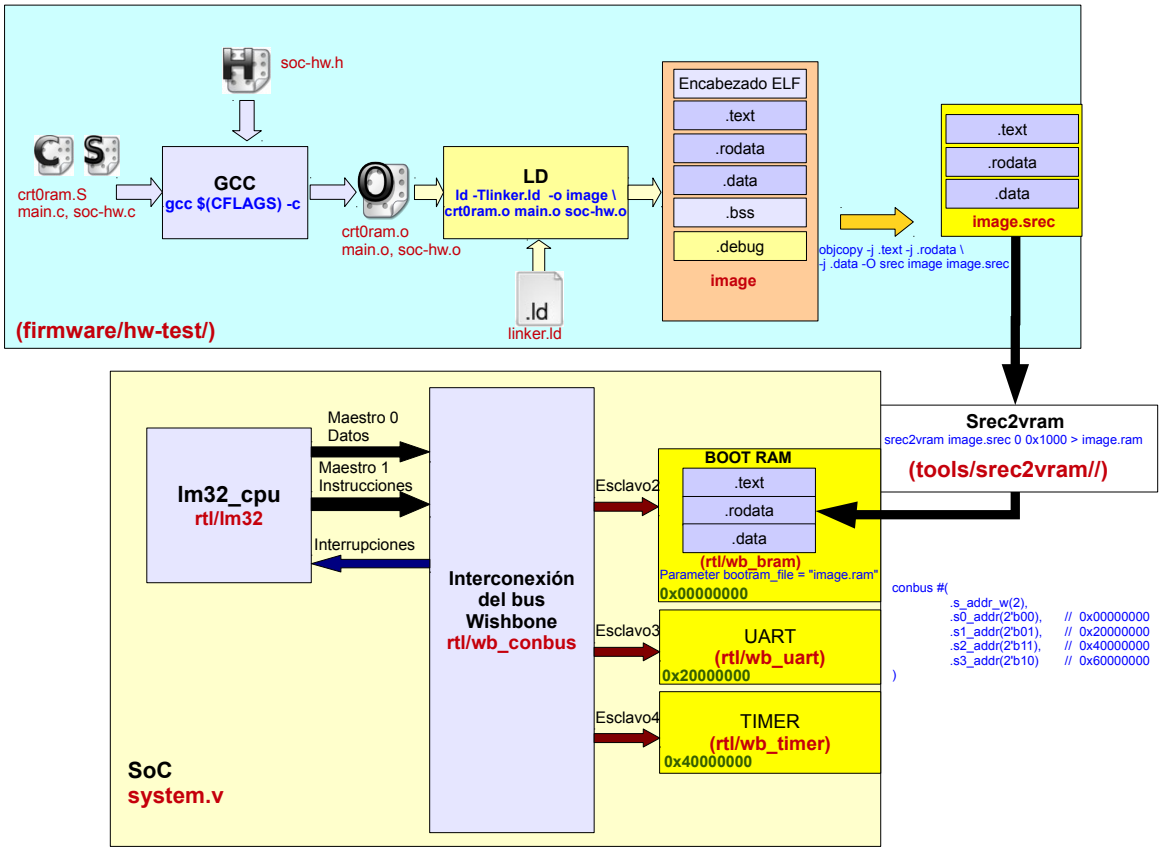

Figura 1.16: Flujo de diseño para el procesador LM32

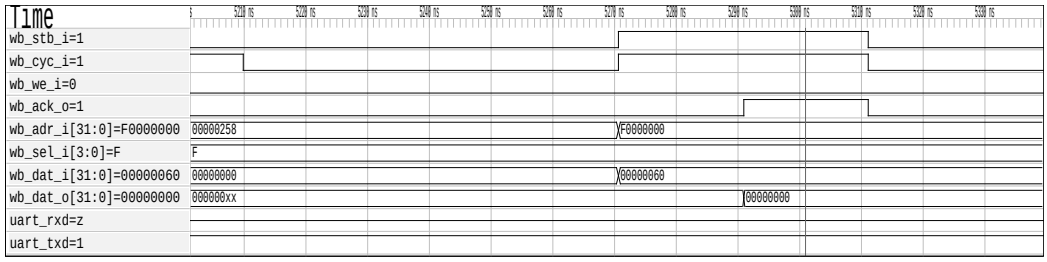

Figura 1.17: Ciclo de lectura del bus wishbone

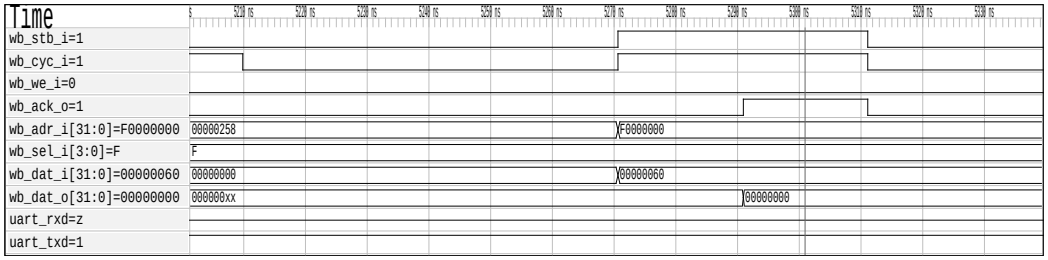

Figura 1.18: Ciclo de escritura del bus wishbone

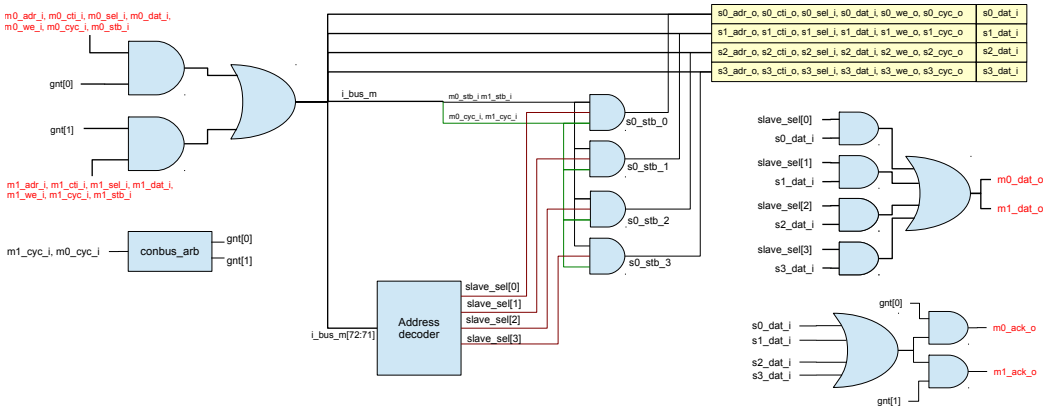

Figura 1.19: Circuito de interconexión del bus wishbone

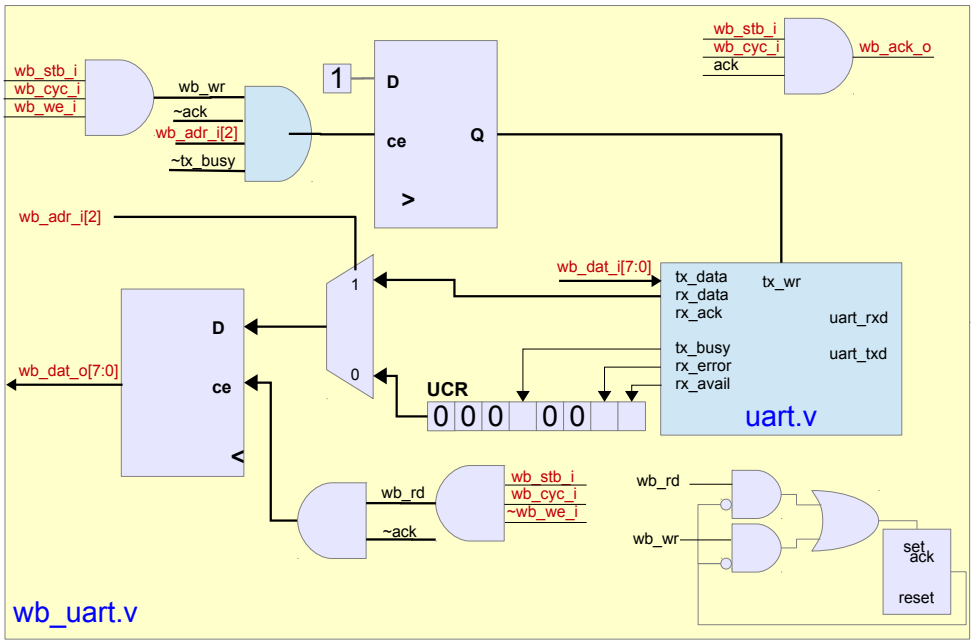

Figura 1.20: Ejemplo de periférico wishbone: UART

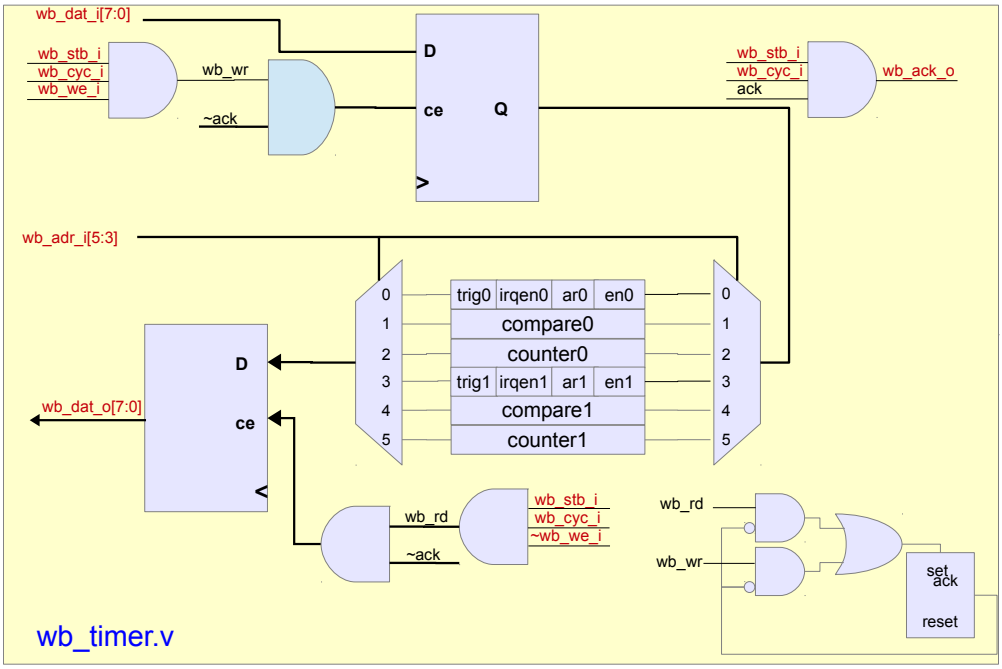

Figura 1.21: Ejemplo de periférico wishbone: TIMER

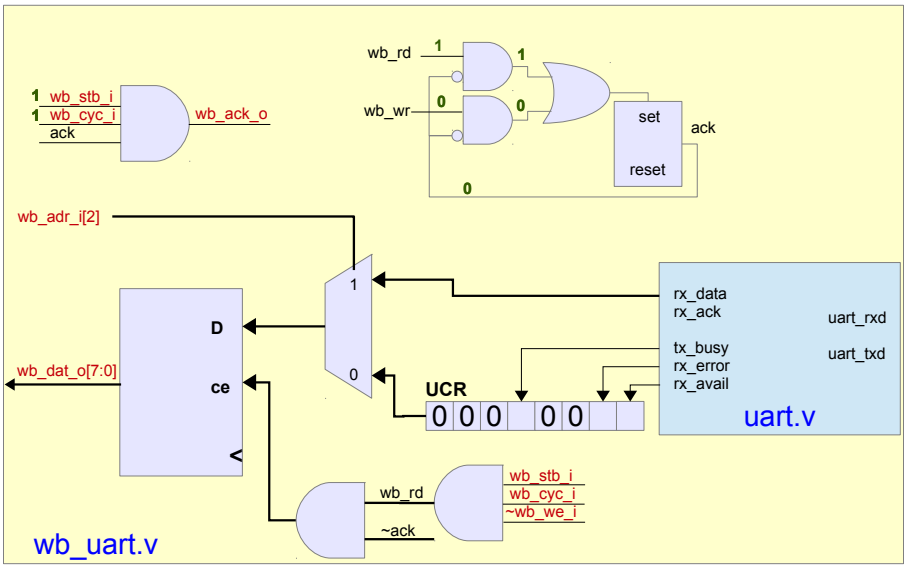

Figura 1.22: Circuito equivalente de lectura de la UART

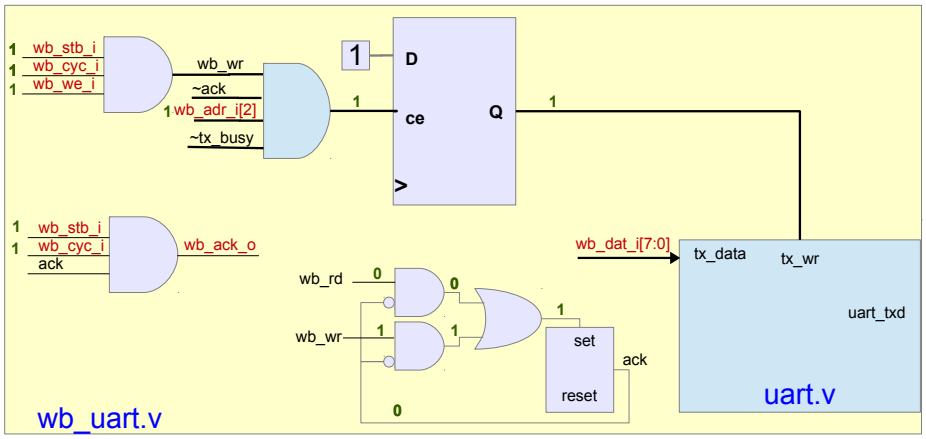

Figura 1.23: Circuito equivalente de escritura de la UART

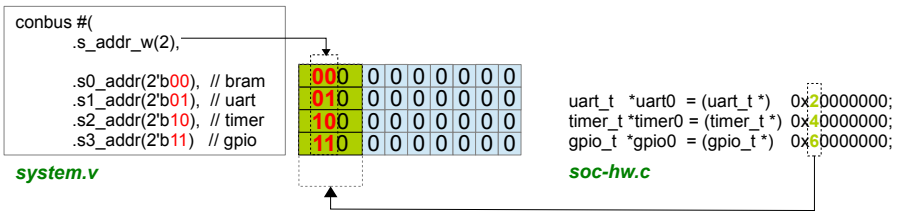

Figura 1.24: Asignación de la dirección de memoria a los periféricos

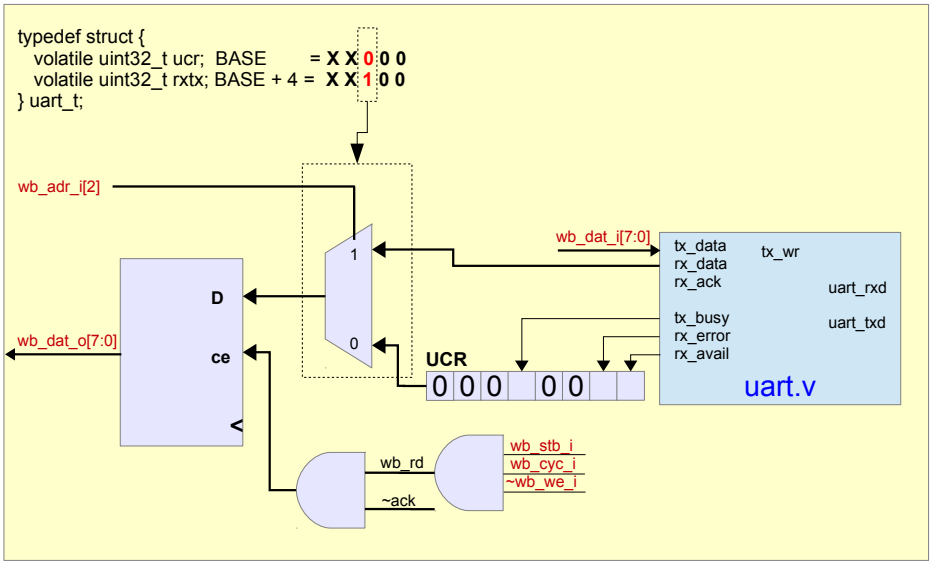

Figura 1.25: Definición de la dirección de los registros internos de la UART

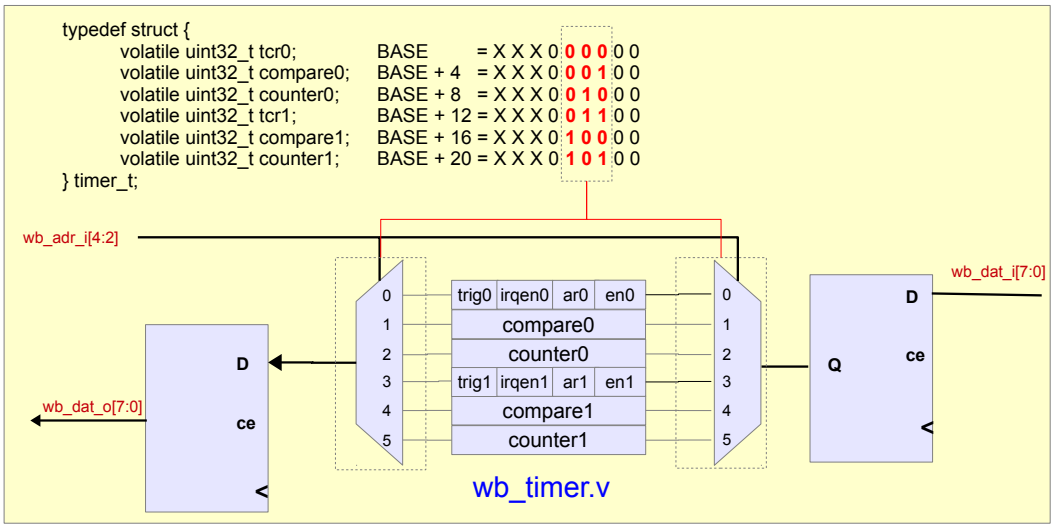

Figura 1.26: Definición de la dirección de los registros internos del TIMER

Bibliografía