11 SISA-I (Simple Instruction Set Architecture-I)

Juan J. Navarro

Toni Juan

Primera versión (SISA-1): 05-Febrero-2002

Última actualización (SISA-I): 29-Noviembre-2010

11.1 Resumen

En este documento se define el lenguaje máquina[1] SISA-I, creado para la enseñanza y el aprendizaje de la arquitectura de un computador RISC. La explicación del SISA-I se hace en 2 grandes apartados y un anexo. En primer lugar se muestra una visión general de la arquitectura, indicando los aspectos relevantes desde el punto de vista del lenguaje máquina (estado del computador, tipos de datos, modos de direccionamiento, formato de las instrucciones, etc.). En el segundo apartado se detalla el conjunto de instrucciones de lenguaje máquina, agrupadas por familias, donde los miembros de cada familia comparten algún tipo de función. Para cada familia se detallan las instrucciones que la forman, los mnemotécnicos usados en el lenguaje ensamblador SISA, el código binario y la función que realiza cada instrucción. Por último, en el anexo se muestra una ficha para cada instrucción en la que se indica el formato en código binario, la semántica de la instrucción, la sintaxis en ensamblador, una descripción textual de su funcionamiento y algún ejemplo de cómo la instrucción modifica el estado del computador. Las fichas están ordenadas alfabéticamente según el mnemotécnico ensamblador de las instrucciones.

1. La definición del nivel de lenguaje máquina de un computador se denomina en ingles "*Instruction Set Architecture*".

11.2 Visión general de la arquitectura

11.2.1 Arquitectura RISC.

SISA-I es una arquitectura RISC de 16 bits. Destacamos los siguientes aspectos:

1. El conjunto de instrucciones es reducido. Solamente hay 22 instrucciones diferentes.
2. Todas las instrucciones de lenguaje máquina son de un tamaño fijo de 16 bits (1 palabra).
3. Todos los operandos fuente y destino de las instrucciones que efectúan cálculos aritméticos, lógicos, comparaciones, etc. tienen los operandos en registros del procesador y dejan el resultado en uno de esos registros. Estas instrucciones tienen dos registros fuente y uno destino que pueden ser distintos. En el procesador hay 8 registros generales de 16 bits para este propósito (`R0`, `R1`,..., `R7`). Para copiar datos entre memoria y registros existen instrucciones de "*load*" (`LD`) y "*store*" (`ST`).
4. La memoria almacena el programa en lenguaje máquina que se está ejecutando y los datos del programa. La memoria se direcciona a nivel de palabra (de 16 bits), esto es, una dirección de memoria hace referencia a una palabra de 16 bits. Las direcciones de memoria son de 16 bits. La capacidad de la memoria es de 2^16 palabras de 16 bits cada una.

11.2.2 Estado del computador

El estado del computador es la información contenida en todos los elementos de memoria del computador (registros, posiciones de memoria, puertos de entrada/salida,...) que son visibles desde el nivel de lenguaje máquina (que pueden ser leídos y/o escritos mediante la ejecución de instrucciones de lenguaje máquina). En el computador suele haber otros elementos de memoria (biestables, registros) que se usan para almacenar resultados intermedios durante las distintas fases de ejecución de una instrucción y que no forman parte del estado del computador[1]. La ejecución de una instrucción de lenguaje máquina provoca una modificación del estado del computador, transformando el estado que tenía el computador antes de su ejecución en un nuevo estado.

El estado del computador de la arquitectura SISA-I lo forman el contenido de los siguientes elementos de memoria:

- Memoria de instrucciones y datos: 2^16 palabras de 16 bits cada una.
- `PC`, registro contador de programa: 16 bits.
- Registros generales: 8 registros de 16 bits cada uno.
- Puertos de entrada/salida: 2^8 puertos de entrada y 2^8 puertos de salida de 16 bits cada uno.

En la definición de un lenguaje máquina (*instructión set architecture*) sólo se especifican los elementos que forman el estado del computador. Se pueden construir distintos procesadores capaces de ejecutar el mismo lenguaje máquina. Cada procesador debe tener los registros que forman parte del estado del

1. El procesador SISP-I-3 tiene algunos registros en la unidad de proceso que no pertenecen al estado del computador que se usan para guardar información temporal durante las fases de ejecución de cada instrucción, pero cuando termina la ejecución de la instrucción el valor de esos registros puede destruirse.

computador, pero puede tener a demás otros registros que no forman parte del estado del computador, que son propios de cada implementación.

A continuación definimos más en profundidad cada uno de estos elementos que forman el estado del computador.

Memoria

Esta arquitectura tiene un único espacio de direcciones de memoria, que se usa tanto para código como para datos, como es habitual[1]. Una dirección de memoria consta de 16 bits y la unidad de direccionamiento es la palabra (word, 16 bits). Se pueden direccionar hasta 2^16 palabras de 16 bits cada una.

El acceso a datos se efectúa mediante las instrucciones de lectura, `LD` (*load*), y escritura, `ST` (*store)*, que copian una palabra (16 bits) de una posición de memoria a un registro o viceversa. La dirección de memoria a la que se accede con estas instrucciones se obtiene según el modo de direccionamiento *registro base más desplazamiento*, que definimos más adelante.

Registro PC. Secuenciación de las instrucciones

SISA-I es una arquitectura con secuenciación implícita de las instrucciones. Tiene un registro especial denominado `PC` (*Program Counter*) que contiene la dirección de memoria sobre la que efectuar la búsqueda de la instrucción a ejecutar[2]. Todas las instrucciones, como resultado de su ejecución, dejan el valor del `PC` incrementado en una unidad, indicando que la siguiente instrucción a ejecutar es la que se encuentra a continuación de la actual. Se incrementa en una unidad porque 1 es el tamaño en palabras de una instrucción y la unidad de direccionamiento de memoria es la palabra (16 bits). A esto se le denomina secuenciación implícita: si no se indica lo contrario, después de ejecutar una instrucción se ejecuta la que se encuentra a continuación (en la dirección siguiente de memoria). Para romper la secuenciación implícita hay que ejecutar una instrucción de ruptura de secuencia (instrucción de salto).

Las instrucciones de ruptura de secuencia son las únicas instrucciones que durante su ejecución pueden modificar el valor del `PC` de otra forma que no sea incrementándolo en una unidad. En SISA-I hay dos instrucciones de salto. Son dos instrucciones de salto condicional en función del valor de un registro general. La instrucción `BZ` (*Branch on Zero*) rompe la secuencia (no continúa el flujo secuencial de ejecución de instrucciones) si el contenido del registro especificado en la instrucción es cero, mientras que la instrucción `BNZ` rompe la secuencia si el registro no contiene el valor cero. Estas instrucciones calculan la dirección destino del salto (dirección de la siguiente instrucción a ejecutar) de modo relativo al `PC`, esto es, suman un desplazamiento al `PC`, para formar el nuevo `PC` que se usará para

1. No obstante, las dos primeras implementaciones de este lenguaje que realizamos, los computadores basados en los procesadores SISP-I-1 y SISP-I-2, son versiones Hardvard de esta arquitectura. Esto es, tienen dos memorias diferenciadas de 2^16 palabras cada una, una para instrucciones y otra para datos. La de datos es una memoria RAM, de lectura/escritura, y es en esa memoria donde se accede con las instrucciones de load y store. A la memoria de instrucciones accede el procesador para buscar cada una de las instrucciones del programa que se está ejecutando. Como no se puede escribir en esta memoria con ninguna instrucción de store (ya que la arquitectura SISA no tiene definido un store (ni un load) sobre esta memoria, porque supone que existe una única memoria) se usa una memoria ROM, de sólo lectura, en la que se escribe el programa cuando esta se fabrica. Así, para cambiar de programa hay que cambiar de ROM. El computador basado en el procesador SISP-I-3 sí que tiene una única memoria para instrucciones y datos, como define el SISA-I.
2. También se le denomina `IP` (*Instruction Pointer*).

buscar la siguiente instrucción. El desplazamiento se encuentra en la propia instrucción de salto, codificado en complemento a dos con 8 bits. Así, después de ejecutar una instrucción de salto se puede pasar a ejecutar una instrucción que se encuentra a una distancia de entre -128 a +127 instrucciones antes o después de la instrucción de salto.

En este documento, para no resultar repetitivo, al explicar qué hace cada instrucción al ejecutarse, no se indica explícitamente que como resultado de la ejecución de la instrucción el `PC` queda incrementado en una unidad, apuntando a la siguiente instrucción a ejecutar. Sólo se indicará como se modifica el `PC` al especificar las instrucciones de ruptura de secuencia.

Registros generales

Existe un conjunto de registros accesible con instrucciones de lenguaje máquina. Está formado por 8 registros de propósito general de 16 bits: `R0, R1,...,R7`. Estos registros se usan:

- como operandos fuente en instrucciones aritméticas y de comparación con números naturales y enteros, en instrucciones de operaciones lógicas bit a bit, en instrucciones de ruptura de secuencia condicional (sobre el que se evalúa la condición del salto), en todas las instrucciones de acceso a memoria (para contener la dirección base del acceso), en las instrucciones de almacenamiento en memoria, *store* y en la de salida de datos, *output*, (para contener el dato a escribir en memoria y en el puerto de salida);
- como operando destino, para almacenar el resultado, en instrucciones aritméticas y de comparación con números naturales y enteros, en instrucciones de operaciones lógicas bit a bit, en las instrucciones de carga de un registro desde memoria, *load*, con un valor inmediato `MOVI`, `MOVHI`, o desde un puerto de entrada, input.

Entrada/Salida

El espacio de direcciones de los puertos de entrada y salida está separado del espacio de memoria. Existen dos espacios separados, uno para la entrada de datos, INPUT y otro para la salida de datos, OUTPUT. Cada uno de estos dos espacios contiene 2^8 puertos de 16 bits cada uno. Las instrucciones `IN` (*input*) y `OUT` (*output*) realizan la lectura y escritura de los puertos del entrada y salida, respectivamente. Son las únicas instrucciones para acceder a los espacios de entrada y salida. Una misma dirección de puerto puede referirse a dos registros físicos diferentes, uno de lectura que se accede con la instrucción `IN`, y otro de escritura al que se accede con `OUT`. No obstante, dependiendo de la implementación, una misma dirección puede referirse a un único registro físico de lectura/escritura, pudiéndose acceder a él tanto con la instrucción `IN` como con la `OUT`.

11.2.3 Tipos de Datos

Los datos que son operandos fuente o destino de las instrucciones se encuentran en elementos de memoria del computador que forman parte del estado del computador: los registros generales del procesador, la memoria, los puertos de entrada/salida y el `PC`.

Los tipos de datos operados por las instrucciones de esta arquitectura son de un único tamaño: una palabra o *word* (16 bits)[1]. No obstante, en algunas instrucciones hay campos de 6 u 8 bits que codifican números naturales en binario o enteros en complemento a 2, que se usan como operandos fuente

(inmediatos). Los valores numéricos de estos campos se representan con 16 bits (en binario o en complemento a 2, según si el campo representa un número natural o entero) antes de ser operados.

Los bits dentro de una palabra, en esta arquitectura, se numeran de 0 a `15`, siendo el bit 0 el bit de la derecha, que representa el bit de menor peso y el bit `15`, el de la izquierda, el bit de mayor peso.

Los tipos de datos, según como los interpretan las instrucciones que los manipulan, son:

- Vectores de 16 bits sobre los que se efectúan operaciones lógicas bit a bit (bitwise).
- Valores booleanos `FALSE` y `TRUE`, que se codifican en binario en un vector de 16 bits mediante el valor 0 y distinto de cero, respectivamente.
- Números naturales codificados en binario con 16 bits (unsigned integers) sobre los que se hacen operaciones aritméticas y de comparación. Las direcciones de memoria y de los puertos de entrada/salida se consideran números naturales codificados en binario.
- Números enteros codificados en complemento a 2 con 16 bits[1] (signed integers) sobre los que se hacen operaciones aritméticas y de comparación.

11.2.4 Modos de direccionamiento

Los operandos fuente y destino de una instrucción se encuentran en alguno de los elementos de memoria cuyo contenido forma el estado del computador. Denominamos modos de direccionamiento a las diferentes formas de especificar, en las instrucciones, dónde se encuentran los operandos fuente y dónde se debe escribir el resultado de una instrucción. Los modos de direccionamiento son las distintas formas que tiene el computador de obtener, para cada operando, el tipo de elemento de memoria de donde se debe leer, o en el que se debe escribir, y su dirección concreta, si es en memoria o en un puerto de entrada/salida, o el número de registro del conjunto de registros.

En general, cada modo de direccionamiento se especifica en la instrucción de una forma diferente. En la arquitectura SISA-I la información de qué modos de direccionamiento se usan para cada operando está implícitamente codificada en el código de operación de la instrucción. Así, el código de operación de la instrucción indica el formato de la instrucción, la función (o parte de ella, ya que algunas instrucciones tienen otros campos para terminar de definirla) y los modos de direccionamiento para los operandos fuente y destino.

Los distintos modos de direccionamiento que aparecen en las instrucciones SISA-I se explican a continuación.

Modo Registro. El operando está (si es fuente) o debe escribirse (si es destino) en un registro general. En la instrucción debe codificarse la información para saber de qué registro se trata (número, o dirección de registro). Los tres bits necesarios para especificar el número de registro de cada operando

1. La definición de palabra (*word*) como un vector, o una tira, de 16 bits no es un estándar, hay arquitecturas en las que se denomina palabra al vector de 32 bits.

1. En Complemento a 2 el bit de más peso, bit 15, es el bit de signo, con peso -2^{15}. El resto de bits tienen el mismo peso y signo que en la interpretación de un número natural codificado en binario.

que usa el modo registro se codifican en campos específicos de la instrucción. El modo registro se usa en el SISA-I para los operandos fuente y destino de todas las instrucciones que efectúan cálculos aritméticos, lógicos, comparaciones, etc. Estas instrucciones no usan ningún otro modo de direccionamiento, excepto la `ADDI`, en la que el segundo operando fuente se encuentra en la propia instrucción (usa el modo inmediato, como se indica a continuación).

Modo inmediato. El operando fuente se encuentra codificado en un campo de la propia instrucción (sólo se usa para operandos fuente). En SISA-I hay dos instrucciones de movimiento que tienen el operando fuente en modo inmediato, `MOVI` y `MOVHI` y una de cálculo, `ADDI`, que tiene el segundo operando fuente en este modo. Para `MOVI` y `MOVHI` el operando inmediato es un vector de 8 bits y para `ADDI` es de 6 bits.

Modo registro base más desplazamiento. Este es un modo de direccionamiento para acceder a operandos en memoria. Se calcula la dirección de memoria del operando sumando al contenido de un registro un valor que puede ser positivo o negativo y que se codifica en la propia instrucción en complemento a dos. El registro general se llama registro base y contiene una dirección de memoria mientras que el valor que se encuentra en la instrucción se denomina desplazamiento y tiene menos bits de los que requiere una dirección de memoria. Este modo permite acceder a datos que se encuentran a una distancia fija (codificada en el desplazamiento de la instrucción) en torno a una dirección de memoria que se encuentra en un registro (que hace de puntero, porque apunta a una dirección de memoria) que puede ir cambiando cada vez que se ejecute la instrucción. En SISA-I todas las instrucciones de acceso a memoria tanto de lectura, `LD`, como de escritura, `ST`, calculan la dirección de memoria con este modo de direccionamiento. El registro base es uno cualquiera de los registros generales del procesador y el desplazamiento está codificado en complemento a 2 en un campo de 6 bits en la propia instrucción.

Modo implícito. Cuando el operando está (si es fuente), o debe escribirse (si es destino) en un registro que no se encuentra codificado en ningún campo específico de la instrucción, porque el registro es fijo para esa instrucción, se dice que se usa el modo de direccionamiento *registro implícito*. El registro está implícitamente codificado en el código de operación de la instrucción y todas las instrucciones con ese código de operación usan el mismo registro. En SISA-I, como en prácticamente todos los lenguajes máquina, podemos decir que todas las instrucciones acceden a un operando fuente y destino usando el modo implícito. Este operando es el registro especial `PC`, que no se especifica en la instrucción, pero que es leído y escrito por todas las instrucciones, para indicar cuál es la siguiente instrucción a ejecutar. Además del acceso al `PC`, solamente hay una instrucción que tiene uno de sus dos operandos fuentes que podemos denominar como implícito, porque no se especifica en la instrucción en un campo separado. Esta es la instrucción `MOVHI Rd, C`, en la que el registro Rd hace de fuente (implícito) y destino.

Por último, decir que también se usa el término *modo de direccionamiento* como la forma de calcular la dirección de la siguiente instrucción a ejecutar en las instrucciones de ruptura de secuencia (esto es, cómo se obtiene la dirección que se carga en el `PC`). En SISA-I hay dos instrucciones de ruptura de secuencia condicional, `BZ` y `BNZ`. Estas instrucciones calculan la dirección destino del salto (cuando se debe romper la secuenciación implícita) sumando al registro `PC` un desplazamiento que se encuentra codificado en complemento a dos en un campo de 8 bits en la propia instrucción. Podríamos llamarlo

modo de direccionamiento registro base implícito más desplazamiento (pues el registro base `PC` no se especifica explícitamente en estas instrucciones), pero se suele llamar **modo relativo al PC**.

11.3 Instrucciones

En esta sección se especifica cada una de las instrucciones SISA-I agrupadas por familias. Las instrucciones de una familia comparten alguna función y/o el formato de sus instrucciones. Para cada familia, en las secciones 11.3.4 a 11.3.10, se indica el formato en lenguaje máquina (campos de bits de las instrucciones en binario y su significado), las instrucciones que forman la familia, su semántica (los cambios que produce la ejecución de la instrucción en el estado del computador) usando una notación simbólica, la sintaxis en lenguaje ensamblador, una descripción textual de la semántica y algunos ejemplos de uso.

La sintaxis en lenguaje ensamblador y la explicación de cómo el programa ensamblador traduce las instrucciones de lenguaje ensamblador a lenguaje máquina, no forman parte del ISA (Instruction Set Architecture), o de la descripción del lenguaje máquina, que es lo que hacemos en este capítulo. No obstante, incluimos aquí la sintaxis de las instrucciones en el lenguaje ensamblador y alguna explicación del paso de ensamblador a máquina. Se pueden definir distintos lenguajes ensamblador para un mismo lenguaje máquina. Aquí usamos los mnemotécnicos del ensamblador que hemos denominado Lenguaje ensamblador SISA-I.

11.3.1 Notación

En adelante, para especificar la semántica de cada instrucción o familia de instrucciones, se usará en varias ocasiones la notación que se indica a continuación. En los casos en que una notación se usa una sola vez, se explica su significado en el momento de usarla. La notación usada en el formato de las instrucciones se define en la siguiente sección que trata del formato.

En esta sección, se efectúan explicaciones generales para algunos grupos de símbolos y luego, para cada símbolo se denota a la izquierda la sintaxis usada y a la derecha su significado. También se dan ejemplos de uso.

Bit y campo de bits de un vector de bits

`X<i>` — Bit `i` del vector de bits `X` (para cualquier símbolo `X`, normalmente una letra mayúscula). Los bits del vector `X` se numeran del `0` al `n-1` empezando por la derecha (por el bit de menor peso, si por ejemplo el vector representa un número natural en binario). Ejemplo:

Si `X` tiene 16 bits y representa un número entero codificado en complemento a 2, `X<15>` es el bit de signo y `X<4>` es el bit de peso 2^4.

`X<j..k>` — Campo de bits del vector de bits `X` (para `j>k` y para cualquier símbolo `X`). En concreto, es el vector de `j-k+1` bits formado por los bits del `k` al `j`, ambos incluidos, del vector `X`. Ejemplo:

Si `X = 1001110111010110` entonces `X<5..3> = 010`.

Contenido de Registros, Memoria y Entrada/Salida

Registros

Cuando nos referimos al vector de bits contenido en el registro `x` usamos el símbolo `Rx`. El símbolo `x` puede ser un dígito del 0 al 7 (ya que es como se numeran los 8 registros generales) o la letra `d`, `a` o `b`. Cuando `x` es un número nos referimos a un registro concreto. Cuando es una letra, nos referimos al registro destino de la instrucción (cuando `x` es igual a `d`), o a uno de los registros que son operandos fuente de la instrucción (cuando `x` es igual a `a` o a `b`).

`Rx` Vector de 16 bits contenido en el registro general `x`, para `x` = 0..7, `d`, `a`, `b`.

Memoria

`MEM[A]` Vector de 16 bits contenido en la palabra (word) de memoria con dirección `A`, siendo `A` un vector de 16 bits.

Entrada/Salida

`INPUT[P]` Vector de 16 bits contenido en el registro `P` del espacio de entrada de datos, siendo `P` un vector de 8 bits.

`OUTPUT[P]` Vector de 16 bits contenido en el registro `P` del espacio de salida de datos, siendo `P` un vector de 8 bits.

Interpretación del valor que representa un vector de bits

Numero Natural

X_u Valor del número natural (*unsigned integer)* representado en binario por el vector de bits `X`. Para un vector de 16 bits:

$$X = x_{15}x_{14} \ \ldots \ x_2x_1x_0 \qquad x_i \in \{0,1\}, \quad X_u = \sum_{i=0}^{15} x_i 2^i$$

Ejemplo: `R3` = `0xFFF9`, $R3_u$ = 65.529

Número entero

X_s Valor del número entero (*signed integer*, o simplemente *integer*) representado en binario por el vector de bits `X`. Para un vector de 16 bits:

$$X = x_{15}x_{14} \ \ldots \ x_2x_1x_0 \qquad x_i \in \{0,1\}, \quad X_s = -x_{15}2^{15} + \sum_{i=0}^{14} x_i 2^i$$

Ejemplo: `R3` = `0xFFF9`, $R3_s$ = -7

Booleano

$X_{boolean}$ Valor booleano representado por el vector de bits X. Si Xu == 0 entonces $X_{boolean}$ = FALSE, en otro caso $X_{boolean}$ = TRUE.

Funciones sobre vectores de bits

SEXT(X) Vector de 16 bits formado por el vector X (de menos de 16 bits) al que se le ha extendido el bit de signo por la izquierda hasta formar un vector de 16 bits. Por ejemplo:

```
SEXT(101101)= 1111111111101101
```

Operadores sobre vectores de bits

Efectúan la operación que indica el operador sobre los dos vectores de bits que se especifican a la izquierda y derecha del operador (excepto para el operador NOT, ~, que sólo tiene un operando que se especifica a la derecha del operador).

Para todos los operadores, los operandos fuente y el resultado son vectores de 16 bits.

Algunas operaciones pueden realizarse sobre números naturales codificados en binario o sobre números enteros en complemento a dos. Se usa el mismo símbolo de operador para los dos tipos de datos siempre y cuando los 16 bits del resultado sean los mismos si se realiza la operación considerando operandos naturales o enteros. Este es el caso de la suma y la resta ya que los 16 bits de menor peso de la operación son los mismos para ambas interpretaciones y no se detecta desbordamiento ni para naturales ni para enteros. Cuando el resultado de la operación es diferente para naturales y para enteros, como por ejemplo en las comparaciones, se diferencian los símbolos del operador, añadiendo una u (unsigned) al símbolo original cuando se trata de números naturales. El caso de números enteros es el caso por defecto, en el que no se le añade nada al símbolo original. Así, < es el símbolo para menor considerando los dos operandos enteros y <u es el símbolo para menor considerando naturales.

Sentencias

`if (x) y else z` Sentencia condicional. x es una expresión condicional sobre vectores de bits que da como resultado el valor 1 o 0. Las sentencias y y z se expresan con la notación que se explica en esta sección. Si x es 1, se efectúa la sentencia y, pero si es 0 se efectúa la sentencia z (el término "else z" puede omitirse).

Usamos los siguientes símbolos condicionales: <, <=, ==, <u, <=u. Los dos primeros para las condiciones menor y menor o igual interpretando los vectores de 16 bits de los operandos como enteros en complemento a dos (*signed integers* o simplemente *integers*). El tercero, el de igualdad, se puede usar tanto para enteros como para naturales ya que tanto en binario como en complemento a dos, existe una representación única para cada número. Los

dos últimos son para las condiciones de menor y menor o igual interpretando los vectores de 16 bits de los operandos como números naturales en binario (*Unsigned integers*).

Asignación

← Asigna vectores de bits. Los vectores, fuente y destino, a derecha e izquierda de la flecha respectivamente, deben tener el mismo número de bits.

Ejemplos:

`Rd ← M[Ra]`

Carga `Rd` con los 16 bits de la posición de memoria cuya dirección es el contenido de Ra.

`Rd<15..8> ← I<7..0>`

Escribe en los 8 bits de mayor peso de `Rd` los 8 bits de menor peso de la instrucción. Deja inalterados los 8 bits de menor peso de `Rd`.

Constante en ensamblador

`C` Constante que aparece como un operando fuente en algunas instrucciones especificadas en lenguaje ensamblador. Por ejemplo, `ADDI Rd, Ra, C`.

La constante `C` puede expresarse en lenguaje ensamblador como un número en decimal con o sin signo. El ensamblador (programa traductor de lenguaje ensamblador a lenguaje máquina) se encarga de codificar el valor de la constante `C` en una tira de bits, que coloca en el campo correspondiente de la instrucción (campo denominado `N6` si tiene 6 bits y `N8` si tiene 8). La codificación se hace en binario o en complemento a 2 dependiendo de la instrucción que se esté traduciendo. En algunas instrucciones (`ADDI`, `LD` y `ST`) en las que el campo `N6` se tiene que interpretar en complemento a 2, en tiempo de ejecución de la instrucción y antes de operar con la tira de bits, se forma un vector de 16 bits a partir de los 6 bits de `N6`, extendiendo el bit de signo. Para otras instrucciones (`MOVI`, `BZ` y `BNZ`) la extensión de signo a 16 bits se hace a partir de los 8 bits de `N8`. Por último, otras instrucciones (MOVHI, IN y OUT) usan en tiempo de ejecución el campo `N8` tal cual.

La constante `C` también se puede expresar en ensamblador como una tira de bits en notación hexadecimal. En ese caso, para que el ensamblador sepa que se está usando notación hexadecimal, en vez de decimal que es la opción por defecto, se precede el vector de dígitos por los símbolos `0x` (un cero seguido de una equis), por ejemplo `0x1D`. En el caso de usar notación hexadecimal, el ensamblador representa en binario los dígitos hexadecimales y trunca los bits de más peso (que deberían valer 0) o completa con ceros a la izquierda hasta obtener los 6 bits del campo `N6` o los 8 de `N8`, según de qué instrucción se trate.

La constante `C`, que puede aparecer como operando en una instrucción en ensamblador, también puede ser una etiqueta o un símbolo definido previamente como una constante, o incluso puede ser una expresión aritmética con etiquetas, símbolos de constantes y constantes (v.g: `L+4`).

Si la constante no puede representarse en el número de bits del campo correspondiente, el ensamblador dará un error de ensamblado.

Cuando explicamos una instrucción que en lenguaje ensamblador tiene una constante `C` decimos algo así como: "La constante `C` especificada en la sentencia ensamblador se codifica en complemento a dos (o en binario, depende de la instrucción de que se trate) en `6` bits en el campo `N6` de la instrucción en lenguaje máquina" (o en 8 bits en el campo `N8`).

11.3.2 Formato de las instrucciones

Todas las instrucciones del lenguaje máquina SISA son de tamaño fijo de 16 bits. Como se muestra en la Figura 11.1 hay 3 formatos de instrucción distintos. Cada formato tiene asociada una "forma" de descomponer los 16 bits de la instrucción en distintos campos. Cada campo codifica una información específica.

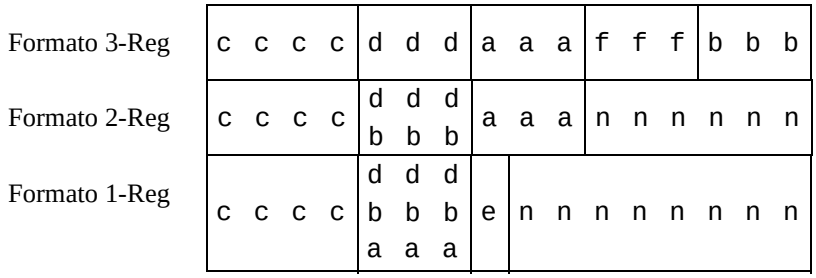

Fig. 11.1 Formatos de las instrucciones SISA-I.

En la explicación de las instrucciones, denotamos con `I` el vector de 16 bits que codifica en lenguaje máquina la instrucción o familia de instrucciones que estamos tratando. Si queremos destacar los distintos campos que forman la instrucción de acuerdo con su formato, separamos los campos por un espacio. Ejemplo: `I` = `1000 110 111 010 110`

Para denotar los bits que forman cada uno de lo distintos campos de una instrucción específica o una familia de instrucciones, usamos letras diferentes. Además, para que resulte menos engorroso a la vista no indicamos los subíndices que marcan el peso de cada uno de los bits del campo (se entiende que el bit de la derecha es el de menor peso).

Todas las instrucciones tienen un campo de código de operación (`cccc`), situado en los bits 15 a 12 de la instrucción, `I<15..12>` = `cccc` (más formalmente: `I<15..12>` = $c_3c_2c_1c_0$). Además, existen campos que pueden verse como una extensión del código de operación, ya que codifican distintas funciones para el mismo valor del campo `cccc`. Las instrucciones con formato 3-Reg tienen el campo

`fff`, `I<5..3>` = $f_2f_1f_0$, que codifica 8 funciones para el mismo código de operación. Además, las instrucciones del formato 1-Reg tienen el campo `e` de un bit, `I<8>`, que codifica dos posibles extensiones del código de operación.

Cada registro especificado en una instrucción se codifica en binario en un campo de 3 bits, ya que hay 8 registros generales (`R0`,..., `R7`). El campo `ddd`, vector de bits $d_2d_1d_0$, codifica en binario el número de registro `d` que es destino de la instrucción, `Rd`. Los registros que son operandos fuente se codifican en los campos `aaa`, para `Ra` y `bbb` para `Rb`.

Las instrucciones que necesitan codificar 3 registros, dos fuentes y un destino, usan el formato 3-Reg. Realizan operaciones aritmético-lógicas y de comparación. Además de los campos de tres bits para codificar el registro destino `Rd` y los dos fuentes `Ra` y `Rb` hay un campo de tres bits, `F` = `fff`, para codificar la función concreta de cada una de las 8 instrucciones que comparten el mismo código de operación.

El formato 2-Reg es para las instrucciones de acceso a memoria (*load* `LD` y *store* `ST`) y para la instrucción `ADDI`, suma con inmediato. Las instrucciones de acceso a memoria requieren un registro fuente `Ra`, como registro base para obtener al dirección de memoria y otro registro fuente, `Rb`, o destino, `Rd`, para el dato a leer o escribir, según se trate de la instrucción `LD` o `ST`. Para estas instrucciones el campo de 6 bits, `nnnnnn` que denominamos `N6` (o más formalmente, `N6` = $n_5n_4n_3n_2n_1n_0$), codifica en complemento a 2 el desplazamiento a sumar al registro base `Ra` para obtener la dirección de memoria a la que se accede. En el caso de la instrucción `ADDI` el campo `N6` representa el número en complemento a 2 a sumar al contenido del registro fuente `Ra`. En la instrucción `ADDI` el resultado de la suma se escribe en el registro destino `Rd`.

Por último, las instrucciones que sólo requieren especificar un registro usan el formato 1-Reg. En este formato hay un bit `e` que sirve como extensión del código de operación. En este formato, hay un campo con un vector de 8 bits, `nnnnnnnn`, que denominamos `N8` que codifica:

- el valor de un número entero en complemento a 2 que hace de operando inmediato en las instrucciones `MOVI` y `MOVHI`,
- el desplazamiento en complemento a 2 para las instrucciones de ruptura de secuencia condicional en modo relativo al `PC`, `BZ` y `BNZ`, o
- la dirección de un puerto de entrada/salida en las instrucciones `IN` y `OUT`.

Un ejemplo de formato de una familia de instrucciones es el siguiente. El vector de 16 bits

`I` = `0001 ddd aaa fff bbb`

denota la familia de las instrucciones de comparación. Para `fff` = `011` es la instrucción `CMPEQ Rd, Ra, Rb`.

Más adelante se indica claramente qué información contiene cada campo de cada instrucción. Por ejemplo, el campo `I<11..9>` puede codificar un registro destino `Rd` o un registro fuente `Rb`, dependiendo del código de operación concreto.

11.3.3 Codificación de las instrucciones

La siguiente tabla muestra el código de operación, campo `I<15..12>` = `cccc`, de todas las instrucciones SISA-I, afinando el significado del resto de campos de la instrucción. Las instrucciones se encuentran ordenadas por valor creciente del código de operación. Los códigos de operación `1000` a `1111` no se usan para instrucciones SISA-I, quedando disponibles para ampliaciones futuras del lenguaje. En la Figura 11.2, a la derecha de cada conjunto de instrucciones que comparten el mismo código de operación, se indica el nombre genérico del conjunto, o de la instrucción cuando sólo hay una, y los mnemotécnicos usados en ensamblador para referirnos a cada una de las instrucciones, en orden ascendente según el campo de función `fff`. La presencia de un guión, en vez de un mnemotécnico, indica que el código de función correspondiente no se usa (se reserva para posibles ampliaciones del lenguaje). Por ejemplo, de las instrucciones con código de operación `0000`, la que tiene código de función `fff` = `000` es la `AND`, la `ADD` tiene código de función `100`, etc.

15 14 13 12	11 10 9	8	7 6 5 4 3 2 1 0	Op. Lógicas y Aritméticas	`AND, OR, XOR, NOT` `ADD, SUB, SHA, SHL`
0 0 0 0	d d d	a a a	f f f b b b		
0 0 0 1	d d d	a a a	f f f b b b	Comparación con y sin signo	`CMPLT, CMPLE, -, CMPEQ` `CMPLTU, CMPLEU, -, -`
0 0 1 0	d d d	a a a	n n n n n n	Add inmediato	`ADDI`
0 0 1 1	d d d	a a a	n n n n n n	Load	`LD`
0 1 0 0	b b b	a a a	n n n n n n	Store	`ST`
0 1 0 1	d d d	0	n n n n n n n n	Mover Inmediato	`MOVI`
	d d d a a a	1			`MOVHI`
0 1 1 0	b b b	0	n n n n n n n n	Salto condicional modo relativo al PC	`BZ`
		1			`BNZ`
0 1 1 1	d d d	0	n n n n n n n n	Input	`IN`
	b b b	1		Output	`OUT`
1 x x x				No usado	8 códigos de operación

Fig. 11.2 Tabla resumen del formato y codificación de las 22 instrucciones SISA-I

11.3.4 Instrucciones aritmético-lógicas

Binario: NOT: `0000 ddd aaa 011 000`

Resto: `0000 ddd aaa fff bbb`

Código, Mnemotécnico, Función y Nombre:

f f f	MNEMO	FUNC	Nombre
0 0 0	AND	&	Bitwise And
0 0 1	OR	|	Bitwise Or
0 1 0	XOR	^	Bitwise Xor
0 1 1	NOT	~	Bitwise Not
1 0 0	ADD	+	Add
1 0 1	SUB	-	Sub
1 1 0	SHA	sha	Shift Arithmetic
1 1 1	SHL	shl	Shift Logic

Semántica: NOT: `Rd ← ~Ra`

SHA y SHL: `Rd ← Ra FUNC Rb<4..0>`

Resto: `Rd ← Ra FUNC Rb`

Ensamblador: NOT: `NOT Rd, Ra`

Resto: `MNEMO Rd, Ra, Rb`

Descripción: Escribe en `Rd` los 16 bits de menor peso del resultado de la operación aritmética o lógica que especifica la instrucción usando como operandos fuente `Ra` y `Rb` (excepto la `NOT` que sólo usa `Ra`). En ningún caso se detecta si el resultado de la operación aritmética sobre los valores representados en los operandos es representable o no en 16 bits, ni para representación de naturales en binario ni para enteros en complemento a 2.

Las instrucciones lógicas (`AND`, `OR`, `XOR` y `NOT`) operan bit a bit (*Bitwise Logical Operations*). El bit `i` del resultado es la operación lógica del bit `i` de cada uno de los dos operandos, para `i = 0,1,...,15.` En el caso de la instrucción `NOT` sólo hay un operando fuente, `Ra`.

Las instrucciones aritméticas `ADD` y `SUB` obtienen los 16 bits de menor peso del resultado de sumar y restar respectivamente, dos números enteros de 16 bits en complemento a dos o dos números naturales de 16 bits en binario, ya que en ningún caso se detecta si el resultado es representable o no en 16 bits según la codificación usada.

Las instrucciones SHA (Shift Arithmetic) y SHL (Shift Logic) escriben en Rd el resultado de desplazar Ra a derecha o a izquierda tantos bits como indican los 5 bits de menor peso de Rb interpretados como un número en complemento a 2. Valores positivos indican desplazamiento a la izquierda y valores negativos desplazamientos a la derecha. En los desplazamientos a la izquierda se copian ceros en los bits de menor peso. En los desplazamientos a la derecha la acción depende de la instrucción: para SHA se copia el bit de signo en los bits de mayor peso, ya que la instrucción es aritmética. Para SHL se copian ceros en los bits de mayor peso, ya que la instrucción es lógica.

11.3.5 Instrucciones de operación con inmediato

Sólo hay una instrucción de operación en la que el segundo operando fuente es inmediato.

Suma con Inmediato

Binario: `0010 ddd aaa nnnnnn`

Semántica: `Rd ← Ra + SEXT(N6)`

Ensamblador: `ADDI  Rd, Ra, C`

Descripción: Escribe en Rd los 16 bits de menor peso de la suma del contenido de Ra con el vector de 16 bits resultante de extender el bit de signo del campo N6 = nnnnnn.

Puede usarse para: a) sumar una constante con rango de -32 a +31 a un número entero interpretando el resultado como entero, b) sumar una constante en el rango 0 a 31 a un número natural interpretando el resultado como natural y c) restar una constante en el rango 0 a 32 a un número natural interpretando el resultado como natural.

La constante C expresada en la sentencia ensamblador se codifica con 6 bits en complemento a dos en el campo N6 de la instrucción.

11.3.6 Instrucciones de comparación de enteros y naturales (con y sin signo)

Binario: `0001 ddd aaa fff bbb`

Código, Mnemotécnico, Función y Nombre:

f f f	MNEMO	CMP	Nombre
0 0 0	CMPLT	<	Less than (integers); menor que (enteros)
0 0 1	CMPLE	<=	Less or Equal (integers); menor o igual (enteros)
0 1 0	---	---	---
0 1 1	CMPEQ	==	Equal than; igual que

f f f	MNEMO	CMP	Nombre
1 0 0	CMPLTU	<u	Less than (unsigned); menor que (naturales)
1 0 1	CMPLEU	<=u	Less or Equal (unsigned); menor o igual (naturales)
1 1 0	---	---	---
1 1 1	---	---	---

Semántica: `if (Ra CMP Rb) Rd ← 1 else Rd ← 0`

Ensamblador: `MNEMO Rd, Ra, Rb`

Descripción: Se evalúa la condición especificada en la instrucción sobre los valores representados por los vectores de bits contenidos en `Ra` y `Rb`. El resultado de la evaluación de la condición es el valor `1` (`TRUE`) o `0` (`FALSE`), codificado en binario con 16 bits, según se cumpla o no la condición. El resultado de la evaluación se escribe en `Rd`.

Cuando el mnemotécnico termina en `u` la comparación se efectúa considerando los operandos números naturales codificados en binario (*Unsigned integers*). Cuando el mnemotécnico no termina en `u`, las comparaciones se efectúan considerando los operandos números enteros codificados en complemento a dos (*signed integers* o simplemente *integers*).

La instrucción `CMPEQ` sirve tanto para números naturales como para enteros, ya que tanto en binario como en complemento a dos, existe una representación única para cada número.

11.3.7 Instrucciones de acceso a memoria

Binario: `LD: 0011 ddd aaa nnnnnn`

`ST: 0100 bbb aaa nnnnnn`

Código, Mnemotécnico y Nombre:

c c c c	MNEMO	Nombre
0 0 1 1	LD	Load (word)
0 1 0 0	ST	Store (word)

Semántica: `LD: Rd ← MEM[SEXT(N6)+Ra]`

`ST: MEM[SEXT(N6)+Ra] ← Rb`

Ensamblador: `LD: LD Rd, C(Ra)`

`ST: ST C(Ra), Rb`

Descripción: Se extiende el bit de signo de los 6 bits del vector `N6` hasta completar 16 bits antes de sumarlos al contenido del registro `Ra` para obtener la dirección de memoria a la cual se accede.

La instrucción `LD` lee la palabra (16 bits) de memoria cuya dirección está especificada en los campos `Ra` y `N6` de la instrucción y la escribe en `Rd`.

La instrucción `ST` escribe el contenido de `Rb` (16 bits) en la dirección de memoria especificada en los campos `Ra` y `N6` de la instrucción.

Con estas instrucciones se puede acceder a posiciones de memoria que se encuentran a una distancia entre -32 y +31 palabras en torno a la dirección que contiene `Ra`. El campo `N6` se interpreta como un número entero codificado en complemento a 2 con 6 bits, denominado desplazamiento.

La dirección de memoria es un número natural codificado en binario con 16 bits (con rango de 0 a 65.536). Aunque se calcule sumando un número natural (el contenido del registro base que se interpreta como una dirección) con un número entero (el desplazamiento que se codifica en la instrucción con 6 bits en complemento a 2), el resultado de la suma siempre se interpreta como un número natural codificado en binario con 16 bits, ya que es una dirección de memoria.

La constante `C` que aparece en la especificación de la instrucción en ensamblador se codifica con 6 bits en complemento a dos en el campo `N6` de la instrucción de lenguaje máquina.

11.3.8 Ruptura de secuencia (saltos)

Binario: `0110 bbb e nnnnnnnn`

Código, Mnemotécnico y Nombre:

e	MNEMO	Nombre
0	BZ	Branch on zero
1	BNZ	Branch on not zero

Semántica:

```
BZ:  if (Rb==0)  PC ← PC + SEXT(N8)
           else   PC ← PC + 1
BNZ: if (Rb<>0)PC ← PC + SEXT(N8)
           else   PC ← PC + 1
```

Ensamblador: `MNEMO  Rb, label`

Descripción: Se extiende el bit de signo de los 8 bits del campo `N8`, hasta completar 16 bits, antes de sumarlos al `PC` para obtener la dirección destino en caso de

salto tomado. Si se cumple la condición especificada en la instrucción (`Rb==0` para `BZ` y `Rb<>0` para `BNZ`) se carga en el `PC` la dirección de salto tomado que se ha calculado. Si no se cumple la condición se carga el `PC` con la dirección de la siguiente instrucción en secuencia (que se encuentra en la dirección `PC+1`).

Con estas instrucciones de salto se puede romper la secuencia de ejecución y pasar a ejecutar una instrucción que se encuentra a una distancia de -128 a 127 instrucciones en torno a la instrucción de salto.

La etiqueta `label` que aparece en la sentencia ensamblador es el nombre de la etiqueta definida en la línea de código ensamblador donde se encuentra instrucción a ejecutar si se cumple la condición que provoca la ruptura de secuencia. El programa ensamblador codifica en complemento a 2 en el vector de 8 bits `N8` el valor que se obtiene de restar el valor de la etiqueta menos la dirección en la que se encuentra la instrucción de salto.

11.3.9 Movimiento registro - inmediato

Binario: `0101 ddd e nnnnnnnn`

Código, Mnemotécnico y Nombre:

e	MNEMO	Nombre
0	MOVI	MOV Immediate
1	MOVHI	MOV High Immediate

Semántica: `MOVI:` `Rd ← SEXT(N8)`

`MOVHI:` `Rd<15..8> ← N8`

Ensamblador: `MNEMO Rd, C`

Descripción: En la instrucción `MOVI`, se extiende el bit de signo del vector de 8 bits `N8` para formar el vector de 16 bits que se carga en el registro `Rd`.

La instrucción `MOVI` puede usarse para cargar en un registro general un número entero en complemento a 2 constante que se encuentre en el rango de -128 a + 127. También puede usarse para cargar en un registro general un número entero en binario, que se encuentre en el rango de 0 a 127 o de 65.408 a 65.535.

En la instrucción `MOVHI`, el vector de 8 bits `N8` se escribe en los 8 bits de mayor peso de `Rd`, y se mantienen inalterados los 8 bits de menor peso de `Rd`. Otra forma de especificar la semántica de la instrucción es decir que se escribe en `Rd` los 16 bits formados por los 8 bits del campo `N8` a los que se les concatena por la derecha los 8 bits de menor peso de `Rd`. Esto quiere decir que esta instrucción tiene un operando fuente implícito, que es igual al

operando destino, ya que `Rd` actúa como fuente (los 8 bits de menor peso) y como destino (los 16 bits).

La constante `C` especificada en la sentencia en ensamblador `MOVI` se codifica en 8 bits en complemento a 2 en el campo `N8` de la instrucción en lenguaje máquina, mientras que para la instrucción `MOVHI`, se codifica en binario.

Ejemplo: Para cargar una constante arbitraria en un registro se puede usar una secuencia `MOVI` seguida de `MOVHI`, tal y como muestra el siguiente ejemplo: cargar `0x34ab` en `R4`

```
MOVI    R4, 0xab
MOVHI   R4, 0x34
```

11.3.10 Entrada/Salida

Binario: `IN: 0111 ddd 0 nnnnnnnn`

`OUT: 0111 bbb 1 nnnnnnnn`

Semántica: `IN: Rd ← INPUT[N8]`

`OUT: OUTPUT[N8] ← Rb`

Ensamblador `IN: IN Rd, C`

`OUT: OUT C, Rb`

Descripción: La instrucción `IN` lee los 16 bits del puerto (o registro) del espacio de entrada (`INPUT`) cuya dirección está especificada en el campo `N8` de la instrucción y los escribe en `Rd`.

La instrucción `OUT` escribe el contenido de `Rb` (16 bits) en el puerto (o registro) del espacio de salida (`OUTPUT`) cuya dirección está especificada en el campo `N8` de la instrucción.

La constante `C` especificada en la sentencia en ensamblador es la dirección del registro de entrada o salida que se codifica en binario con 8 bits en el campo `N8` de la instrucción, `I<7..0>`.

ANEXO: Las 22 instrucciones SISA-I

En este apartado se muestra una ficha para cada instrucción. Las fichas están ordenadas alfabéticamente según el mnemotécnico ensamblador de las instrucciones. En cada ficha se indica el formato en código binario de la instrucción, su semántica usando la notación explicada en la sección 11.3, la sintaxis en ensamblador de la instrucción, una descripción textual de su funcionamiento, algún comentario y ejemplos de uso y ejemplos de cómo la instrucción modifica el estado del computador.

En general, en cada ejemplo de cómo la ejecución de una instrucción modifica el estado del computador, se especifica primero el estado del computador antes de la ejecución de la instrucción (estado antes), indicándose solamente el contenido de los elementos de memoria que serán leídos o escritos por la instrucción, después se especifica la instrucción que se ejecuta (instrucción) y por último se especifica el estado del computador después de ejecutarse la instrucción, indicándose solamente el valor de los elementos de memoria que han sido modificados. El vector de bits que contienen los registros y posiciones de memoria se indica en hexadecimal, pero en las instrucciones que tiene sentido interpretar esos vectores de bits como números naturales o enteros, se da entre paréntesis el valor según la interpretación que se haga. Para simplificar, excepto para las instrucciones de ruptura de secuencia, no se especifica el valor del PC ni antes ni después de la ejecución de la instrucción, aunque todas las demás instrucciones lo modifican incrementándolo en 1.

ADD ADD

Binario: `0000 ddd aaa 100 bbb`

Ensamblador: `ADD Rd, Ra, Rb`

Semántica: `Rd` ← `Ra + Rb`

Descripción: Escribe en `Rd` los 16 bits de menor peso del resultado de sumar el contenido de los registros `Ra` y `Rb`. Puede usarse tanto para la suma de números enteros (integers) como de naturales (unsigned integers).

Comentarios: En ningún caso se detecta si el resultado de la suma de los valores representados en los operandos es representable o no en 16 bits, ni para representación de naturales en binario ni para enteros en complemento a 2.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_s$ = -7 y $R3_u$ = 65.529), `R4` = `0x0008` ($R4_s$ = 8 y $R4_u$ = 8) y `R5` = `0xFFFF` ($R5_s$ = -1 y $R5_u$ = 65.535)

Instrucción: `0x0AE4 (ADD R5, R3, R4)`

Estado después: `R5` = `0x0001` ($R5_s$ = 1 y $R5_u$ = 1)

Observación: Se ha producido desbordamiento interpretando los operandos y el resultado como números naturales.

Ejemplo 2: Estado antes: `R0` = `0x0001` ($R0_s$ = 1 y $R0_u$ = 1), `R4` = `0x7FFF` ($R4_s$ = 32.767 y $R4_u$ = 32.767) y `R5` = `0xFFFF` ($R5_s$ = -1 y $R5_u$ = 65.535)

Instrucción: `0x0A24 (ADD R5, R0, R4)`

Estado después: `R5` = `0x8000` ($R5_s$ = -32.768 y $R5_u$ = 32.768)

Observación: Se ha producido desbordamiento interpretando los operandos y el resultado como números enteros.

ADD Immediat ADDI

Binario: `0010 ddd aaa nnnnnn`

Ensamblador: `ADDI Rd, Ra, C`

Semántica: `Rd ← Ra + SEXT(N6)`

Descripción: Escribe en `Rd` los 16 bits de menor peso de la suma del contenido de `Ra` con el vector de 16 bits resultante de extender el bit de signo del campo `N6`.

Comentarios: En ningún caso se detecta si el resultado de la operación de los valores representados en los operandos es representable o no en 16 bits, ni para representación de naturales en binario ni para enteros en complemento a 2.

Se pueden sumar valores inmediatos al registro fuente comprendidos entre -32 y +31.

La constante `C` expresada en la sentencia ensamblador se codifica con 6 bits en complemento a dos en el campo `N6` de la instrucción.

Ejemplo: Estado antes: `R3` = `0xFFF9` ($R3_s$ = -7 y $R3_u$ = 65.529) y `R5` = `0xFFFF` ($R5_s$ = -1 y $R5_u$ = 65.535)

Instrucción: `0x2AFE (ADDI R5, R3, -2)`

Estado después: `R5` = `0xFFF7` ($R5_s$ = -9 y $R5_u$ = 65.527)

bitwise logical AND — AND

Binario: `0000 ddd aaa 000 bbb`

Ensamblador: `AND Rd, Ra, Rb`

Semántica: `Rd ← Ra & Rb`

Descripción: Escribe en `Rd` el resultado del producto lógico (operación lógica `And`) bit a bit del contenido de los registros `Ra` y `Rb`. En concreto, el bit `i` del resultado es la operación lógica `And` del bit `i` de cada uno de los dos operandos, para `i = 0,1,...,15.`

Ejemplo: Estado antes: `R6` = `0xF505`, `R7` = `0xAAA3`

Instrucción: `0x0F87  (AND R7, R6, R7)`

Estado después: `R7` = `0xA001`

Branch on Not Zero — BNZ

Binario: `0110 bbb 1 nnnnnnnn`

Ensamblador: `BNZ Rb, label`

Semántica: `if (Rb<>0) PC ← PC + SEXT(N8)`

`else PC ← PC + 1`

Descripción: Realiza una ruptura de secuencia condicional en función del contenido del registro `Rb`. La dirección destino de salto tomado se especifica en modo relativo al `PC`.

Se extiende el bit de signo de los 8 bits del campo `N8` hasta completar 16 bits, antes de sumarlos al `PC` para obtener la dirección destino de salto tomado. Si alguno de los 16 bits del contenido de `Rb` es distinto de `0` se carga en el `PC` la dirección de salto tomado que se ha calculado. Si no se cumple la condición (si los 16 bits del contenido de `Rb` valen todos `0`) se incrementa en 1 el valor del `PC`.

Comentarios: Con estas instrucciones de salto se puede romper la secuencia de ejecución y pasar a ejecutar una instrucción que se encuentre a una distancia de -128 a 127 instrucciones en torno a la instrucción de salto.

La etiqueta `label` que aparece en la sentencia ensamblador es el nombre de la etiqueta definida en la línea de código ensamblador donde se encuentra la instrucción a ejecutar si se cumple la condición que provoca la ruptura de secuencia. El programa ensamblador codifica en complemento a 2 en el vector de 8 bits `N8` el valor que se obtiene de restar el valor de la etiqueta menos la dirección en la que se encuentra la instrucción de salto.

Ejemplo: Estado antes: `PC` = `0x0FFF` y `R5` = `0xFFFF`

Instrucción: `0x6BF5` (`BNZ R5, lazo`) con `lazo` `0x0FF4`)

Estado después: `PC` = `0x0FF4`

Branch on Zero BZ

Binario: `0110 bbb 0 nnnnnnnn`

Ensamblador: `BZ Rb, label`

Semántica: `if (Rb==0) PC ← PC + SEXT(N8)`

`else PC ← PC + 1`

Descripción: Realiza una ruptura de secuencia condicional en función del contenido del registro `Rb`. La dirección destino de salto tomado se especifica en modo relativo al `PC`.

Se extiende el bit de signo de los 8 bits del campo `N8`, hasta completar 16 bits, antes de sumarlos al `PC` para obtener la dirección destino de salto tomado. Si los 16 bits del contenido de `Rb` son todos `0` se carga en el `PC` la dirección de salto tomado que se ha calculado. Si no se cumple la condición (si alguno de los 16 bits del contenido de `Rb` valen `1`) se incrementa en 1 el valor del `PC`.

Comentarios: Con estas instrucciones de salto se puede romper la secuencia de ejecución y pasar a ejecutar una instrucción que se encuentre a una distancia de -128 a 127 instrucciones en torno a la instrucción de salto.

La etiqueta `label` que aparece en la sentencia ensamblador es el nombre de la etiqueta definida en la línea de código ensamblador donde se encuentra la instrucción a ejecutar si se cumple la condición que provoca la ruptura de secuencia. El programa ensamblador codifica en complemento a 2 en el vector de 8 bits `N8` el valor que se obtiene de restar el valor de la etiqueta menos la dirección en la que se encuentra la instrucción de salto.

Ejemplo: Estado antes: `PC` = `0x0FFF` y `R5` = `0x0000`

Instrucción: `0x6AF5` (`BNZ R5, lazo`) con `lazo` `0x0FF4`)

Estado después: `PC` = `0x0FF4`

CoMPare if EQual — CMPEQ

Binario: `0001 ddd aaa 011 bbb`

Ensamblador: `CMPEQ Rd, Ra, Rb`

Semántica: `if (Ra == Rb) Rd ← 1 else Rd ← 0`

Descripción: Si el contenido del registro `Ra` es igual que el contenido del registro `Rb`, escribe un 1 en el registro `Rd`, y en caso contrario escribe un 0. El valor 1 o 0 que se escribe en `Rd` (que representa el resultado `TRUE` o `FALSE` de la evaluación de la condición) se codifica en binario con 16 bits.

Comentarios: La instrucción puede usarse para saber si dos números son iguales, tanto para números naturales como para enteros, ya que tanto en binario como en complemento a dos, existe una representación única para cada número.

Ejemplo 1: Estado antes: `R2` = `0xFFF9` ($R2_s$ = -7 y $R2_u$ = 65.529), `R4` = `0x0008` ($R4_s$ = 8 y $R4_u$ = 8) y `R5` = `0x3A00` ($R5_{boolean}$ = `TRUE`)

Instrucción: `0x1A9C (CMPEQ R5, R2, R4)`

Estado después: `R5` = `0x0000` ($R5_{boolean}$ = `FALSE`)

Ejemplo 2: Estado antes: `R0` = `0xFFFF` ($R0_s$ = -1 y $R0_u$ = 65.535), `R4` = `0xFFFF` ($R4_s$ = -1 y $R4_u$ = 65.535) y `R5` = `0x0000` ($R5_{boolean}$ = `FALSE`)

Instrucción: `0x1A1C (CMPEQ R5, R0, R4)`

Estado después: `R5` = `0x0001` ($R5_{boolean}$ = `TRUE`)

CoMPare if Less or EQual — CMPLE

Binario: `0001 ddd aaa 001 bbb`

Ensamblador: `CMPLE Rd, Ra, Rb`

Semántica: `if (Ra <= Rb) Rd ← 1 else Rd ← 0`

Descripción: Si el contenido del registro `Ra` es menor o igual que el contenido del registro `Rb`, interpretados como números enteros codificados en complemento a 2, escribe un 1 en el registro `Rd`, y en caso contrario escribe un 0. El valor 1 o 0 que se escribe en `Rd` (que representa el resultado `TRUE` o `FALSE` de la evaluación de la condición) se codifica en binario con 16 bits.

Comentarios: En todos los casos el resultado de la comparación es el correcto.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_S$ = -7), `R4` = `0x0008` ($R4_S$ = 8) y `R5` = `0x0000` ($R5_{boolean}$ = FALSE)

Instrucción: `0x1ACC (CMPLE R5, R3, R4)`

Estado después: `R5` = `0x0001` ($R5_{boolean}$ = TRUE)

Ejemplo 2: Estado antes: `R0` = `0x0001` ($R0_S$ = 1), `R4` = `0x7FFF` ($R4_S$ = 32.767) y `R5` = `0x00FF` ($R5_{boolean}$ = TRUE)

Instrucción: `0x1B08 (CMPLE R5, R4, R0)`

Estado después: `R5` = `0x0000` ($R5_{boolean}$ = FALSE)

CoMPare if Less or EQual, for Unsigned — CMPLEU

Binario: `0001 ddd aaa 101 bbb`

Ensamblador: `CMPLEU Rd, Ra, Rb`

Semántica: `if (Ra <=u Rb) Rd ← 1 else Rd ← 0`

Descripción: Si el contenido del registro `Ra` es menor o igual que el contenido del registro `Rb`, interpretados como números naturales codificados en binario, escribe un 1 en el registro `Rd`, y en caso contrario escribe un 0. El valor 1 o 0 que se escribe en `Rd` (que representa el resultado `TRUE` o `FALSE` de la evaluación de la condición) se codifica en binario con 16 bits.

Comentarios: En todos los casos el resultado de la comparación es el correcto.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_u$ = 65.529), `R4` = `0x0008` ($R4_u$ = 8) y `R5` = `0xFFFF` ($R5_{boolean}$ = `TRUE`)

Instrucción: `0x1AEC (CMPLEU R5, R3, R4)`

Estado después: `R5` = `0x0000` ($R5_{boolean}$ = `FALSE`)

Ejemplo 2: Estado antes: `R0` = `0x0001` ($R0_u$ = 1), `R4` = `0x7FFF` ($R4_u$ = 32.767) y `R5` = `0x0000` ($R5_{boolean}$ = `FALSE`)

Instrucción: `0x1A2C (CMPLEU R5, R0, R4)`

Estado después: `R5` = `0x0001` ($R5_{boolean}$ = `TRUE`)

CoMPare if Less Than — CMPLT

Binario: `0001 ddd aaa 000 bbb`

Ensamblador: `CMPLT Rd, Ra, Rb`

Semántica: `if (Ra < Rb) Rd ← 1 else Rd ← 0`

Descripción: Si el contenido del registro `Ra` es menor que el contenido del registro `Rb`, interpretados como números enteros codificados en complemento a 2, escribe un 1 en el registro `Rd`, y en caso contrario escribe un 0. El valor 1 o 0 que se escribe en `Rd` (que representa el resultado `TRUE` o `FALSE` de la evaluación de la condición) se codifica en binario con 16 bits.

Comentarios: En todos los casos el resultado de la comparación es el correcto.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_S$ = -7), `R4` = `0x0008` ($R4_S$ = 8) y `R5` = `0xFFFF` ($R5_{boolean}$ = `TRUE`)

Instrucción: `0x1B03  (CMPLT R5, R4, R3)`

Estado después: `R5` = `0x0000` ($R5_{boolean}$ = `FALSE`)

Ejemplo 2: Estado antes: `R0` = `0x0001` ($R0_S$ = 1), `R4` = `0x7FFF` ($R4_S$ = 32.767) y `R6` = `0x0000` ($R6_{boolean}$ = `FALSE`)

Instrucción: `0x1C04  (CMPLT R6, R0, R4)`

Estado después: `R6` = `0x0001` ($R6_{boolean}$ = `TRUE`)

CoMPare if Less Than, for Unsigned — CMPLTU

Binario: `0001 ddd aaa 100 bbb`

Ensamblador: `CMPLTU Rd, Ra, Rb`

Semántica: `if (Ra <u Rb) Rd ← 1 else Rd ← 0`

Descripción: Si el contenido del registro `Ra` es menor que el contenido del registro `Rb`, interpretados como números naturales codificados en binario, escribe un 1 en el registro `Rd`, y en caso contrario escribe un 0. El valor 1 o 0 que se escribe en `Rd` (que representa el resultado `TRUE` o `FALSE` de la evaluación de la condición) se codifica en binario con 16 bits.

Comentarios: En todos los casos el resultado de la comparación es el correcto.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_u$ = 65.529), `R4` = `0x0008` ($R4_u$ = 8) y `R5` = `0x0000` ($R5_{boolean}$ = FALSE)

Instrucción: `0x1B23 (CMPLTU R5, R4, R3)`

Estado después: `R5` = `0x0001` ($R5_{boolean}$ = TRUE)

Ejemplo 2: Estado antes: `R0` = `0x0001` ($R0_u$ = 1), `R2` = `0x7FFF` ($R2_u$ = 32.767) y `R5` = `0xA30F` ($R5_{boolean}$ = TRUE)

Instrucción: `0x1AA0 (CMPLTU R5, R2, R0)`

Estado después: `R5` = `0x0000` ($R5_{boolean}$ = FALSE)

INput IN

Binario: `0111 ddd 0 nnnnnnnn`

Ensamblador: `IN Rd, C`

Semántica: `Rd ← INPUT[N8]`

Descripción: Lee los 16 bits del registro (o puerto) del espacio de entrada (INPUT) cuya dirección está especificada en el campo `N8` de la instrucción y los escribe en el registro `Rd`.

Comentarios: La constante `C` especificada en la sentencia ensamblador es la dirección del registro de entrada que se codifica en binario con 8 bits en el campo `N8` de la instrucción.

Ejemplo 1: Estado antes: `R3` = `0xAF09`, `INPUT[0xBA]` = `0x8000`

Instrucción: `0x76BA  (IN R3, 0xBA`; o también `IN R3, 186)`

Estado después: `R3` = `0x8000`

Ejemplo 2: Estado antes: `R0` = `0xFFFF`, `INPUT[0x03]` = `0x0001`

Instrucción: `0x7003  (IN R0, 0x03`; o también `IN R0, 3)`

Estado después: `R0` = `0x0001`

LoaD LD

Binario: `0011 ddd aaa nnnnnn`

Ensamblador: `LD Rd, C(Ra)`

Semántica: `Rd ← MEM[SEXT(N6)+Ra]`

Descripción: Lee la palabra (16 bits) de memoria cuya dirección está especificada en los campos `Ra` y `N6` de la instrucción y la escribe en `Rd`. La dirección de memoria a la cual se accede se calcula previamente sumando el contenido del registro Ra con el resultado de efectuar la extensión de signo a 16 bits del campo `N6` de 6 bits.

Comentarios: Con estas instrucciones se puede acceder a elementos de memoria que se encuentran a una distancia entre -32 y +31 palabras, módulo 2^{16}, en torno a la dirección que contiene `Ra`. El campo `N6` se interpreta como un número entero codificado en complemento a 2 con 6 bits, denominado desplazamiento.

La dirección de memoria es un número natural codificado en binario con 16 bits (con rango de 0 a 65.535). Aunque se calcule sumando un número natural en binario con un número entero en complemento a dos, el resultado de la suma siempre se interpreta como un número natural codificado en binario con 16 bits, ya que es una dirección de memoria.

Durante el proceso de ensamblado, la constante `C` que aparece en la especificación de la instrucción en lenguaje ensamblador es codificada por el ensamblador con 6 bits en complemento a dos en el campo `N6` de la instrucción de lenguaje máquina.

Ejemplo 1: Estado antes: `R2` = `0x3F02`, `R7` = `0x0A34`, `MEM[k]` = `(k+3)` mod `65.536` para `k` = `0,.., 65.535` (v.g.: `MEM[0x3EFD]= 0x3F00`)

Instrucción: `0x3EBB (LD R7, -5(R2))`

Estado después: `R7` = `0x3F00`

Ejemplo 2: Estado antes: `R2` = `0x000A`, `R7` = `0xFF00`, `MEM[k]` = `(k+3)` mod `65.536` para `k = 0,.., 65.535` (v.g.: `MEM[0xFFFA]= 0xFFFD`)

Instrucción: `0x3EB0 (LD R7, -16(R2))`

Estado después: `R7` = `0xFFFD`

MOV High Immediate MOVHI

Binario: `0101 ddd 1 nnnnnnnn`

Ensamblador: `MOVHI Rd, C`

Semántica: `Rd<15..8>` ← `N8`

Descripción: El vector de 8 bits `N8` se escribe en los 8 bits de mayor peso del registro `Rd` y se mantienen inalterados los 8 bits de menor peso de `Rd`.

Comentarios: Otra forma de especificar la semántica de la instrucción es decir que escribe en `Rd` los 16 bits formados por los 8 bits del campo `N8` a los que se les concatena por la derecha los 8 bits de menor peso de `Rd`. Esto quiere decir que esta instrucción tiene un operando fuente implícito, que es igual al operando destino, ya que `Rd` actúa como fuente (los 8 bits de menor peso) y como destino (los 16 bits).

Durante el proceso de ensamblado, la constante `C,` que es un número natural que aparece en la especificación de la instrucción en lenguaje ensamblador, es codificada por el ensamblador con 8 bits en binario en el campo `N8` de la instrucción de lenguaje máquina.

Ejemplo: Para cargar una constante de 16 bits en un registro general se puede usar una secuencia `MOVI` seguida de `MOVHI`, tal y como muestra el siguiente ejemplo: cargar `0x34ab` en `R4`

```
MOVI    R4, 0xab
MOVHI   R4, 0x34
```

MOV Immediate MOVI

Binario: `0101 ddd 0 nnnnnnnn`

Ensamblador: `MOVI Rd, C`

Semántica: `Rd ← SEXT(N8)`

Descripción: Extiende el bit de signo del vector de 8 bits `N8` para formar el vector de 16 bits que carga en el registro `Rd`.

Comentarios: Se puede usar esta instrucción para cargar en un registro general un número entero en complemento a 2 constante, que se encuentre en el rango de -128 a + 127. También puede usarse para cargar en un registro general un número entero en binario, que se encuentre en el rango de 0 a 127 o de 65.408 a 65.535.

Durante el proceso de ensamblado, la constante `C,` que es un número entero que aparece en la especificación de la instrucción en lenguaje ensamblador, es codificada por el ensamblador con 8 bits en complemento a dos en el campo `N8` de la instrucción de lenguaje máquina.

Ejemplo 1: Se puede cargar el valor -1 en `R5` (escribir 16 unos en `R5`) mediante la instrucción:

```
MOVI    R5, -1
```

Ejemplo 2: Para cargar una constante de 16 bits en un registro general se puede usar una secuencia `MOVI` seguida de `MOVHI`, tal y como muestra el siguiente ejemplo: cargar `0x34ab` en `R4`

```
MOVI    R4, 0xab
MOVHI   R4, 0x34
```

bit wise logical NOT — NOT

Binario: `0000 ddd aaa 011 000`

Ensamblador: `NOT Rd, Ra`

Semántica: `Rd` ← `~Ra`

Descripción: Escribe en `Rd` el resultado de la complementación lógica (operación lógica `Not`, tambien denominada negación lógica) bit a bit del contenido del registro `Ra`. En concreto, el bit `i` del resultado es la operación lógica `Not` del bit `i` del operando, para `i = 0,1,...,15.`

Ejemplo: Estado antes: `R6` = `0xF505`, `R7` = `0xAAA3`

Instrucción: `0x0F98  (NOT R7, R6)`

Estado después: `R7` = `0x0AFA`

bit wise logical OR — OR

Binario: `0000 ddd aaa 001 bbb`

Ensamblador: `OR Rd, Ra, Rb`

Semántica: `Rd ← Ra | Rb`

Descripción: Escribe en `Rd` el resultado de la suma lógica (operación lógica `Or`) bit a bit del contenido de los registros `Ra` y `Rb`. En concreto, el bit `i` del resultado es la operación lógica `Or` del bit `i` de cada uno de los dos operandos, para `i = 0,1,...,15.`

Ejemplo: Estado antes: `R6` = `0xF505`, `R7` = `0xAAA3`

Instrucción: `0x0F8F  (OR R7, R6, R7)`

Estado después: `R7` = `0xFFA7`

OUTput OUT

Binario: `0111 bbb 1 nnnnnnnn`

Ensamblador: `OUT C, Rb`

Semántica: `OUTPUT[N8]` ← `Rb`

Descripción: Escribe los 16 bits contenidos en `Rb` en el registro (o puerto) del espacio de salida (OUTPUT), cuya dirección está especificada en el campo `N8` de la instrucción.

Comentarios: La constante `C` especificada en la sentencia en ensamblador es la dirección del registro de salida que se codifica en binario con 8 bits en el campo `N8` de la instrucción.

Ejemplo 1: Estado antes: `R2` = `0x3F02`, `OUTPUT[0xBA]` = `0xA12C`

Instrucción: `0x75BA` (`OUT 0xBA, R2`; o también `OUT 186, R2`)

Estado después: `OUTPUT[0xBA]` = `0x3F02`

Ejemplo 2: Estado antes: `R0` = `0xFFFF`, `OUTPUT[0x03]` = `0x0001`

Instrucción: `0x7103` (`OUT 0x03, R0`; o también `OUT 3, R0`)

Estado después: `OUTPUT[0x03]` = `0xFFFF`

SHift Arithmetic SHA

Binario: `0000 ddd aaa 110 bbb`

Ensamblador: `SHA Rd, Ra, Rb`

Semántica: `Rd` ← `Ra sha Rb`

Descripción: Escribe en `Rd` el resultado de desplazar `Ra` a derecha o a izquierda tantos bits como indican los 5 bits de menor peso de `Rb` interpretados como un número en complemento a 2. Valores positivos indican desplazamientos a la izquierda y valores negativos desplazamientos a la derecha, pudiéndose desplazar hasta 15 bits a la izquierda y hasta 16 bits a la derecha. En los desplazamientos a la izquierda se copian ceros en los bits de menor peso. En los desplazamientos a la derecha se copia el bit de signo en los bits de mayor peso.

Comentarios: Otra forma de interpretar la instrucción como operador aritmético, es que devuelve los 16 bits de menor peso de la representación en complemento a 2 del número entero $Ra_S \times 2^{Rb<4..0>_S}$.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_S$ = -7), `R4` = `0x0008` ($R4_S$ = 8) y `R5` = `0xFFFF` ($R5<4..0>_S$ = -1)

Instrucción: `0x08F5  (SHA R4, R3, R5)`

Estado después: `R4` = `0xFFFC` ($R4_S$ = -4)

Observación: Interpretando como enteros el operando `R3` y el resultado `R4` (`R5<4..0>` siempre se debe interpretar como entero), se ha efectuado una división entera por 2 con resto positivo.

Ejemplo 2: Estado antes: `R5` = `0x000A` ($R5_S$ = 10 y $R5_U$ = 10), `R6` = `0x7FFF` ($R6_S$ = 32.767 y $R6_U$ = 32.767) y `R7` = `0x0003` ($R7<4..0>_S$ = 3)

Instrucción: `0x0BB7  (SHA R5, R6, R7)`

Estado después: `R5` = `0xFFF8` ($R5_S$ = -8 y $R5_U$ = 65528)

Observación: Interpretando como enteros o como naturales el operando `R6` y el resultado `R5` (`R7<4..0>` siempre se debe interpretar como entero), se ha efectuado una multiplicación por 8, pero para ambas interpretaciones se ha producido desbordamiento.

SHift Logic SHL

Binario: `0000 ddd aaa 111 bbb`

Ensamblador: `SHL Rd, Ra, Rb`

Semántica: `Rd` ← `Ra shl Rb`

Descripción: Escribe en `Rd` el resultado de desplazar `Ra` a derecha o a izquierda tantos bits como indican los 5 bits de menor peso de `Rb` interpretados como un número en complemento a 2. Valores positivos indican desplazamientos a la izquierda y valores negativos desplazamientos a la derecha. En los desplazamientos a la izquierda se copian ceros en los bits de menor peso. En los desplazamientos a la derecha se copian ceros en los bits de mayor peso.

Comentarios: Otra forma de interpretar la instrucción `SHL` como operador aritmético es que devuelve los 16 bits de menor peso de la representación en binario del número natural $Ra_u \times 2^{Rb<4..0>_s}$

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_u$ = 65.529), `R4` = `0x0008` ($R4_u$ = 8) y `R5` = `0xFFFF` ($R5<4..2>_s$ = -1)

Instrucción: `0x08FD (SHL R4, R3, R5)`

Estado después: `R4` = `0x7FFC` ($R4_u$ = 32.761)

Observación: Interpretando como naturales el operando `R3` y el resultado `R4` (`R5<4..0>` siempre se debe interpretar como entero), se ha efectuado una división entera por 2.

Ejemplo 2: Estado antes: `R5` = `0x000A` ($R5_s$ = 10 y $R5_u$ = 10), `R6` = `0x7FFF` ($R6_s$ = 32.767 y $R6_u$ = 32.767) y `R7` = `0x0003` ($R7<4..0>_s$ = 3)

Instrucción: `0x0BBF (SHL R5, R6, R7)`

Estado después: `R5` = `0xFFF8` ($R5_s$ = -8 y $R5_u$ = 65528)

Observación: Interpretando como enteros o como naturales el operando `R6` y el resultado `R5` (`R7<4..0>` siempre se debe interpretar como entero), se ha efectuado una multiplicación por 8, pero para ambas interpretaciones se ha producido desbordamiento.

STore ST

Binario: `0100 bbb aaa nnnnnn`

Ensamblador: `ST C(Ra), Rb`

Semántica: `MEM[SEXT(N6)+Ra] ← Rb`

Descripción: Escribe el contenido de `Rb` (16 bits) en la posición de memoria cuya dirección está especificada en los campos `Ra` y `N6` de la instrucción. La dirección de memoria a la cual se accede se calcula previamente sumando el contenido del registro Ra con el resultado de efectuar la extensión de signo a 16 bits del campo `N6` de 6 bits.

Comentarios: Con estas instrucciones se puede acceder a elementos de memoria que se encuentran a una distancia entre -32 y +31 palabras, módulo 2^{16}, en torno a la dirección que contiene `Ra`. El campo `N6` se interpreta como un número entero codificado en complemento a 2 con 6 bits, denominado desplazamiento.

La dirección de memoria es un número natural codificado en binario con 16 bits (con rango de 0 a 65.535). Aunque se calcule sumando un número natural en binario con un número entero en complemento a dos, el resultado de la suma siempre se interpreta como un número natural codificado en binario con 16 bits, ya que es una dirección de memoria.

Durante el proceso de ensamblado, la constante `C` que aparece en la especificación de la instrucción en lenguaje ensamblador es codificada por el ensamblador con 6 bits en complemento a dos en el campo `N6` de la instrucción de lenguaje máquina.

Ejemplo 1: Estado antes: `R2` = `0x3F02`, `R7` = `0x0A34`, `MEM[k]` = `(k+3)` mod `65.536` para `k = 0,.., 65.535` (v.g.: `MEM[0x3EFD]= 0x3F00`)

Instrucción: `0x4EBB (ST -5(R2), R7)`

Estado después: `MEM[0x3EFD]` = `0x0A34`

Ejemplo 2: Estado antes: `R2` = `0x000A`, `R7` = `0xFF00`, `MEM[k]` = `(k+3)` mod `65.536` para `k = 0,.., 65.535` (v.g.: `MEM[0xFFFA]= 0xFFFD`)

Instrucción: `0x4EB0 (ST -16(R2), R7)`

Estado después: `MEM[0xFFFA]` = `0xFF00`

SUBtract SUB

Binario: `0000 ddd aaa 101 bbb`

Ensamblador: `SUB Rd, Ra, Rb`

Semántica: `Rd` ← `Ra - Rb`

Descripción: Escribe en `Rd` los 16 bits de menor peso del resultado de restar al contenido del registro `Ra` el contenido de `Rb`. Puede usarse tanto para la resta de números enteros (integers) como de naturales (unsigned integers).

Comentarios: En ningún caso se detecta si el resultado de la resta de los valores representados en los operandos es representable o no en 16 bits, ni para representación de naturales en binario ni para enteros en complemento a 2.

Ejemplo 1: Estado antes: `R3` = `0xFFF9` ($R3_s$ = -7 y $R3_u$ = 65.529), `R4` = `0x0008` ($R4_s$ = 8 y $R4_u$ = 8) y `R5` = `0xFFFF` ($R5_s$ = -1 y $R5_u$ = 65.535)

Instrucción: `0x0AEC (SUB R5, R3, R4)`

Estado después: `R5` = `0xFFF1` ($R5_s$ = -15 y $R5_u$ = 65.521)

Observación: No se ha producido ningún desbordamiento.

Ejemplo 2: Estado antes: `R0` = `0x0001` ($R0_s$ = 1 y $R0_u$ = 1), `R4` = `0x7FFF` ($R4_s$ = 32.767 y $R4_u$ = 32.767) y `R5` = `0xFFFF` ($R5_s$ = -1, $R5_u$ = 65.535)

Instrucción: `0x0A2C (SUB R5, R0, R4)`

Estado después: `R5` = `0x8002` ($R5_s$ = -32.766 y $R5_u$ = 32.770)

Observación: Se ha producido desbordamiento en naturales.

bit wise logical XOR XOR

Binario: `0000 ddd aaa 010 bbb`

Ensamblador: `XOR Rd, Ra, Rb`

Semántica: `Rd ← Ra ^ Rb`

Descripción: Escribe en `Rd` el resultado de la suma lógica exclusiva (operación lógica `Xor`, tambien denominada `Or` exclusiva) bit a bit del contenido de los registros `Ra` y `Rb`. En concreto, el bit `i` del resultado es la operación lógica `Xor` del bit `i` de cada uno de los dos operandos, para `i = 0,1,...,15.`

Ejemplo: Estado antes: `R1` = `0xF505`, `R2` = `0xAAA3`

Instrucción: `0x0491 (XOR R2, R2, R1)`

Estado después: `R2` = `0x5FA6`