

Instituto Tecnológico de Costa Rica
Campus Central de Cartago



Escuela Ingeniería Mecatrónica

Microprocesadores y Microcontroladores
Profesor: Rodolfo Piedra Camacho

Tarea 3

Estudiantes:

Catalina Salazar Chaves – 2021075170
Sebastián Villalobos Mora - 2018227618

Respuestas Teóricas

1) ¿Cuántos registros posee el ISA de RISCv32?

Para el ISA de RISCv32, se tienen 32 registros de 32 bits cada uno.

2) Describa la funcionalidad de los campos: Opcode y Funct3

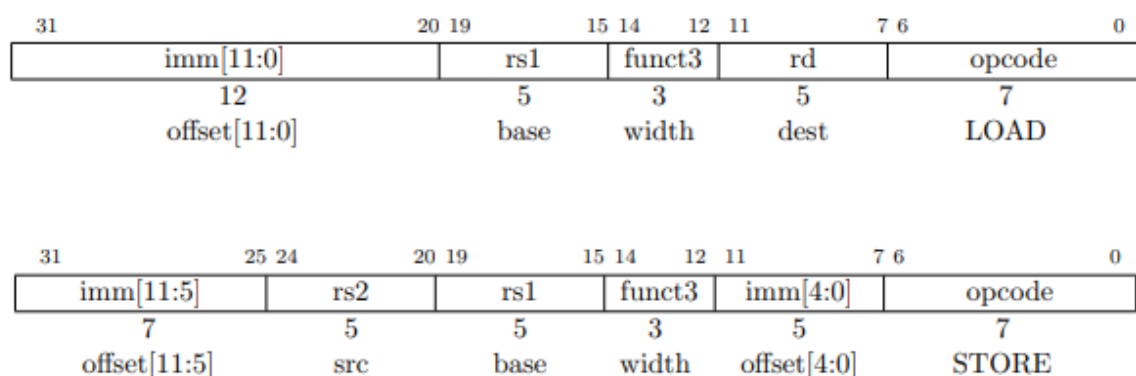
El campo Opcode ocupa los 7 bits menos significativos identifica el tipo general de operación que se va a ejecutar. Pueden ser operaciones aritméticas, lógicas, de carga, de almacenamiento, o de salto, entre otras.

El Funct3 se usa para especifica operaciones detalladas, dentro de la categoría indicada por el Opcode. Es decir, diferencia entre operaciones que comparten el mismo Opcode, por ejemplo, si Opcode indica una operación aritmética, el Funct3 indica cuando se trata de una resta.

3) Para la arquitectura ARM se vieron tres formas distintas de decodificación según el tipo de instrucción: Memoria, Aritmético Lógicas y de control de Flujo. Haga una relación entre la decodificación ARM vista en clase y los tipos de decodificación RISCv: I, R, S, B y J.

Tal como en la decodificación ARM, se pueden relacionar los tipos de decodificación de RISCv dentro de las categorías definidas por el tipo de instrucción. Comenzando por las instrucciones de memoria, en RISCv se utilizan principalmente con el formato I (para cargar) y el formato S (para almacenar). Para facilitar la comparación, se presentan las decodificaciones a continuación:

I y S:



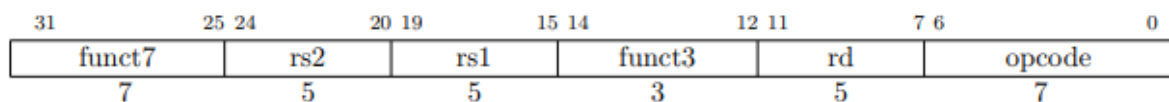
Memoria en ARM:



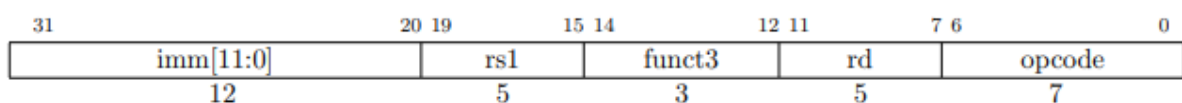
La dirección efectiva donde se carga o se almacena un dato se obtiene a partir de la suma de rs1 y el offset definido por el inmediato con extensión de signo, similar a los bits PW. Al cargar, se copia un valor de memoria al registro en rd, comparable con Rd en su equivalente de ARM. Al almacenar, se copia un valor de registro en la dirección rs2 de memoria, cuya equivalencia en ARM también es Rd. En este caso, el bit 31 siempre contiene el signo del inmediato, similar al bit U en ARM, mas no hay una comparación directa para el bit I.

Después, en cuanto a las instrucciones aritméticas, en ARM se tienen las operaciones entre registros y entre un registro y un valor inmediato. En RISC-V sucede algo similar, donde las operaciones entre registros usan el formato de R y las operaciones con inmediato usan el formato I, en contraste al uso de Src2 en ARM. Se presenta la decodificación a continuación para una mejor comparación:

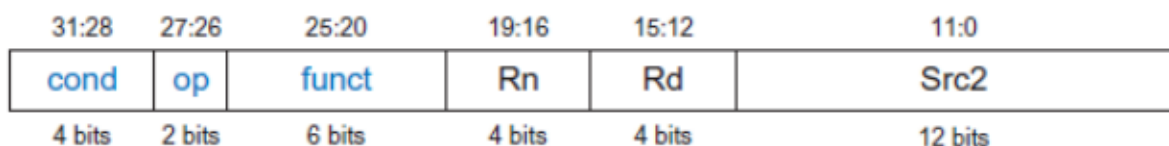
R:



I:



Data processing (ARM):

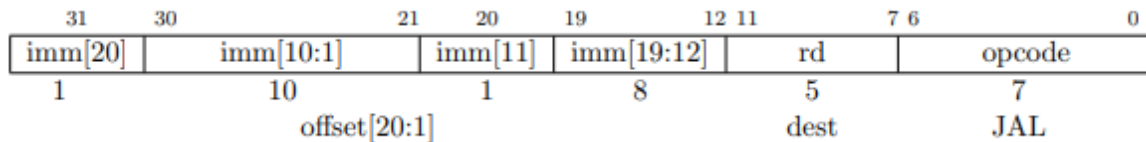


En tanto el formato R como el I, el registro de destino de las operaciones corresponde a rd, comparable a Rd en ARM. Para I, funct3 cumple la función de funct, los bits de imm[11:0] contienen el inmediato, como Src2, y rs1 cumple el papel de Rn, en este

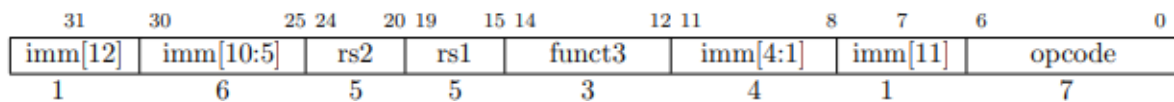
caso. Para R, funct7 y funct3 definen el tipo de operación como lo hace funct, mientras que rs1 y rs2 son los registros operandos, equivalentes a Rm y Rn en ARM.

Con respecto a la categoría de control de flujo, las instrucciones de salto condicional siguen el formato B, mientras que los incondicionales usan el formato J. Comparando a la decodificación de estas instrucciones en ARM:

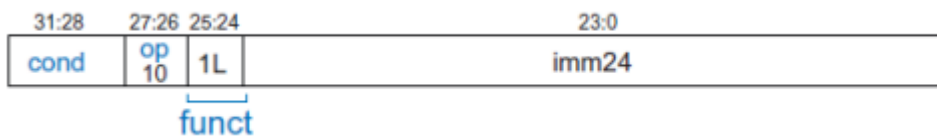
J:



B:



Control de Flujo ARM



En este caso, para J, los valores del inmediato pueden equivaler a imm24, donde en rd contiene la dirección de la instrucción después del salto (PC+4). Al hablar de B, los valores inmediatos (a pesar de estar separados) también equivalen a imm24 en ARM. No hay valores comparables a rs2 ni rs1 pues esto es parte del mecanismo para condicionales de RISC-V.

4) ¿Qué funcionalidad particular tiene el registro x0?

La particularidad que tiene el registro x0 es que tiene un valor constante, que es 0. Además, no permite escritura, es decir, no se pueden almacenar valores en x0.

5) El inmediato en las instrucciones tipo B, S y J está “desordenado”. Que implicaciones tiene esto para la implementación de una unidad de extensión de signo.

Tiene varias implicaciones, pues como el valor de los inmediatos están desorganizados, se le agrega complejidad a la decodificación. La unidad de extensión de signo debe identificar los bits relevantes, es decir, se debe realizar lógica adicional

para manejar las posiciones y consecuentes combinaciones de los bits en las instrucciones. Esto afecta el rendimiento, pues reconstruir el inmediato implica ciclos adicionales de decodificación y ejecución de instrucciones. Sin embargo, para aliviar el problema se puede utilizar el paralelismo, donde la unidad de extensión de signo reconstruye y extiende el valor del inmediato al mismo tiempo.

6) La arquitectura ARM utiliza las banderas NZCV para realizar operaciones condicionales como BEQ. Explique de que forma la arquitectura RISC-V consigue una funcionalidad similar, comente sobre las limitaciones de la estrategia de RISC-V.

Para poder trabajar operaciones condicionales, RISC-V utiliza un mecanismo de condicionalidad que compara valores de registros directamente. Por ejemplo, para la instrucción BEQ, se comparan los valores de rs1 y rs2, y si son iguales, entonces se ejecuta el salto. Además, calcula el offset para este salto utilizando el inmediato en la instrucción, que, dependiendo de su formato, puede estar desordenado.

Este tipo de mecanismo implica que cada instrucción condicional requiere una comparación entre registros. Esto significa que incrementan las instrucciones necesarias para ejecutar los condicionales en comparación a ARM. Además, esto afecta el rendimiento, y, en los casos de instrucciones tipo B o J, tener una desorganización del valor inmediato requiere lógica adicional, pues se tiene que reconstruir el valor correcto antes de ejecutar la instrucción. Por otra parte, no existe tanta flexibilidad como en ARM al hablar de los tipos de condiciones que se pueden evaluar, pues se tienen que realizar incluso más comparaciones adicionales para poder cumplir con condiciones como mayor que, menor que, etc.

7) Explique la diferencia entre las instrucciones JAL y BEQ.

Estas instrucciones son para realizar saltos, JAL (Jump and Link) salta a una dirección y guarda la dirección de la siguiente instrucción, el salto es sin condición. BEQ (Branch if Equal) hace un salto, pero si se verifica que dos registros son iguales, este es un salto condicional y no deja guardada la dirección de la siguiente instrucción.

8) ¿La siguiente instrucción se puede soportar en la arquitectura RISC-V? LW R1, [R2, R3]. Explique su respuesta.

No es soportada. La instrucción LW requiere en su formato que la dirección de memoria se calcule utilizando el registro base y un desplazamiento inmediato. Se intentan usar dos registros, R2 y R3, en la dirección de memoria. Puede usar un registro como base con un desplazamiento inmediato, pero no se puede usar dos registros directamente para calcular la dirección sin un desplazamiento.

9) ¿Como RISC-V da soporte a la instrucción NOP?

La instrucción NOP (No Operation) se logra mediante operaciones que tienen un efecto nulo. La instrucción “addi” que suma cero al registro 0 es una forma común de implementar un NOP. La instrucción no altera ningún estado. En los ensambladores, también se puede dejar un espacio en blanco, si el objetivo es hacer una pausa o espacio en el flujo de ejecución durante el desarrollo o depuración. En algunos entornos, también se puede definir un macro para simplificar la escritura de NOP. Por ejemplo:

```
.macro NOP  
    addi x0, x0, 0  
.end_macro
```

10) Explique porque en las instrucciones de corrimiento de bits (SLLI, SRAI, etc) el valor shamt solo es de 5 bits.

Las razones se relacionan con la representación de registros y el diseño de la arquitectura. Los registros tienen un tamaño de 32 bits, lo que hace que los desplazamientos más allá de 31 bits no tengan valor práctico, 31 es 2^5 menos 1, donde los 5 bits provienen de esta potencia 5. La reducción de 5 bits de desplazamiento simplifica el hardware que implementa la decodificación de instrucciones. Además, deja suficiente campo para instrucciones, como el código de operación y operandos, volviendo la arquitectura más simple y eficiente. La mayoría de las aplicaciones de desplazamientos como la multiplicación o división por potencias de dos se puede implementar eficientemente dentro del rango de 31 bits, no hay gran necesidad de mayores desplazamientos.

11) ¿Qué hace la instrucción LUI?

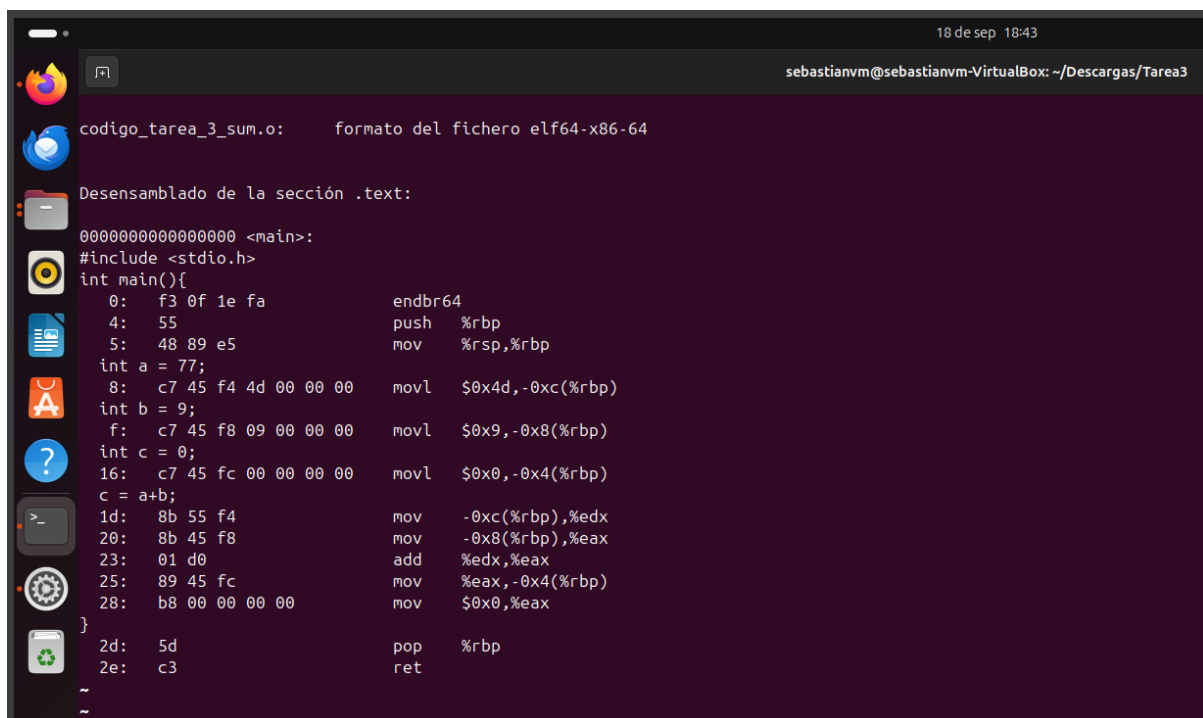
En la arquitectura RISC-V se utiliza para cargar un valor inmediato en la parte alta (los 20 bits más significativos) de un registro. Es un componente esencial en el manejo de valores grandes y dirección de memoria dentro del espacio de 32 o 64 bits. Generalmente se usa seguida de una instrucción ORI o ADDI para completar un valor.

12) Describa el concepto de “cross-compilación”

Es el proceso de compilar código fuente en un sistema para generar un ejecutable que se ejecutará en otro sistema diferente. Es fundamental en el desarrollo de software para diversas plataformas, más si éstas tienen distintas arquitecturas de hardware, sistemas operativos o configuraciones. Desde un único entorno se utiliza esta técnica para crear aplicaciones para distintas plataformas y arquitecturas.

Compilación de Código para RISCV

1a)



```
18 de sep 18:43
sebastianvm@sebastianvm-VirtualBox: ~/Descargas/Tarea3

codigo_tarea_3_sum.o:      formato del fichero elf64-x86-64

Desensamblado de la sección .text:
0000000000000000 <main>:
#include <stdio.h>
int main(){
    0:  f3 0f 1e fa                endbr64
    4:  55                        push    %rbp
    5:  48 89 e5                  mov     %rsp,%rbp
    int a = 77;
    8:  c7 45 f4 4d 00 00 00      movl    $0x4d,-0xc(%rbp)
    int b = 9;
    f:  c7 45 f8 09 00 00 00      movl    $0x9,-0x8(%rbp)
    int c = 0;
    16: c7 45 fc 00 00 00 00      movl    $0x0,-0x4(%rbp)
    c = a+b;
    1d:  8b 55 f4                  mov     -0xc(%rbp),%edx
    20:  8b 45 f8                  mov     -0x8(%rbp),%eax
    23:  01 d0                    add     %edx,%eax
    25:  89 45 fc                  mov     %eax,-0x4(%rbp)
    28:  b8 00 00 00 00          mov     $0x0,%eax
}
    2d:  5d                        pop     %rbp
    2e:  c3                        ret

~
~
```

3)

```

sebastianvm@sebastianvm-VirtualBox: ~/Descargas/Tarea3$ riscv32-unknown-elf-gcc -ol -g codigo_tarea_3_sum.c -o codigo_tarea_3_sum
sebastianvm@sebastianvm-VirtualBox: ~/Descargas/Tarea3$ riscv32-unknown-elf-gcc -ol -g -c codigo_tarea_3_sum.c -o codigo_tarea_3_sum.o
sebastianvm@sebastianvm-VirtualBox: ~/Descargas/Tarea3$ riscv32-unknown-elf-objdump -S codigo_tarea_3_sum.o

codigo_tarea_3_sum.o:      formato del fichero elf32-littleriscv

Desensamblado de la sección .text:

00000000 <main>:
#include <stdio.h>
int main(){
    0:  fe010113      addi    sp,sp,-32
    4:  00812e23      sw      s0,28(sp)
    8:  02010413      addi    s0,sp,32
    int a = 77;
    c:  04d00793      li      a5,77
   10:  fef42623      sw      a5,-20(s0)
    int b = 9;
   14:  00900793      li      a5,9
   18:  fef42423      sw      a5,-24(s0)
    int c = 0;
   1c:  fe042223      sw      zero,-28(s0)
    c = a+b;
   20:  fec42703      lw      a4,-20(s0)
   24:  fe842783      lw      a5,-24(s0)
   28:  00f707b3      add     a5,a4,a5
   2c:  fef42223      sw      a5,-28(s0)
   30:  00000793      li      a5,0
}
   34:  00078513      mv      a0,a5
   38:  01c12403      lw      s0,28(sp)
   3c:  02010113      addi    sp,sp,32
   40:  00000067      ret
sebastianvm@sebastianvm-VirtualBox: ~/Descargas/Tarea3$

```

4)

```

int c = 0;
1c:  fe042223      sw      zero,-28(s0)
c = a+b;
20:  fec42703      lw      a4,-20(s0)
24:  fe842783      lw      a5,-24(s0)
28:  00f707b3      add     a5,a4,a5
2c:  fef42223      sw      a5,-28(s0)
30:  00000793      li      a5,0

```

La línea 5 (int c = 0;) lo que hace es un store word “sw” donde guarda el valor de cero (0) en la dirección de memoria s0 + (-28). La línea 6 (c = a+b;) lo que hace es en el registro a4 cargar el valor con signo extendido que se tiene almacenado en s0 + (-20), luego hace algo similar cargando el valor con signo extendido que se tiene en s0 + (-24) pero con el registro a5. Luego se hace una suma que dejará el resultado en a5, de lo que se tiene en a4 y a5. Luego un store word del resultado que se tiene en a5 en la dirección de memoria s0 + (-28). Por último, con load inmediato “li” se almacena el registro a5 el inmediato 0.

5) Lo que hace “li” es un load de un valor inmediato en la dirección rd, esta se puede implementar como se ve a continuación:

li usage	Base Instruction(s)
li a0,-2048	addi a0,x0,-2048
li a0,2048	lui a0,0x1 addi a0,a0,-2048

Si se tiene un inmediato de hasta 12 bits se puede implementar con solamente un “addi” y se realiza la suma con el registro x0. Si se presenta un inmediato con más de 12 bits se realiza primero una “lui” que toma los 12 bits menos significativos y los coloca en el registro. Luego, se vuelve a aplicar un “addi” como en el primer caso, pero se da entre el registro a0 y el complemento del valor inmediato.

6)

```
sebastianvm@sebastianvm-VirtualBox:~/Descargas/Tarea3$ riscv32-unknown-elf-gcc -ol -g codigo_tarea_3_mul.c -o codigo_tarea_3_mul
sebastianvm@sebastianvm-VirtualBox:~/Descargas/Tarea3$ riscv32-unknown-elf-gcc -ol -g -c codigo_tarea_3_mul.c -o codigo_tarea_3_mul.o
sebastianvm@sebastianvm-VirtualBox:~/Descargas/Tarea3$ riscv32-unknown-elf-objdump -S codigo_tarea_3_mul.o

codigo_tarea_3_mul.o:      formato del fichero elf32-littleriscv

Desensamblado de la sección .text:

00000000 <main>:
#include <stdio.h>
int main(){
    0:  fe010113      addi    sp,sp,-32
    4:  00812e23      sw      s0,28(sp)
    8:  02010413      addi    s0,sp,32
    int a = 15;
    c:  00f00793      li      a5,15
    10:  fef42623      sw      a5,-20(s0)
    int b = 20;
    14:  01400793      li      a5,20
    18:  fef42423      sw      a5,-24(s0)
    int c = 0;
    1c:  fe042223      sw      zero,-28(s0)
    c = a*b;
    20:  fec42703      lw      a4,-20(s0)
    24:  fe842783      lw      a5,-24(s0)
    28:  02f707b3      mul     a5,a4,a5
    2c:  fef42223      sw      a5,-28(s0)
    30:  00000793      li      a5,0
}
    34:  00078513      mv      a0,a5
    38:  01c12403      lw      s0,28(sp)
    3c:  02010113      addi    sp,sp,32
    40:  00008067      ret
sebastianvm@sebastianvm-VirtualBox:~/Descargas/Tarea3$
```

7) Como no se tiene una pseudoinstrucción que pueda realizar la operación “mul” directamente esto no es posible. Como se sabe que una multiplicación es la suma aplicada una cantidad determinada sobre el número en cuestión (4x3 es sumar 3 veces 4), se puede adaptar con varios comandos del set de enteros que cambiaría la

forma en que se compila para realizar la operación. Con “add” se puede sumar un número cierta cantidad definida por lo que de esta forma si se puede lograr la multiplicación de dos enteros con las operaciones del set de enteros de RISCv32.