Práctico 8 - OdC

Lautaro Bachmann

Contents

1)

Como resolver

Hay que tener en cuenta que los numeros son signados y que por ende al extenderlos hay que prestar atencion al bit mas significativo para saber si llenar los bits faltantes con 0 o 1

1)

```
00 0000 0000 0000 0000 0000 0001
=
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
```

2)

```
10 0000 0000 0000 0000 0000 0000
=
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0000 0000 0000 0000 0000 0000
```

2)

Como resolver

1)

Ir a la greencard y ver que tipo de instruccion son

2)

En base al tipo de instruccion ver como está conformada, e ir completando los campos (opcode, etc) uno a uno para luego unirlo todo.

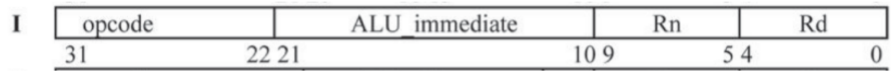
1)

ADDI es una instuccion de formato I.

STUR es una instruccion de formato D.

2)

ADDI X9, X9, #0



opcode=10 0100 0100

ALU_immediate=0000 0000 0000

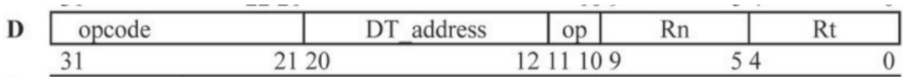
Rn=01001

Rd=01001

Instruccion en binario 0b1001 0001 0000 0000 0000 0001 0010 1001

Instruccion en hexa hex = 0x91000129

STUR X10, [X11, #32] (COMPLETAR)



opcode=11111000000

DT_address=000100000

op= ??

Rn= 01011

Rt= 01010

3)

Como resolver

1. Ver los campos que tiene para determinar el tipo de instruccion.
2. Revisar greencard para ver como se escribe la instruccion en assembler
3. Pasar los campos a binario para obtener la representacion binaria

1)

op=0x658, Rm=13, Rn=15, Rd=17, shamt=0

Tipo de la instruccion

Por los campos proporcionados podemos saber que la instruccion es de tipo **R**.

Instruccion en asssembler

SUB X17 X15 X13

Representacion binaria

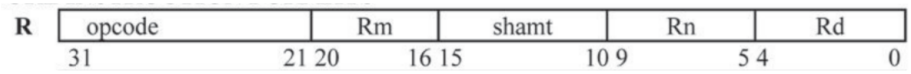
op=11001011000

Rm=01101

shamt=000000

Rn=01111

Rd=10001



1100 1011 0000 1101 0000 0001 1111 0001

2)

op=0x7c2, Rn=12, Rt=3, const=0x4

Tipo de la instruccion

Por los campos proporcionados podemos saber que la instruccion es de tipo **D**.

Instruccion en asssembler

LDUR X3 [X12, #4]

Representacion binaria (COMPLETAR)

opcode=11111000010

DT_address = 000000100

op=??

Rn = 00110

Rt = 00011

11111000010 000000100 ?? 00110 00011

4)

Como resolver

1. Transformar de binario a hexadecimal.
2. Chequear los primeros 11 bits para determinar de que operacion se trata

1)

Pasar de binario a hexa

```
1000 1011 0000 0000 0000 0000 0000 0000
= 8B000000
```

Instruccion que representa en memoria

```
opcode = 10001011000 = 0x458 = ADD
Rm = 00000           = X0
shamt = 000000
Rn = 00000           = X0
Rd = 00000           = X0
```

ADD X0 X0 X0

2)

Pasar de binario a hexa

```
1101 0010 1011 1111 1111 1111 1110 0010
= D2BFFFE2
```

Instruccion que representa en memoria (COMPLETAR)

```
opcode = 1 1010 0101 = 0x195
lsl = 01
MOV_im = 1111111111111111
Rd = 00010
```

5)

Como resolver

En big endian el valor mas significativo es guardado en la primer direccion de almacenamiento

Pasos:

1. Unir el contenido en memoria teniendo en cuenta que está guardado en formato big endian.
2. Determinar opcode con primeros 11 bits
3. Ver el tipo de la operacion

Intruccion 1

0x10010000: 0x8B010029

Pasemos de hexa a binario

```
0x8B010029
= 1000 1011 0000 0001 0000 0000 0010 1001
```

Separemos la instruccion en partes

```
opcode = 100 0101 1000 = 0x458 = ADD
Rm = 00001             = X1
shamt = 000000
Rn = 00001             = X1
Rd = 01001             = X9
```

ADD X9 X1 X1

Instruccion 2

0x10010004: 0x8B010121

Pasemos de hexa a binario

```
0x8B010121
= 1000 1011 0000 0001 0000 0001 0010 0001
```

Separemos la instruccion en partes

```
opcode = 100 0101 1000 = 0x458 = ADD
Rm = 00001             = X1
shamt = 000000
Rn = 01001             = X9
Rd = 00001             = X1
```

ADD X1 X9 X1

Codigo en assembly

```
ADD X9 X1 X1 // X9 = X1 + X1
ADD X1 X9 X1 // X1 = X9 + X1
```

Valor final de X1

```
X1
= X9 + X1
= X1 + X1 + X1
= X1 * 3
```

6)

Como resolver

1. Chequear si los numeros utilizados deben tener signo o no
2. Chequear si los numeros utilizados no superan el tamaño permitido
3. Chequear si los registros de la instruccion pueden ser utilizados

1. LSL XZR, XZR, 0

Se puede codificar en codigo maquina.

2. ADDI X1, X2, -1

No se puede codificar en codigo maquina ya que el ALUImm es no signado

3. ADDI X1, X2, 4096

No se puede codificar en codigo maquina ya que ALUImm solo soporta valores menores a 4095 (0xFFF)

4. EOR X32, X31, X30

No se puede codificar en codigo maquina ya que Rd solo soporta valores hasta 31 (0b11111)

5. ORRI X24, X24, 0x1FFF

No se puede codificar en codigo maquina ya que ALUImm solo soporta valores menores a 4095 (0xFFF)

6. STUR X9, [XZR,#-129]

Se puede codificar en codigo maquina.

7. LDURB XZR, [XZR,#-1]

Se puede codificar en codigo maquina.

8. LSR X16, X17, #68

No se puede codificar en codigo maquina ya que shamt solo soporta valores hasta 63 (0b111111)

9. MOVZ X0, 0x1010, LSL #12

No se puede codificar en codigo maquina ya que #12 es un valor no valido teniendo en cuenta como está codificado el lsl del MOVZ

10. MOVZ XZR, 0xFFFF, LSL #48

Se puede codificar en codigo maquina.

7)

Como resolver

Por cada linea ensamblar la instruccion correspondiente a codigo maquina

a)

```
      MOVZ X0, 0x1, LSL #48 // 110100101 11 0000000000000001 00000
                            // 1101 0010 1110 0000 0000 0000 0010 0000
                            // 0xD2E00020

  L1: SUBI X0,X0,#1         // 1101000100 000000000000 00000 00000
                            // 1101 0001 0000 0000 0000 0000 0000 0000
                            // 0xD1000000

      CBNZ X0, L_1          // 10110101 1111111111111111111 00000
                            // 1011 0101 1111 1111 1111 1111 1110 0000
                            // 0xB5FFFFE0
```

b) (COMPLETAR)

MOVZ X0, 0xFFFF, LSL #32 // 110100101 10 1111111111111111 00000 // 1101 0010 1101 1111 1111 1111 1110 0000

L1: SUBIS X0,X0,#1 B.NE L1

c (COMPLETAR)

8)

Como resolver

Tener en cuenta que el campo de inmediato del branch tiene 26 bits.

Por lo cual, como el primer bit es el bit de signo, tenemos lo siguiente:

- Maxima cantidad de **instrucciones** hacia adelante de B:

$$2^{25} - 1 = 1FFFFFF$$

- Maxima cantidad de **posiciones de memoria** hacia adelante:

$$(2^{25} - 1) * 4 = 7FFFFFC$$

Tener tambien en cuenta que se parte desde el PC.

Tener en cuenta erratas del BR, ya que es de tipo R

(1) y (2)

1. Ver capacidad en bits de la instruccion a usar.
2. Ver si partiendo del PC y sumandole la maxima cantidad que se puede avanzar se llega al resultado deseado.

3)

1. Guardar direccion de memoria a la que quiero saltar en un registro
2. Hacer un branch register con el registro del punto anterior

1)

Explicacion

Resolucion

0x000FFFFC > 0x00014000 ⇒ Es posible llegar con una sola instruccion
0x000FFFFC < 0x00114524 ⇒ No es posible llegar con una sola instruccion
0x000FFFFC < 0x0F000200 ⇒ No es posible llegar con una sola instruccion

2)

Explicacion

- Maxima cantidad de **instrucciones** hacia adelante de B:

$$2^{25} - 1 = 1FFFFFF$$

- Maxima cantidad de **posiciones de memoria** hacia adelante:

$$(2^{25} - 1) * 4 = 7FFFFFC$$

$$\begin{aligned} \text{posMaxima} &= PC + 0x07FFFFFC \\ &= 0x00000600 + 0x07FFFFFC \\ &= 0x080005FC \end{aligned}$$

Resolucion

0x080005FC > 0x00014000 ⇒ Es posible llegar con una sola instruccion
0x080005FC > 0x00114524 ⇒ Es posible llegar con una sola instruccion
0x080005FC < 0x0F000200 ⇒ No es posible llegar con una sola instruccion

3) (COMPLETAR)

9)

Como resolver

- Maxima cantidad de **posiciones de memoria** hacia adelante de B: $(2^{25} - 1) * 4 =$ 7FF FFFC

Pasos

1. Dividir posicion de memoria a la que quiero llegar por la cantidad maxima de posiciones que puedo avanzar y redondear para arriba de ser necesario.

Osea:

$$\text{cantInstruccionesB} = \lceil \text{ posicionObjetivo / maxPosAvance } \rceil$$

Resolucion

Como queremos llegar a 0xFFFFFFFC y con cada **B** podemos recorrer 0x07FFFFFC posiciones de memoria, tenemos que:

$$\begin{aligned} \text{cantInstruccionesB} &= \lceil 0xFFFFFFFC \ / \ 0x07FFFFFC \rceil \\ &= \lceil 32.00000092 \rceil \\ &= 33 \end{aligned}$$

10)

Como resolver

.org 0x0000 hace que el programa empiece en la direccion 0

Tener en cuenta que cada instruccion suma cuatro a la direccion de comienzo del programa, por lo cual:

- MOVZ está en la posicion 0
- MOVK está en la posicion 4
- STURW está en la posicion 8
- STURW está en la posicion 12

Este tipo de programa es un programa que se reescribe solo.

Resolucion

```
  .org 0x0000               // Inicia memoria en posicion 0x0
0:  MOVZ X0, 0x0400, LSL #0   // X0 = 0x0400
4:  MOVK X0, 0x9100, LSL #16  // X0 = 0x91000400
8:  STURW X0, [XZR,#12]       // M[XZR + 12] = 0x91000400
// 12: STURW X0, [XZR,#12]    // Esta posicion fue cambiada por la instruccion anterior
```

Desensamblar 0x91000400

0x91000400 = 1001 0001 0000 0000 0000 0100 0000 0000

opcode = 100 1000 1000 = 0x488 = ADDI

$$\underbrace{100100010}_{opcode}\underbrace{000000000001}_{imm}\underbrace{00000}_{Rn}\underbrace{00000}_{Rd}$$

= ADDI X0, X0, #1

Conclusion

```
  .org 0x0000               // Inicia memoria en posicion 0x0
0:  MOVZ X0, 0x0400, LSL #0   // X0 = 0x0400
4:  MOVK X0, 0x9100, LSL #16  // X0 = 0x91000400
8:  STURW X0, [XZR,#12]       // M[XZR + 12] = 0x91000400
// 12: STURW X0, [XZR,#12]    // Esta posicion fue cambiada por la instruccion anterior
12: ADDI X0, X0, #1           // X0 = 0x91000400 + 0x1 = 0x91000401
```

Por ende, el valor que devuelve **X0** en este programa es **0x91000401**