EC sept19-feb20 ETeo2 (enunciado)

Estructura de Computadores (Universitat Oberta de Catalunya)

Ejercicios Teo.2

Esta semana se harán ejercicios sobre:

- El análisis y la escritura de pequeños programas escritos en ensamblador CISCA. En ambos casos los programas manejan datos estructurados de tipo vector (los datos de tipo matriz los dejamos para los ejercicios prácticos). Haciendo énfasis en los modos de direccionamiento y en especial en los relativos: relativo a registro base (denominado relativo en CISCA) y relativo a registro índice (denominado indexado en CISCA). (Ejercicios 2.1.x). Módulo 7, apartado 2.

- La codificación de las instrucciones CISCA. Dado un código ensamblador debéis traducirlo a lenguaje máquina y al revés. Lo que haga el código no nos importa, sólo nos importa cómo está codificado. (Ejercicio 2.2). Módulo 7, apartado 3.

- La ejecución de las micro-operaciones de las que se componen las instrucciones CISCA. Dada una instrucción en ensamblador o en lenguaje máquina indicar la secuencia de micro-operaciones necesaria para su ejecución y al revés. (Ejercicio 2.3). Módulo 7, apartado 4.

También hay que revisar los materiales de todo el módulo 2.

Lo que hacen o lo que tienen que hacer los códigos lo especificamos mediante el lenguaje de alto nivel C o mediante un texto que describe la funcionalidad del código.

En los códigos escritos en C no declaramos las variables que se usan. Suponemos que ya están declaradas. Tampoco debéis reservar espacio en memoria ni inicializar las variables en memoria del programa en ensamblador CISCA. La dirección simbólica de memoria que tenéis que usar en el código CISCA para acceder a una variable, estructurada o no, que aparece en el código en C debe coincidir con el nombre de la variable en C. Por ejemplo, para traducir la siguiente sentencia en C,

```
A = A + V[i];
```

podemos usar el código CISCA siguiente (suponiendo que R0 y R1 no se están usando para almacenar ninguna variable viva en este momento del código):

```
MOV  R1, [i]
MUL  R1, 4
MOV  R1, [V+R1]
ADD  [A], R1
```

Ejercicio Teo.2.1.1

Este primer ejercicio consiste en analizar unos fragmentos de código CISCA que acceden a elementos de un vector y decir lo que hacen. Debes expresarlo en un lenguaje de alto nivel, como el C (sin usar notación de punteros), o usando un pseudo-lenguaje, o explicando claramente lo que hace el código y por qué.

Tenemos almacenado en memoria, a partir de la dirección simbólica `V`, un vector de 100 elementos, donde cada elemento es un número entero codificado en complemento a 2 con 32 bits. Recordad que en la dirección simbólica `V` se encuentra almacenado el elemento `V[0]`, en la dirección `V+4` se encuentra `V[1]`, etc. También tenemos almacenado en memoria, en la dirección simbólica `A`, un número entero codificado en complemento a 2 con 32 bits.

Suponed que en el programa escrito en el lenguaje de alto nivel se ha definido la variable `V` como vector de enteros de 32 bits y la variable `A` como número entero de 32 bits.

a)

```
MOV   R1, V
ADD   R1, 4
MOV   R2, [R1]
MOV   [A], R2
```

Solución: `A = V[1];`
Es la forma más simple de explicar lo que hacen las 4 instr.

b)

```
MOV   R1, [A]
MOV   R2, V
MOV   [R2+12], R1
```

Solución:

c)

```
MOV   R1, 8
MOV   R3, [V+R1]
MOV   [A], R3
```

Solución:

Ejercicio Teo.2.1.2

a)

A partir del siguiente programa escrito en el lenguaje de alto nivel C, que calcula la suma de los elementos de un vector, escribid un programa en ensamblador CISCA que implemente el mismo algoritmo (que sea una traducción a ensamblador CISCA del programa en C).

Considerad que no nos interesa el valor final de las variables `partial_sum`, `i` y `N`. Por ello, para optimizar el código debéis usar registros del procesador para almacenar sus valores.

Suponed que en el programa escrito en el lenguaje de alto nivel se ha definido la variable `V` como vector de enteros de 32 bits y la variable `sum` como número entero de 32 bits, y que se encuentran almacenados a partir de las direcciones simbólicas de memoria `V` y `sum` respectivamente.

Código en C

```
N = 4;
sum = 0;

partial_sum = 0;
i = 0;

while ( i< N)  {
   partial_sum = partial_sum + V[i];
   i = i + 1;
}
sum = partial_sum;
```

Solución:

b)
Tenemos definido un vector, V, de 8 elementos. Cada elemento es un número entero codificado en complemento a 2 con 32 bits:

```
V: 3, -7, 125, 421, -9, 1000, 7, 8
```

Escribid un código en ensamblador que cambie el orden en que se encuentran los elementos del vector, dejando el primer elemento en la última posición, el segundo en la antepenúltima etc. Después de la ejecución del código el vector debe quedar así:

```
V: 8, 7, 1000, -9, 421, 125, -7, 3
```

El bucle principal del programa debería servir para vectores con un número cualquiera de elementos. Antes de este bucle, habría que iniciar el contenido de ciertos registros con los valores apropiados para este ejemplo (vector de 8 elementos)

Solución:

Ejercicio Teo.2.2

a)

Para facilitaros el trabajo os damos resuelto un problema parecido a los que os pedimos que resolváis a continuación.

Dado el siguiente código en ensamblador CISCA:

```
            MOV R1, R2
loop:       DEC R2
            JE end_loop
            MUL R1, R2
            JMP loop
end_loop:   MOV R3,4
            MOV [100+R3], R1
```

Traducidlo a lenguaje máquina y expresarlo en la siguiente tabla. Suponed que la primera instrucción del código se ensambla a partir de la dirección 10000h (que es el valor del registro PC en el estado inicial). En la siguiente tabla usad una fila para codificar cada instrucción. Si suponemos que la instrucción comienza en la dirección @, el valor de cada uno de los bytes de la instrucción con direcciones @+i para i=0, 1, … se debe indicar en la tabla en formato hexadecimal en la columna correspondiente. Recordad que los campos que codifican un desplazamiento en 2 bytes o un inmediato o una dirección en 4 bytes lo hacen en formato *little endian*. Esto quiere decir que hay que escribir los bytes de menor peso (de dirección más pequeña) a la izquierda y los de mayor peso (dirección mayor) a la derecha.

		Bk para k=0..10										
Dirección	**Ensamblador**	0	1	2	3	4	5	6	7	8	9	10
00010000h	MOV R1, R2	10	11	12								
00010003h	DEC R2	25	12									
00010005h	JE end_loop	41	60	09	00							
00010009h	MUL R1, R2	22	11	12								
0001000Ch	JMP loop	40	00	03	00	01	00					
00010012h	MOV R3, 4	10	13	00	04	00	00	00				
00010019h	MOV [100+R3], R1	10	53	64	00	00	00	11				

En negro se indica el código de operación, en rojo la codificación del primer operando y en verde la codificación del segundo operando. En los operandos se codifica primero el modo de direccionamiento y después el operando. Todos los valores están en hexadecimal.

Explicación de la primera instrucción:

1. La instrucción MOV se codifica con el valor 10h (ver la tabla de la página 25)
2. El registro R1 se codifica como un 1 en el dígito más significativo para indicar que se trata de un modo de direccionamiento de tipo registro, y un 1 en el dígito menos significativo para indicar que se trata del registro R1. En total: 11h (ver tabla de la página 26)
3. El registre R2 se codifica como 12h, donde el 2 final indica el registro R2.

Explicación de la tercera instrucción (JE):

1. La instrucción JE se codifica con el valor 41h
2. El modo de direccionamiento de tipo relativo a registre PC se codifica como 60h
3. El desplazamiento del salto se calcula como 00010012h (dirección donde comienza la instrucción MOV R3,4) – 00010009h (dirección donde comienza la instrucción que va después del salto, MUL R1, R2) = 0009h. Este valor **0009**h se codifica en forma little-endian como **09**h **00**h. Las direcciones de las instrucciones posteriores no se conocen hasta que no se codifican, así que no podemos calcular el valor 9 hasta que hemos codificado todo el programa (sin los valores concretos de los modos de direccionamiento de tipo relativo a PC)

Comentarios de las instrucciones MOV R3,4 y MOV [100+R3],R1: el valor de las constantes 4 y 100 se representa en formato little-endian y con 32 bits (4 Bytes):

4d= **00000004**h se representa como **04**h **00**h **00**h **00**h

100d= **00000064**h se representa como **64**h **00**h **00**h **00**h

b)
Dado el siguiente código en ensamblador CISCA:

```
                MOV  R0, [A]
                CMP  R0, [B]
                JLE  Label_1
                DEC  R0
                JMP  Label_2
Label_1:        ADD  [B], 4
Label_2:        MUL  [B], R0
                MOV  [A], R0
```

Traducidlo a lenguaje máquina y rellenar la siguiente tabla. Suponed que la primera instrucción del código se ensambla a partir de la dirección 10F8h (que es el valor del registro PC en el estado inicial). Suponed también que las direcciones simbólicas `A` y `B` valen 20h y 200h respectivamente. En la siguiente tabla usad una fila para codificar cada instrucción, como se hace en los ejemplos de codificación en la documentación de la asignatura y en el ejercicio 0.

		Bk para k=0..10										
Dirección	**Ensamblador**	0	1	2	3	4	5	6	7	8	9	10
000010F8h	MOV R0, [A]											
	CMP R0, [B]											
	JLE Label_1											
	DEC R0											
	JMP Label_2											
	ADD [B], 4											
	MUL [B], R0											
	MOV [A], R0											

c)
Decodificad el siguiente fragmento de código que se encuentra almacenado a partir de la dirección 13F20h de memoria. Esto es, tenéis que hacer la función inversa de la que hace el programa ensamblador cuando traduce un código de lenguaje ensamblador CISCA a lenguaje máquina CISCA. A partir del código en lenguaje máquina del CISCA, que se da en la siguiente tabla, encontrad el código en lenguaje ensamblador del CISCA.

En la siguiente tabla se indica el contenido de cada byte de memoria (en hexadecimal) desde la dirección 13F20 hasta la dirección 13F4F (que es donde se encuentra el código que tenéis que desensamblar). El hecho de que la tabla muestre 16 bytes de memoria (16 columnas) en cada fila de la tabla es puramente estético, podíamos haber dado el contenido de la memoria usando otro número de columnas distinto de 16.

Dirección (Hexa)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
13F2x	10	12	00	01	00	00	00	26	20	30	2A	00	00	00	01	00
13F3x	00	00	41	60	13	00	20	12	20	30	2A	00	00	25	20	30
13F4x	2A	00	00	40	00	27	3F	01	00	10	20	00	01	00	00	12

Para indicar el resultado de este ejercicio, rellenad la siguiente tabla. Esta tabla es como las que hemos usado en los ejercicios anteriores para indicar el código ensamblador y su traducción a lenguaje máquina.

Os damos decodificada la primera instrucción a modo de ejemplo.

Si en el proceso de decodificación encontráis una instrucción ilegal (un código que no corresponde a ninguna instrucción CISCA, tal como se han definido en la documentación de la asignatura) copiad Ins. Ileg. en la fila correspondiente a esa instrucción y aquí termina el ejercicio, ya que no podéis seguir decodificando. En la tabla siguiente añadir o borrar las filas que consideréis oportuno.

		Bk para k=0..10										
Dirección	**Ensamblador**	0	1	2	3	4	5	6	7	8	9	10
00013F20h	**MOV R2, 1**	**10**	**12**	**00**	**01**	**00**	**00**	**00**				
00013F27h												

Ejercicio Teo.2.3

El *ciclo de ejecución* de una instrucción se divide en 3 fases principales
1) Lectura de la instrucción
2) Lectura de los operandos fuente
3) Ejecución de la instrucción y almacenamiento del operando destino

Dar la secuencia de micro-operaciones que hay que ejecutar en cada fase para las siguientes instrucciones del ejercicio Teo.4.0.

		Bk per a k=0..10										
Dirección	**Ensamblador**	0	1	2	3	4	5	6	7	8	9	10
...	...											
00010009h	MUL R1, R2	22	11	12								
0001000Ch	JMP loop	40	00	03	00	01	00					
00010012h	MOV R3, 4	10	13	00	04	00	00	00				
00010019h	MOV [100+R3], R1	10	53	64	00	00	00	11				

Suponed que la lectura de las instrucciones se puede hacer con un solo acceso a memoria.

`MUL R1, R2`

Fase	Micro-operaciones
1	(MAR=00010009h) ← (PC=00010009h), read ; Ponemos el contenido del PC en el registro MAR (MBR=00121122h) ← Memoria ; Leemos la instrucción MUL R1,R2 (PC=0001000Ch) ← (PC=00010009h) + 3 ; Incrementamos el PC en 3 unidades ; (tamaño de la instrucción) (IR=00121122h) ← (MBR=121122h) ; Cargamos la instrucción en el registre IR
2	(No es necesario hacer nada, los operandos fuente son registros)
3	R1 ← R1 * R2

`JMP loop`

Fase	Micro-operaciones
1	
2	
3	

`MOV R3,4`

Fase	Micro-operaciones
1	
2	
3	

MOV [100+R3], R1

Fase	Micro-operaciones
1	
2	
3	