Organización y Arquitectura de Computadoras
2020-1
Práctica 9: Ensamblador

Profesor: José de Jesús Galaviz Casas
Diseño: Roberto Monroy Argumedo

1. Objetivos

Generales:

- El alumno se familiarizará con el proceso interno de ensamblado de un archivo de código fuente.

Particulares:

En el desarrollo de la práctica, el alumno:

- Aplicará los conocimientos teóricos de la codificación de instrucciones.
- Relacionará los conocimientos previos de la programación en lenguaje ensamblador con el lenguaje de alto nivel C.

2. Requisitos

- **Conocimientos previos:**
  - Diseño de conjuntos de instrucciones.
  - Modos de direccionamiento.
  - Programación en el lenguaje C:
    - Tipos de datos.
    - Estructuras de control.
    - Arreglos y apuntadores.
    - Entrada y salida.
  - Codificación ASCII.
- **Tiempo de realización sugerido:**
  10 horas.

- **Número de colaboradores:**
  En parejas
- **Software a utilizar:**
  GCC.

3. Planteamiento

Un proceso de compilación consta de tres fases:

1. El **compilador** traduce el archivo de código fuente escrito en un lenguaje de alto nivel a uno escrito en lenguaje ensamblador.
2. Éste se pasa a un **ensamblador**, un programa que lo traduce a lenguaje máquina, produciendo un **archivo objeto**, el cual contiene las instrucciones y los datos estáticos del programa e información necesaria para colocarlo en la memoria.
3. Generalmente un programa contiene referencias a rutinas en otros archivos o en bibliotecas, por lo que al finalizar el ensamblado, se ejecuta el **enlazador**, un programa que resuelve las referencias externas, creando un archivo ejecutable y terminando así el proceso de compilación.

En esta práctica se desarrollará un ensamblador para un pequeño conjunto de instrucciones de la arquitectura DLX.

4. Desarrollo

En las secciones 4.1 - 4.6 se describen las características del lenguaje que el ensamblador deberá se capaz de traducir a lenguaje máquina. En la sección 4.7 se describe el proceso de ensamblado de un archivo de código fuente y en la sección 4.8 el formato del archivo objeto que se deberá generar.

4.1. Arquitectura DLX

DLX, pronunciado *deluxe*, es una arquitectura RISC de tipo *load-store*, fue diseñada por John L. Hennessy y David A. Patterson con propósitos educativos basándose en MIPS[**Hennessy**]. Cuenta con 32 registros de propósito general y el tamaño de cada instrucción es de 32 bits.

4.2. Conjunto de instrucciones

A continuación se describen las instrucciones que serán reconocidas y traducidas por el ensamblador, el primer término es el mnemónico de la instrucción seguido de los argumentos que recibe:

1. `lw rs dir`
   Carga la palabra que comienza en la dirección `dir` de la memoria de datos en el registro `rs`.
   Ejemplo:
   `lw $01 0($00)`

2. `lh rs dir`
   Carga la media palabra que comienza en la dirección `dir` de la memoria de datos en el registro `rs`.
   Ejemplo:
   `lh $01 0($00)`

3. `lb rs dir`
   Carga el byte almacenado en la dirección `dir` de la memoria de datos en el registro `rs`.
   Ejemplo:
   `lb $01 0($00)`

4. `sw rs dir`
   Guarda la palabra almacenada en el registro `rs` en la dirección `dir` de la memoria de datos.
   Ejemplo:
   `sw $01 0($00)`

5. `sh rs dir`
   Guarda la media palabra almacenada en el registro `rs` en la dirección `dir` de la memoria de datos.
   Ejemplo:
   `sh $01 0($00)`

6. `sb rs dir`
   Guarda el byte almacenado en el registro `rs` en la dirección `dir` de la memoria de datos.
   Ejemplo:
   `sb $01 0($00)`

7. `add rd rs rt`
   Suma el contenido de los registro `rs` y `rt`, guardando el resultado en el registro `rd`.
   Ejemplo:
   `add $01 $01 $01`

8. `addi rd rs imm`
   Suma el contenido del registro `rs` y el valor constante `imm`, guardando el resultado en el registro `rd`.
   Ejemplo:
   `addi $01 $01 42`

9. `sub rd rs rt`
   Resta el contenido del registro `rt` al registro `rs`, guardando el resultado en el registro `rd`.
   Ejemplo:
   `sub $01 $01 $01`

10. `subi rd rs imm`
    Resta la constante `imm` al registro `rs`, guardando el resultado en el registro `rd`.
    Ejemplo:
    `add $01 $01 42`

11. `and rd rs rt`
    Realiza la operación lógica `and` bit a bit con los registros `rs` y `rt`, guardando el resultado en `rd`.
    Ejemplo:
    `and $01 $02 $03`

12. `andi rd rs imm`
    Realiza la operación lógica `and` bit a bit con el registro `rs` y la constante `imm`, guardando el resultado en `rd`.
    Ejemplo:
    `andi $01 $02 31`

13. `or rd rs rt`
    Realiza la operación lógica `or` bit a bit con lo registros `rs` y `rt`, guardando el resultado en `rd`.
    Ejemplo:
    `or $01 $02 $03`

14. `ori rd rs imm`
    Realiza la operación lógica `or` bit a bit con el registro `rs` y la constante `imm`, guardando el resultado en `rd`.
    Ejemplo:
    `ori $01 $02 31`

15. `beq rs rt label`
    Salto condicional, se ejecuta la instrucción marcada por `label` si `rs` es igual a `rt`.
    Ejemplo:
    `beq $01 $02 label`

16. `bgt rs rt label`
    Salto condicional, se ejecuta la instrucción marcada por `label` si `rs` es mayor a `rt`.
    Ejemplo:
    `bgt $01 $02 label`

17. `j label`
    Salto incondicional, se ejecuta la instrucción marcada por `label`.
    Ejemplo:
    `j label`

18. `jr rd`
    Salto incondicional, se ejecuta la instrucción con la dirección guardada en el registro `rd`.
    Ejemplo:
    `jr $01`

4.3. Codificación de las intrucciones

La codificación de las instrucciones del lenguaje dependen de sus operandos y su función, para facilitar la traducción, se clasifican en tres tipos:

4.3.1. Intrucciones tipo I

Para instrucciones con un operando de tipo **I**nmediato:

	Opcode	rs	rd	Inmediato
bits	6	5	5	16

4.3.2. Intrucciones tipo J

Para instrucciones de salto (***J**ump*):

	Opcode	Desplazamiento
bits	6	26

4.3.3. Intrucciones tipo R

Para instrucciones con operandos de tipo **R**egistro:

	Opcode	rs	rt	rd	Func
bits	6	5	5	5	11

4.4. Directivas

El ensamblador deberá implementar cuatro directivas que se describen a continuación:

- `data`. Los siguientes datos serán guardados en la sección de memoria de datos estáticos.
- `text`. Las siguientes instrucciones serán guardadas en la sección de memoria de texto.
- `word`. Los siguientes datos deben ser almacenados como palabras.
- `byte`. Los siguientes datos deben ser almacenados como bytes.

4.5. Modos de direccionamiento

Las direcciones de memoria son asignadas a cada byte, el ensamblador reconocerá dos modos de direccionamiento:

- `c(reg)`. Inmediato `c` más la dirección en el registro `reg`.
- `label`. Etiqueta de una dirección.

El hardware sólo reconocerá el modo `c(reg)`, por lo que la traducción del modo `label`, debe ser de la forma `c` + `reg`.

4.6. Sintaxis del lenguaje ensamblador

- Sólo se permite una instrucción por línea.
- Entre cada término puede aparecer cualquier número de espacios o tabulaciones.
- Los argumentos de las instrucciones se separan por comas (,).
- Los identificadores de los registros comienzan con un signo de pesos (`$`), seguido del número de registro a dos dígitos. Ejemplos: `$00`, `$03`, `$21`, etc.
- Para los inmediatos sólo se admiten enteros escritos en base diez.
- Las etiquetas sólo podrán contar con carcateres alfabéticos. Para señalar una dirección, se coloca la etiqueta al principio de la línea seguido por dos puntos (:).
- Las directivas comienzan con un punto (.).
- Las directivas `data` y `text` siempre aparecerán en este orden y sólo se pueden usar una vez en todo el archivo de código fuente.

4.7. Proceso de ensamblado

El proceso de ensamblado consta de dos fases:

4.7.1. Fase 1

El ensamblador lee cada línea del archivo escrito en lenguaje ensamblador y la divide en piezas llamadas **lexemas**, éstas son: palabras individuales, identificadores de registros, signos de puntuación o números. Por ejemplo, la línea:

```
label: addi $01, $02, 3
```

Contiene 8 lexemas:

label	:	addi	$01	,	$02	,	3

Si la línea comienza con una etiqueta, el ensamblador guarda el identificador junto con la dirección de memoria en una tabla, ésta es llamada **tabla de símbolos**.

4.7.2. Fase 2

Una vez dividido el archivo en lexemas, el ensamblador traduce las instrucciones a código máquina, ayudándose de la tabla de símbolos para sustituir etiquetas por direcciones de memoria. Además el ensamblador debe actuar acorde a las directivas que encuentre. Al finalizar, el ensamblador produce un archivo objeto.

4.8. Formato del archivo objeto

El archivo objeto generado por el enamblador, basado en el formato usado por los sitemas UNIX, constará de tres secciones:

1. **Cabecera**. Se especifican los tamaños del área de texto y el área de datos.
2. **Texto**. Contiene las instrucciones del programa en lenguaje máquina.
3. **Datos**. Contiene los datos codificados en binario.

Generalmente también cuenta con las siguientes secciones, pero las omitiremos porque no serán usadas por el ensamblador ni el simulador:

- **Información de reasignación**. Identifica las instrucciones y datos que dependen de direcciones absolutas.
- **Tabla de símbolos**. Asocia direcciones con etiquetas externas y lista las referencias faltantes.
- **Información de depurado**. Describe la forma en que el programa fue compilado para que un depurador pueda asociar las instrucciones con las líneas del archivo de código fuente original.

5. Entrada

El ensamblador recibirá por medio de la línea de comandos el nombre de un archivo de texto plano, el cual contendrá las instrucciones en lenguaje ensamblador descritas en las secciones 4.1 - 4.6. Puedes encontrar un ejemplo en la figura 1.

6. Salida

El ensamblador escribirá una simulación de un archivo objeto con las características descritas en la sección 4.8. Las secciones de datos y texto serán cadenas de caracteres '0' y '1', simulando la codificación binaria de éstas. Además cada línea será del tamaño de una palabra, es decir, cada línea contará con 32 caracteres.

```
        .data
dato:   .word   1
        .text
        lw      $01, dato
loop:   beq     $01, $00, end
        subi    $01, $01, 1
        j       loop
end:    add     $02, $00, $01
```

Figura 1: Ejemplo de código ensamblador.

7. Variables libres

El alumno definirá los códigos de operación de las instrucciones, además del tamaño de la memoria y los límites de las secciones de datos y texto.

8. Procedimento

Se debe entregar un archivo de código fuente escrito en el lenguaje de programación C con la solución al ejercicio 4. Adjunta las soluciones a los ejercicios 1, 2 y 3 junto con las respuestas a las preguntas en un archivo PDF.

El programa debe estar completamente documentado y cumplir con las convenciones de código.

9. Ejercicios

El archivo objeto producido por el ensamblador será usado en la siguiente práctica para simular la ejecución del programa siguiendo el flujo de datos de la arquitectura DLX, por lo que se debe considerar las repercusiones de las decisiones tomadas en esta práctica.

1. Clasifica las instrucciones de la sección 4.2 según los tipos descritos en 4.3.
2. Asigna cuidadosamente los códigos de operación a cada una de las instrucciones.
3. Define el tamaño de la memoria y las direcciones de inicio de las secciones de memoria para el área de texto y el área de datos.
4. Implementa el ensamblador para el lenguaje descrito.

10. Preguntas

1. Una pseudoinstrucción es una instrucción de lenguaje ensamamblador sin implementación directa en el hardware, su función es simplificar la pro-

gramación sin complicar el hardware, por ejemplo:

`move $rd, $rs` se traduce a `addu $rd, $zero, $rs`

Propón 3 pseudoinstrucciones para el lenguaje ensamblador y su traducción.

2. Sólo se implementaron dos modos de direccionamiento, propón cómo podría simular el ensamblador los siguientes modos:
   - `label` $\pm$ `imm`.
   - `label` $\pm$ `imm(reg)`.