Practica 4 - WinMIPS64

30 de septiembre de 2014

Repaso MSX88

- 4 Registros de Proposito General de 16 Bits
 - Pueden ser visto como 8 registros de 8 Bits
- 65.536 (2¹⁶) bytes de memoria.
- Pila
- Flags
- Memoria de Datos y Programas es la misma
- Muchas instrucciones pueden acceder a la memoria
- Instrucciones de tamaño variable
- Directivas del ensamblador
 - ORG, DW, DB, END, OFFSET

WinMIPS64

- 32 Registros de Proposito General de 64 Bits (r0 .. r31)
 - r0 vale siempre 0
- 32 Registros de punto flotante de 64 Bits (f0 .. f31)
- Memoria de Datos y Programas estan separadas
- 4.096 (2¹²) bytes de memoria para datos.
- No hay pila
- No hay flags
- Acceso a memoria limitado a 2 instrucciones (y sus variantes)
 - LOAD: obtener valores de la memoria
 - STORE: almacenar un valor en la memoria
- Instrucciones de tamaño fijo, 32bits



Ejecucion Secuencial

- Una CPU puede ejecutar una instrucción en varias etapas:
 - Obtener la proxima instrucción
 - Decodificar la instrucción
 - Ejecutar la instrucción
 - Actualizar los resultados
- En el caso del MSX88, cada instruccion va pasando por los diferentes estados y una vez finalizada dicha instrucción, se obtiene la siguiente.
- Suponiendo que cada etapa se realiza en un ciclo de reloj de CPU:
 - Cada instrucción tarda en ejecutarse 4 ciclos
 - 1 instrucción, 4 ciclos
 - 2 instrucciones, 8 ciclos
 - 3 instrucciones, 12 ciclos
 - 10 instrucciones, 40 ciclos
 - n cantidad de instrucciones se ejecutaran en 4 * n ciclos.



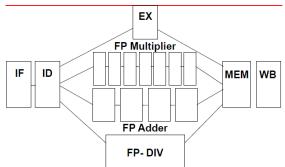
Segmentación de Cause

- Si las etapas son independientes podemos ejecutarlas en paralelo
- Mientras ejecutamos una etapa de una instrucción, ejecutamos otra etapa de otra instrucción.
- De esta manera usamos mejor la CPU, todas las etapas estan en funcionamiento en cada ciclo de CPU.
 - Cada instrucción tarda en ejecutarse 4 ciclos
 - 1 instrucción, 4 ciclos
 - 2 instrucciones, 5 ciclos
 - 3 instrucciones, 6 ciclos
 - 10 instrucciones, 11 ciclos
 - n cantidad de instrucciones se ejecutaran en 3 + n ciclos.
- A esta técnica se la llama Segmentación de Cause

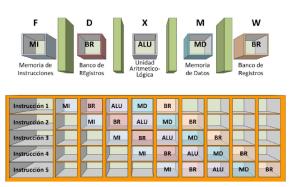
Ciclo de Instruccion de WinMIPS64

- Busqueda (IF)
 - Se accede a la memoria buscando la instrucción
 - Se incrementa el PC
- Decodificación / Búsqueda de Operandos (ID)
 - Se decodifica la instrucción
 - Se accede al banco de registro por los operandos
 - Si es un salto, se calcula el destino y si hay que realizarlo o no
- Ejecución / Dirección Efectiva (EX)
 - Si es una instrucción de proceso, se ejecuta en la ALU
 - Si es acceso a memoria, se calcula la dirección efectiva
 - Si es un salto, se calcula el nuevo PC
- Acceso a Memoria (MEM)
 - Si es un acceso a memoria, se accede
- Almacenamiento (WB)
 - Se almacena el resultado (si existiese) en el banco de registros

- Cada etapa se ejecuta en un ciclo de reloj, salvo que involucre la unidad de punto flotante.
- Las etapas de punto flotante dependen de la operación a realizar:
 - Suma se realiza en 4 ciclos
 - Multiplicación se realiza en 7 ciclos
 - División se realiza en 24 ciclos



Ejemplo



Directivas de Ensamblador

- Directivas generales
 - .data: comienzo de segmento de datos
 - .text o .code: comienzo de segmento de código
- Las directivas que mas vamos a usar son .data y .code.
 Recordar que la memoria de programas y datos son diferentes.
- Las variables van en el segmento de datos, acordarse antes de empezar a definir variables de escribir la directiva .data.
- Las instrucciones van en el segmento de código. Antes de escribir instrucciones, acordarse de usar la directiva .code
- En un programa puedo intercalar directivas .data y .code
- Las etiquetas se definen con un nombre y dos puntos (:), pueden usarse para referenciar partes de un programa (en saltos) como para nombrar variables
- La CPU empieza a ejecutar instrucción a partir de la posición de memoria 0. Por lo que no es necesario usar la directiva .org.



Variables

Las variables se deben guardar en la memoria de datos.

- .word w1: entra un word de dato (64-bits, 8 bytes)
- .byte b1: entra bytes (8-bits, 1 byte)
- .word32 n1: entra número(s) de 32 bit (32-bits, 4 bytes)
- .word16 n: entra número(s) de 16 bit (16-bits, 2 bytes)
- .double f: entra número de punto flotante
- .ascii "cadena": entra string ascii
- .asciiz "cadena": entra string ascii terminado en cero

Variables

```
numero1: .word -50
```

numero2: .word 12302

cadena: .asciiz "Hola∟Mundo"

- Los números pueden ser:
 - Números sin signo, en este caso se guardan en BSS
 - Números con signo, en este caso se guardan en Ca2
- Acordarse, que es nuestro programa el que le da significado a los números

Generalidades

- Hay instrucciones especificas para leer y escribir en memoria.
- Las instrucciones aritmetico-logicas poseen 3 operandos, el primer operando es en el que se va a guardar el resultado, el 2do y 3ro son los parametros de la operación.
- La instrucción *HALT* se utiliza para detener el simulador.
- La instrucción NOP es una instrucción que no realiza ninguna operación. Mas adelante vamos a ver su utilidad.
- El registro *r0* siempre vale 0, no se puede cambiar su valor.
- Las instrucciones de punto flotante solo aceptan como parámetros registros de punto flotante. Salvo las instrucciones de conversión entre punto flotante/punto fijo, no hay instrucciones que acepten diferentes tipos de registros.

Instrucciones Generales con valores inmediatos

Son instrucciones aritmetico/logicas donde el tercer párametro es un valor inmediato.

- DADDI r10, r12, 25: Suma r12 + 25 y lo guarda en r10
- ANDI r8, r10, 11: Hace la operación AND entre r10 y 11 y lo guarda en r8
- ORI r3, r11, 33: Hace la operación OR entre r11 y 33 y lo guarda en r3
- XORI r12, r20, 111: Hace la operación XOR entre r20 y 111 y lo guarda en r12
- SLTI r3, r5, 100: Si r5 es menor que 100 entonces r3 es 1, sino r3 es 0

Notar que todas las instrucciones finalizan con I.

Instrucciones manipulación de Registros

- DADD r1, r2, r3: Suma r2 + r3 y lo guarda en r1
- DSUB r10, r12, r13: Resta r12 r13 y lo guarda en r10
- AND r8, r10, r11: Hace la operación AND entre r10 y 11 y lo guarda en r8
- OR r3, r11, r13: Hace la operación OR entre r11 y r13 y lo guarda en r3
- XOR r1, r20, r1: Hace la operación XOR entre r20 y r1 y lo guarda en r1
- SLT r3, r5, r0: Si r5 es menor que r0 entonces r3 es 1, sino r3 es 0

Accesos a memoria

- Los accesos a memoria se realizan con 2 instrucciones: LOAD y STORE
 - LD r10, desplaz(r15): Leer desde memoria y guardarlo en el registro r10
 - SD r10, desplaz(r15): Guardar en memoria el valor del registro r10
- La dirección desde donde leer/escribir se calcula como la suma de desplaz + r15
- desplaz puede ser el nombre de una variable, o un número.

Accesos a memoria - Variables

```
.data
numero: .word 25
.code
LD r1, numero(r0)
DADDI r5, r0, numero
LD r2, 0(r5)
```

- La instrucción *LD r1*, *numero(r0)* carga en el registro *r*1, lo que vale la variable numero
- La instruccion *LD r2*, *0(r5)* carga en el registro *r*2, el mismo valor
 - Primero cargamos el registro r5 el offset de la variable numero
 - La dirección de LD r2, O(r5) es r5 + 0 = numero + 0 = numero

Accesos a memoria - Tablas

```
.data
tabla: .word 25, 50, 75, 100
.code
DADD r10, r0, r0 ; r10 = 0
LD r2, tabla(r10)
DADDI r10, r10, 8 ; r10 = r10 + 8
LD r3, tabla(r10)
```

- Podemos obtener los diferentes valores de una tabla
- El registro r10 es nuestro indice en la tabla
- Debemos incrementar r10 en 8, ya que cada elemento de la tabla es de 64bits (8 bytes)

Accesos a memoria - Tablas II

```
.data
tabla: .word 25, 50, 75, 100
.code
DADDI r11, r0, tabla ; r11 = tabla
LD r2, 0(r11)
DADDI r11, r11, 8 ; r11 = r11 + 8
LD r3, 0(r11)
```

- Alternativamente podemos realizar lo mismo de antes de otra manera.
- El registro r11 es nuestra posicion en la memoria
- Debemos incrementar r11 en 8, ya que cada elemento de la tabla es de 64bits (8 bytes)



Accesos a memoria - Leer Memoria

Las diferentes variantes de la instrucción LOAD:

- LD r1, offset(r2): Load Doubleword (64 bits)
- LB r1, offset(r2): Load Byte
- LBU r1, offset(r2): Load Byte s/signo
- LH r1, offset(r2): Load Halfword (16 bits)
- LHU r1, offset(r2): Load Halfword s/signo
- LW r1, offset(r2): Load Word (32 bits)
- LWU r1, offset(r2): Load Word s/signo
- LB, LH y LW extienden el bit de signo.

Accesos a memoria - Escribir Memoria

Las diferentes variantes de la instrucción STORE:

- *SD r1, offset(r2)*: Store Doubleword
- SB r1, offset(r2): Store Byte
- SH r1, offset(r2): Store Halfword
- *SW r1, offset(r2)*: Store Word

Control de Flujo

- Salto incondicional:
 - J etiqueta: Saltar a etiqueta
- Salto condicional que compara 2 registros:
 - $BEQ\ r1,\ r2,\ etiqueta$: si r1=r2 saltar a etiqueta
 - BNE r1, r2, etiqueta: si r1 distinto r2 saltar a etiqueta
- Salto condicional que compara con cero:
 - $BEQZ \ r1$, etiqueta: si r1 = 0 saltar a etiqueta
 - BNEZ r1, etiqueta: si r1 no es 0 saltar a etiqueta

Ejemplo 1

```
.data
A: .word 1
B: .word 2
.code
    LD R1, A(R0); Cargo R1 con A
    LD R2, B(R0); Cargo R2 con B
    SD R2, A(R0); Guardo en A R2
    SD R1, B(R0); Guardo en A R1
    HALT
```

Ejemplo 2

```
.data
A:
  .word 1
B:
   .word 6
.code
             R1, A(R0)
      LD
      LD
             R2, B(R0)
LOOP:
      DSLL
             R1, R1, 1
      DADDI
             R2, R2, -1
      BNEZ
             R2, LOOP
      HALT
```

Manejo de una Tabla

```
.data
        TABLA: .word 20, 1, 14, 3, 2, 58, 18, 7, 12, 11
        NUM:
                .word 7
        LONG:
                .word 10
.code
        LD
                R1, LONG(RO)
        LD
                R2, NUM(R0)
        DADD
                R3, R0, R0
        DADD
                R10, R0, R0
LOOP:
        LD
                R4, TABLA(R3)
        BEQ
                R4, R2, LISTO
        DADDI
                R1, R1, -1
        DADDI
                R3, R3, 8
        BNEZ
                R1, LOOP
        J
                FIN
LISTO:
        DADDI
                R10, R0, 1
FIN:
        HALT
```