Trabajo Práctico 2: Data path y pipeline

José Ignacio Sbruzzi, *Padrón Nro. 97.452*
jose_sbruzzi@hotmail.com
Leandro Huemul Desuque, *Padrón Nro. 95.836*
desuqueleandro@gmail.com

2do. Cuatrimestre de 2016
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

Se introdujeron modificaciones sobre datapaths para agregar ciertas funcionalidades a los mismos. El objetivo del trabajo es la práctica y el aprendizaje del funcionamiento del datapath.

1. Introducción

MIPS ES.... LOS DATAPATHS SON...

2. Desarrollo

2.1. Modificación del DP monociclo

La modificación introducida fue una nueva instrucción denominada "load byte unsigned"(mnemónico lbu), que en vez de cargar un word como lw, carga un byte, y no hace extensión de signo: los bits a la izquierda del último son simpre 0.

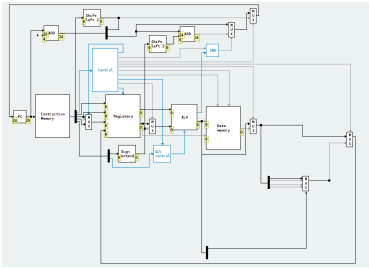

Figura 1: Estado final del datapath monociclo

2.1.1. Procedimiento de la modificación

Luego de notar que incluso al recibir direcciones desalineadas la memoria carga una palabra alineada (es decir, por ejemplo, las direcciones 0, 1, 2 y 3 hacen que se carge la palabra en la dirección 0), se decidio hacer que lbu funcionara exactamente igual que lw, pero removiendo 3 bytes, haciendo parecer de esta manera que se carga un solo byte.

Para esto, se agregó una señal a la unidad de control llamada çortar". La señal impacta el sistema por medio de un multiplexor que define si se guardará en el register file la palabra obtenida de memoria o una versión modificada que mantiene un sólo byte. El word modificado se calcula incluso si la señal çortar.es 0.

La modificación del word es llevada a cabo por medio de un *Distributor* que divide el word en sus 4 bytes. Luego, por medio de un multiplexor se realiza la elección de uno de esos 4 bytes. Finalmente, el byte seleccionado se concatena a 24 ceros, quedando así el byte seleccionado en la parte menos significativa del word.

2.1.2. Pruebas

A continuación se muestra el código assembler utilizado como prueba. El CPU pasa la prueba si en el registro t0 queda el valor 0x000000aa, el registro t2 queda en 0x000000bb, el t3 en 0x000000cc y el t4 en 0x000000dd.

```
.data
prueba: .word 0xAABBCCDD, 0x11223344, 0x22222222, 0x33333333

.text

mostrar:
        la $t5, prueba
        lbu $t0, 0($t5)
        lbu $t1, 1($t5)
        lbu $t2, 2($t5)
        lbu $t3, 3($t5)
```

Figura 2: Prueba utilizada para verificar el correcto funcionamiento de la modificación

2.2. Modificación del DP pipeline: jal

Se decidió agregar la instrucción jal antes de la bgezal para aprovechar el hardware añadido. Inicialmente se notó que el los branches se resuelven en la etapa MEM: es allí donde se modifica el PC. El guardado de la dirección de retorno en el registro ra puede realizarse en la etapa WB, ya que jal -tal como hace beq- "flushea"todos los registros interetapa, con lo cual al ejecutarse el WB del jal, la instrucción que sigue estará en la etapa IF.

Con el objetivo de hacer la menor cantidad de modificaciones posibles, no se cambia la condición del salto: se siguen comparando dos registros, solo que esta vez siempre se comparan los registros zero. Así, *jal offset* equivale a *beq zero, zero, offset* con la única diferencia de que se guarda PC+4 en RA.

Entonces, se agregó una señal "GuardarPC.a la unidad de control, la cual se guarda en los registros interetapa siguientes a ID. Además se agregó NewPC (presente en ID/EX) a a los registros EX/MEM y MEM/WB.

Se Agregan también dos multiplexores en la etapa WB controlados por la señal "GuardarPC": uno permite fijar la señal WriteAddress en 31 (correspondiente a RA) y otro que permite fijar WriteData en NewPC.

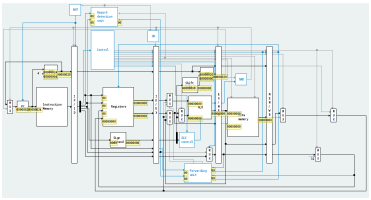

Figura 3: Datapath pipeline con jal

2.2.1. pruebas

La prueba utilizada se expone a continuación. Si es exitosa, t1 termina en 0, t0 termina en 1 y ra en 0c.

```
.data
prueba: .word 0xAABBCCDD, 0x11223344, 0x22222222, 0x33333333

.text

mostrar:
        add $t1, $zero, $ra
        addi $t0, $zero, 1
        jal funcioncita
        addi $t0, $zero, 2
funcioncita:
        add $t2, $zero, $ra
```

Figura 4: Datapath pipeline con jal

2.3. Modificación del DP pipeline: bgeal y bgezal

Una tarea solicitada fue la implementación de una instrucción BGEZAL. En vez de implementar bgezal, se decidió implementar bgeal, y hacer que bgezal sea una pseudoinstrucción. La pseudoinstrucción se traduce como una única instrucción (se cambia uno de los argumentos por el registro zero), con lo que no hay pérdidas de velocidad y se gana versatilidad.

La compuerta AND de la etapa MEM reúne la información proveniente de las señales Branch (proveniente de la unidad de control, fijada en la instrucción) y Zero (proveniente de la ALU, es 1 si el resultado de la ALU es 0). La ALU implementa la operación slt, con lo cual, por medio de modificaciones de software a partir de la instrucción JAL se puede obtener BGEAL. Se aprovechó que slt deja el bit Zero en 1 si la condición "mayor o igual"se cumple (pero no "menor estricto").

2.3.1. modificación bgeal

Las modificaciones respecto de JAL, así, no se hicieron sobre el archivo que describe el hardware (.cpu) sino sobre el que describe el software (.set). Se tomó la línea de la sección ïnstructions"que describe beq y sólo se reemplazaron el opcode (por uno nuevo, distinto del de JAL) y el nombre mismo de la instrucción.

En la sección çontrol"se copió la línea que corresponde al beq, pero se cambió el valor de RegWrite y GuardarPC a 1 (tal como se había hecho para JAL) y se asignó un nuevo ALUop. El nuevo ALUop se hizo corresponder con la operación slt en la sección .ªlu"del mismo archivo.

Luego de hacer pruebas se detectó que el registro RA se guardaba siempre, se dé o no el salto, con lo que se hizo necesario almacenar el valor de salida del componente AndBranch (que registra si realmente se llevó a cabo el branch) en el registro interetapa MEM/WB, y luego usar una compuerta AND y un multiplexor para filtrar el valor de la señal RegWrite.

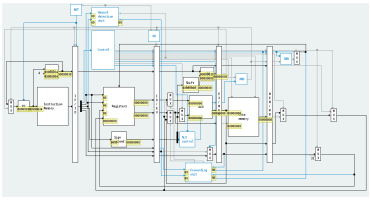

Figura 5: Datapath pipeline con las modificaciones necesarias para bgeal

2.3.2. modificación bgezal

Se agregó la pseudoinstrucción bgezal, que es bgeal pero con su segundo argumento en el registro 0.

2.3.3. pruebas

Se utilizaron tres pruebas, una de las cuales está representada a continuación. Las otras dos son variantes en las que se cambia el -5 por un número positivo y por 0. De ser un número positivo o 0, al final de la ejecución RA es 0c y t0 es aaa, de lo contrario, RA es 0 y t0 es bbb.

```
1  .data
2  prueba: .word 0xAABBCCDD, 0x11223344, 0x22222222, 0x33333333
3
4  .text
5
6  mostrar:
7            addi $t0, $zero, 0x0AAA
8
9            addi $t1, $zero, -5
10           bgezal $t1, funcioncita
11           addi $t0, $zero, 0x0BBB
12 funcioncita:
13           add $t2, $zero, $ra
14
```

Figura 6: Prueba de bgezal.

2.3.4. mejoras posibles

Una vez implementada bgeal, se podría implementar jal como una pseudoinstrucción y de esta manera se reduciría el número de instrucciones que implementa el datapath.

2.4. Modificación del DP pipeline: sb

La instrucción sb (store byte) almacena el byte más bajo del registro pasado en memoria. El problema más importante que presenta esta instrucción es el hecho de que DrMips maneja la memoria de a words: es imposible guardar o cargar un único byte. Así, la instrucción debería cargar un word, superponerle el byte a guardar, y luego guardar ese word a memoria. Otro problema es que es imposible guardar y cargar datos a memoria en el mismo ciclo, con lo que la instrucción debería ocupar la etapa MEM durante dos ciclos. Esto va en contra de la idea central del pipeline, con lo que se hace necesario implementar sb como una pseudoinstrucción que tenga al menos dos instrucciones: una de carga y otra de almacenamiento. Las operaciones intermedias en las que se transforma el word también son complicadas, ya que requieren la aplicación de varias máscaras de bits.

#define SB(Rs,Imm,Rt,aux) addi at, zero, 3 addi aux, Rt, Imm and at, at, aux //el byte a superponer está en at

addi aux, zero, 0 beq aux, at, Cero

addi aux, zero, 1 beq aux, at, Uno

addi aux, zero, 2 beq aux, at, Dos

addi aux, zero, 3 beq aux, at, Tres

Cero: lw aux, Imm(Rt) addi at, zero, 0xFF sll at, at, 24 and aux, aux, at sll Rs,Rs,24 add Rs, at, Rs j Fin Uno: lw aux, Imm(Rt) addi at, zero, 0xFF sll at, at, 16 and aux, aux, at sll Rs,Rs,16 add Rs, at, Rs j Fin Dos: lw aux, Imm(Rt) addi at, zero, 0xFF sll at, at, 8 and aux, aux, at sll Rs,Rs,8 add Rs, at, Rs j Fin Tres: lw aux, Imm(Rt) addi at, zero, 0xFF and aux, aux, at add Rs, at, Rs Fin:nop

lw at, Imm(Rt)

3. Resultados

3.1. Ventajas de implementar lbu/bgeal/etc...

A continuación se muestra una función assembly que ejecuta el lbu.

Puede concluirse que ejecutar una operación equivalente al lbu descripto lleva como 1000000 instrucciones o ciclos más pq es monocycle

4. Conclusiones

Referencias

[1] Gardner, Martin. “Mathematical Games - The fantastic combinations of John Conway’s new solitaire game ’life‘ ” Scientific America, 223. pp. 120-123. ISBN 0-89454-001-7. Archivado del original en: https://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

Consultado en septiembre 2016.

[2] Sitio web de GXemul http://gxemul.sourceforge.net/

Consultado en septiembre 2016.