Trabajo práctico 3: Datapath

Augusto Arturi (#97498)
turitoh@gmail.com

Matias Rozanec (#97404)
rozanecm@gmail.com

Agustin Miguel Payaslian (#96885)
payas17@hotmail.com

Grupo Nro. # - 2do. Cuatrimestre de 2017

66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

30.nov.2017

Resumen

El objetivo del presente trabajo es familiarizarse con la arquitectura de una CPU MIPS, específicamente con el datapath y la implementación de instrucciones. Para ello se agregarán diversas instrucciones de CPU provistas por el simulador DrMIPS.

1. Introducción

Aquí se comenta en forma escueta cómo está constituido el presente informe, donde básicamente se encuentran dos secciones principales: Desarrollo y Conclusiones.
En Desarrollo se encuentran breves comentarios sobre la implementacion de las distintas instrucciones. En la seccion conclusiones se discuten los resultados obtenidos.

2. Desarrollo

2.1. Implementación

Para cada una de las instrucciones a agregar, lo primero que se estudió fue el código de la instrucción. De esta forma, se pudo escribir la instrucción en el archivo .set correspondiente al datapath que se pedía modificar. Una vez agregada la instrucción en el archivo .set, se procedió a analizar la arquitectura. En base a este análisis se pudo ver qué modificaciones había que llevar adelante en el datapath para que la instrucción agregada efectivamente haga lo que debe hacer.

2.2. Agregado de instrucciones

En todos los casos, los códigos específicos de cada instrucción han sido consultados en manuales y bibliografía autorizada.

2.2.1. Instrucciones `sll` y `srl` en el datapath `pipeline.cpu`

Para agregar estas dos instrucciones, se debieron agregar las siguientes líneas:

```
"sll":  {"type": "R", "args": ["reg", "reg", "int"], "
   fields": {"op": 0, "rs": "#2", "rt": "#3", "rd": "
   #1", "shamt": "#3", "func": 0}}


"srl":  {"type": "R", "args": ["reg", "reg", "int"], "
   fields": {"op": 0, "rs": "#2", "rt": "#3", "rd": "
   #1", "shamt": "#3", "func": 2}}
```

En ambos casos se trata de instrucciones de tipo `R`, cuyo `opcode` es 0, y cuyos argumentos 1, 2 y 3 corresponden a los registros `rd`, `rs` y `shamt` respectivamente. En el caso de `sll`, `func` recibe el valor 0, mientras que en el caso de `srl`, este código es 2.
Lugo hubo que vincular los códigos de función con las operaciones de la ALU

propiamente dicho. Esto se logra mediantes las siguientes líneas:
En la sección de control de la ALU:

```
{"aluop": 2, "func": 0,  "out": {"Operation": 13}}
{"aluop": 2, "func": 2,  "out": {"Operation": 14}}
```

En la sección de operaciones de la ALU:

```
"13": "sll"
"14": "srl"
```

2.2.2. Instrucción `blt` en el datapath `pipeline.cpu`

Esta vez se agregó la instrucción como pseudo instrucción. Esto es porque la arquitectura no puede ser modificada de forma que se pueda soportar esta operación. Las pseudo instrucciones son muy útiles a pesar de no ser implementeadas nativamente, ya que le simplifican mucho las tareas al programador. Cada pseudo instrucción está compuesta por varias instrucciones que sí están implementadas en hardware nativament.
Para agregar esta pseudo instrucción, se debió agregar la siguiente línea.

```
"blt":  {"args": ["reg", "reg", "offset"], "to": ["slt
   ␣$1,␣#1,␣#2", "bne␣$1,␣$0,␣#3"]}
```

2.2.3. Instrucción `j` en el datapath `unicycle.cpu`

```
"j":    {"type": "J", "args": ["target"], "fields": {"
   op": 2, "target": "#1"}, "desc": "PC␣=␣target"}
```

Esta instrucción ya se encontraba implementada en el datapath sugerido.
En cuanto al análisis de hazards, se llegó a la conclusión de que no es posible que en este datapath haya hazards, debido a que todo ocurre en un solo ciclo. Los hazards ocurren en sistemas con pipeline, donde una instrucción depende del resultado de otra previa de una forma en que ésta esté expuesta por el solapamiento de instrucciones en el pipeline. Como en este caso no hay pipeline, no pueden ocurrir hazards.

2.2.4. Instrucción `jr` en el datapath `pipeline.cpu`

En este caso se trata de una operación de tipo `R`, cuyo `opcode` es 0, y su número de función es 8. El argumento que recibe corresponde a los bits del registro `rs`.
Para agregar la instrucción, hubo que agregar la siguiente línea en el archivo .set correspondiente: Para agregar esta pseudo instrucción, se debió agregar la siguiente línea.

```
"jr":   {"type": "R", "args": ["reg"], "fields": {"op"
   : 0, "rs": "#1", "rt": "0", "rd": "0", "shamt": 0,
   "func": 8}, "desc": "The␣jr␣instruction␣loads␣the␣
   PC␣register␣with␣a␣value␣stored␣in␣a␣register."},
```

2.2.5. Instrucción `jalr` en el datapath `pipeline.cpu`

Formato de la instrucción:
`JALR rs` (`rd` = 31 implied)
`JALR rd, rs`
Descripción:
$rd \leftarrow return\_addr, PC \leftarrow rs$
En este caso se vuelve a tratar de una pseudo instrucción. Ésta es compuesta por las operaciones `addi` y `jr`, ésta última implementada anteriormente.
La instrucción completa es: `jalr rd, rs`. En caso de omitir el campo `rd`, se tomará su valor como igual a 31. Como acá hay dos casos posibles, hubo que agregar dos líneas en el apartado de pseudo instrucciones en el archivo .set.

```
"jalr": {"args": ["reg", "reg"], "to": ["addi␣#1,␣$0,␣
    $ra", "jr␣#2"]},


"jalr": {"args": ["reg"], "to": ["addi␣31,␣$0,␣$ra", "
    jr␣#1"]}
```

2.3. Modificaciones en el datapath

2.3.1. Instrucciones `sll` y `srl` en el datapath `pipeline.cpu`

Para agregar estas instrucciones, el problema que se presentó fue que la arquitectura no estaba preparada para enviarle a la alu los bits `shamt` desde el código de instrucción correspondiente a las instrucciones de bit shifting (que son 6 en total: `sll`, `srl`, `sra`, `sllv`, `srlv` y `srav`). Por lo tanto, hubo que modificar el datapath de forma que la ALU pueda recibir dichos bits.
Lo primero que se hizo, es agregar los cables necesarios para hacer llegar los bits de `shamt` hasta la ALU. Como éstos son solo 5 y la ALU recibe 32 por entrada, hubo que agregar otros (32-5) bits y concatenarlos con los 5 que queríamos hacer entrar. Esto se realizó mediante un generador de un valor constante (componente `Constant` de tamaño (32-5), con todos los bits en 0) y un `Concatenator`, que concatena todos los bits, obteniendo así el valor de `shamt` pero en 32 bits, listo para entrar a la ALU.
Como la ALU no puede recibir una entrada más, hubo que agregar un multiplexor que permita seleccionar si permitir entrar la entrada que ya está conectada en este momento o la entrada con el valor de `shamt`. Para lograr esto, se cableó la salida del multiplexor `MuxReg` a la entrada 0 del nuevo multiplexor, `MuxShamt`. A la otra entrada de este multiplexor se cableó el valor de `shamt`. La salida de este nuevo multiplexor se conecta a la entrada de la ALU que quedó libre anteriormente.
Queda ahora para resolver qué usar como selector de este multiplexor.
Los bits de `shamt` deben entrar en la ALU únicamente cuando se trata de operaciones de bit shifting. Para esto hay que tener en cuenta los siguientes códigos: `opcode` y `func`.
`opcode` es 0 para todas las instrucciones de tipo `R` menos una. Los códigos `func` tienen los 3 bits más significativos en 0 en todos los casos de operaciones de bit shifting. Para implementar el selector del nuevo multiplexor se utilizaron compuertas tipo `or` y `not` sobre los bits mencionados.

2.3.2. Instrucción `blt` en el datapath `pipeline.cpu`

Al agregase como una pseudo Instrucción, no hubo que realizar modificaciones en el datapath.

2.3.3. Instrucción `j` en el datapath `unicycle.cpu`

Como se mencionó anteriormente, esta instrucción ya estaba incluída, por lo que no hubo que realizar modificaciones.

2.3.4. Instrucción `jr` en el datapath `pipeline.cpu`

En este caso hubo que considerar una entrada alternativa al Program Counter, por lo que se agregó un multiplexor que recibía por una entrada lo que hasta ahora recibía el Program Counter, y por la otra el valor del registro que contiene la dirección a la que nos interesa ir. La salida de este multiplexor va a la entrada del Program Counter. Mediante el uso de compuertas `or` y `not`, estudiando el código de operación correspondiente a esta instrucción, se implementó el selector de este nuevo multiplexor.

2.3.5. Instrucción `jalr` en el datapath `pipeline.cpu`

Al agregase como una pseudo Instrucción, no hubo que realizar modificaciones en el datapath.

3. Conclusiones

66:20 Organización de computadoras
Trabajo práctico 3: Data Path

1. Objetivos

El objetivo de este trabajo es familiarizarse con la arquitectura de una CPU MIPS, específicamente con el datapath y la implementación de instrucciones. Para ello, se deberán agregar instrucciones a diversas configuraciones de CPU provistas por el simulador DrMIPS [**?**]

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección **??**), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo[1], y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

4. Recursos

Usaremos el programa DrMIPS [**?**] para configurar y simular el data path de un procesador MIPS [**?**], tanto uniciclo como multiciclo.

5. Descripción.

5.1. Introducción

El programa DrMIPS nos permite evaluar distintos diseños de datapath para procesadores MIPS32, al darnos la posibilidad de organizarlo como queramos. Si bien sólo puede haber uno de algunos de los componentes del DP (como el registro de PC o la unidad de control), podemos poner sumadores, multiplexores, extensores de signo y conexiones arbitrariamente. También es

[1] `http://groups.yahoo.com/group/orga6620`

posible modificar el conjunto de instrucciones. Además de la estructura lógica del DP, DrMips nos permite escribir programas simples y simular su ejecución en el DP, mostrando los valores que toman las diversas entradas y salidas de cada elemento. El programa se puede conseguir en `https://bitbucket.org/brunonova/drmips/wiki/Home`, o se puede descargar para Ubuntu, ya sea desde el repositorio de Ubuntu (aunque la versión está desactualizada) o autorizando un repositorio externo (ver [**?**]).

5.2. Datapaths

El programa viene con algunos DP ya implementados, a saber:
Uniciclo:

- `unycicle.cpu`: El DP uniciclo por defecto.
- `unycicle-no-jump.cpu`: Variante más simple del DP uniciclo que no soporta la instrucción `j`.
- `unycicle-no-jump-branch.cpu`: Una variante aún más simple que no soporta `jump` ni `branch`.
- `unycicle-extended.cpu`: Una variante que soporta instrucciones adicionales, como multiplicación y división.

Multiciclo:

- `pipeline.cpu`: El DP de pipeline por defecto, implementa detección de hazards. Los DP de pipeline no soportan la instrucción `j` (salto).
- `pipeline-only-forwarding.cpu`: Variante del DP de pipeline que implementa forwarding pero no genera stalls (genera resultados incorrectos).
- `pipeline-no-hazard-detection.cpu`: Otra variante que no hace hazard detection de ninguna manera (genera resultados incorrectos).
- `pipeline-extended.cpu`: Una variante que soporta instrucciones adicionales, como multiplicación y división, como `unycicle-extended.cpu`.

5.3. Instrucciones a implementar

1. Implementar las instrucciones `sll` y `srl` en el datapath `pipeline.cpu`.
2. Implementar la instrucción `blt` en el DP `pipeline.cpu`. ¿Se puede hacer nativamente, o hay que emplear pseudoinstrucciones? Justificar.
3. Implementar la instrucción `j` en el DP `unicycle.cpu`. Verificar que no se produzcan hazards.
4. Implementar la instrucción `jr` en el DP `pipeline.cpu`.
5. Implementar la instrucción `jalr` en el DP `pipeline.cpu`.

6. Implementación.

Los archivos antes mencionados, así como los archivos `.set` que contienen los datos del conjunto de instrucciones, están en formato JSON [**?**], y se pueden modificar con un editor de texto. Se sugiere uno que pueda hacer *color syntax highlighting*, como el `gedit` que viene con el Ubuntu. La explicación de los formatos se encuentra en el archivo `configuration-en.pdf` que se distribuye con el programa.

7. Pruebas

En todos los casos debe verificarse que la instrucción se ejecute correctamente. Esto implica que el PC tome el valor deseado, y además que en el caso del DP multiciclo no se produzcan hazards, como ser la ejecución de la instrucción siguiente al salto, o en el caso de utilizar el valor de un registro, que éste tenga el valor correcto.

8. Informe.

Se debe entregar:

- Informe describiendo el desarrollo del trabajo práctico.
- Capturas de pantalla de los DP modificados.
- Programas de prueba.
- CD conteniendo los DP, los programas de prueba y los conjuntos de instrucciones usados en cada caso.
- Este enunciado.

9. Fecha de entrega.

- Vencimiento: Jueves 30 de Noviembre de 2017.

Referencias

[1] DrMIPS, `https://bitbucket.org/brunonova/drmips/wiki/Home`.

[2] PPA de Bruno Nova, `https://launchpad.net/~brunonova/+archive/ubuntu/ppa`.

[3] ECMA-404 The JSON Data Interchange Standard, `http://www.json.org/`.

[4] "Computer organization and design: the hardware-software interface", John Hennessy, David Patterson. Capítulo 5.