

Trabajo práctico 2: Data path y pipeline

Tomas Franco, *Padrón Nro. 91.013*

`tomasnfranco@gmail.com`

Pablo Ascarza, *Padrón Nro.*

`pabloqac87@gmail.com`

Diego Meller, *Padrón Nro. 91.299*

`dmeller@fi.uba.ar`

Grupo Nro. - 2do. Cuatrimestre de 2016

66.20 Organización de Computadoras

Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

Utilizamos el programa llamado DrMIPS[1] para simular un procesador MIPS32[2] y poder modificarlo para poder utilizar 3 nuevas instrucciones, una para un procesador uniciclo[2] y las otras dos en un procesador pipeline[2]

1. Enunciado

Se adjunta el enunciado al informe.

2. Desarrollo

Se modificaron los archivos `.cpu` y `.set` cuyo formato es JSON[3] correspondientes a cada arquitectura de cada tipo de procesador para las instrucciones pedidas.

2.1. lbu

Instrucción a desarrollar

```
1 | lbu Rs, Imm(Rt)
```

(Load byte unsigned). Esta instrucción de tipo I carga en Rs el valor del byte almacenado en la posición resultante de sumar el valor del registro Rt y el valor de 16 bits Imm.

En un principio se penso hacer la instrucción como una pseudo-instrucción,

Pseudoinstrucción Equivalente

```
1 | lw Rs, Imm(Rt)
2 | li Rtemp, 255
3 | and Rs, Rs, Rtemp
```

pero se eligió modificar el data path dado que eso daría una mejoría en el tiempo ya que se va a ejecutar una sola instrucción en vez de tres (al ser monociclo cada instrucción tarda lo mismo que es lo que dure la instrucción mas larga).

La modificación al **Data Path** fue:

1. Agregar luego de la salida del Data Memory un bifurcador que envia el word completo a un multiplexor y a un distribuidor.
2. El distribuidor toma solo el byte (de los bits menos significativos) y lo envía a un componente que rellena con ceros el byte hasta volver a formar una palabra (“rellenador” de ceros).
3. Del “rellenador” de ceros va a la otra entrada del multiplexor.
4. Se agrego una línea de control llamada “LoadByte” que le indica al multiplexor cuando tomar solo el byte. Si esta línea esta apagada toma el word completo.
5. Del multiplexor se envía la salida hacia el multiplexor que ya existía que elegía si tomaba el dato de la Memoria o del resultado de la ALU.
6. Ordenamos los ítems y los cables entre los mismos para que el DP se vea bien.

En cuanto al **set de instrucciones** lo que se hizo fue:

1. Agregar la instrucción propiamente dicha, con código de operación 36 (utilizamos un código cercano al loadword que no este utilizado en esta arquitectura).
2. Agregar a la unidad de control que dicha operación utiliza las mismas líneas de control que el loadword agregando línea de control “LoadByte” en 1.

2.2. sb

Instrucción a desarrollar

```
1 | sb Rs, Imm(Rt)
```

(Store byte). Esta instrucción de tipo I almacena el byte menos significativo de Rs en la posición resultante de sumar el valor del registro Rt y el valor de 16 bits Imm.
Esta instrucción también se podría reemplazar con una pseudo-instrucción,

Pseudoinstrucción Equivalente

```
1 | lw $1, Imm(Rt)
2 | li $9, -256
3 | and $1, $1, $9
4 | or $1, $1, Rs
5 | sw $1, Imm(Rt)
```

pero se eligio crear una mezcla de una pseudo-instrucción con una nueva instrucción que si modifica el datapath para no tener que utilizar algún registro auxiliar (temporal) como el \$9 (\$t1).

Pseudoinstrucción Final

```
1 | lw $1, Imm(Rt)
2 | sbr $1, Rs, $0
3 | nop
4 | sw $1, Imm(Rt)
```

La instrucción nueva **sbr** requirió la modificación al **Data Path**:

1. Del ForkEXR1 que distribuye el resultado de la ALU se tomo una rama hacia un nuevo distribuidor donde se toman los bits 8 a 31 (los 3 bytes mas significativos).
2. Se agrego un Fork que obtiene el ReadData2 del EX/MEM y lo envia, además de donde enviaba antes, a un nuevo distribuidor donde se toman los bits 0 a 7 (el byte menos significativo)
3. Se agrego un Concatenador que toma de esos 2 distribuidores los bytes quedando un word nuevamente.
4. Se agrego una línea de control llamada WriteByte.
5. Se agrego un multiplexor que toma o lo que trae la memoria o lo que trae el concatenador y se elige por el WriteByte.
6. Ordenamos los ítems y los cables entre los mismos para que el DP se vea bien.

En cuanto al **set de instrucciones** lo que se hizo fue:

1. Agregar la instrucción sbr, con código de operación 44.
2. Agregar la pseudo-instrucción sb con las llamadas a las instrucciones.
3. Agregar a la unidad de control que la operación sbr utiliza las líneas de control ALUSrc, RegDst, RegWrite, MemToReg y la nueva línea WriteByte.
4. Se agrego la operación 3 de la ALU que le dice que utilice el OR para ser mas ordenados.

2.2.1. Hazards de Datos

El único Hazard que se podría dar es el RAW (Read after Write) pero se salvo haciendo un nop entre el sbr y el sw, ya que era complicado modificar mas el data path para que utilice el Forward Unit sin cambiar la funcionalidad básica de las otras instrucciones.

2.3. bgezal

Instrucción a desarrollar

`1 | bgezal Rs, Imm`

Esta instrucción de tipo I compara el valor del registro Rs con cero, y si es mayor o igual salta a la posición $(PC + 4 + Imm * 4)$, almacenando el valor de PC +4 en el registro ra.

En este caso es mucho mas simple y queda mas prolijo (al tener que hacer un “and link”) hacerlo modificando el Data Path que pensando una pseudo-instrucción equivalente.

La modificación al **Data Path** fue:

1. Agregar 2 multiplexores entre EX/MEM y MEM/WB para elegir si guardar un valor en un registro o si guardar el PC+4 en el ra
2. Para ambos multiplexores se agrego en los Registros Pipeline ID/EX y EX/MEM el item JAL, que fue agregado desde la unidad de control, si se utiliza el bgezal estara en 1 y sino sera 0 siguiendo el camino que usaba antes para las demas instrucciones.
3. El primer multiplexor toma (en caso de que el JAL este activo) el valor del PC+4 que esta almacenado en el Registro Pipeline EX/MEM en RA, para ello también se agrego un Bifurcador entre el Registro Pipeline ID/EX y el ADD que calcula el target del branch para almacenarlo. Ese valor se lo pasa al registro pipeline MEM/WB así guarda en el ra dicha dirección.
4. El segundo multiplexor toma (en caso del que el JAL este activo) una constante con valor 31 (número de registro para el ra) y lo guarda en el registro pipeline MEM/WB para guardar el PC+4 allí.
5. Ordenamos los ítems y los cables entre los mismos para que el DP se vea bien.

En cuanto al **set de instrucciones** lo que se hizo fue:

1. Agregar la instrucción propiamente dicha, con código de operación 5 (utilizamos un código cercano al beq que no este utilizado en esta arquitectura).
2. Agregar a la unidad de control que dicha operación utiliza las mismas líneas de control que el beq agregando línea de control “JAL” en 1, también se cambio el ALUOp para que llame a una nueva operación de control de la ALU ($ALUOp = 3$).
3. En el control de la ALU cuando el ALUOp es 3 (código que creamos nosotros) la ALU hara la operación 7 (slt) dejando la línea de control “Zero” en 1 cuando el Rs sea mayor o igual a cero y en 0 cuando Rs sea menor que cero.

2.3.1. Hazards de Control

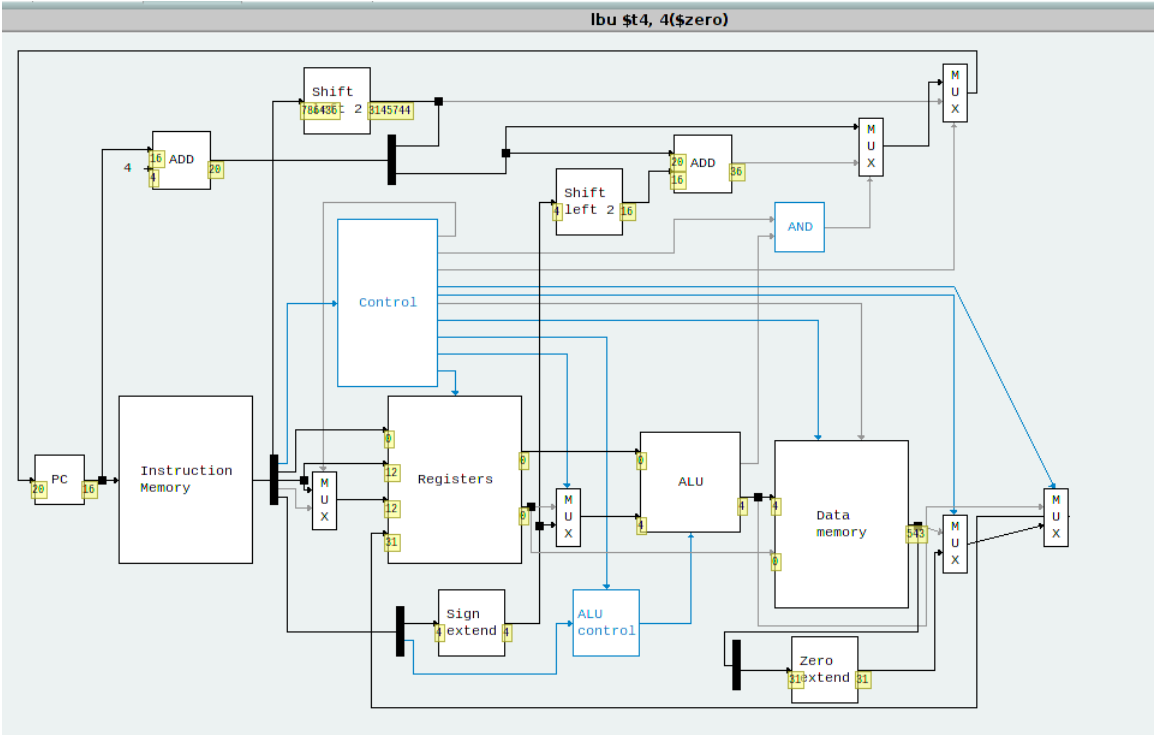
El Hazard de control se soluciona con un flush que hace la arquitectura cuando finaliza la etapa de memoria por lo que las instrucciones que empezaron a buscarse, identificarse y ejecutarse se pierden sin modificar ningún registro o la memoria.

2.3.2. Hazards de Datos

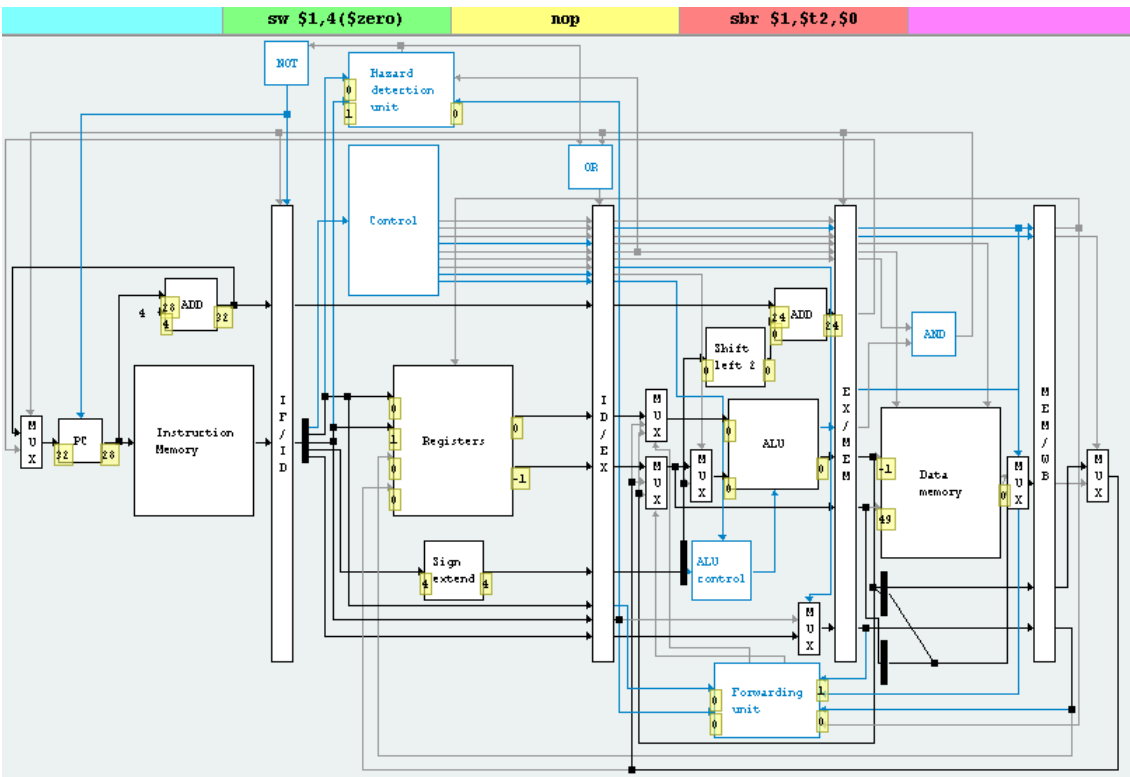
El único Hazard que se podría dar es el RAW (Read after Write) pero dado que la instrucción se copió del beq el mismo se soluciona con un Forwarding que lo maneja la unidad de forwarding.

3. DataPaths Modificados

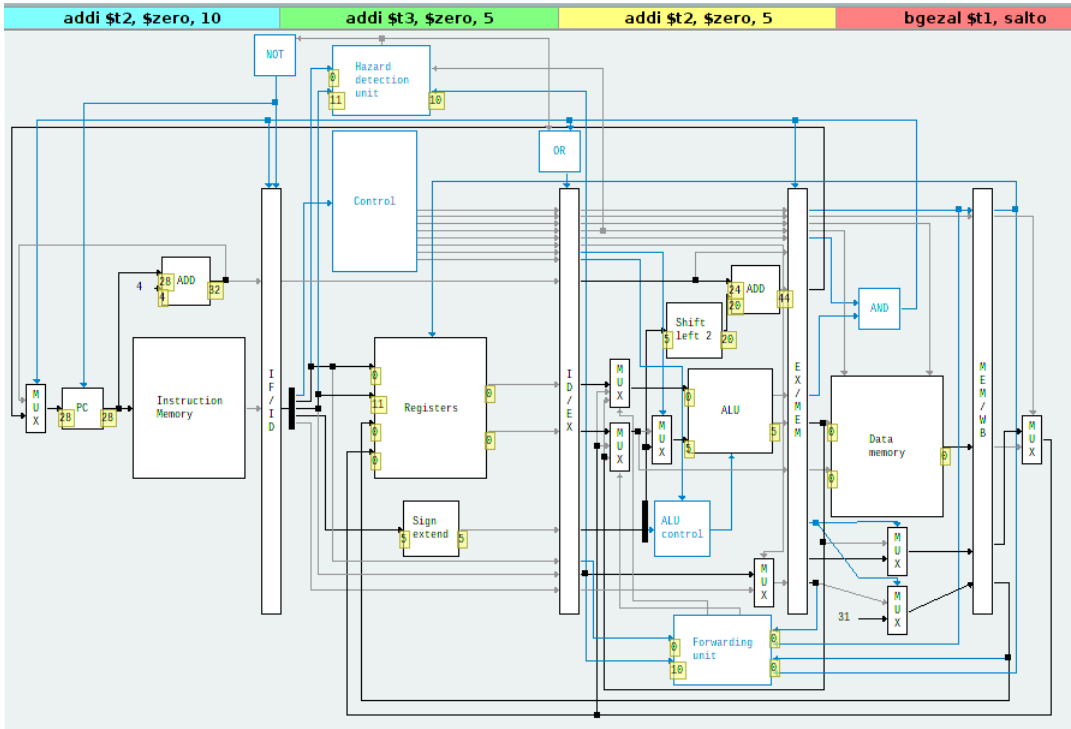
3.1. Monociclo Ibu



3.2. Pipeline sb



3.3. Pipeline bgezal



4. Programas de Prueba

4.1. lbu

```
1 | li    $t1, -5 #carga un número de mas de 1 byte (FFFFFFFB)
2 | sw    $t1, 4($zero) # guardo el número para poder cargarlo
3 | lbu   $t4, 4($zero) # cargo el primer byte del número (11111011 = FB = 251)
4 | sw    $t4, 12($zero) # guardo la palabra que tendra solo un byte con el número (FB = 251)
```

4.2. sb

```
1 | li    $t1, -1 #Cargo un word con todos los bits en 1
2 | sw    $t1, 4($zero) #Lo guardo en la memoria 4($zero) = FFFFFFFF
3 | li    $t2, 0x31 #Cargo un word pero con solo un byte de datos
4 | sb    $t2, 4($zero) #Guardo el byte cargado en t2 en la memoria 4($zero) = FFFFFFF31
```

4.3. bgezal

```
1 | li    $t1, 100 #carga algo mayor o igual a zero
2 | bgezal $t1, salto #hago el branch, que debería tomarse y dejar en ra 8
3 | addi   $t2, $zero, 5 #instrucción en PC = 8, va a hacer flush de esto
4 | addi   $t3, $zero, 5 #instrucción en PC = 12, idem
5 | salto:
6 | addi   $t2, $zero, 10 #lugar a donde va a saltar PC = 16
7 | sw     $ra, 4($zero) #guardo ra para ver que valor tomo. (ra = 8)
```

5. Conclusiones

- Vale la pena modificar el DataPath cuando es un ciclo y la pseudo-instrucción llevaría mas de una instrucción ya que en el unicycle se ejecuta de una instrucción a la vez y todas tienen el mismo tiempo que es un ciclo de reloj que dura cuanto dure la instrucción mas larga.
- En el Pipeline el riesgo de datos para el bgezal se salvo con Forwarding gracias a que las instrucciones básicas ya tenían implementado dicho mecanismo y que el DrMIPS acepta la unidad de Forwarding.
- El sb trajo tanta complejidad que la catedra pidió que se haga con una mezcla de una pseudoinstrucción y una instrucción nueva interna, esto hizo que tengamos que volver a pensar la instrucción, dado que lo que pensamos para la primera entrega no servía y volver a modificar el datapath, también trajo un riesgo de datos que tuvimos que salvar con una instrucción nop al no saber manejar correctamente la unidad de Forwarding teniendo que perder un ciclo del pipeline.
- Editar el JSON del DrMIPS es mas complicado de lo que se pensaba, primero hay que bosquejar lo que se quiere agregar en una hoja a mano para poder después pasarlo al programa y no olvidarse de ningún ítem tanto elementos, cables o atributos. De todas formas ayudo el editor de texto a cometer menos errores.
- Ordenar los cables y los elementos del procesador fue mas complicado todavía, si el DrMIPS tuviese una interfaz gráfica seria de gran utilidad para esa parte.

Referencias

- [1] DrMIPS, <https://brunonova.github.io/drmips>.
- [2] “Computer organization and design: the hardware-software interface”, John Hennessy, David Patterson. Capítulo 5.
- [3] ECMA-404 The JSON Data Interchange Standard, <http://www.json.org/>.