Pipeline-Project

Jose Eduardo Huamani Ñaupas
Jose Francisco Paca Sotero
Sergio Sebastian Lezama Orihuela
Héctor Sebastian Nieto Paz

November 2024

1 Introduction

El pipeline permite que diferentes etapas de múltiples instrucciones se ejecuten simultáneamente. Esto implica dividir las instrucciones en varias etapas, como búsqueda de instrucciones (IF), decodificación (ID), ejecución (EX), acceso a memoria (MEM) y escritura de resultados (WB). La principal ventaja de un pipeline es que permite mejorar el rendimiento al aumentar la cantidad de instrucciones que se procesan por ciclo de reloj.

En este proyecto, hemos realizado modificaciones primero a un single cycle para poder probar las instrucciones y luego se pasaron a pipeline para mejorar su eficiencia y soportar casos como hazards, dependencias entre instrucciones y predicción de saltos, entre otros.

2 Instrucciones Soportadas

- **Operaciones Aritmeticas**
  - `add`, `qadd`
  - `sub`, `qsub`
  - `mul`, `mla`
- **Operaciones Logicas**
  - `and`, `bic`, `orr`, `eor`
  - `cmp`, `cmn`, `tst`, `teq`
- **Mov y Shift**
  - `mov`
  - `lsr`, `asr`, `lsl`, `ror`

- **Load y Store**
  - Offset, Pre-offset, Post-offset, y Literales variants for `ldr` and `str`
- **Branch Operations**
  - `b`
- **Flags**
  - Flags: Negative (`N`), Zero (`Z`), Carry (`C`), Overflow (`V`), Saturated (`Q`)

3 Modificaciones del Pipeline

En este proyecto, hemos modificado significativamente el datapath del pipeline, especialmente la fase de **Execute** y **Fetch**, el **register file** y el modulo de control para soportar las varias instrucciones. En las siguientes secciones explicaremos cada cambio realizado, empezando desde el *register file*, después el datapath y finalmente el control.

faltan imagenes.

3.1 Modificaciones al Register File

Inicialmente, la implementación de un procesador pipeline ilustrada en el libro ***Digital Design and Computer Architecture*** contiene un **register file** con 2 puertos de lectura y 1 puerto de escritura. Para soportar instrucciones que utilizan 3 operadores a la ves, como **MLA**, o modificar registros durante operaciones de preindex y postindex, notamos que la implementación original es limitante. Aun así, es posible aun utilizar dicha implementación agregando fases adicionales al pipeline, las cuales se encarguen de realizar estas operaciones. Sin embargo, con fines de mantener un pipeline de 5 fases, modificamos el register file.

Añadimos 2 nuevas puerta al register file, las cuales nos permitirá leer un tercer registro, RA3 y RD3. Esto sera útil tanto para instrucciones como el **MLA**, al igual para los shifteos usando registros en operandos de otra instrucción.

Ademas, añadimos otras 2 nuevas puerta adicional para escribir en nuevo registro, WA1 Y WD1. Esto nos permitirá realizar operaciones donde necesitemos escribir a 2 registros, como un load que realiza preindex o postindex. Ahora que tenemos una nueva compuerta de escritura, necesitamos una señal adicional que indique cuando debamos escribir en este nuevo puerto, la cual llamaremos WriteBack.

Con estos nuevos cambios, nuestro register file tiene la capacidad de soportar muchas mas operaciones que la implementación original. Sin embargo, con estos cambios vienen nuevos hazards que resolveremos mas adelante.

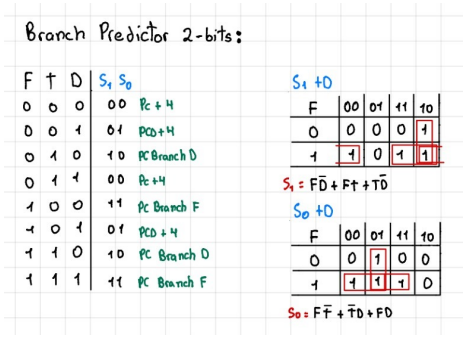

Figure 1: Register File Modificado

3.2 Modificaciones del Datapath

Iniciamos en el Fetch, donde nosotros obtenemos la instrucción y la pasamos a un registro para la siguiente fase. Mas adelante veremos la implementación del branch predictor en la fase de Fetch. Al pasar a la etapa de Decode, comenzamos a ver los cambios.

Hemos implementado nuevos **muxes** para cada uno de los puertos donde se recibe el valor del registro el cual queremos leer. Esto es principalmente debido a la particular manera en la cual las instrucciones de multiplicación están codificadas, ya que no tienen parecer en lo que respecta a registros con las instrucciones de data-proccessing o memoria. Observamos que para nuestro tercer registro, ademas de ser utilizado para el tercer operando de **MLA**, tambien es util para leer los registros los cuales son utilizados para las operaciones de **register-shifting**. La nueva señal que reciben estos nuevos muxs, es llamada MulOp, y provienen del modulo de control.

Similar a la implementación original, pasamos las señales provenientes del register file, tanto como el valor de los registros y el address de cada registro, ademas de nuevos valores como el **shamnt5** y otros hacia la fase de Execute. Ademas, pasamos las nuevas señales provenientes del control, como el MulOp.

Pasando a la etapa del Execute, los principales cambios fueron la agregación de nuevos muxes, principalmente para solucionar el hazard conocido que se obtiene tras una operación de read-after-write para nuestro nuevo operador el cual llamaremos *C*. Ademas, a todos los operadores (A,B,C) se les agrego un nuevo mux a cada uno, tras el anterior, debido a la creación de un nuevo hazard, pues ahora tenemos una puerta de escritura adicional, WA1. Debido a esto, se debe crear nuevas señales provenientes del hazard, similares a la implementación original, llamada ForwardIndex, debido a que este hazard ocurre cuando se realiza una operación de preindex o postindex, y uno de los operandos es el registro a modificar.

La parte central del datapath muestra cómo se genera el operando B desplazado. Primero se utiliza un multiplexor controlado por la señal RegShift para decidir si el desplazamiento (shift) es a través del valor de un registro o un

valor literal. Dependiendo de cual se habilite, se pasa un valor particular que representaría la cantidad de rotaciones o shifting que se hará, dependiendo de la operacion manejada por el ShiftControl. La salida del bloque de desplazamiento, ShiftedSrcB, va hacia un mux, con señal MulOp, que elige entre el ShiftedSrcB y WriteData, debido a que las operaciones de multiplicación no soportan shifting ni rotaciones. Tras eso sigue la implementación original, donde esta el mux que elige entre un registro o un inmendiate dependiendo del ALUSrc. Por ultimo, antes de entrar este segundo operando al ALU, hay un mux adicional, con señal saturated, el cual nuevamente elije entre el segundo operando actual o WriteData.

Por otro lado, también se realizan rotaciones para lo que es valores inmediatos. Podemos observar en la parte inferior se haya un mux, cuya señal es MemtoReg, el cual elige entre un valor de inmediato de 12 bits, si es una operación de memoria, o un inmediato con rotación, en caso no lo sea.

Una ves que se tiene el segundo Operando B, los operandos A,B,C entran al ALU y se genera el resultado, ya sea una operacion de preindex, add, o logica. Dependiendo de que operacion sea, este resultado ALUResult entrara a un mux en el cual se elegira entre el ALUResult o el SrcA, el cual es elegido con una señal proveniente del control PostIndex.

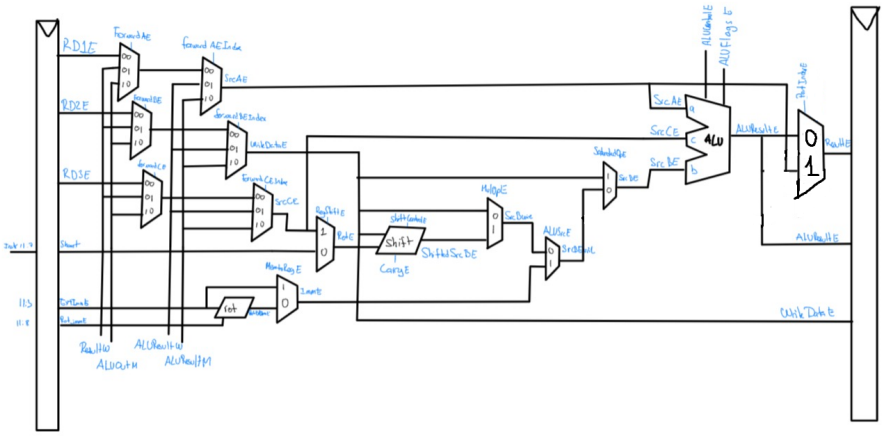

Figure 2: Execute datapath

4 Hazards

Los tipos de hazards que se deben manejar son los siguientes:

- **Data Hazards:** Ocurren cuando una instrucción depende de los resultados de una instrucción anterior.

- **Control Hazards:** Surgen cuando el flujo de control (por ejemplo, saltos) no se conoce con antelación.
- **Structural Hazards:** Se producen cuando el hardware no puede soportar la ejecución simultánea de varias instrucciones.

Las modificaciones al control del pipeline incluyen la inserción de unidades de control que gestionan estas situaciones, como la detección de hazards y la inserción de stalls para evitar que se tomen decisiones incorrectas debido a datos incorrectos.

4.1 Forwarding/Bypassing

El *forwarding* o *bypassing* es una técnica para resolver los hazards de datos sin tener que insertar un stall en el pipeline. Si un dato que se necesita está disponible en una de las etapas anteriores del pipeline, puede ser "re-enviado" (forwarded) a la etapa que lo necesite, sin esperar a que la instrucción anterior termine de ejecutarse.

4.2 Stall y Flush

Los *stalls* se insertan en el pipeline para evitar que las instrucciones accedan a datos incorrectos antes de que se resuelvan. Un *flush* es una operación que descarta las instrucciones en el pipeline cuando se toma una decisión incorrecta, como un salto condicional erróneo.

5 Branch Predictor con Branch Target Buffer (BTB)

El diseño propuesto del *Branch Predictor* incluye la incorporación de un *Branch Target Buffer (BTB)* en la etapa de *Fetch* para mejorar la predicción de las ramas. Este diseño aprovecha la resolución temprana del *Branch Taken* que se obtiene en la etapa de *Decode*, facilitando así una predicción más precisa del salto antes de la resolución final.

5.1 Funcionamiento del Branch Predictor

En este sistema, en la etapa de *Fetch*, el *BTB* recibe como entrada el *Program Counter (PC)* y los dos bits más significativos de la instrucción [27:26], que corresponden al *opcode* de la operación. Si la operación es de tipo *branch*, el *BTB* consulta su *array de registros*, que tiene una longitud de 33 bits por entrada. Los primeros 32 bits de cada registro almacenan la *dirección de destino* de la rama, mientras que el bit restante almacena un valor que indica si la rama fue tomada en la última ejecución.

El índice de este *array de registros* se obtiene a partir de los *últimos 7 bits* del *PC* de la rama, omitiendo los dos primeros bits que siempre son cero debido

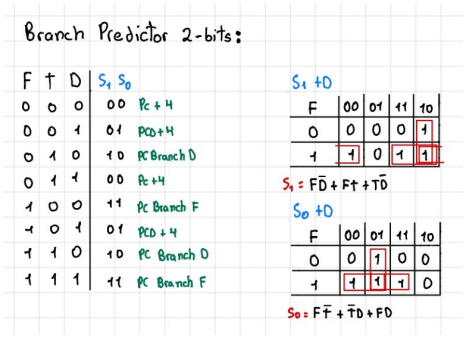

Figure 3: Mapas de Karnaugh para el Branch Mux

a que el *PC* debe ser múltiplo de 4. Así, el índice se corresponde con los bits [8:2] del *PC*.

Si la *instrucción de rama* no se encuentra en el *BTB*, el *Branch Predictor* predice que la rama no será tomada. Sin embargo, si la instrucción está presente en el *BTB*, se utiliza la información almacenada para predecir si la rama será tomada o no. Si en la etapa de *Decode* se determina que la rama fue efectivamente tomada, el *BTB* se actualiza con la nueva *dirección de destino* y se marca como *tomada*.

5.2 Manejo de Branch Missed y Hazards

Para manejar posibles *hazards*, se introduce la lógica de un *Branch Missed*. Este hazard se activa si la *dirección de destino de la rama* es diferente de *PC + 4*. Si el *Branch Target Address* coincide con el valor de *PC + 4*, el flujo de instrucciones continúa sin cambios. En caso contrario, se activa un *multiplexor de 4 entradas* en la etapa de *Fetch*, el cual seleccionará uno de los siguientes valores de *PC*:

- *PC + 4*, que es el valor del contador de programa estándar.
- *PC de la rama tomada + 4*, que es el valor del *PC* de la rama que fue tomada.
- La predicción del *Branch Predictor* sobre si la rama será tomada o no.
- El valor de *PC* como se determina en la etapa de *Decode*.

Esto asegura que, si la predicción de la rama realizada en la etapa de *Fetch* es incorrecta, el *PC* se ajustará en consecuencia en la etapa de *Decode*.

6 Waveforms y Código de Assembly

Las waveforms representan la actividad del procesador en las distintas etapas del pipeline durante la ejecución de un conjunto de instrucciones. A continuación

se presenta un ejemplo de un código en Assembly y su equivalente en código de máquina, utilizado para las pruebas del proyecto.

6.1 Código de Assembly

Aquí tienes el conjunto de instrucciones en Assembly utilizadas en las pruebas:

```
movs r0, #0    # Mueve el valor inmediato 0 al registro r0 y actualiza los flags
mov r1, #0     # Mueve el valor inmediato 0 al registro r1
mvns r2, #1    # Mueve el complemento bit a bit del inmediato 1 al registro r2
tst r0, r1     # Realiza AND lógico entre r0 y r1, y actualiza los flags
teq r0, r1     # Realiza XOR lógico entre r0 y r1, y actualiza los flags
cmp r2, r1     # Compara r2 y r1 (resta sin guardar el resultado) y actualiza los flags
cmn r0, r2     # Suma r0 y r2 (sin guardar el resultado) y actualiza los flags
```

6.2 Código de Máquina

El código de máquina equivalente es el siguiente:

```
E3B00000   # movs r0, #0
E3A01000   # mov r1, #0
E3F02001   # mvns r2, #1
E1100001   # tst r0, r1
E1300001   # teq r0, r1
E1520001   # cmp r2, r1
E1700002   # cmn r0, r2
```

6.3 Explicación de las Instrucciones

A continuación se explica el funcionamiento de cada instrucción:

- **movs r0, #0 (E3B00000):** Copia el valor inmediato 0 al registro r0 y actualiza los flags del procesador (N, Z, C, V).
- **mov r1, #0 (E3A01000):** Copia el valor inmediato 0 al registro r1. No afecta los flags.
- **mvns r2, #1 (E3F02001):** Copia el complemento bit a bit de 1 al registro r2 y actualiza los flags.
- **tst r0, r1 (E1100001):** Realiza una operación AND lógica entre r0 y r1, y actualiza los flags según el resultado. No almacena ningún valor.
- **teq r0, r1 (E1300001):** Realiza una operación XOR lógica entre r0 y r1, y actualiza los flags según el resultado. No almacena ningún valor.
- **cmp r2, r1 (E1520001):** Realiza una operación de resta entre r2 y r1, sin almacenar el resultado, pero actualiza los flags.

- **cmn r0, r2 (E1700002):** Realiza una operación de suma entre r0 y r2, sin almacenar el resultado, pero actualiza los flags.