Implementación de Memoria RAM en Nand2Tetris

Nucleo Digital

September 13, 2024

1 Introducción

En esta práctica de grupo, trabajamos en la implementación de varios chips necesarios para construir una memoria RAM en el simulador de Nand2Tetris. El objetivo fue crear diferentes módulos que, combinados, permiten la creación de memorias desde un simple bit hasta una memoria de 16K registros de 16 bits. En este documento, explicamos los principales resultados obtenidos, cómo se probó cada componente y, especialmente, cómo se utilizaron componentes clave como multiplexores (Mux) y demultiplexores (DMux) para controlar el flujo de datos en cada módulo de memoria.

2 Elementos de Entrega

2.1 Códigos Implementados

A continuación adjuntamos los códigos implementados para cada uno de los chips, junto con una explicación detallada de su funcionalidad y de los componentes utilizados.

2.1.1 Bit.hdl

El chip `Bit` es el componente más simple de almacenamiento. Almacena un valor binario (0 o 1), que puede mantenerse o actualizarse dependiendo de la señal de carga (`load`). Un multiplexor (`Mux`) se encarga de seleccionar si el valor almacenado debe mantenerse (cuando `load` es 0) o si debe reemplazarse por el nuevo valor de entrada (`in`). Además, un flip-flop de retardo (`DFF`) es utilizado para mantener el valor entre ciclos de reloj.

```
/**
 * Bit:
 * Si load=1, almacena el valor de in.
 * Si load=0, mantiene el valor almacenado.
 */
CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
```

```
    DFF(in=dffIn, out=dffOut);
    Mux(a=dffOut, b=in, sel=load, out=dffIn);
    DFF(in=dffIn, out=out);
}
```

2.1.2 PC.hdl

El Program Counter (`PC`) es un contador programable de 16 bits que permite tres operaciones: incremento del valor actual, carga de un nuevo valor y reseteo. Los multiplexores (`Mux16`) se encargan de controlar el flujo de datos entre estas operaciones. Por ejemplo, uno de los multiplexores selecciona entre incrementar el valor actual o cargar uno nuevo, y otro permite decidir si el valor debe resetearse. La suma para el incremento se realiza con el chip `Inc16`, que suma 1 al valor actual.

```
/**
 * PC:
 * - Incrementa (inc=1)
 * - Carga un valor (load=1)
 * - Reset (reset=1)
 */
CHIP PC {
    IN in[16], load, inc, reset;
    OUT out[16];

    PARTS:
    Inc16(in=out5, out=out1);
    Mux16(a=out5, b=out1, sel=inc, out=out2);
    Mux16(a=out2, b=in, sel=load, out=out3);
    Mux16(a=out3, b=false, sel=reset, out=out4);
    Register(in=out4, load=true, out=out5, out=out);
}
```

2.1.3 Register.hdl

El chip `Register` permite almacenar un valor de 16 bits, y está compuesto por 16 instancias del chip `Bit`. Cada uno de los bits del registro es gestionado de forma individual, permitiendo que el registro completo almacene un valor de 16 bits y lo mantenga o lo actualice dependiendo de la señal de carga (`load`).

```
/**
 * Register:
 * Registro de 16 bits. Usa 16 bits para almacenar valores.
 */
CHIP Register {
    IN in[16], load;
    OUT out[16];

    PARTS:
    Bit(in=in[0], load=load, out=out[0]);
```

```
    Bit(in=in[1], load=load, out=out[1]);
    Bit(in=in[2], load=load, out=out[2]);
    // Resto de los bits...
}
```

2.1.4 RAM8.hdl

El chip `RAM8` implementa una memoria de 8 registros de 16 bits. Aquí, un `DMux8Way` se encarga de distribuir la señal de carga a uno de los 8 registros, en función de la dirección proporcionada. Cada registro puede actualizarse o mantener su valor dependiendo de la señal de carga recibida. Por su parte, un `Mux8Way16` selecciona la salida del registro correcto, de acuerdo a la dirección dada.

```
/**
 * RAM8:
 * Memoria de 8 registros de 16 bits.
 */
CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];

    PARTS:
    DMux8Way(in=load, sel=address, a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);
    Register(in=in, load=load0, out=out0);
    Register(in=in, load=load1, out=out1);
    // Resto de los registros...
    Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address, out=out);
}
```

2.1.5 RAM64.hdl

El chip `RAM64` amplía la capacidad del chip `RAM8` para manejar 64 registros de 16 bits. Para lograr esto, utilizamos 8 bloques de `RAM8`, controlados por un `DMux8Way` que distribuye la señal de carga a los bloques correspondientes, y un `Mux8Way16` que selecciona la salida de uno de los bloques en función de la dirección de 6 bits.

```
/**
 * RAM64:
 * Memoria de 64 registros de 16 bits.
 */
CHIP RAM64 {
    IN in[16], load, address[6];
    OUT out[16];

    PARTS:
    DMux8Way(in=load, sel=address[3..5], a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);
    RAM8(in=in, load=load0, address=address[0..2], out=out0);
    RAM8(in=in, load=load1, address=address[0..2], out=out1);
```

```
    // Resto de los bloques RAM8...
    Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address[3..5], out=out);
}
```

2.1.6 RAM512.hdl

El chip `RAM512` extiende la memoria para manejar 512 registros de 16 bits, utilizando 8 bloques de `RAM64`. Similar a los chips anteriores, el `DMux8Way` se encarga de distribuir la señal de carga a los diferentes bloques de `RAM64`, mientras que el `Mux8Way16` selecciona la salida correcta en función de la dirección proporcionada.

```
/**
 * RAM512:
 * Memoria de 512 registros de 16 bits.
 */
CHIP RAM512 {
    IN in[16], load, address[9];
    OUT out[16];

    PARTS:
    DMux8Way(in=load, sel=address[6..8], a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);
    RAM64(in=in, load=load0, address=address[0..5], out=out0);
    RAM64(in=in, load=load1, address=address[0..5], out=out1);
    // Resto de los bloques RAM64...
    Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address[6..8], out=out);
}
```

2.1.7 RAM4K.hdl

El chip `RAM4K` sigue la misma estructura que `RAM512`, pero amplía la capacidad de memoria a 4096 registros de 16 bits. Utiliza bloques de `RAM512` y los componentes `DMux8Way` y `Mux8Way16` para distribuir la señal de carga y seleccionar la salida correspondiente, respectivamente.

```
/**
 * RAM4K:
 * Memoria de 4K registros de 16 bits.
 */
CHIP RAM4K {
    IN in[16], load, address[12];
    OUT out[16];

    PARTS:
    DMux8Way(in=load, sel=address[9..11], a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);
    RAM512(in=in, load=load0, address=address[0..8], out=out0);
    RAM512(in=in, load=load1, address=address[0..8], out=out1);
    // Resto de los bloques RAM512...
    Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address[9..11], out=out);
}
```

2.1.8 RAM16K.hdl

Finalmente, `RAM16K` extiende la memoria a 16,384 registros de 16 bits, utilizando bloques de `RAM4K`. Utiliza un `DMux4Way` para distribuir la señal de carga a los 4 bloques de `RAM4K`, y un `Mux4Way16` para seleccionar la salida correspondiente.

```
/**
 * RAM16K:
 * Memoria de 16K registros de 16 bits.
 */
CHIP RAM16K {
    IN in[16], load, address[14];
    OUT out[16];

    PARTS:
    DMux4Way(in=load, sel=address[12..13], a=load0, b=load1, c=load2, d=load3);
    RAM4K(in=in, load=load0, address=address[0..11], out=out0);
    RAM4K(in=in, load=load1, address=address[0..11], out=out1);
    RAM4K(in=in, load=load2, address=address[0..11], out=out2);
    RAM4K(in=in, load=load3, address=address[0..11], out=out3);
    Mux4Way16(a=out0, b=out1, c=out2, d=out3, sel=address[12..13], out=out);
}
```

2.2 Resultados

Todos los chips mencionados fueron implementados y probados usando los archivos de prueba (`.tst`) y comparación (`.cmp`) proporcionados por Nand2Tetris. Las pruebas confirmaron que los chips funcionan correctamente y no se encontraron diferencias en la *diff table*, lo cual indica que el comportamiento es el esperado según las especificaciones.

2.3 Metodología

Cada componente se probó individualmente para asegurarse de que cumpliera con su funcionalidad. Comenzamos con los chips más simples, como `Bit` y `Register`, y luego avanzamos hacia componentes más complejos como `RAM8`, `RAM64`, y finalmente `RAM16K`. La modularidad en el diseño permitió que la capacidad de la memoria se ampliara de forma progresiva y consistente.