Segundo laboratorio GRUPO MACACO

Estudiantes:
Luis Mario Toscano Palomino.
Jhon Anderson Vargas Gómez.
Jeider Torres Martínez.
Samuel David Traslaviña Mateus.
Luis Alejandro Vecino Daza.

Profesor:
Carlos Jaime Barrios Hernández

Grupo:
A1

Arquitectura de computadores
Universidad Industrial de Santander
12 de septiembre de 2024

Tabla de contenido

1er punto

- ¿Cuál es el objetivo de cada uno de esos proyectos con sus palabras y describa qué debe hacer para desarrollarlo?

Desarrollo:

2do Proyecto: Construcción de la ALU del Hack Computer

El objetivo de este proyecto es armar todos los chips que se describen en el Capítulo 2 del curso, hasta llegar a la Unidad Lógica Aritmética (ALU) de la computadora Hack. La ALU es un componente súper importante porque se encarga de todas las operaciones aritméticas y lógicas de la computadora. Lo interesante es que para construirla solo podemos usar los chips que ya vimos en el Capítulo 1 y los que vayamos armando durante este proyecto. O sea, es como ir construyendo un rompecabezas donde cada pieza depende de la anterior, y al final tienes algo tan útil como la ALU, que es el cerebro de las operaciones.

Pasos para desarrollarlo:

1. **Revisar los componentes básicos del capítulo 1**: Antes de empezar a construir la ALU, es crucial que entiendas cómo funcionan los chips básicos que ya desarrollaste, como los multiplexores (MUX), demultiplexores (DMUX), compuertas lógicas (AND, OR, NOT), y otros circuitos fundamentales. Estos serán las bases que utilizarás para crear componentes más complejos.

2. **Construir los chips intermedios**: El siguiente paso es empezar a construir chips intermedios como el **Half Adder** y el **Full Adder**. Estos chips te permitirán realizar operaciones aritméticas, que son esenciales para la ALU. Debes enfocarte en entender cómo se suman bits binarios y cómo el acarreo (carry) se propaga en las operaciones.

3. **Diseñar el ALU**: Una vez que tengas listos los chips intermedios, comienza a diseñar la ALU. Aquí es donde todo lo que has aprendido hasta ahora entra en juego. La ALU debe ser capaz de realizar operaciones aritméticas como suma y resta, así como operaciones lógicas como AND, OR, y NOT. Asegúrate de que el diseño permita seleccionar qué operación se debe realizar, utilizando un conjunto de bits de control.

4. **Prueba y depura de la ALU**: Una vez que tengas el diseño completo, es esencial que lo pruebes. Utiliza simulaciones para asegurarte de que la ALU realice correctamente cada operación (suma, resta, AND, OR, etc.). Si algo falla, revisa paso a paso el diseño y corrige cualquier error en los chips que hayas construido.

3er Proyecto: Construcción de una Unidad de Memoria RAM

El objetivo de este proyecto es armar todos los chips que se describen en el **Capítulo 3**, hasta llegar a una **Unidad de Memoria de Acceso Aleatorio (RAM)**. La RAM es fundamental porque es donde el computador guarda datos de manera temporal mientras está funcionando. La diferencia en este proyecto es que partimos de un componente más básico, el **DFF** (Flip-Flop), que básicamente es un circuito de memoria primitivo. Con eso y los chips que ya construimos en los capítulos anteriores, vamos poco a poco armando la RAM. Lo interesante es que, así como con la ALU, aquí estamos construyendo todo desde cero, pero ahora enfocado en cómo se almacenan y se manejan los datos.

Pasos para desarrollarlo:

1. **Familiarización con el DFF (Flip-Flop)**: Para este proyecto, el punto de partida es el **D Flip-Flop (DFF)**. Este es un chip básico que almacena un bit de información y será el bloque fundamental para construir la RAM. Revisa cómo funciona, entendiendo cómo almacena y actualiza el valor de un bit.

2. **Construcción de registros básicos**: Usando los DFFs, tu siguiente paso es construir registros, que son simplemente grupos de flip-flops que almacenan varios bits. Por ejemplo, un registro de 8 bits tendría 8 DFFs conectados entre sí. Estos registros te permitirán almacenar datos en memoria.

3. **Diseñar la memoria RAM de manera jerárquica**:
    - **Memoria pequeña (RAM8)**: Empieza construyendo una pequeña unidad de memoria, como una **RAM8**, que es capaz de almacenar 8 palabras de datos. Para esto, necesitarás usar múltiples registros y un sistema de control que te permita seleccionar qué registro leer o escribir.
    - **Aumenta la capacidad**: Una vez que tengas la RAM8 funcionando, puedes aumentar la capacidad de la memoria construyendo unidades más grandes como **RAM64**, **RAM512**, y eventualmente una **RAM16K**. Esto se logra organizando varias unidades más pequeñas en un sistema jerárquico donde puedes seleccionar el bloque correcto de memoria y el registro dentro de ese bloque.

4. **Crear un circuito de control**: La memoria RAM necesita un mecanismo que permita seleccionar en qué dirección (o ubicación) almacenar o leer datos. Esto implica usar multiplexores y demultiplexores que seleccionen el registro adecuado dentro de la memoria.

5. **Prueba y verifica**: Igual que con la ALU, debes probar cada etapa de tu memoria RAM usando simulaciones. Asegúrate de que puedes leer y escribir correctamente en cualquier ubicación de la memoria.

2do punto

- Explique las principales diferencias entre la lógica aritmética y la lógica secuencial.

Desarrollo:

Lógica Aritmética (Combinacional):

La lógica aritmética también se conoce como lógica combinacional, y básicamente se ocupa de operaciones que se realizan sin tener memoria. En otras palabras, el resultado de una operación en un circuito de lógica aritmética solo depende de las entradas actuales, no de lo que pasó antes.

Ejemplos comunes de lógica aritmética incluyen operaciones como sumar, restar, multiplicar o comparar números. Piensa en una calculadora básica: cuando introduces dos números y presionas el botón para sumar, el resultado solo depende de esos dos números. La calculadora no recuerda qué números sumaste antes; solo hace la operación con lo que le diste en ese momento.

Resumen de la lógica aritmética:
- Sin memoria: No recuerda resultados anteriores.
- Depende solo de las entradas actuales.
- Ejemplos: sumadores, restadores, comparadores.

Lógica Secuencial:

La lógica secuencial es diferente porque sí utiliza memoria. Esto significa que el resultado de las operaciones depende no solo de las entradas actuales, sino también de lo que ocurrió anteriormente. Los circuitos de lógica secuencial pueden "recordar" lo que pasó antes, por lo que su salida depende tanto de las entradas actuales como del estado previo del sistema.

Un ejemplo sencillo de lógica secuencial es un reloj digital. El reloj recuerda la hora actual y actualiza la hora con el tiempo. No solo depende de lo que está sucediendo ahora, sino que también toma en cuenta la hora que tenía previamente.

Resumen de la lógica secuencial:
- Con memoria: Recuerda los estados previos.
- Depende de las entradas actuales y del estado anterior.
- Ejemplos: contadores, registros, memorias.

Diferencia Clave:

La diferencia principal es que la lógica aritmética no tiene memoria, solo se fija en las entradas actuales para dar una salida, mientras que la lógica secuencial sí tiene memoria y puede recordar lo que pasó antes, usando esa información para decidir la salida actual.

Espero que esta explicación haya sido clara. Si te imaginas la lógica aritmética como una calculadora que solo resuelve problemas matemáticos en el momento y la lógica secuencial como un reloj que "recuerda" la hora y la actualiza, puedes entender la diferencia fundamental entre estos dos tipos de lógica.

Bonus

- ¿Qué tipo de unidades aritmeticológicas existen?

Desarrollo:

Las Unidades Aritmético-Lógicas (ALU) son componentes clave en los procesadores y otros sistemas digitales, ya que se encargan de realizar operaciones aritméticas y lógicas. Existen varios tipos de ALUs, que se pueden clasificar en función de sus características, capacidades y aplicaciones. A continuación, te explico los principales tipos de ALUs que existen:

ALU Simple o Básica

Esta es la forma más sencilla de una ALU. Está diseñada para realizar solo las operaciones aritméticas y lógicas básicas. Por lo general, las operaciones que soporta incluyen:

- Aritmética: suma, resta.
- Lógica: AND, OR, NOT, XOR.

Se utilizan principalmente en procesadores de baja potencia o en sistemas embebidos donde no se necesitan operaciones complejas.

ALU Compleja o Avanzada

Este tipo de ALU extiende las capacidades de una ALU simple y es capaz de realizar operaciones aritméticas y lógicas más complejas. Además de las operaciones básicas, estas ALUs pueden realizar:

- Multiplicación y división.
- Operaciones de desplazamiento (shift) y rotación de bits.
- Comparaciones de magnitud (mayor que, menor que, igual a).

Estas ALUs son utilizadas en procesadores de propósito general como los que se encuentran en los PCs y servidores.

ALU de Punto Fijo (Fixed-Point ALU)

Este tipo de ALU está diseñado para trabajar con números enteros y representaciones de punto fijo. En estas ALUs, los números tienen una posición fija para el punto decimal, lo que permite realizar cálculos aritméticos con números enteros de manera eficiente.

Se usan en aplicaciones donde no es necesario trabajar con números fraccionarios o cuando la precisión no es crítica, como en muchos microcontroladores.

ALU de Punto Flotante (Floating-Point ALU o FPU)

La ALU de punto flotante, también conocida como Unidad de Punto Flotante (FPU), está diseñada para trabajar con números en formato de punto flotante, lo que permite realizar cálculos con números fraccionarios o muy grandes/muy pequeños. Estas ALUs son fundamentales en aplicaciones que requieren alta precisión numérica, como gráficos por computadora, simulaciones científicas y aplicaciones de procesamiento de señales.

Las operaciones que soportan incluyen:

- Suma y resta de punto flotante.
- Multiplicación y división de punto flotante.
- Operaciones especiales como raíz cuadrada y exponenciales.

ALU Vectorial o SIMD (Single Instruction, Multiple Data)

Esta ALU es capaz de realizar la misma operación aritmética o lógica en varios datos simultáneamente. Esto es particularmente útil en aplicaciones que requieren procesamiento paralelo, como el procesamiento de gráficos, multimedia y procesamiento de señales.

Este tipo de ALUs se encuentran en unidades de procesamiento gráfico (GPUs) y en algunas CPUs modernas que incluyen instrucciones SIMD para acelerar tareas como la manipulación de imágenes o el procesamiento de grandes volúmenes de datos.

ALU Escalar

Una ALU escalar opera sobre un único par de operandos en cada ciclo de reloj. Este es el tipo más tradicional de ALU que se encuentra en muchos procesadores clásicos. A diferencia de una ALU vectorial, no puede manejar múltiples datos al mismo tiempo.

ALU Especializada

Estas ALUs están diseñadas para un propósito específico y optimizadas para un tipo particular de operación. Por ejemplo, en algunas arquitecturas de procesadores gráficos o de inteligencia artificial, las ALUs pueden estar diseñadas para operaciones especializadas como la multiplicación de matrices o convoluciones en redes neuronales.

ALU Configurable o Programable

Una ALU configurable permite cierta flexibilidad en las operaciones que puede realizar. Este tipo de ALU puede ser reconfigurado para realizar diferentes tipos de operaciones aritméticas o lógicas según las necesidades del programa. Se utilizan en procesadores de propósito general avanzados y en arquitecturas reconfigurables como los FPGA (Field Programmable Gate Arrays).

Anexos

En esta sección de anexos, se presentan los códigos y explicaciones detalladas correspondientes a los proyectos desarrollados como parte del curso, siguiendo los lineamientos descritos en los capítulos 2 y 3 del libro guía. Los proyectos incluyen la construcción de componentes fundamentales de una computadora, específicamente:

- Primer proyecto: Diseño y construcción de todos los chips necesarios para implementar la Unidad Lógica Aritmética (ALU) del Hack Computer, utilizando únicamente los componentes previamente desarrollados en el capítulo 1 y los que se fueron construyendo a lo largo de este proyecto.

- Segundo proyecto: Desarrollo de los chips necesarios para la implementación de una Unidad de Memoria RAM, haciendo uso de compuertas primitivas (DFF) y los chips creados en los capítulos anteriores.

A continuación, se detalla el código y la lógica detrás de cada uno de los componentes creados para ambos proyectos, mostrando paso a paso el proceso de construcción y las decisiones técnicas tomadas. Este anexo servirá como un recurso de referencia para comprender mejor la estructura interna de una ALU y una unidad de memoria RAM, así como su funcionamiento en el contexto de la arquitectura del Hack Computer.

“Incluir imágenes de los hdl y algún tipo de descripción o explicación”

Project 2.

Chip HalfAdder:

```
CHIP HalfAdder {
    IN a, b;    // 1-bit inputs
    OUT sum,    // Right bit of a + b
        carry;  // Left bit of a + b

    PARTS:
    Xor(a=a,b=b,out=sum);
    And(a=a,b=b,out=carry);
}
```

El HalfAdder es un circuito combinacional básico que suma dos bits de entrada, generando dos salidas: la suma (sum) y el acarreo (carry). La salida sum es el resultado de una operación XOR entre los dos bits de entrada, representando la suma de estos sin considerar ningún acarreo. Por otro lado, la salida carry es el resultado de una operación AND entre los bits de entrada, indicando si hay un acarreo hacia el siguiente bit de mayor orden.

Chip FullAdder:

```
CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c

    PARTS:

    Xor(a=a,b=b,out=d);
    Xor(a=c,b=d,out=sum);

    And(a=a,b=b,out=ab);
    And(a=d,b=c,out=abc);
    Or(a=abc,b=ab,out=carry);
}
```

El FullAdder amplía la funcionalidad del HalfAdder al sumar tres bits de entrada (dos bits más un bit de acarreo) y producir dos salidas: suma (sum) y acarreo (carry). La salida sum se obtiene sumando los tres bits de entrada mediante dos HalfAdders en cascada, mientras que la salida carry indica si hay un acarreo hacia la siguiente posición de bit, combinando los acarreo intermedios generados por los HalfAdders.

Chip Add16:

```
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    HalfAdder(a = a[0], b = b[0], sum = out[0], carry= carry1);
    FullAdder(a = a[1], b = b[1], c = carry1, sum = out[1], carry = carry2);
    FullAdder(a = a[2], b = b[2], c = carry2, sum = out[2], carry = carry3);
    FullAdder(a = a[3], b = b[3], c = carry3, sum = out[3], carry = carry4);
    FullAdder(a = a[4], b = b[4], c = carry4, sum = out[4], carry = carry5);
    FullAdder(a = a[5], b = b[5], c = carry5, sum = out[5], carry = carry6);
    FullAdder(a = a[6], b = b[6], c = carry6, sum = out[6], carry = carry7);
    FullAdder(a = a[7], b = b[7], c = carry7, sum = out[7], carry = carry8);
    FullAdder(a = a[8], b = b[8], c = carry8, sum = out[8], carry = carry9);
    FullAdder(a = a[9], b = b[9], c = carry9, sum = out[9], carry = carry10);
    FullAdder(a = a[10], b = b[10], c = carry10, sum = out[10], carry = carry11);
    FullAdder(a = a[11], b = b[11], c = carry11, sum = out[11], carry = carry12);
    FullAdder(a = a[12], b = b[12], c = carry12, sum = out[12], carry = carry13);
    FullAdder(a = a[13], b = b[13], c = carry13, sum = out[13], carry = carry14);
    FullAdder(a = a[14], b = b[14], c = carry14, sum = out[14], carry = carry15);
    FullAdder(a = a[15], b = b[15], c = carry15, sum = out[15], carry = carry16);
}
```

El Add16 es un sumador de 16 bits que suma dos números de 16 bits. La salida (out) es el resultado de sumar cada par de bits de las entradas, bit a bit, utilizando FullAdders para manejar los acarreos entre las posiciones de bit, asegurando así la correcta propagación del acarreo a través de las 16 posiciones.

Chip Inc16:

```
CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
    //Mira si la posicion 0 es (1 o 0) y con eso determina el carry
    Not( in = in[0], out = out[0] );

    HalfAdder( a = in[1], b = in[0], sum = out[1], carry = carry1 );
    HalfAdder( a = in[2], b = carry1, sum = out[2], carry = carry2 );
    HalfAdder( a = in[3], b = carry2, sum = out[3], carry = carry3 );
    HalfAdder( a = in[4], b = carry3, sum = out[4], carry = carry4 );
    HalfAdder( a = in[5], b = carry4, sum = out[5], carry = carry5 );
    HalfAdder( a = in[6], b = carry5, sum = out[6], carry = carry6 );
    HalfAdder( a = in[7], b = carry6, sum = out[7], carry = carry7 );
    HalfAdder( a = in[8], b = carry7, sum = out[8], carry = carry8 );
    HalfAdder( a = in[9], b = carry8, sum = out[9], carry = carry9 );
    HalfAdder( a = in[10], b = carry9, sum = out[10], carry = carry10 );
    HalfAdder( a = in[11], b = carry10, sum = out[11], carry = carry11 );
    HalfAdder( a = in[12], b = carry11, sum = out[12], carry = carry12 );
    HalfAdder( a = in[13], b = carry12, sum = out[13], carry = carry13 );
    HalfAdder( a = in[14], b = carry13, sum = out[14], carry = carry14 );
    HalfAdder( a = in[15], b = carry14, sum = out[15], carry = carry15 );

}
```

El Inc16 es un incrementador de 16 bits que incrementa un número de 16 bits en uno. La salida (out) es el resultado de añadir 1 al número de entrada. Este proceso se realiza utilizando un HalfAdder para el bit menos significativo y FullAdders para manejar los acarreos a través de las posiciones de bit restantes.

Chip ALU:

```
CHIP ALU {
    IN
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute (out = x + y) or (out = x & y)?
        no; // negate the out output?
    OUT
        out[16], // 16-bit output
        zr,      // if (out == 0) equals 1, else 0
        ng;      // if (out < 0)  equals 1, else 0

    PARTS:

    Mux16(a= x, b= false, sel= zx, out=zxout );
    Mux16(a= y, b= false, sel= zy, out=zyout );

    Not16(in= zxout, out= notzxout);
    Not16(in= zyout, out= notzyout);

    Mux16(a= zxout, b= notzxout, sel= nx, out= nxout);
    Mux16(a= zyout, b= notzyout, sel= ny, out= nyout);

    Add16(a = nxout, b = nyout, out = sum);
    And16(a= nxout, b= nyout, out= and);
    Mux16(a= and, b= sum, sel= f, out= fout);

    Not16(in= fout, out= notfout);
    Mux16(a= fout, b= notfout, sel= no, out= aux);

    And16(a= aux, b= aux, out= out);
}
```

La ALU es un componente crucial del procesador que realiza diversas operaciones aritméticas y lógicas sobre dos números de 16 bits. Las señales de control (zx, nx, zy, ny, f, no) determinan la operación específica a realizar. Por ejemplo, zx y zy pueden poner a cero las entradas x y y, respectivamente, mientras que nx y ny pueden negar estas entradas. La señal f decide si la operación es una suma (x + y) o una operación AND (x & y), y no puede negar el resultado de la operación seleccionada. La salida (out) es el resultado de la operación, y los indicadores (zr y ng) muestran si la salida es cero o negativa, respectivamente. La ALU permite una amplia gama de operaciones, combinando sumas y operaciones lógicas, y manipulando las entradas y salidas según las necesidades específicas del cálculo.

Project 3.

Chip Bit:

```
CHIP Bit {
    IN in, load;
    OUT out;


    PARTS:
    Mux(a=outloop, b=in, sel=load, out=out1);
    DFF(in=out1, out=outloop, out=out);
}
```

El chip Bit es la unidad más básica de almacenamiento de memoria, capaz de guardar un solo bit de información. Utiliza un DFF para retener el bit de manera continua. Una señal de carga (`load`) determina si el valor de entrada (`in`) debe ser almacenado en el bit.

Chip Register:

```
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]);
    Bit(in=in[3], load=load, out=out[3]);
    Bit(in=in[4], load=load, out=out[4]);
    Bit(in=in[5], load=load, out=out[5]);
    Bit(in=in[6], load=load, out=out[6]);
    Bit(in=in[7], load=load, out=out[7]);
    Bit(in=in[8], load=load, out=out[8]);
    Bit(in=in[9], load=load, out=out[9]);
    Bit(in=in[10], load=load, out=out[10]);
    Bit(in=in[11], load=load, out=out[11]);
    Bit(in=in[12], load=load, out=out[12]);
    Bit(in=in[13], load=load, out=out[13]);
    Bit(in=in[14], load=load, out=out[14]);
    Bit(in=in[15], load=load, out=out[15]);
}
```

El Register es un conjunto de 16 bits, diseñado para almacenar un valor de 16 bits. Este chip está compuesto por 16 chips Bit, cada uno de los cuales guarda un bit del valor de 16 bits. Una señal de carga (`load`) controla si el valor de entrada de 16 bits debe ser almacenado en el registro.

Chip Ram8:

```
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);
    Register(in=in, load=load2, out=out2);
    Register(in=in, load=load3, out=out3);
    Register(in=in, load=load4, out=out4);
    Register(in=in, load=load5, out=out5);
    Register(in=in, load=load6, out=out6);
    Register(in=in, load=load7, out=out7);


    Mux8Way16(a=out0, b=out1, c=out2, d=out3,
            e=out4, f=out5, g=out6, h=out7,
            sel=address, out=out);
}
```

El RAM8 es una memoria compuesta por 8 registros de 16 bits cada uno. Utiliza una lógica de direccionamiento para seleccionar el registro en el que se desea operar. La selección del registro se realiza mediante una señal de dirección de 3 bits (`address`), permitiendo elegir uno de los 8 registros disponibles para lectura o escritura.

Chip Ram64:

```
CHIP RAM64 {
    IN in[16], load, address[6];
    OUT out[16];

    PARTS:
    // access logico
    DMux8Way(in=load, sel=address[3..5],
            a=load0, b=load1, c=load2, d=load3,
            e=load4, f=load5, g=load6, h=load7);

    // RAM array
    RAM8(in=in, load=load0, address=address[0..2], out=out0);
    RAM8(in=in, load=load1, address=address[0..2], out=out1);
    RAM8(in=in, load=load2, address=address[0..2], out=out2);
    RAM8(in=in, load=load3, address=address[0..2], out=out3);
    RAM8(in=in, load=load4, address=address[0..2], out=out4);
    RAM8(in=in, load=load5, address=address[0..2], out=out5);
    RAM8(in=in, load=load6, address=address[0..2], out=out6);
    RAM8(in=in, load=load7, address=address[0..2], out=out7);

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

El RAM64 expande la capacidad de almacenamiento a 64 registros de 16 bits cada uno. Está construido a partir de 8 chips RAM8 y utiliza una señal de dirección de 6 bits (`address`). Los 3 bits más significativos seleccionan uno de los chips RAM8, y los 3 bits menos significativos seleccionan el registro dentro del chip RAM8 elegido.

Chip Ram512:

```
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);
    RAM64(in=in, load=load2, address=address[0..5], out=out2);
    RAM64(in=in, load=load3, address=address[0..5], out=out3);
    RAM64(in=in, load=load4, address=address[0..5], out=out4);
    RAM64(in=in, load=load5, address=address[0..5], out=out5);
    RAM64(in=in, load=load6, address=address[0..5], out=out6);
    RAM64(in=in, load=load7, address=address[0..5], out=out7);


    Mux8Way16(a=out0, b=out1, c=out2, d=out3,
            e=out4, f=out5, g=out6, h=out7,
            sel=address[6..8], out=out);
}
```

El RAM512 extiende aún más la capacidad a 512 registros de 16 bits. Se construye a partir de 8 chips RAM64 y utiliza una señal de dirección de 9 bits (`address`). Los 3 bits más significativos seleccionan uno de los chips RAM64, mientras que los 6 bits menos significativos seleccionan el registro dentro del chip RAM64 seleccionado.

Chip Ram4K:

```
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);

    // RAM array
    RAM512(in=in, load=load0, address=address[0..8], out=out0);
    RAM512(in=in, load=load1, address=address[0..8], out=out1);
    RAM512(in=in, load=load2, address=address[0..8], out=out2);
    RAM512(in=in, load=load3, address=address[0..8], out=out3);
    RAM512(in=in, load=load4, address=address[0..8], out=out4);
    RAM512(in=in, load=load5, address=address[0..8], out=out5);
    RAM512(in=in, load=load6, address=address[0..8], out=out6);
    RAM512(in=in, load=load7, address=address[0..8], out=out7);

    // output logic
    Mux8Way16(a=out0, b=out1, c=out2, d=out3,
            e=out4, f=out5, g=out6, h=out7,
            sel=address[9..11], out=out);
}
```

El RAM4K proporciona una memoria de 4096 registros de 16 bits cada uno. Está formado por 8 chips RAM512 y utiliza una señal de dirección de 12 bits (`address`). Los 3 bits más significativos seleccionan uno de los chips RAM512, y los 9 bits restantes seleccionan el registro dentro del chip RAM512.

Chip Ram16K:

```
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);

    // RAM array
    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);

    // output logic
    Mux4Way16(a=out0, b=out1, c=out2, d=out3,
              sel=address[12..13], out=out);
}
```

El RAM16K es una memoria de 16384 registros de 16 bits cada uno, construida a partir de 4 chips RAM4K. Utiliza una señal de dirección de 14 bits (`address`), donde los 2 bits más significativos seleccionan uno de los chips RAM4K, y los 12 bits restantes seleccionan el registro dentro del chip RAM4K.

Chip PC:

```
CHIP PC {
    IN in[16],load,inc,reset;
    OUT out[16];

    PARTS:
    // prefix logic: inc -> load -> reset
    Add16(a=outloop, b[0]=true, b[1..15]=false, out=outloopInc);
    Mux16(a=outloop, b=outloopInc, sel=inc, out=t0);
    Mux16(a=t0, b=in, sel=load, out=t1);
    Mux16(a=t1, b=false, sel=reset, out=t2);

    // loading logic
    Or(a=load, b=inc, out=loadOrInc);
    Or(a=loadOrInc, b=reset, out=loadRAM);

    // timestep
    RAM8(in=t2, load=loadRAM, out=outloop, out=out);
}
```

El PC es un registro especializado que contiene la dirección de la próxima instrucción a ser ejecutada por la CPU. Este registro puede ser cargado con un nuevo valor, incrementado en 1, o mantenido en su valor actual, dependiendo de las señales de control. Es esencial para el control del flujo en la ejecución de programas.