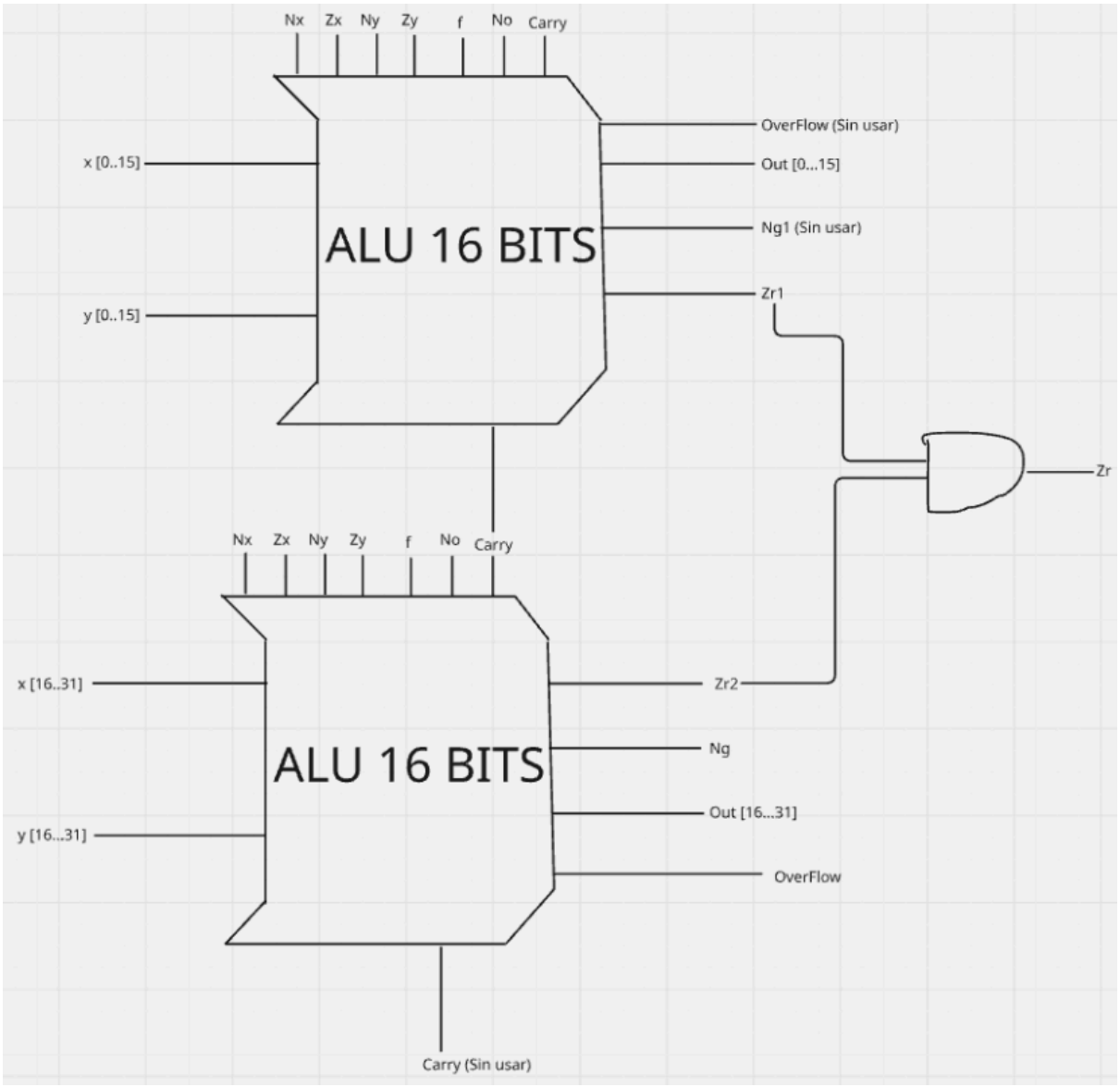
PARCIAL ORGANIZACIÓN DE COMPUTADORES

1. **Modularidad:** Ventajas y desventajas de usar dos ALU16 vs. una ALU32 monolítica. (Ej. ventajas: reutilizo; desventajas: latencia en carry.)

La modularidad es una estrategia de diseño que consiste en construir sistemas más complejos a partir de la división, donde se pueda reutilizar diseños más simples que faciliten la misma. Para ello construimos la ALU32 Monolítica y la ALU32 a partir de 2 ALU's16 para realizar un comparativo.

ALU32 a partir de 2 ALU's16



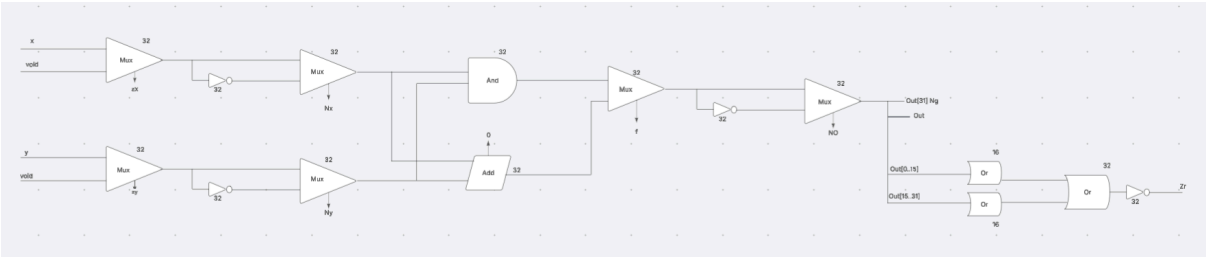
Esta ALU es el ejemplo del uso de la modularidad, ya que tomamos el diseño ya realizado de una ALU16 y las conectamos entre ellas (16 Bits parte baja, 16 Bits parte alta) al igual que el carry de la parte la parte baja con la alta, logrando así los 32 Bits, sin necesidad de

hacer grandes modificaciones. Esto trae una serie de ventajas y desventajas que abordaremos a partir de los principios de la modularidad.

Ventajas	
Reutilización de componentes	Se aprovecha el diseño e implementacion de una ALU16, evitando crear nuevamente los componentes
Abstracción y Simplicidad	Divide el problema complejo (32 bits) en bloques más pequeños y manejables (16 bits), lo que lo permite mayor facilidad de la implementación, además expone un diagrama más claro, ya que “oculta” los elementos que la componen, lo que permite una idea clara y específica de lo que se necesita.
Mantenibilidad	Al tener partes más simples es más fácil identificar los errores y resolverlos, ya que no necesitas enfocarte en la mirada global de la ALU32, sino en sus componentes de forma independiente.
Escalabilidad	Así como se construye un ALU32 a partir de ALU’s16 se puede escalar a ALU64 a partir de ALU32, lo que sigue con los principios de modularidad.

Desventajas	
Independencia	Bajo los principios de modularia lograr independencia es muy importante ya que cada parte puede realizar su trabajo independiente de la otra, pero en este caso, las ALU’s deben de estar interconectadas, por lo que una debe de esperar un resultado (carry) de la otra para seguir realizando su trabajo.
Latencia por propagación del acarreo	La suma y resta requieren que el carry (ripple carry - propagación del carry) se transfiera a la otra ALU, por lo que implica un tiempo extra en el procesamiento
Complejidad de control	Aunque se logra una abstracción que permite mayor facilidad del problema trae consigo el reto de hacer las interconexiones de manera correcta. Por ejemplo con el manejo del overflow.

ALU32 Monolítica



Una ALU monolítica de 32 bits sería un solo bloque de hardware diseñado específicamente para trabajar con entradas y salidas de 32 bits, lo que implica la construcción de cada componente para que pueda trabajar con esas entradas y salidas, aquí no se trabaja la modularidad y trae consigo una serie de ventajas y desventajas:

Ventajas	
Diseño más compacto	Al estar todo construido de acuerdo el tamaño de las entradas y salidas no es necesario la conexión adicional (submódulos), como lo sería con el carry, las flags (zr,ng, overflow) y la interpretación de las salidas.
Menor latencia si se usa optimización del carry (carry lookahead)	Si se utilizará en un sistema monolítico el ripple carry implicaría mayor tiempo y gasto energético pasando bit a bit, por su estructura monolítica, permite la implementación de carry lookahead sin necesidad de rediseñar la estructura de módulos ya que no los usan, por lo que su implementación consistirá en que cada Bit pueda predecir si va a haber un acarreo o no a partir de la **generación** de carry (si dos bits son 1 entonces generará un carry) y sino **propagará** el carry

Desventajas	
Mayor complejidad del diseño y sin reutilizar	Requiere construir todo los componentes que necesita para su operación, lo que implica mayor esfuerzo y tiempo, aunque pueda generar la facilidad de que siempre tiene las mismas piezas cada vez se hará más engorroso su construcción
Dificultades para la escalabilidad	Cada vez que se requiera aumentar el tamaño de procesamiento requerirá volver a hacer todo, además que cada vez se tendrán más compuertas que manejar
Mantenimiento	Encontrar los errores implica mayor esfuerzo, puesto que no tienes la posibilidad de revisión por módulos

2. **Signed vs. unsigned:** Cambios necesarios para soportar ambos tipos de operaciones. (Ej. banderas adicionales para unsigned overflow.)

Las ALU's están diseñadas para realizar operaciones binarias tanto para números con signo (signed) como para número sin ellos (unsigned), la diferencia radica en la implementación del resultado (Bit más significativo) y las flags de estado que generan.

Unsigned

Los números unsigned se representan en binario puro, donde todos los bits corresponden a la magnitud del número. En este caso no existe un bit de signo: el rango de un entero de 32 bits va de 0 a ($2^{32} – 1$) Para poder soportar estas operaciones debe generar y entender un carry para el manejo de desbordamientos.

Signed

Los números con signo en el Hardware también se representan con complemento de dos, donde el Bit más significativo (MSB) determina si el número es positivo (0) negativo (1), esto cambia la interpretación del rango de números, ya que representa negativos desde -2^{31} a $+2^{31} -1$, esta representación permite hacer operaciones de suma o resta con una lógica binaria estandar, de lo contrario tocaría implementar representaciones de signo-magnitud, donde el MSB daría el signo y el resto de bits el valor (Ej: 1000101= -5) esto implicaría un la ALU debería de identificar si es positivo o negativo antes de la operación para luego hacer las operaciones entre las magnitudes, esto implicaría pasos extras y además se tendrían 2 ceros distintos (-0, +0), lo cual es un error. Otra implementación que podría hacer es el representación complemento de 1 (Ej: 1111010= -5), en está las operación deben gestionar de forma distinta el carry agregándole un +1 cuando hay carry final. Estas implementaciones traerán consigue flags y conexiones adicionales (uno para la suma y otro para la resta), lo que lo haría poco lógico y más complejo su diseño y operación. Por esto se utiliza la representación complemento 2 que permite hacer las operaciones de forma lógica binaria como lo hacen los números sin signos. Adicionalmente, maneja las siguentes flags.

Overflow: desbordamiento (excede el rango de número representativos) en signed

Zr: Indican si el resultado es 0

Ng: Indicar si el resultado es positivo o negativo

Carry: Propagación

En la ALU se encuentran las flags necesarias para poder hacer manejos de unsigned y signed dependiendo del contexto, generando todas las flags y es el proceso de identificar lo hace el set de instrucciones.

3. **Carry propagation:** Cómo implementarías un carry-lookahead y qué implicaciones tendría. (Ej. reduce latencia pero aumenta compuertas.):

Un carry lookahead adder evita tener que esperar a que cada bit calcule y entregue su carry al siguiente (como en un sumador normal). En su lugar, usa lógica adicional para **anticipar** si un bloque de bits va a generar o propagar un carry, de modo que varios se calculan **en paralelo**. Esto hace que la suma sea mucho más rápida (no crece linealmente con el número de bits), aunque la desventaja es que se necesitan **más compuertas y conexiones**, lo que aumenta el consumo y la complejidad del diseño. En el repositorio hemos implementado este CLA, completamente funcional, y la manera simplificada de predecir cada carry es esta:

c0 = cin

$c1 = g0 \vee (p0 \cdot c0)$

$c2 = g1 \vee (p1 \cdot g0) \vee (p1 \cdot p0 \cdot c0)$

$c3 = g2 \vee (p2 \cdot g1) \vee (p2 \cdot p1 \cdot g0) \vee (p2 \cdot p1 \cdot p0 \cdot c0)$

$c4 = g3 \vee (p3 \cdot g2) \vee (p3 \cdot p2 \cdot g1) \vee (p3 \cdot p2 \cdot p1 \cdot g0) \vee (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$

Como vemos, entre más adders tenemos, más operaciones lógicas utilizamos, por eso es que, por lo general, las implementaciones son solo de 4 bits, ya que hacerlo de más bits representaría un crecimiento mucho más considerable. En conclusión, es mucho más rápido que el ripple adder, pero es mucho más complejo y utiliza muchas más compuertas lógicas

4. **Optimización:** Si tu diseño actual consume demasiadas compuertas lógicas, ¿qué técnicas aplicarías para reducir el uso de hardware sin perder funcionalidad? (Ej. multiplexores compartidos.)

El diseño de una ALU32 a partir de dos ALU16 consume muchas compuertas por la duplicación de operaciones y señales de control. Para optimizar el hardware, se necesita aplicar técnicas de simplificación y reutilización de recursos.

Se podría aplicar una reutilización del adder, de modo que tanto la suma como la resta se ejecuten en el mismo circuito, haciendo la resta a través del complemento a dos y compuertas XOR controladas, en lugar de utilizar bloques separados. Igualmente,la organización jerárquica de la ALU32 en dos ALU16 posibilita compartir señales de control entre ellas, minimizando una lógica que puede ser redundante. Estas podrían ser algunas técnicas que hacen posible disminuir el uso de compuertas, lo que lo hace más eficiente, sin que afecte la capacidad de la ALU para hacer operaciones.

También se puede implementar una optimización del carry, como el carry lookahead (calcula en paralelo si un bloque de Bits genera un carry y se propaga), carry select adder (divide el sumador en bloques y realiza en paralelo la posibilidad de que el cin = 0 o cin= 1, y es seleccionado por un multiplexor) y carry skip (que a partir de la propagación de un conjunto de bit los “salta”). Estas técnicas son más rápido suelen gastar más energía y compuertas.

5. **Escalabilidad:** Estrategia para extender el diseño a 64 o 128 bits sin reescribir todo. (Ej. enlazar más ALU16 con carry chain.)

La escalabilidad de una ALU32 hacia 64 o 128 bits se puede lograr por medio de un enfoque de módulo sin necesidad de tener que reescribir todo el diseño. La estrategia principal consistiría en aprovechar la estructura jerárquica de la ALU32, que ya tenemos, basada en la interconexión de varios bloques ALU16, y extender el mismo a mayor cantidad de módulos. De esta manera, para construir una ALU64 bastaría con enlazar cuatro bloques ALU16 en cadena, propagando el carry out de un bloque como carry in del siguiente. De la misma manera, una ALU128 podría formarse a partir de ocho bloques ALU16 organizados de la misma manera.

Esta forma por módulos permite mantener el diseño mas uniforme, ya que las señales no necesitan redefinirse, sino únicamente extenderse a cada uno de los bloques. Además, se facilita la reutilización del hardware ya diseñado y probado, lo que reduciría el tiempo de implementación y minimizaría los errores. Sin embargo no es eficiente en tiempo y energía escalarlo de esta forma, de continuar con un sistema modular lo más razonable sería hacerlo con dos ALU's de la mitad de bits que se desea obtener, es decir, una ALU64 estaría construida por 2 ALU's de 32Bits y una ALU128 con 2 ALU's de 64 lo que mantendría los principios de modularidad del punto 1 de forma más simple y clara.

En la industria la forma más utilizada es la división de a 4, 8, 16 Bits con optimización de carries como el carry lookahead para mejorar la eficiencia y su rapidez.