Entrega 1 Organización de Computadores, por Miguel Villegas Nicholls.

Descripción del Diseño.

La **ALU32** es una unidad aritmético-lógica de 32 bits construida de forma modular a partir de dos bloques **ALU16**, los cuales procesan en paralelo la parte baja y alta de los operandos. Este enfoque favorece la reutilización de componentes previamente verificados y simplifica la escalabilidad del diseño a anchos mayores. El diseño implementa operaciones aritméticas y lógicas siguiendo la convención establecida en la ALU de 16 bits del curso Nand2Tetris, extendida a 32 bits. Entre sus capacidades se encuentran la suma, la resta, la conjunción lógica (AND), así como la negación condicional de entradas y salidas, además de la detección de condiciones de estado como cero, negativo y overflow.

Señales de entrada

- x[32], y[32]: Operandos de 32 bits.
- zx, nx: Controlan el **zeroing** (forzar a cero) y la **negación** de x.
- zy, ny: Controlan el **zeroing** y la **negación** de y.
- f: Selecciona entre operación aritmética (f=1, suma/resta) o lógica (f=0, AND).
- no: Indica si el resultado debe ser negado antes de la salida.

Señales de salida

- out[32]: Resultado de 32 bits de la operación.
- zr: Bandera de **cero**, activada si el resultado es 0x0000_0000.
- ng: Bandera de **negativo**, refleja el bit más significativo del resultado (bit 31).
- overflow: Bandera de **overflow aritmético con signo**, activada si el resultado excede el rango representable en complemento a dos de 32 bits.

Decisiones clave de diseño

1. **Modularidad mediante dos ALU16**:
   El diseño se compone de una ALU de 16 bits para la parte baja (x[0..15], y[0..15]) y otra para la parte alta (x[16..31], y[16..31]). Esto permite reutilizar el bloque existente y facilita la escalabilidad a arquitecturas de mayor ancho.

2. **Propagación de carry**:
   El carry de la ALU baja se propaga como entrada a la ALU alta **solo en operaciones aritméticas** (f=1). En operaciones lógicas (f=0) la propagación se inhibe, garantizando que las salidas no se vean afectadas por arrastres innecesarios.

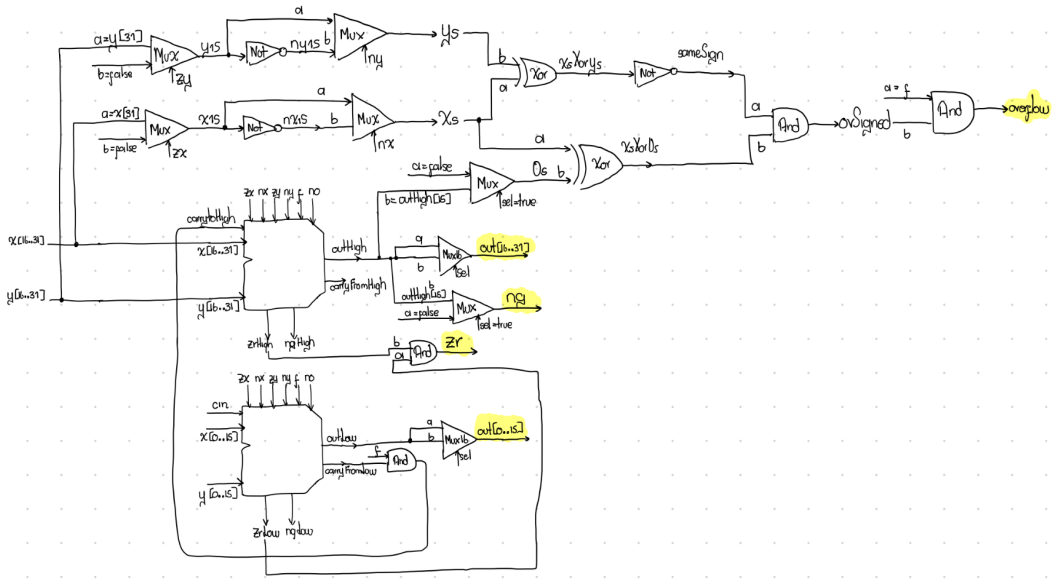
3. **Construcción de salida de 32 bits**:
   Los resultados parciales (outLow y outHigh) se combinan en un bus de salida de 32 bits mediante multiplexores de paso, asegurando compatibilidad y evitando errores de direccionamiento en el simulador.

4. **Cálculo de banderas de estado**:
   - **Zero (zr)**: Se activa únicamente cuando tanto la parte baja como la alta son cero.
   - **Negative (ng)**: Refleja el bit 31 (MSB) del resultado completo, correspondiente al bit 15 de la parte alta.
   - **Overflow (overflow)**: Se calcula comparando los signos de los operandos normalizados (Xs, Ys) y el signo del resultado (Os). Se activa si los operandos tienen el mismo signo pero el resultado difiere, y solo en modo aritmético (f=1).

Con este diseño, la ALU32 logra un balance entre **modularidad, claridad y funcionalidad**, cumpliendo con las operaciones fundamentales de una arquitectura de 32 bits y permitiendo extender el modelo fácilmente a anchos superiores.

Diagrama/Esquema del Diseño.

Diagrama ALU de 32 bits:



Preguntas de Pensamiento Crítico.

1. **Modularidad:** Ventajas y desventajas de usar dos ALU16 vs. una ALU32 monolítica. (Ej. ventajas: reutilizo; desventajas: latencia en carry.)
   a. La modularidad es un principio fundamental en diseño digital porque permite reutilizar componentes, simplificar la verificación y mejorar la mantenibilidad. Usar dos ALU16 para construir una ALU32 ofrece claras ventajas: se aprovecha un bloque

probado y confiable, lo cual reduce la complejidad de diseño y facilita el escalamiento a arquitecturas más grandes. Además, esto se alinea con la filosofía de los procesadores reales, donde bloques más pequeños se encadenan. Sin embargo, la principal desventaja es la **latencia en la propagación del carry**: al depender de la salida de la ALU baja para la entrada de la ALU alta, la suma se vuelve más lenta frente a una ALU monolítica optimizada. Asimismo, duplicar componentes puede generar mayor consumo de compuertas frente a un diseño específico de 32 bits. En conclusión, la decisión entre modularidad y monolitismo depende de si se prioriza simplicidad y reutilización (ALU16) o rendimiento absoluto y optimización de hardware (ALU32 dedicada).

2. **Signed vs. unsigned:** Cambios necesarios para soportar ambos tipos de operaciones. (Ej. banderas adicionales para unsigned overflow.)
    a. El soporte para operaciones con y sin signo implica diferencias en cómo se interpretan las banderas de estado. En complemento a dos (signed), la bandera de **overflow** se calcula comparando los signos de los operandos y del resultado, mientras que en operaciones sin signo lo importante es el **carry out** como indicador de desbordamiento. Para soportar ambos, sería necesario introducir banderas adicionales: por ejemplo, `OF` (overflow) para signed y `CF` (carry flag) para unsigned. También sería útil agregar un bit de control que indique al hardware si la operación debe evaluarse bajo semántica signed o unsigned, de manera que el circuito active la lógica de detección adecuada. Aunque las operaciones básicas de suma, resta o AND son las mismas a nivel de bits, la interpretación cambia radicalmente en el nivel lógico. Así, se trata menos de modificar la ALU en su núcleo y más de extender la lógica de **estado** que acompaña a cada operación.

3. **Carry propagation:** Cómo implementarías un carry-lookahead y qué implicaciones tendría. (Ej. reduce latencia pero aumenta compuertas.)
    a. El método clásico de propagación de acarreo en un ripple-carry adder es secuencial: cada bit espera el resultado del anterior, lo que introduce latencia proporcional al número de bits. Un **carry-lookahead adder (CLA)** resuelve este problema calculando de manera anticipada los términos de generación (`Gi = Ai·Bi`) y propagación (`Pi = Ai ⊕ Bi`) del acarreo, permitiendo obtener el carry de cada etapa a partir de expresiones lógicas combinacionales. Implementar un CLA en una ALU de 32 bits reduciría significativamente la latencia, ya que el cálculo del carry se hace en paralelo en grupos de bits, logrando tiempos cercanos a log(n) en lugar de lineales. No obstante, esta técnica aumenta de manera notable el consumo de compuertas y la complejidad del circuito, porque se requiere mucha lógica extra para calcular los `Pi` y `Gi`. En síntesis, el CLA representa un **trade-off clásico**: sacrificar área y energía para obtener menor retardo, lo cual es fundamental en procesadores de alto rendimiento.

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.)
    a. Reducir el consumo de compuertas en una ALU es un problema de optimización de recursos. Una primera estrategia es la **compartición de hardware**: en lugar de tener circuitos separados para operaciones como suma y resta, se puede usar el mismo sumador con lógica adicional que complemente el operando en operaciones de resta. Asimismo, operaciones como OR pueden implementarse a partir de NOT y AND mediante leyes de De Morgan, evitando duplicar circuitos. Otra técnica es usar **multiplexores compartidos**: en vez de tener caminos de datos paralelos, se reutiliza la misma ruta con selectores. También es común aplicar técnicas de **factorización booleana** para simplificar expresiones lógicas. Finalmente, si el objetivo es un diseño más económico y no tanto velocidad, puede preferirse el ripple-carry frente al carry-lookahead, ya que reduce significativamente el número de compuertas. Todas estas decisiones requieren balancear simplicidad, costo en hardware y rendimiento esperado del sistema.

5. **Escalabilidad:** Estrategia para extender el diseño a 64 o 128 bits sin reescribir todo. (Ej. enlazar más ALU16 con carry chain.)
    a. La escalabilidad en diseño digital se logra aplicando el principio de modularidad. Una estrategia eficiente para ampliar a 64 o 128 bits es encadenar más bloques `ALU16C`, de modo que se construya una ALU mayor simplemente replicando módulos ya probados. Este enfoque reduce el esfuerzo de rediseño y facilita la verificación, pues basta comprobar la correcta propagación del carry entre módulos. Por ejemplo, una ALU64 se implementaría con cuatro ALU16C en cascada y una ALU128 con ocho. Si el rendimiento es crítico, se podrían agrupar los bloques en unidades de 32 bits y usar técnicas de carry-lookahead a nivel de bloque, reduciendo la latencia global. Además, mantener una interfaz uniforme (x[n], y[n], banderas) asegura compatibilidad con arquitecturas superiores. En resumen, la clave para escalar es diseñar bloques básicos reutilizables y definir un protocolo claro de comunicación entre ellos, de forma que la extensión sea un proceso de ensamblaje más que de rediseño.