Integrante: Andres Felipe Prieto Lozano

# Universidad EAFIT

# Curso: Organización de Computadores

---

**PRIMER PARCIAL:**

**ALU 32 bit**

**Preguntas de Analisis**

**Nombre:**

**Andres Felipe Prieto Lozano**

**Profesor:**

**Edison Valencia**

**Materia:**

**Organización de Computadores**

**Universidad EAFIT**

**Escuela de Ciencias aplicadas e Ingeniería**

**Medellín**

**2025**

Integrante: Andres Felipe Prieto Lozano

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

Para el desarrollo del ejercicio de la creacion de una ALU 32 BIT la realice usando dos ALU de 16 BIT, esto debido a ciertas Ventajas, las cuales son:

1. Reutilización:

   Esto debido a que solamente se diseña un ALU16 ( en mi caso se llama ALU16C ) para luego en el archivo .HDL del ALU32, en vez de hacer muchas compuertas y hacerlo algo confuso, solamente hay que usar 2 ALU16 haciéndolo mas sencillo en la elaboración.

2. Simplicidad:

   Usamos cadenas de bits de 16, haciendo que sea más corto en cuanto a la depuración en posibles errores.

3. Escalabilidad:

   Ahora, será más sencillo pasar a 48/64 bits, ya que podremos usar varios ALU16 o ALU32, en vez de usar muchísimas compuertas en un solo archivo, haciéndolo más sencillo escalar

Aunque también pueden haber algunas desventajas:

1. Se podría generar una latencia de propagación de carry en cadena ripple entre ambas mitades, ya que la ALU16 ALTA tendra que esperar que la ALU16 baja produzca el CarryOUT, generando que el tiempo por operacion sea en serie, lo

que provocaria que para crear el ALU64 tenga una latencia mucho mayor, en vez de si la hacemos MONOLITICA, podriamos generar que carrys se calculen en paralelo

2. Fisicamente, hay que usar mas cables para la señal de interconexion entre carrryLOW y carryHIGH
3. Se necesita una compuerta EXTRA AND para detectar el ZR

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

a. Deberíamos añadir un control, mode = ( 0 signed / 1 unsigned )
b. Capturar el carry final( carryHigh ya lo tengo )
c. Luego de realizar las operaciones
d. El Flag ( ng ), solamente usar cuando el mode = 0
e. Usar overflow signed cuando mode = 0, tal cual esta
f. OverFlow unsigned. usa el carry High como indicador cuando mode=1
g. el ZR sigue igual

Integrante: Andres Felipe Prieto Lozano

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

1. Para este caso, dividiría los 32 bits en bloques de 4 bits
2. Para cada bit, calcularia dos señales:
    a. G_i = a_i AND b_i, genera carry
    b. P_i = a_i XOR b_i Propaga carry
3. Para cada bloque de 4 bits:
    a. G_block = combinación de G y P (si el bloque produce un carry)
    b. P_block = P0·P1·P2·P3 (si dejaría pasar un carry de entrada)
4. Calcular carries entre bloques en paralelo
5. Con el carry de entrada de cada bloque, dentro del bloque se generan los carries de cada bit usando las mismas fórmulas (pero sólo 4 niveles).
6. Finalmente cada suma: sum_i = P_i XOR carry_de_bit.

Esto generará menos operaciones en Serie y se harán más rápidas en paralelo

**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.)**

1. Compartir / eliminar multiplexores innecesario.
    a. En mi ALU32.hdl puse dos Mux16 con sel=false que solo copian low y high a out. Eso es redundante: puedo cablear directamente out[0..15]=low y out[16..31]=high y ahorro dos Mux16 completos.
2. Reutilizar señales ya calculadas
    a. Ya genero zrLow y zrHigh de cada mitad; combinarlos con un solo And me da zr sin construir una reducción grande de 32 bits. También uso directamente el bit de signo de la mitad alta (resSign) para ng (no necesito más lógica que un OR con false, que también podría quitar y cablear directo).
    b. Evitar lógica duplicada en normalización de entradas
    c. Las señales de control (zx,nx,zy,ny,f,no) se aplican dentro de cada ALU16C. Si quisiera reducir compuertas, podría normalizar cada operando (zero / negate) una sola vez a 32 bits antes de dividirlo en mitades, y luego alimentar dos bloques más simples.

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

Reutilizo el mismo bloque base (ALU16C) que ya uso dos veces en la ALU32. Para 64 bits pondría 4 bloques en cadena; para 128 bits, 8. Solo copio el patrón que ya tengo: cada bloque maneja un segmento de 16 bits del bus ([0..15], [16..31], etc.).

Paso el carry igual que ahora: el primer bloque recibe cIn=false; su cOut entra como cIn del siguiente, y así sucesivamente (ripple). No toco la lógica interna.

El flag ng lo sigo tomando del bit de signo del resultado del último bloque (lo que hoy llamo resSign en 32 bits).

El flag zr lo obtengo AND-eando los zr de cada bloque, igual que hoy hago zr = zrLow AND zrHigh, solo extendido (por ejemplo en cadena).

El overflow lo calculo igual que ahora, usando solo el bit más alto: overflow = f AND (xMSB == yMSB) AND (resMSB != xMSB). Solo cambio las referencias a los índices del último bit (63 o 127).

Las señales de control (zx,nx,zy,ny,f,no) van a todos los bloques en paralelo, como ya están duplicadas ahora para las dos mitades.