Respuestas a preguntas de pensamiento crítico

1. **Modularidad: ¿dos ALU16 o una ALU32 monolítica?**

Ventajas de usar "dos ALU16":

- Reuso/bit-slice: mismo bloque 16-bit se replica; simplifica verificación y mantenimiento.
- Flexibilidad SIMD: puedes operar en paralelo en dos palabras de 16 bits (ejemplo: DSP: dos sumas saturadas de 16 bits/clk).
- Timing local: trayectos internos más cortos; facilita cumplir frecuencia en nodos lentos o con ruteo difícil.
- Rendimiento/energía dinámicos: desactivar un slice cuando se trabaja a 16 bits; clock/power-gating por bloque.
- Rendimiento de fabricación: bloques pequeños suelen tener mejor yield y permiten floorplanning modular.

Desventajas de usar "dos ALU16":

- Latencia de carry inter-bloques: si encadenas carry (para 32-bit), el límite crítico incluye el salto de 16→17.
- Complejidad de control: más señales (modo 2×16 vs 1×32), multiplexores de caminos y banderas por mitad.
- Área extra: compuertas de control y multiplexores; posible duplicación de flags/comparadores.
- Operaciones de 32 bits no triviales. Ejemplo: multiplicación/división requieren coordinación entre slices.

Ventajas de ALU32 monolítica:

- Trayecto crítico optimizable globalmente: un único CLA/CSA/CSLA jerárquico sobre 32 bits.
- Menos control: interfaz más simple; una sola lógica de flags.
- Menos ruteo inter bloques: menos saltos de frontera.

Desventajas de ALU32 monolítica

- Menos reutilización: verificación y ECOs pueden ser más costosos.
- Menos flexibilidad SIMD (a menos que decidamos añadir partición interna/byte masking).
- Posible peor frecuencia si usas ripple carry largo sin aceleración.

2. **Signed vs. unsigned: Cambios necesarios para soportar ambos tipos de operaciones.**

Suma/resta usan el mismo sumador de complemento a dos; lo que cambia es cómo interpretas flags y qué operación de desplazamiento eliges.

Flags y condiciones:

- Carry-out (C): relevante para unsigned (overflow cuando Cout=1 en suma o Cout=0 tras A−B).
- Overflow con signo (V): V = C_n XOR C_{n-1} (carry a la salida del MSB vs carry interno). Indica overflow signed.
- Negativo (N): bit de signo del resultado (MSB).
- Zero (Z): resultado == 0.
- Less-than:
    - Unsigned: tras A−B, A<B si C=0 (borrow).
    - Signed: A<B si N XOR V = 1.

Operaciones específicas:

- Shifts: necesitas LSR (logical shift right) para unsigned y ASR (arithmetic shift right) para signed; LSL es común. Un barrel shifter debe soportar LSR/ASR/ROL/ROR.
- Comparadores: o bien reutilizas A−B y decodificas flags, o añades comparador dedicado (más área, menos latencia de branch).
- Multiplicación/División: agrega soporte signed (p. ej., Booth/Booth modificado) y extensión de signo en los operandos.
- Saturación/Redondeo (opcional): modos saturados suelen depender del tipo (p. ej., saturación signed a 0x7FFF/0x8000).

Interfaz de control sugerida:

- Campos de operación separados: op_addsub, op_logic, op_shift, is_signed, sat_en.
- Rutas de datos compartidas y decodificación de flags dependiente de is_signed.

3. Carry propagation: implementar Carry-Lookahead (CLA)

Objetivo: reducir latencia del carry frente a ripple. Aumenta compuertas y ruteo.

Señales por bit:

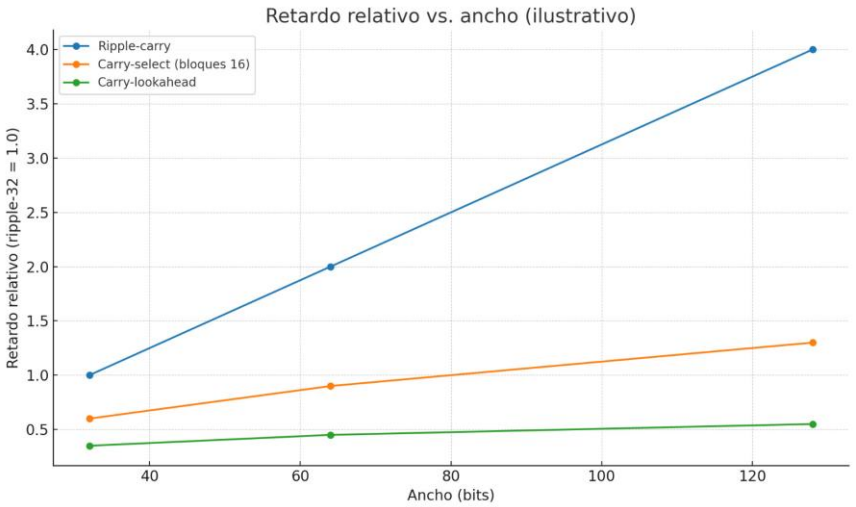
- Generate: G_i = A_i & B_i
- Propagate: P_i = A_i ^ B_i
- Carry: C_{i+1} = G_i | (P_i & C_i)
- Suma: S_i = P_i ^ C_i

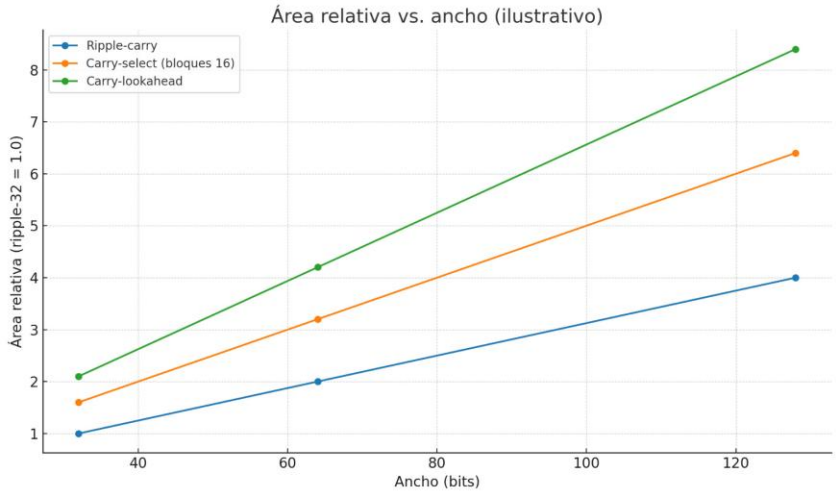
Implicaciones:

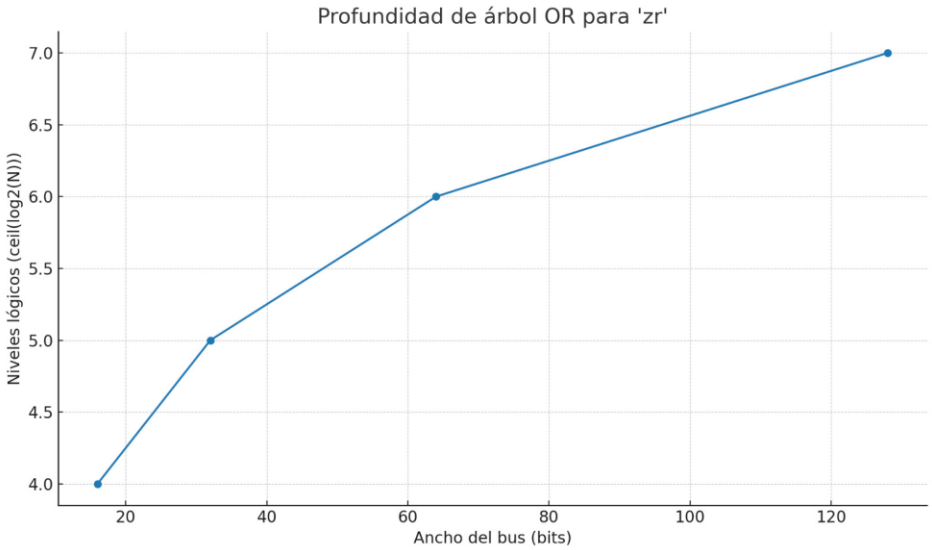
- Latencia ↓: de O(n) (ripple) a ~O(log n) jerárquico.
- Área ↑: puertas adicionales para productos lógicos y OR de varios términos.
- Ruteo ↑: redes de P/G globales; cuidado con fan-out.
- Energía ↑ (típicamente): más conmutación en redes largas.
- Compromisos: alternativas como carry-select (CSLA con multiplexores) o prefix adders (Kogge-Stone, Brent-Kung) si apuntas a frecuencias muy altas.

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

- Compartir Mux de “zero” (zx/zy). Patrón: Mux16(a=bus, b=false, sel=z*). Evita ANDs con máscaras por bit.
- Negación condicional barata. Un Not16 por bus + Mux16(a=bus, b=~bus, sel=n*); normalmente menos área que Xor16 bit a bit.
- Preproceso único reutilizado. Calcula xx y yy una sola vez (después de zx/nx y zy/ny) y úsalo para AND y ADD.
- Flags desde el resultado final. zr con árbol OR (Or8Way→Or16Way→Not); ng = MSB del resultado. No dupliques lógica de flags.
- Adder de área mínima. Ripple-carry (cadena de FullAdder o Add16CF); mínimo en compuertas (a costa de retardo).
- Propaga carry sólo en suma. cin_next = f AND cout_prev (cuando f=0, no hay lógica extra).
- Constantes compartidas. Reusa false/true en todos los Mux.
- Módulos auxiliares reutilizables. Define una vez Add16CF, Or16Way, MSB16 y reutiliza en todos los anchos.







5. **Escalabilidad: Estrategia para extender el diseño a 64 o 128 bits sin reescribir todo.**

Plan modular por "slices" de 16 bits:

- Slice16 con cin/cout (usa el mismo patrón: preproceso → AND/ADD con Add16CF → no → res + cout).
- ALU32 = 2×Slice16: c1 = f AND c0 (con c0=false).
- ALU64 = 4×Slice16: c[i+1] = f AND c[i] (cadena).
- ALU128 = 8×Slice16: igual idea.

Flags a N bits:

- zr = NOT(OR en árbol de todos los slices).
- ng = MSB del último slice (tras no).
- cout = cout del último slice (si f=1).
- ovf (sólo para suma) = (sx==sy) AND (ss!=sx) usando signos de la suma del último slice antes de no.

Opcional (más rápido sin tocar el arnés):

- Cambia sólo el "motor" del adder por bloque a carry-select (dos sumas en el bloque alto, con cin=0/1 y un Mux) o carry-lookahead. El resto del cableado (preproceso, f, no, flags y zero-tree) queda igual.