Defensa de Talleres 1 y 2

Organización del Computador 1

Primer cuatrimestre 2025

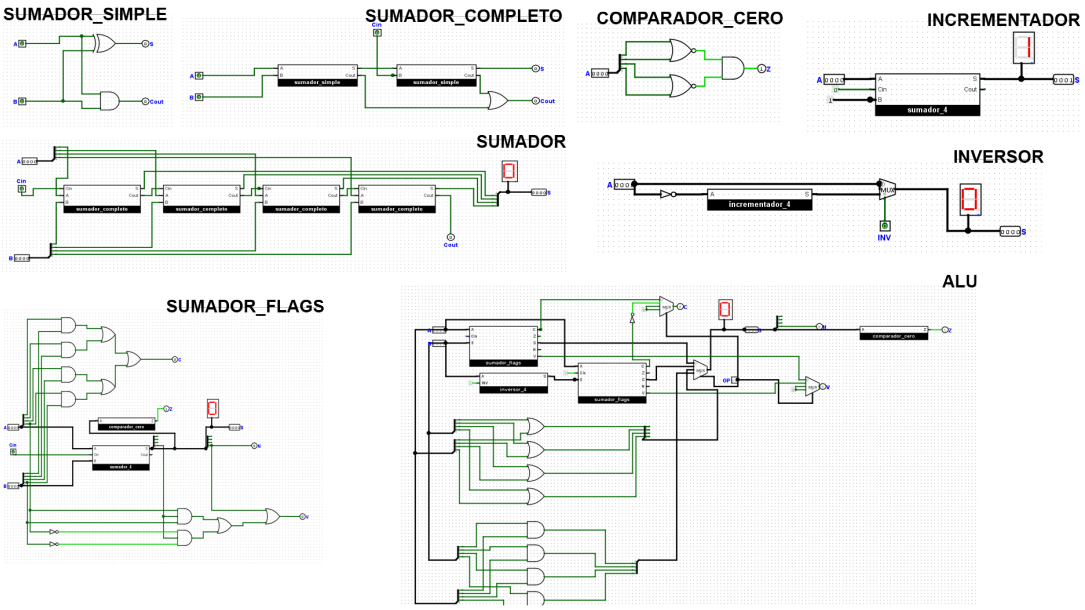
1 Taller 1

1.1 Idea general

Implementación de una **ALU de 4 bits** que suma, resta y evalua ciertas condiciones y sus componentes: sumadores, comparador, incrementador e inversor. Se verifican los resultados finales con flags de acuerdo a lo devuelto y en las tablas se puede ver la diferencia en la interpretación de diferentes operaciones según sea sin signo ó complemento a 2.

1.2 Componentes

- **Sumador simple**: implementa una suma de un bit sin carry de entrada
- **Sumador completo**: agrega el carry de entrada y salida, para poder usarse "en cadena"
- **Sumador de 4 bits**:
  - Cuatro sumadores completos conectados en cadena, suma números de 4 bits.
  - Se verifican overflow (V) y carry_out (C), obteniendo los resultados esperados para suma sin y con signo (complemento a 2)
- **Comparador_cero**: detecta si el resultado de una operación es cero
- **Incrementador**: Suma 1 a un input
- **Inversor**: si INV = 0 devuelve el input original, si INV = 1 devuelve la operación $(\neg A + 1)$, el inverso en complemento a 2. Todo esto es realizado con un *multiplexor*
- **Sumador_flags**: Se valida una suma con 4 flags, de acuerdo al resultado. Z indica res = 0, C indica si la suma produjo carry, V indica si la suma produjo (complemento a 2) overflow, N indica si la suma produjo (complemento a 2) un negativo
- **ALU**: En AND y OR; C = 0, V = 0
  - Suma (OP 00): operación normal de suma con 2 inputs
  - Resta (OP 01): se realiza la operación A - B $\equiv$ A + $(\neg B + 1)$. El C (que debe devolver el *borrow*) se invierte ya que $Borrow \equiv \neg Carry$ en complemento a 2
  - AND (OP 10): se realiza un AND bit a bit
  - OR (OP 11): se realiza un OR bit a bit



Circuitos realizados en el taller 1

2 Taller 2

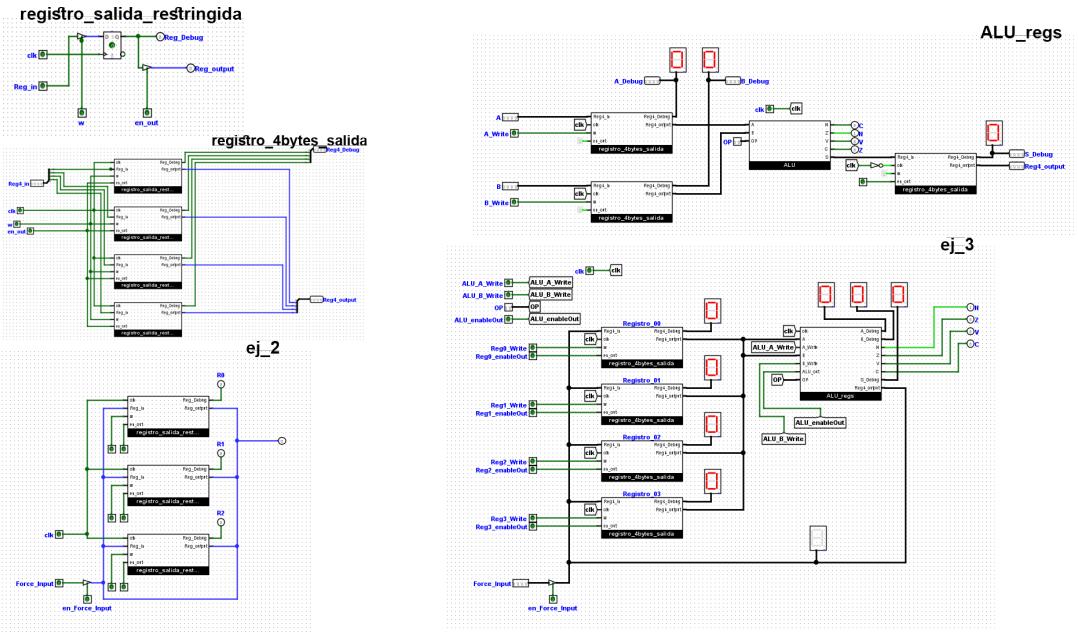
2.1 Idea general

Idea similar al Taller 1 de implementación de un ALU para operaciones básicas, pero agregando más cosas. El objetivo fue evaluar el funcionamiento y la interacción entre registros, señales de control y las operaciones de la ALU, el manejo de *flujo de datos*.

2.2 Componentes

- **registro_salida_restringida**: Recibe una señal de entrada (*reg_in*), una señal de escritura (*w*) que permite la escritura del valor cuando hay flanco de subida de *clk* y una señal de habilitación de salida (*en_out*). Se controla cuándo se guarda un valor y cuándo se lo devuelve.
- **registro_4bytes_salida**: Recibe un dato de 4 bits y separa bit a bit, guardando el valor si $clk = \uparrow$ (ó 1) y $w = 1$. *clk* y *w* son señales de control comunes a todos los bits. El valor del bus es devuelto (Reg4_output) $\iff$ $en\_out = 1$
- **ALU_regs**: Recibe 2 registros para operandos A y B, cada uno controlado por su A_write y B_write. Se define qué operación es devuelto y almacenado según el valor en *OP*.
- **ej_02**: Recibe 3 registros (R0 a R2). Se carga un valor a un registro y luego se transfiere entre registros a través de un bus de datos compartido, utilizando señales de control (y *clk*).
  - **Force_Input**: Se pasa un input para iniciar el pasaje de datos
  - **en_Force_Input**: Habilita que se tome el valor de Force_Input o desde la salida de un registro random ("enable force input").
  - **clk**: Habilita la escritura en el flanco de subida ($\uparrow$ ó 1)
  - **w**: Habilita la escritura en cada reg
  - **en_out**: Habilita que el valor del registro se mande al bus
- **ej_03**: Recibe 4 registros (R0 a R3).

  Todos los items anteriores (del ej_02) son válidos y podrían ir acá. Abajo enumero los que no describí antes
  - **RegX_write**: Habilita escritura
  - **RegX_EnableOut** Habilita la salida al bus
  - **ALU_A_Write y ALU_B_Write**: Valor de los regs A y B
  - **OP**: Código para seleccionar la operación
  - **ALU_EnableOut**: Habilita la salida del res del ALU al bus
  - **Acá entra en juego la importancia del $\neg clk$ descripta en el punto de la teoría**



Circuitos realizados en el taller 2

3 Anotaciones extra

3.1 Interpretación de datos

⋆ Se tienen n bits

- **Sin signo**
  - **Rango de representación**: desde 0 a $(2^n - 1)$ | de 0 a 15 en 4 bits
  - Se suma y resta como en decimal
  - No hay negativos, solo positivos
- **Complemento a 2**
  - **Rango de representación**: desde (-2^{n-1}) hasta $(2^{n-1} - 1)$ | de -8 a 7 en 4 bits
  - Hay negativos y dependen del MSB (bit más significativo). MSB = 1 → negativo, MSB = 0 → positivo
  - Para obtener el negativo de un número se invierten todos los bits y se suma 1 (-A $\equiv \neg A + 1$)
- **¿Cuándo se produce *overflow* en complemento a 2?**:
  Cuando se suman dos números con el mismo signo y el res es del signo opuesto.
  Recordar que en este caso A - B $\equiv$ A + $(\neg B + 1)$, así que la regla sigue aplicando

3.2 Componentes implementados

Componente	Función
Buffer	**"Interruptor"**. Tiene *una* entrada y salida, pero solo la habilita el valor según una señal de control
MUX	**Selector de datos**. Recibe *varias* entradas y según el valor de una señal de control elige cuál de esas entradas habilitar a la salida
Clock	**Señal de control**. "Sincroniza" las operaciones de un circuito (les dice cuándo deben activarse, básicamente), según su valor sea 0 (flanco de bajada ↓) ó 1 (flanco de subida ↑). Generalmente los datos son captados en el flanco de subida (1)
ALU	**Realiza operaciones lógicas simples** (suma, resta, AND, OR) y genera flags. Recibe 2 operandos y una señal de control (OP) y devuelve lo asignado según el caso. Incluye todos los componentes anteriores en su implementación