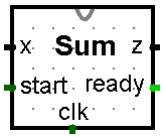
Pregunta 1

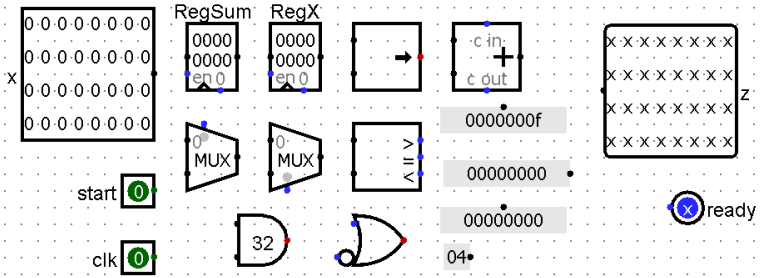


En la entrada *x* (de 32 bits), el circuito *sum* de la derecha recibe 8 números de 4 bits cada uno. Cuando *start* se pone en 1, debe llevar la salida *ready* a 0 y calcular en varios ciclos del reloj la suma de los 8 números. Al terminar el cálculo debe colocar *ready* en 1 y entregar el resultado de la suma en la salida *z* (de 32 bits). Por ejemplo, si *x* es 0x4a9 (en hexadecimal), los números son 0, 0, 0, 0, 0, 4, 10 y 9. La salida *z* debe ser 23 porque 4+10+9 es 23.

Implemente el circuito *sum* **usando un solo sumador** de 32 bits. Use además un registro *RegSum* de 32 bits para ir almacenando las sumas parciales de los 8 números y otro registro *RegX* para almacenar inicialmente *x* y luego desplazarlo en 4 bits en cada ciclo. También necesitará usar multiplexores, un desplazador (shifter), un comparador, un *and* bit a bit con entradas y salidas de 32 bits, más otras compuertas. Use este algoritmo para realizar el cálculo:

```
Siempre
  z = RegSum
  ready = RegX==0
En el pulso de bajada del reloj
  if (start || !ready) {
    RegSum = start ? 0 : (RegSum + RegX & 0xf)
    RegX = start ? x : (RegX >> 4)
  }
```

Recuerde que no puede usar más de un sumador. Torpedo de compuertas:



Pregunta 2

Parte a.- La figura muestra un extracto del estado actual de un *caché* de 4 KB (2^{12} bytes) de 1 grado de asociatividad con 256 líneas de 16 bytes. Por ejemplo en la línea 6b del caché (en hexadecimal) se almacena la línea de memoria que tiene como etiqueta 86b (es decir, la línea que va de la dirección 86b0 a la dirección 86bf).

línea cache	*etiqueta*	*contenido*
9c	79c	
6b	86b	
34	a34	

Un programa accede a las siguientes direcciones de memoria (en hexadecimal): 79c0, 86b4, a34c, 56b0, 79c4, a348, d340, 79c8, 86b8. Indique qué accesos a la memoria son aciertos en el caché, cuáles son desaciertos y rehaga la figura mostrando el estado final del cache. Por ejemplo el acceso 79c0 es un acierto.

Parte b. La tabla de la derecha muestra las instrucciones Risc-V ejecutadas por un programa. Haga un diagrama que muestre el ciclo en que se ejecuta cada etapa de las instrucciones, considerando una arquitectura (i) en pipeline con etapas *fetch*, *decode* y *execute*, y (ii) superescalar, con 2 pipelines con las mismas etapas de (i). Suponga que el salto en E no ocurre y el salto en F sí ocurre (no hay ningún tipo de predicción de saltos).

```
A add  a1,a2,a3
B andi t4,a1,7
C addi t5,t4,1
D add  t6,t4,a4
E blt  a1,t4,I
F beq  t5,a1,P
G add  ...
H sub  ...
I xor  ...
J andi ...
...
P addi a3,a2,1
Q sub  t7,a4,t5
```

Pregunta 3

La función *palindro* del cuadro de la derecha verifica que un arreglo de *n* enteros es palíndrome, es decir que al invertirlo se lee el mismo arreglo. Entrega *n* si *a* es palíndrome. En caso contrario entrega el índice del elemento en donde falla la verificación. Programe la función *palindro_par* con la misma funcionalidad de *palindro*, pero realizando la verificación en paralelo con *p* procesos pesados (creados con *fork*). Recibe los mismos parámetros que *palindro*, más el parámetro *p*, es decir: `int palindro_par(int *a, int n, int p);`

```
int palindro(int *a,
             int n) {
  for ( int i= 0;
        i<n/2; i++) {
    if(a[i]!=a[n-1-i])
      return i;
  }
  return n;
}
```

Ayuda: Subdivida el intervalo [0, *n*/2[en *p* subintervalos y verifique cada subintervalo en uno de los procesos hijos creados con *fork*. No intente llamar a *palindro* en los hijos. No le servirá. Copie el código de *palindro* en *palindro_par* y modíquelo para que sí le sirva. Si uno de los hijos termina tempranamente porque resolvió que el arreglo no es palíndrome, **no** necesita detener el resto de los hijos. Espere pacientemente a que terminen sus verificaciones. **¡Revise que su solución posee paralelismo!**