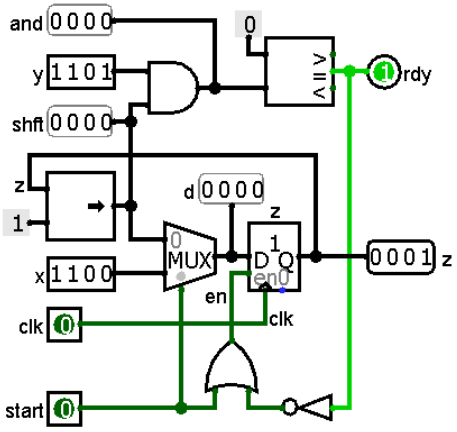
Pregunta 1

I. Considere el circuito de la figura. **Complete** la tabla de abajo con los valores que toma cada señal hasta el ciclo 6 del reloj. El valor de la entrada *x* se mantiene constante en 0b1100 y el de *y* constante en 0b1101. Los valores de *shft*, *d*, *and* y *z* se expresan en la tabla en binario.



II. **Modifique** el circuito agregando una salida *n* de manera que cuando *rdy* vuelve a ser 1, en *n* se indique la cantidad de ciclos en los que *rdy* permaneció en 0. Para el ejemplo de la parte *I*, cuando *rdy* vuelve a 1, *n* debe ser 3 y permanecer constante hasta que *start* se ponga en 1 nuevamente. Dibuje solo la parte que necesita modificar, más una que otra compuerta para mostrar el contexto en donde introduce la modificación.

ciclo	*start*	*shft*	*d*	*en*	*and*	*rdy*	*z*
1	0	0000	0000	0	0000	1	0000
2	1	0000	0000	1	0000	1	0000
3	0					0	1100
4	0						
5	0						
6	0						

Pregunta 2

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 4c del caché (en hexadecimal) se almacena la línea de memoria que tiene como etiqueta b4c (es decir, la línea que va de la dirección b4c0 a la dirección b4cf).

línea cache	*etiqueta*	*contenido*
15	915	
4c	b4c	
e0	5e0	

Un programa accede a las siguientes direcciones de memoria (en hexadecimal): b4c0, 9154, 4e00, 4e08, 7154, b4c0, 4e0c, 14c0. **Indique** qué accesos a la memoria son aciertos en el caché, cuáles son desaciertos y **rehaga la figura** mostrando el estado final del caché. Por ejemplo el acceso b4c0 es un acierto.

B) La tabla de la derecha muestra las instrucciones Risc-V ejecutadas por un programa. Haga diagramas que muestren 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 las 2 pipelines con las mismas etapas de (i). Suponga para (i) y (ii) que el salto en D sí ocurre (y no hay ningún tipo de predicción de saltos).

Programa

```
A and a5,s4,t0
B addi a4,t1,4
C sub a6,t2,a5
D bgt a6,a5, M
E
F
...
M sub a2, s0, a0
N ori a1, a1, 4
```

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

*int palindro_par(int *a, int n, int p);*

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!**