

# App dCir, simulador de circuitos digitales

## Tutorial

### Introducción

Este simulador está concebido con propósitos docentes. Su objetivo es profundizar en el conocimiento básico de los sistemas digitales.

La simulación se focaliza en el hardware, concretado en el dibujo de un circuito. Este dibujo puede generarse de dos formas: usando los propios recursos de la aplicación y con cualquier otro procedimiento disponible: a partir de aplicaciones gráficas o, simplemente y más adecuado, a partir de la fotografía de un dibujo realizado a mano alzada. Los formatos aceptados son .png y .jpg. Los elementos importantes del dibujo son los bloques constitutivos (puertas, flip-flops, etc.) y sus interconexiones. Cualquier otro elemento se considera estético. Posteriormente, si se hace uso de un dibujo externo, se debe trasladar la información en él contenida a la aplicación, antes de poder proceder a su simulación. Durante la simulación, se visualiza cómo fluye la información por el circuito, con una atención especial a su velocidad de propagación.

### Generación del proyecto

Para simular un circuito, la primera acción consiste en hacer que la aplicación lo reconozca. Esto se consigue generando un proyecto.

Luego de llamar a la aplicación (dCir.exe) aparece una ventana con el menú principal (fig.1). El desplegable *Project* contiene varias opciones de inicio (fig.2): se puede abrir un proyecto previamente creado, guardar un proyecto actual, crear uno nuevo o terminar. Al seleccionar *new*, aparece la fig.3. Debe definirse el directorio que alojará al proyecto, el nombre del fichero del proyecto, el fichero que contiene el dibujo con el circuito digital y diferentes parámetros de operación. El nombre del proyecto puede evitar la extensión pero, de darse, debe ser .prj. El fichero gráfico no es opcional, aunque puede tratarse de una imagen en blanco.

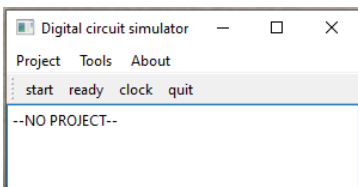


Fig.1: ventana principal

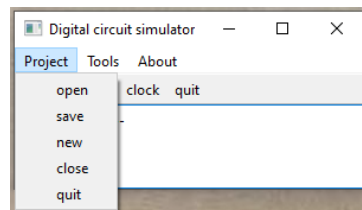


Fig.2: menú del proyecto

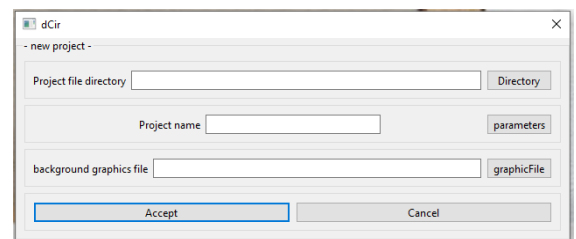


Fig.3: detalles del proyecto

Para ilustrar el proceso de creación de un nuevo proyecto, hacemos uso de un simple circuito con un inversor de 4 bits (fig.4). En este caso, el circuito ha sido generado con las herramientas gráficas de Word.

Para trasladar a *dCir* la información contenida en este circuito procederemos como sigue:

- 1) **Caminos** por los que fluye la información. Se trata de establecer los cables que sirven de nexo entre módulos.
  - a) Clicar el botón derecho sobre la ventana del circuito. Aparece la fig.5.
  - b) Clicar botón izquierdo sobre *Paths*; aparece la fig. 6
  - c) Establecer la cantidad de bits asociados al camino y clicar sobre *open*.
  - d) Clicar botón izquierdo sobre el inicio del camino, arrastrar hasta el posible cambio de dirección, soltar, volver a clicar, arrastrar, etc. hasta llegar al destino (fig.7).
  - e) Clicar el botón derecho y seleccionar *close* para terminar el camino, si el camino es correcto, sino clicar *cancel* (fig.6).

Este procedimiento se puede agilizar. Clicando *path\_mode* en la barra de herramientas se omiten los dos primeros pasos.

Una vez cerrado un camino, sobre él aparece la información  $p(x, y)$ . Se debe interpretar como *path*(identificador, nº de bits). Se puede actualizar su número de bits así como eliminarlo del circuito (fig.6). Para ello, se requiere de su identificador.

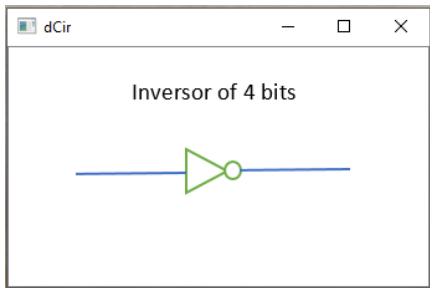


Fig.4: inversor de 4 bits

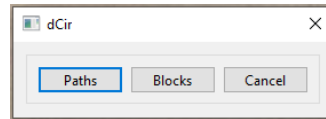


Fig.5: creación de caminos

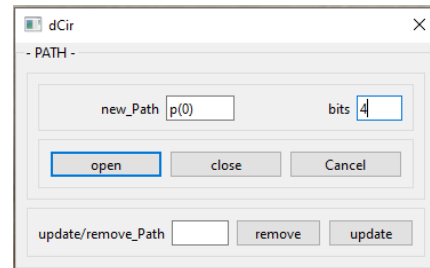


Fig.6: apertura de un camino de 4 bits

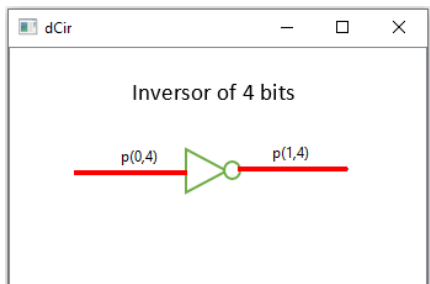


Fig.7: caminos establecidos

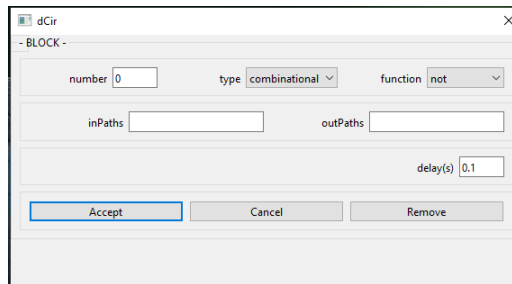


Fig.8: inserción de un bloque

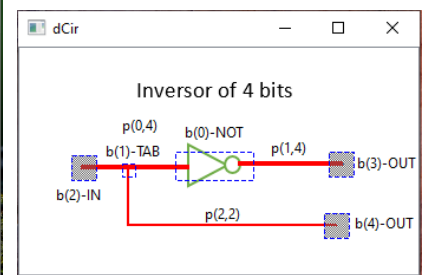


Fig.9: proyecto concluido

- 2) **Bloques** que procesan la información. Una vez establecidos los caminos, se deben insertar los bloques.
  - a) Clicar el botón derecho sobre la ventana del circuito. Aparece la fig.5.
  - b) Clicar botón izquierdo sobre *Blocks*; aparece la fig. 8. En el desplegable *type* se dispone de los tipos de bloques *combinational*, *sequential*, *node*, *portIn*, *portOut*, *tap* y *memory*. Para cada tipo de bloque, se deben definir diferentes parámetros. En *inPaths* y *outPaths* se deben dar los números de referencia de los caminos que entran y salen del bloque, separados por comas. En *delay* se indica el tiempo de retardo del bloque, en segundos. Los tipos *combinational* y *sequential* disponen de subtipos.
  - c) Cuando todos los parámetros del bloque están definidos y aceptados, sobre el dibujo del bloque aparece un rectángulo en cuyo interior se indica la referencia al bloque y su tipo.

Este procedimiento se puede agilizar. Clicando *block\_mode* en la barra de herramientas se omite el primer paso.

Los caminos que introducen información al circuito deben incluir el bloque *portIn*, los caminos que extraen información incluyen el bloque *portOut*. Las derivaciones en los caminos pueden ser de los tipos *node* y *tap*. El tipo *node* tiene un camino de entrada y varios caminos de salida, como separador de caminos, o viceversa, como concentrador. El tipo *tap* sirve para derivar un camino secundario de otro principal. En cualquier caso, deben indicarse los bits que componen los caminos. Por ejemplo, si de un camino principal de 4 bits ( $p(0,4)$ ) se deriva el camino de 2 bits ( $p(2,2)$ ), como *pathsIn* se indica 0 (ó 0[3:0]) y como *pathsOut* 2[2:1] si se desean extraer los bits 2 y 1 del camino principal. No se requiere especificar los bits si ambos caminos son del mismo número de bits (fig.9). Tampoco es necesario que el camino secundario parta de un lugar próximo al camino principal. Ello hace posible mejorar la estética del circuito al evitar cruces de caminos.

Con estos dos pasos el proyecto queda concluido y puede guardarse en un fichero con extensión .prj. Se trata de un fichero de texto plano que puede editarse. Ahora ya se puede proceder a la simulación clicando sobre el botón *start* (fig.10). Por ejemplo, si introducimos el valor 0x6 en el bloque de entrada (clic derecho sobre el bloque -> *init value=0x6*) puede apreciarse que la salida contiene el valor inverso (0x9) y que el camino inferior adquiere el valor de los bits intermedios (0x3). La simulación se puede detener/continuar en cualquier momento actuando sobre el botón *paused/ready*.

**NOTA:** los bloques requieren de una separación mínima.

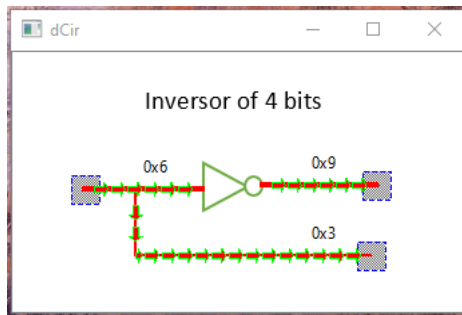


Fig.10: simulación

En *Tools->configuration* se pueden definir ciertos parámetros tales como el tipo, color y velocidad de avance de las flechas, qué información se visualiza, etc.

El botón *clock* del menú principal es útil para los circuitos secuenciales. Genera un flanco ascendente que tiene como consecuencia que los *flip-flops* del circuito adquieran la información de entrada de forma síncrona.

Un subtipo de bloque combinacional es **others**. Permite definir cualquier bloque combinacional. Para ello, solicita un fichero de texto plano que incluye su tabla de la verdad (fig.11).

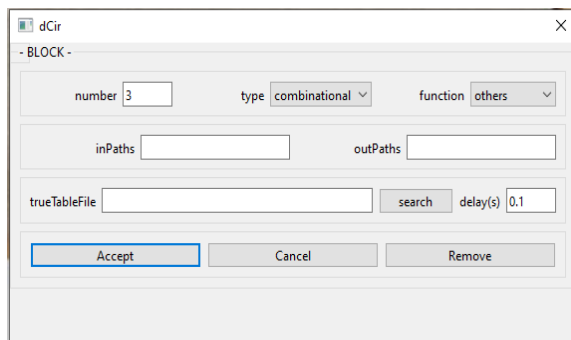


Fig.11: bloque combinacional genérico

#input	output
0b0x0	0b10
0b0x1	0b00
0b10x	0b00
0b110	0b11
0b111	0b01

Fig.12: Ejemplo de bloque combinacional genérico con tres bits de entrada y dos bits de salida

El fichero debe tener extensión .tt (true table) y contiene dos columnas: entradas, a la izquierda, y salidas, a la derecha. Se acepta el carácter 'X' como indicador de indiferencia y los formatos numéricos: binario (0b...), hexadecimal (0x...) y decimal. El carácter '#' denota línea de comentario. Deben estar contempladas todas las combinaciones posibles de la entrada (fig.12).

Un subtipo de bloque secuencial es **FSM**. Permite definir cualquier máquina de estados finita. Para ello, solicita un fichero de texto plano que incluye su tabla de la verdad (fig.13).

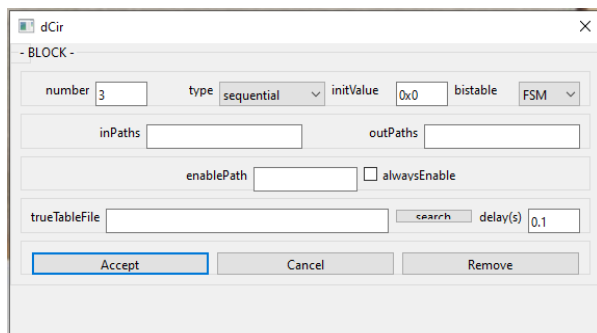


Fig.13: bloque máquina de estados finita

#SIMPLE MACHINE FSM			
#status	input	status+	output
0x0	0bXXX	0xC	0x0018
0x6	0bX0X	0x7	0x6304
0x6	0bX1X	0x9	0x6304
0x7	0bXXX	0x0	0x7321
0x9	0bXXX	0x0	0x9041
0xA	0bXXX	0x0	0xA321
0xB	0bXXX	0xC	0xB318
0xC	0b110	0x0	0xC202
0xC	0b0XX	0x6	0xC202
0xC	0b10X	0xA	0xC202
0xC	0b111	0xB	0xC202

Fig.14: Ejemplo de máquina de estados

Es similar al bloque combinacional *others* solo que contiene cuatro columnas: estado\_actual, entradas, estado\_futuro y salidas, por ese orden y de izquierda a derecha (fig.14). Para cada estado deben estar contempladas todas las combinaciones posibles de las entradas. Se dispone del camino de entrada *enablePath* para permitir o inhibir su operación.

Otro bloque considerado es **memory** (fig.15). Como antes, se solicita un fichero de texto plano, con extensión .tt, que contiene dos columnas: dirección y datos.

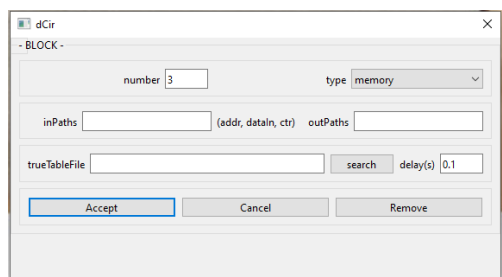


Fig.15: bloque memoria

El bloque *memory* dispone de tres caminos de entrada: dirección (*addr*), datos (*dataIn*) y control (*ctr*, para lectura/escritura) y uno de salida (*outPath*). La escritura está sincronizada con el botón *clock*, pero no la lectura.

Si la memoria debe incluir el código máquina de un programa, existe la opción *tools->assembler* para ensamblar un fichero de texto plano que incluya el código fuente (fig.16). Esta herramienta parte de un fichero .asm, que incluye al código fuente, y genera dos ficheros con extensiones .dasm y .tt. El primero de ellos incluye un desensamblado del código fuente y el segundo es el fichero que debe incluirse en el bloque *memory* (fig 17). Existe la opción de visualizar en tiempo real el estado de la memoria con *tools->viewMemory*.

```

;Algoritmo en lenguaje de programación de bajo nivel
; (Assembler específico para este µP)

A      equ VAR+0    ;Declare A=0x64, B=0x65, C=0x66, I=0x67
B      equ VAR+1
C      equ VAR+2
I      equ VAR+3
ONE    equ CONST+0  ;Declare ONE=0x68, ZERO=0x69
ZERO   equ CONST+1

.org 0    ;program-memory starts at address 0 (0x0)
begin:   MOV ZERO,C   ;C=0
        MOV ZERO,I   ;I=0
loop:    CMP I,B      ;FZ=1 if I=B
        BEQ end      ;if FZ=1 (I=B) then go to end (end address at 0x08)
        ADD A,C       ;C=C+A
        ADD ONE,I     ;I++
        CMP ZERO,ZERO ;FZ=1 always since ZERO=ZERO
        BEQ loop      ;loop is repeated B times (loop address at 0x02)
end:     ...          ;Contains the result

.org 100          ;data-memory starts at address 100 (0x64)
VAR:     .word 11,5   ;Memory allocated for variables A=11,B=5
.org 104          ;data-memory starts at address 104 (0x68)
CONST:   .word 1,0    ;Memory allocated for constants ONE=1,ZERO=0

```

```

#SIMPLE MACHINE MEMORY
#CODE
#addr      Contain
0x00       0xB4E6
0x01       0xB4E7
0x02       0x73E5
0x03       0xC008
0x04       0x3266
0x05       0x3467
0x06       0x74E9
0x07       0xC002

#DATA
0x64       0x000B
0x65       0x0005
0x68       0x0001
0x69       0x0000

```

Fig.16: Ejemplo de código ensamblador