Guía para las herramientas del laboratorio

Introducción

A través de esta guía aprenderás a utilizar las herramientas para programar y depurar un sistema basado en un microprocesador Z80 compatible.

Mediante un ejemplo sencillo veremos cómo realizar las siguientes tareas:

1. Ensamblar un programa.
2. Poner en marcha el sistema procesador Z80 incluyendo memoria y puertos, en dos alternativas:
   2.1. Utilizando un simulador del procesador Z80 que se ejecuta en el PC.
   2.2. Utilizando un sistema real cargado en la placa DE0.
3. Cargar y probar el programa con la ayuda de un debugger que corre en el PC.

Antes de Comenzar...

Antes de comenzar es necesario descargar y guardar en un directorio adecuado, el paquete de herramientas *z80-tools* de la página web del curso, que contiene todos los programas necesarios para programar y probar los diseños realizados. Además, para trabajar con la placa DE0 se requiere tener instalado el Quartus II, versión 9.0 o superior con las licencias correspondientes. En todos los casos se recomienda tener instalado también el editor de texto Notepad++[1] versión 6.3 o superior. Para más información sobre la instalación del Quartus II, remitirse a la *guía del curso Diseño Lógico (Eva de Diseño Lógico: sección Laboratorios → Guía para realizar el primer diseño con Quartus II)*.

El paquete de herramientas incluye:

- *binutils-z80*: Esta es la herramienta de compilación y linkeado para traducir el programa en lenguaje assembler al código de máquina que interpreta el Z80 y que debe cargarse en la memoria del procesador.
- *gdb-z80*: GNU Debugger para el sistema Z80 que usamos en el curso. Permite

1 Notepad++ es un editor de texto para Windows de código abierto que se puede descargar de: http://notepad-plus-plus.org/ . Se recomienda utilizar este editor para escribir los programas en assembler.

ejecutar nuestros programas en forma controlada e inspeccionar el estado del sistema Z80.

- *jtagcon*: Programa auxiliar encargado de realizar la comunicación entre la PC y el microprocesador dentro de la placa DE0. Se utiliza para eso la misma conexión USB a través de la cual se configura el chip FPGA con el conexionado del sistema (solamente para placa DE0).
- *lab-intup-de0*: Este directorio incluye un proyecto de Quartus que contiene el sistema con el procesador, una memoria ROM con un programa de inicialización (monitor.hex), memoria RAM, y algunos puertos de entrada salida.

  (solamente para placa DE0).
- *QEMU-z80*: Un simulador de Z80 basado en el emulador QEMU.
- *scripts-notepad++*: Archivos de configuración del editor Notepad++ que permiten utilizarlo para ejecutar todas las herramientas anteriores sin la necesidad de un intérprete de comandos. Es importante destacar que Notepad++ no es necesario para correr las herramientas de desarrollo, pero sí es altamente recomendado.

Configuración previa de Notepad++

Se agregarán comandos al menú *Macro* de Notepad++ para poder invocar scripts de compilación desde dentro del editor. Asimismo se agregará la definición de sintaxis coloreada para poder visualizar en forma más amigable programas en lenguaje ensamblador de Z80. Para poder utilizar los scripts de configuración, se requiere la instalación previa del *plugin NppExec* utilizando el *Plugins Admin* de Notepad++.

Instalación de plugin NppExec mediante Plugins Admin

El Plugins Admin se encuentra en el menú de plugins del Notepad++ *(Plugins → Plugins Admin)*.

Al abrir el *Plugins Admin*, este actualiza y muestra la lista de plugins disponibles en el repositorio de plugins de Notepad++. Se debe seleccionar *NppExec* de la lista e instalarlo. (Si no se ve el NppExec en la lista de plugins disponibles, es probable que ya esté instalado, confirmarlo en la pestaña *Installed*).

Instalación de los scripts

El paquete de herramientas incluye los archivos `NppExec.ini` y `npes_saved.txt`. Estos dos archivos deben copiarse en el directorio `"Datos_de_aplicacion"\Notepad++\plugins\config` donde se guarda la configuración del plugin *NppExec*, sobrescribiendo, si existieran, los archivos del mismo nombre (respaldarlos previamente).

La ubicación del directorio `"Datos_de_aplicacion"` varía dependiendo de la versión

de sistema operativo Windows utilizada. Para Windows 7 o superior es:

```
C:\Users\nombre_usuario\AppData\Roaming\Notepad++\plugins\config
```

En todos los casos dicha ruta puede averiguarse ejecutando el comando `echo %APPDATA%` en una consola de comandos de Windows. Por ejemplo, para el usuario "Julio" en Windows 10:

```
C:\Users\Julio>echo %AppData%
C:\Users\Julio\AppData\Roaming
```

La carpeta *AppData* es una carpeta oculta, por lo que por defecto no aparece en el explorador de archivos. Para acceder a ella se presentan 2 alternativas:

1. Marcar *"Mostrar archivos ocultos"* en opciones de Vista del explorador de archivos.
2. Tocando 2 veces sobre la barra superior donde se muestra la ruta actual se puede escribir la ubicación `%APPDATA%` para abrir la carpeta en cuestión.

Configuración de los scripts

Debido a que los scripts deben saber dónde se encuentran las herramientas del curso (carpeta `z80-tools`) y el programa Quartus de Altera, es necesario modificar el archivo `npes_saved.txt` antes de poder utilizar los scripts.

Para esto se debe abrir el archivo `npes_saved.txt` y cambiar las siguientes líneas:

```
// Directorios de Herramientas
SET Z80TOOLS_DIR = C:\z80-tools
SET QUARTUS_PATH = C:\Program Files\altera\90\quartus

SET BINUTILS_DIR = $(Z80TOOLS_DIR)\binutils-z80
SET QEMU_DIR = $(Z80TOOLS_DIR)\qemu-z80
SET GDB_DIR = $(Z80TOOLS_DIR)\gdb-z80
SET JTAG_DIR = $ (Z80TOOLS_DIR)\jtagcon
```

Cada una de estas líneas debe apuntar al directorio donde se encuentra cada una de las herramientas del curso. Si no se modificó la estructura de directorios, las únicas líneas que hay que modificar son las dos primeras (marcadas con azul).

Una vez completados los pasos anteriores **se debe reiniciar (cerrar y abrir) Notepad++**.

Luego de reiniciado deben aparecer seis nuevos elementos en el menú *Macro* de

Notepad++ que usaremos para la ejecución de comandos, como se explica más adelante.

Instalación del resaltado de sintaxis

Notepad++ cuenta también con un resaltado de sintaxis que permite resaltar las palabras reservadas del lenguaje para que el programa que estemos editando sea más fácil de leer. Este resaltado de sintaxis se especifica en el archivo "`z80-tools\Notepad_pp_scripts\z80.xml`" y debe ser cargado como lenguaje de usuario en Notepad++. Es importante notar que el archivo `z80.xml` anterior funciona **solamente en la versión de Notepad++ 6.2 o superior**. Para versiones anteriores se puede utilizar el archivo `z80_npp_anteriores_a_6_2.xml` suministrado en el mismo directorio.

Para instalar el resaltado de sintaxis se debe:

1. Configurar idioma inglés para la interfaz de usuario (si no estaba ya configurado). *(Configuración → Preferencias → Generales → Localización)*
2. Abrir la interfaz para configurar los lenguajes definidos por el usuario *(Language → User Defined Language → Define your language)* y presionar el botón *"Dock"*.
3. Importar el archivo con la definición del lenguaje ensamblador Z80: Presionar el botón "*Import*", navegar hasta el archivo "`Notepad_pp_scripts/z80.xml`" y seleccionarlo.
4. Presionar el botón *"Undock"* y cerrar la ventana con la interfaz de configuración de lenguajes definidos por usuario.
5. Si lo desea vuelva a configurar idioma español para la interfaz de usuario. *(Settings → Preferences → General → Localization)*

Como fue mencionado anteriormente, no es necesario utilizar el Notepad++ para utilizar las herramientas del curso. Se recomienda analizar los comandos invocados por cada uno de los macros de Notepad++ (Anexo 4) para comprender mejor el funcionamiento de las herramientas.

En lo que sigue, se explica cómo ejecutar los programas de desarrollo desde Notepad++.

1-Ensamblado del programa.

El ensamblado es el proceso de traducción de un programa fuente escrito en lenguaje ensamblador (comprensible por humanos) a código de máquina (binario, comprensible por el microprocesador).

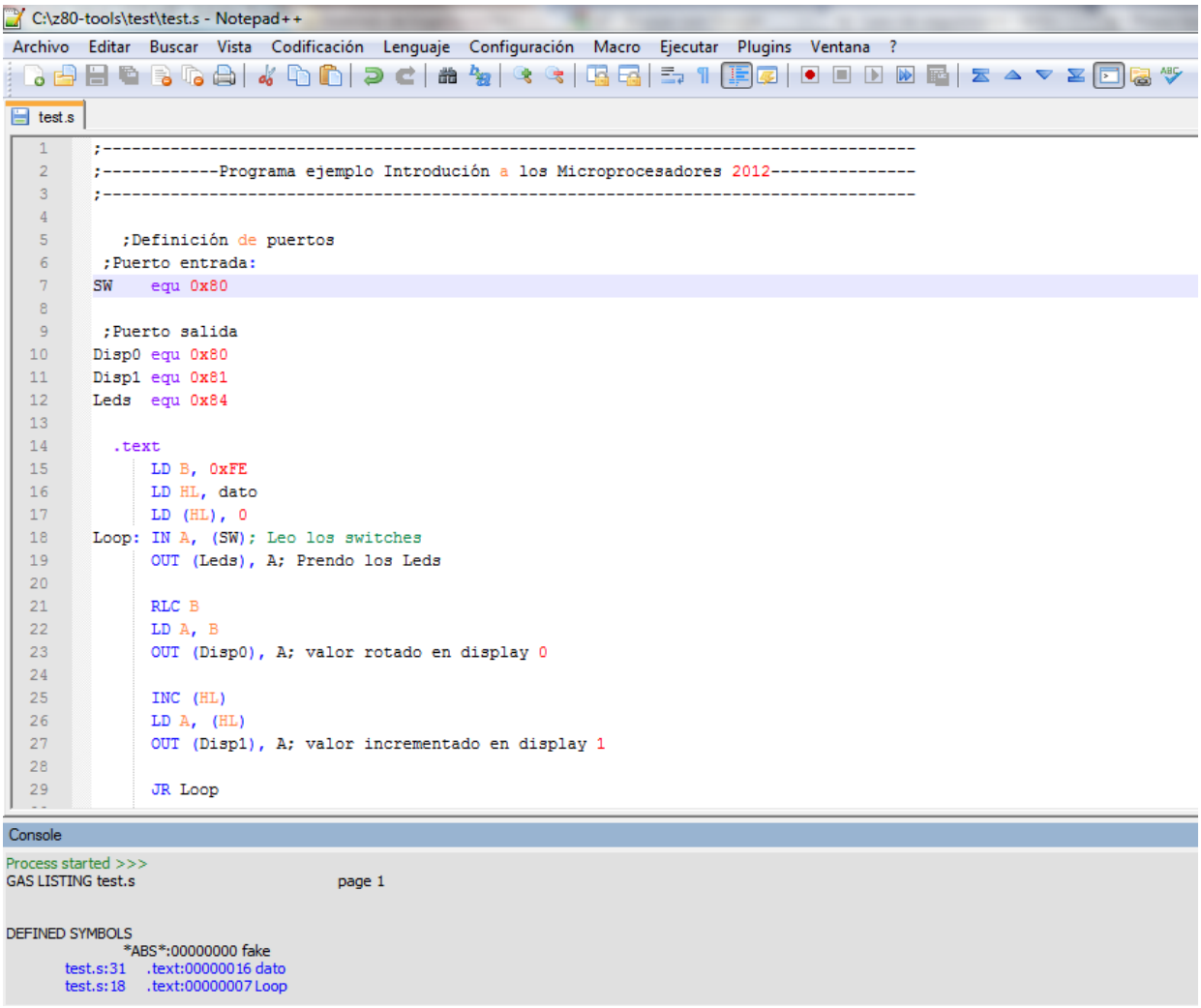
El código fuente en lenguaje ensamblador que se desee compilar debe estar en un archivo con extensión `.s`. En nuestro caso partiremos del ejemplo `test.s` incluido en el directorio `z80-tools\test`.

Para crear un nuevo programa basta con escribir el código utilizando un editor de texto y

guardar el archivo con extensión `.s`.

Para ensamblar un programa se deben seguir los siguientes pasos:

1. Abrir en Notepad++ el archivo con código fuente `test.s`
2. Una vez abierto el código, ir a *Macro* → *Compilar.* Este macro realiza la traducción en dos etapas: el comando `z80-coff-as` que ensambla el código fuente y genera el archivo `test.o`; seguido por el comando `z80-coff-ld` que "*linkea*" el archivo `test.o` y genera el archivo binario final `test`.
3. Revisar errores del assembler o el linker y volver al paso 2 de ser necesario.



Los resultados del ensamblado se muestran en la consola del NppExec. En caso de errores o advertencias, éstas se resaltan en rojo, mientras que algunas de las salidas de interés se resaltan en azul.

En caso en que haya algún error de sintaxis, estos son reportados por el ensamblador de

la siguiente forma:

```
Nombre archivo: Assembler messages:
Nombre archivo: n° línea: Error: Tipo de Error
```

Por ejemplo:

```
test.s: Assembler messages:
test.s: 15: Error: Unknown instruction 'outo' (la instrucción outo
no existe, es OUT)
```

En el caso en que haya error de linkeo, obtenemos mensajes de la siguiente forma:

```
Nombre archivo: C:\directorio\archivo.s: n° línea: Tipo de Error
```

Por ejemplo:

```
test.o: C:\z80-tools\test\test.s:15: undefined reference to 'Ledso'
(La etiqueta debe ser Leds)
```

El programa fuente debe tener la línea:

```
.text
```

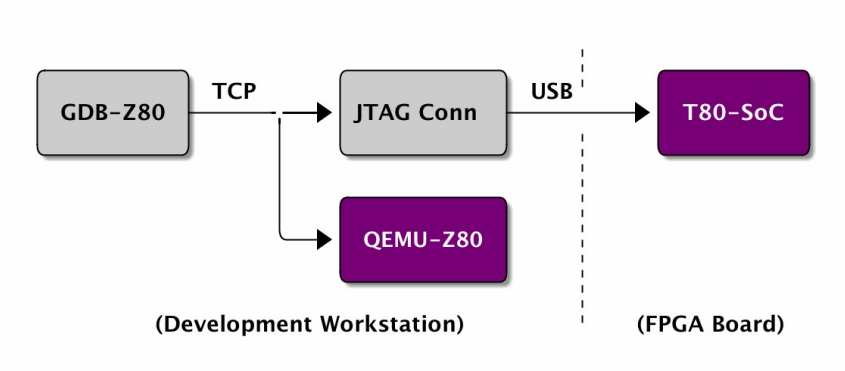
antes del comienzo de las instrucciones.

Cuando se invoca el linker con el comando `z80-coff-ld`, se le indica que nuestro programa debe ser cargado a partir de la dirección `0xB000` y que la primera instrucción de nuestro programa está también en esa dirección (opción `-Ttext 0xB000` y `-e 0xB000` en la invocación a `z80-coff-ld`)

Al final del tutorial se presentan enlaces que llevan a la documentación del assembler y del linker.

2-Puesta en marcha del sistema procesador Z80 en conjunto con el debugger gdb

Una vez compilado exitosamente nuestro programa, vamos a necesitar alguna plataforma sobre la cual probar el funcionamiento de nuestro código. Para esto tenemos dos opciones: una es trabajar con el sistema cargado en la placa DE0 del curso y la otra es trabajar con un simulador del Z80 en nuestro PC (Development Workstation en la figura).



Nota: Tener en cuenta que la Tarea 1 del curso se realizará sobre el simulador del Z80, mientras que los laboratorios se realizarán sobre la placa DE0, por lo que se recomienda en una primer instancia, aprender a utilizar el simulador y luego, para los laboratorios, aprender a utilizar la placa DE0.

2.1- Trabajando con el simulador en el PC

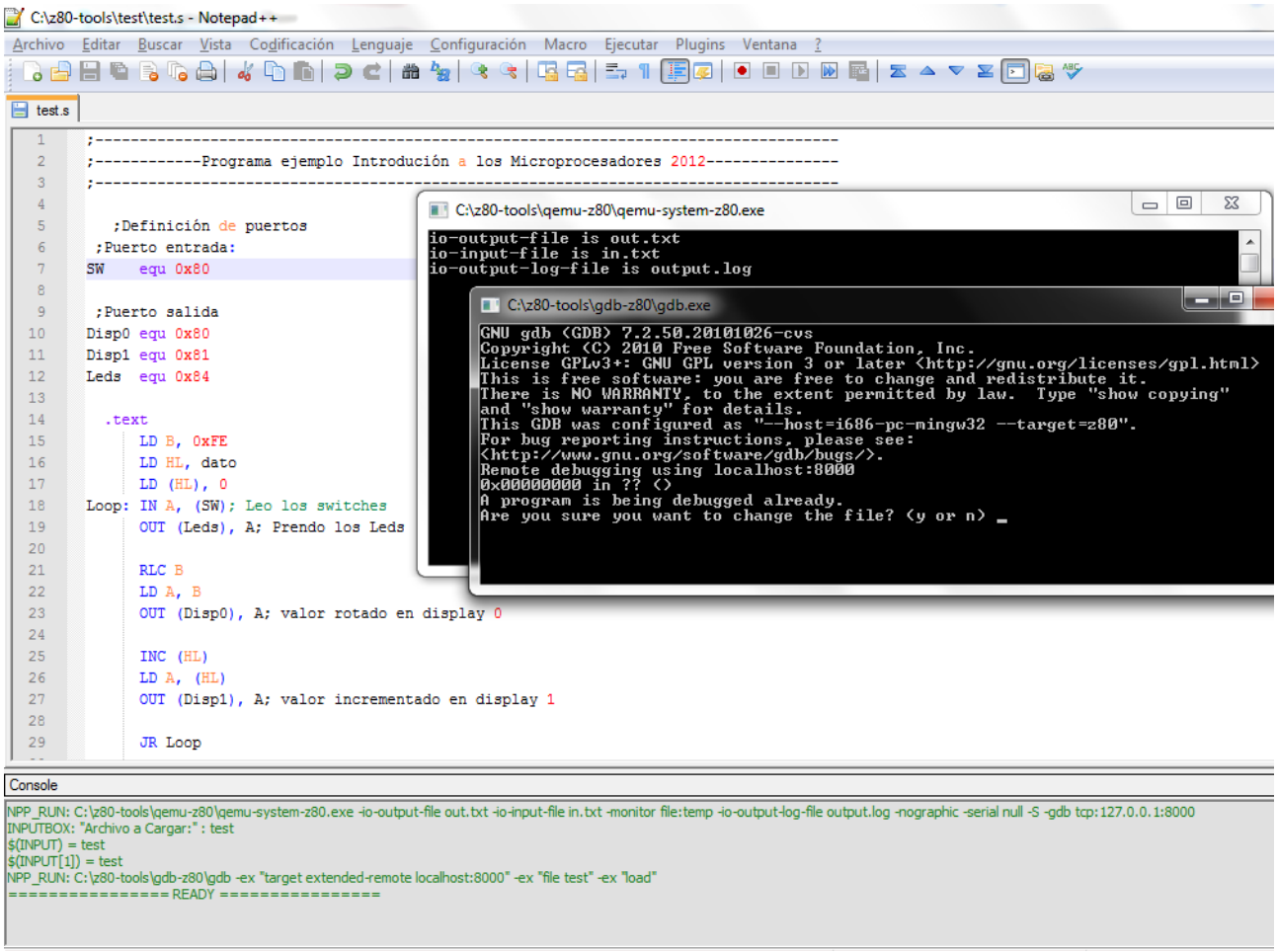
Para correr dicho simulador junto con el debugger gdb, simplemente hay que ir a Macro → QEMU + Gdb. Este proceso arranca el simulador del Z80 que se abre en una ventana y queda a la espera de comandos por parte del debugger, que a su vez, se ejecuta en otra ventana.

Además de arrancar el simulador y el debugger, el macro carga en la memoria del sistema Z80 simulado el programa que esté abierto en el Notepad++, que debe haber sido previamente ensamblado con éxito.

Cuando comienza a ejecutarse el gdb se muestra el siguiente mensaje:

```
A program is being debugged already
Are you sure you want to change the file? (y or n)
```

Seleccionar `yes (y)` y dar enter para que gdb termine de cargar el programa y quede listo para ser utilizado.



Como se aprecia en la figura, una vez completada la secuencia anterior, además de la ventana original de Notepad++, quedarán abiertas dos ventanas más: una con la línea de comandos del debugger gdb en la que normalmente se trabajará, y otra con la consola del simulador QEMU a través de la cual se nos despliega la información intercambiada con los puertos de entrada/salida.

El simulador emula el comportamiento de un procesador Z80 con 16KB de ROM `(0x0000-0x3FFF)` y 48KB de RAM `(0x4000-0xFFFF)`. Los puertos de entrada y salida se emulan a través de archivos de texto en disco.

Puertos de entrada: cada vez que el simulador ejecuta una instrucción IN, en lugar de tomar el valor a leer desde el dispositivo físico (inexistente en este caso), lo toma desde un archivo de texto de nombre `in.txt` en el mismo directorio donde está el programa fuente `(test.s)`. Cada línea del archivo `in.txt` contiene una dupla “dirección: valor”, p. ej. `0x90: 0x0F`, que especifica que el valor a leer en la dirección `0x90` de entrada debe ser `0x0F`.

Puertos de salida: cada vez que el simulador ejecuta una instrucción OUT, éste escribe el valor de salida en un archivo `out.txt`. El archivo `out.txt` contiene 256 lineas, una por cada puerto de salida del Z80. Además la dirección de salida accedida y el valor escrito

en el puerto se despliegan en la consola del programa QEMU, que fue abierta automáticamente por el por el Notepad++ al invocar "*Macro -> QEMU + gdb*". También se agrega un renglón con la misma información al final del archivo `outfile.log` que se crea al ejecutar el QEMU.

2.2 Trabajando con un sistema real cargado en la placa DE0

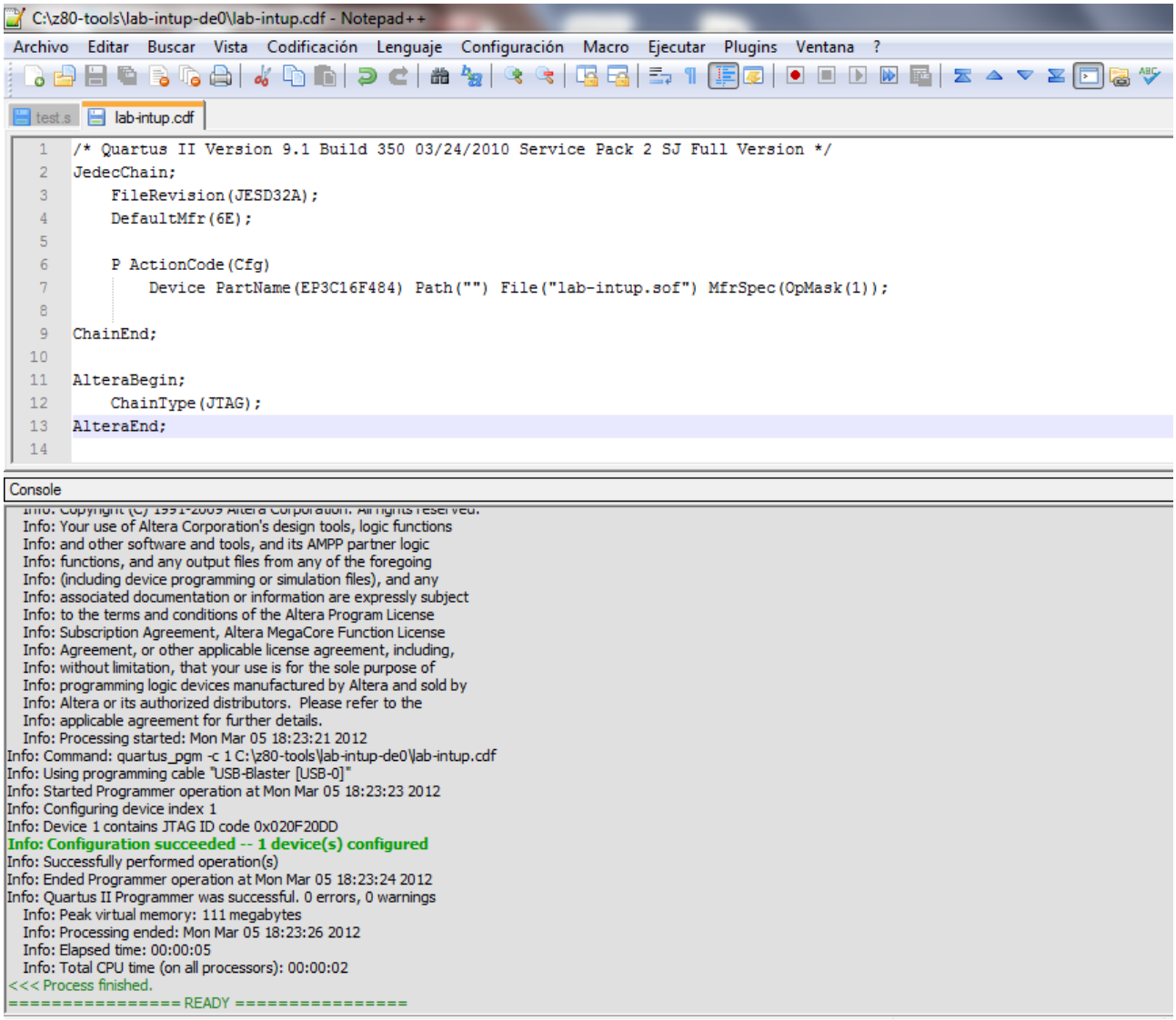
2.2.1 Grabar el conexionado del sistema dentro del chip de la placa DE0

El circuito con el sistema completo (procesador Z80, memorias y puertos) debe ser grabado dentro del chip FPGA de la placa DE0 antes de poder ser utilizado. Para ello se debe:

1. Conectar la placa DE0 y encenderla.
2. Abrir en Notepad++ el archivo `.cdf` [2] que se encuentra en el directorio `lab-intup-de0`. (Verificar que el archivo `.cdf` contenga el nombre del archivo `.sof` que se encuentra en el mismo directorio y que su sección path esté vacía.)
3. Seleccionar *Macro → Grabar sistema en FPGA* en el menú de Notepad++.

En caso de presentarse errores, las líneas correspondientes se resaltan en rojo, mientras que en caso de éxito, alguna línea de interés se resalta en verde.

2 El archivo .cdf se puede abrir con el Notepad++ para editarlo, no es más que un archivo de texto con rutas y comandos para el Quartus II.



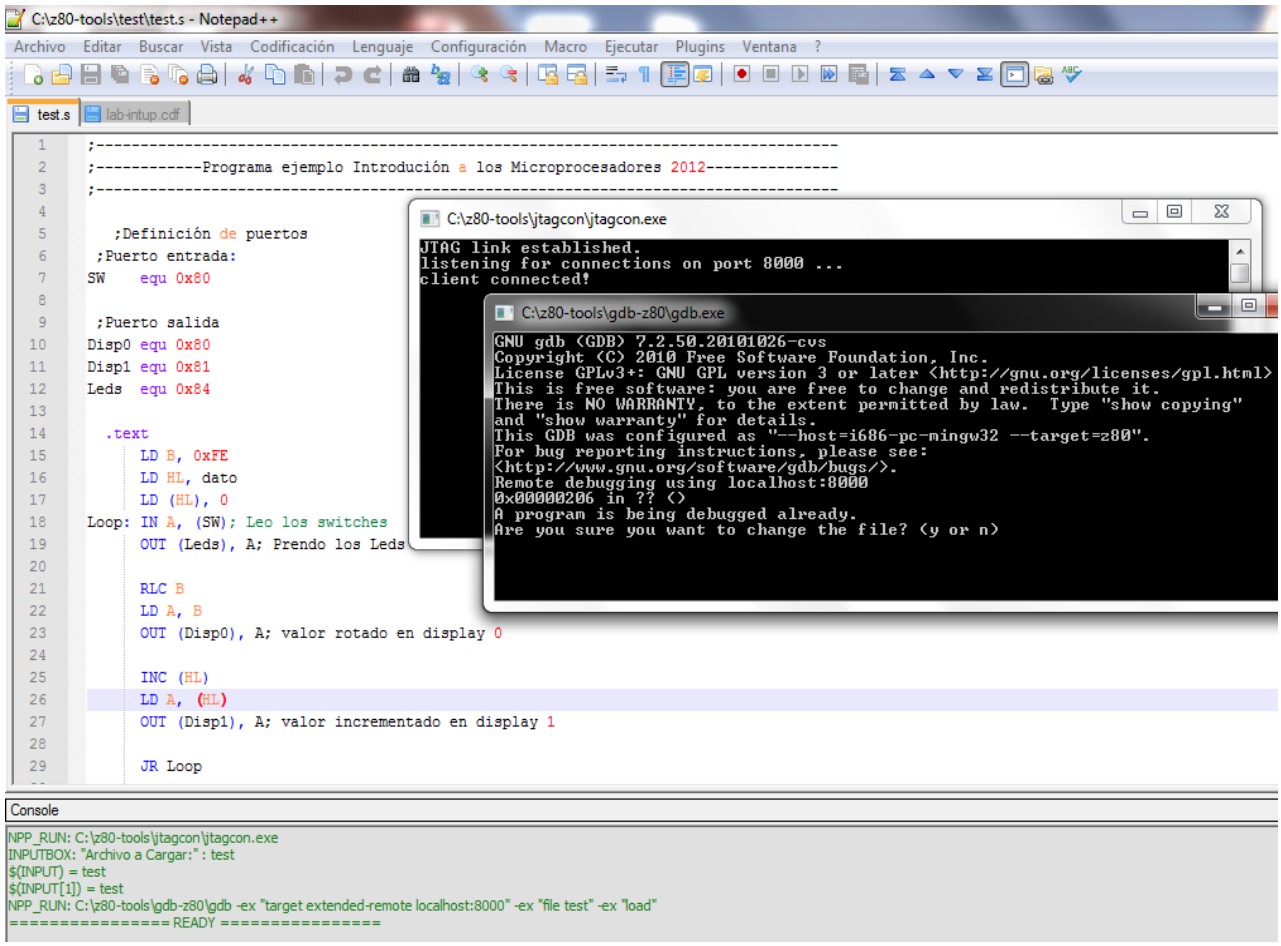
Al programar la placa, se instancia en el FPGA un sistema completo formado por un procesador Z80 más memorias y puertos junto con el programa `monitor.hex` grabado en su memoria ROM. El programa monitor se comienza a ejecutar luego de un reset del Z80 y queda a la espera de comandos por parte del debugger para cargar nuestro programa en memoria y ejecutarlo. En el apartado *"Utilización del debugger"* se describe el uso del debugger para ejecutar nuestro programa en forma controlada y examinar el contenido de los registros o la memoria.

El sistema incluye puertos de entrada para los switches SW[9..0] y los pulsadores BUTTON[2..0] y puertos de salida para manejar los leds LEDG[7..0] y los displays de 7 segmentos HEX3, ...HEX0. En el anexo 3 se indican en detalle los recursos disponibles y su mapa de direcciones.

2.2.2 Establecer la comunicación entre el PC y la placa

Una vez programado el sistema en la placa, se debe proceder a abrir un canal de comunicación a

través del puerto USB entre la PC y el sistema dentro de la placa DE0. Para hacer esto hay que ir a *Macro → jtagcon+gdb.* Este proceso abre `jtagcon` en una ventana y el debugger (`gdb`) en otra como se aprecia en la figura.



El programa `jtagcon.exe` se ejecuta en el PC y establece una conexión con el programa monitor cargado en la memoria del Z80 a través del puerto USB. Luego, el jtagcon queda a la espera de comandos por parte del debugger (escuchando el puerto tcp 8000 del computador personal) y los retransmite al monitor. El programa jtagcon queda corriendo después de desplegar el mensaje "`listening for connections on port 8000`". Esta ventana puede dejarse minimizada mientras se trabaja con el debugger.

Además de ejecutar `jtagcon` y el debugger, el macro carga en la memoria del procesador el programa seleccionado.

Nota: La primera vez que se ejecuta `jtagcon.exe` puede aparecer una ventana de seguridad de Windows pidiendo autorización al usuario para ejecutarlo. En dicho caso, seleccionar la opción "Desbloquear".

3 Utilización del debugger

Una vez abierto el debugger, simplemente hay que ejecutar los comando deseados para depurar el programa. A continuación damos una lista con los comandos más útiles. Al final del tutorial se da un link con más información sobre los comandos del gdb.

Comandos gdb

El debugger gdb cuenta con gran cantidad de comandos para depurar el sistema. Algunos comandos útiles son:

Breakpoints:

- `break etiqueta`: para insertar un break point en el lugar de la etiqueta.
- `info break`: Muestra información de los break point.
- `del i`: elimina el break point número `i`.

Ejecutar el programa:

- `cont`: ejecuta el programa hasta el próximo break.
- `nexti`: ejecuta una instrucción assembler, over call.
- `stepi`: ejecuta una instrucción assembler, into call.

Desplegar información:

- `info registers`: muestra el contenido de los registros.
- `disassemble dir,+30`: desensambla el contenido de la dirección `dir` y 30 lugares más.
- `disassemble etiqueta`: desensambla el contenido de la etiqueta
- `list`: muestra el código fuente.
- `x/20bx dir`: Muestra 20 elementos de tamaño byte (b) en hexadecimal (segunda 'x') a partir de la dirección `dir`.

Modificar registros o memoria:

- `set $a= 0x55`: carga `0x55` en el registro A del Z80.
- `set $bc=0x0ff0`: modifica el par de registros BC del Z80.

Atención que gdb no sabe modificar el registro b solo. El comando:

- `set $b=0x0f` : crea una variable interna a gdb ("convenience variables") con nombre $b
- `set {char}0xNNNN=0x55` : escribe UN byte en la dir 0xNNNN con el valor 0x55
- `set {int}0xNNNN = 0x1234` : escribe DOS bytes, `0x34` den la dirección NNNN y `0x12` en la dirección 0xNNNN+1

Lo que le estamos indicando con el texto entre llaves (son llaves, no paréntesis) es que el dato almacenado en 0xNNNN es del tipo char (que ocupa un byte) o del tipo int (que ocupa dos bytes en memoria). En lugar de la dirección numérica se puede poner un símbolo definido con una etiqueta.

Puertos (No válido para el QEMU, sólo funciona en placa DE0):

- `monitor in (0x90)`: lee el contenido del puerto `0x90`, en nuestro caso, lee los switches.
- `monitor out (0xA0), 0x55` : escribe en el puerto `0xA0` el valor `0x55`.

Comienzo y fin de sesión:

- `quit`: salir de gdb.
- `target` : conecta al dispositivo destino donde se ejecutará nuestro programa.
- `file` : selecciona archivo a depurar y carga tabla de símbolos al debugger.
- `load` : carga el programa en la memoria del sistema destino.
- `disconnect` : termina la conexión con el *target*, sea éste el monitor corriendo en la placa DE0 o el simulador.

Ayuda:

- `help`: nos da una lista con todos los comandos del gdb.
- `help comando`: nos describe el comando 'comando'.

4- Ejemplo de ejecución de prueba

A continuación presentamos un ejemplo de ejecución de los comandos en el gdb, con la rutina `test.s` cargada en el sistema:

Luego de cargar el programa, como fue explicado anteriormente, se puede desensamblar el código de la rutina utilizando el comando `disassemble`:

```
(qemu-gdb)disassemble 0xb000, +20
Dump of assembler code from 0xb000 to 0xb014:
=> 0x0000b000:  ld b,0xfe
   0x0000b002:  ld hl,0xb016
   0x0000b005:  ld (hl),0x00
   0x0000b007 <Loop+0>: in a,(0x80)
   0x0000b009 <Loop+2>: out (0x84),a
   0x0000b00b <Loop+4>: rlc b
   0x0000b00d <Loop+6>: ld a,b
   0x0000b00e <Loop+7>: out (0x80),a
   0x0000b010 <Loop+9>: inc (hl)
   0x0000b011 <Loop+10>:        ld a,(hl)
   0x0000b012 <Loop+11>:        out (0x81),a
End of assembler dump.
(qemu-gdb)
```

Posteriormente, se ejecutaron las tres primeras instrucciones del código `"LD B, 0xFE"`, `"LD HL, dato"` e `"IN A, (SW)"` con el comando `stepi` que ejecuta de a una instrucción por paso. Observar que la instrucción que aparece luego de ejecutar el comando `stepi` no es la instrucción realizada por el procesador, sino la que le sigue.

Una vez que el procesador ejecutó la instrucción IN para leer los switches, se puede observar el contenido de los registros utilizando el comando `info registers`, para corroborar la correcta lectura del puerto de entrada. El contenido del acumulador deberá corresponder al estado de los switches (si estamos trabajando con la placa) o a lo especificado en el archivo `in.txt` para la dirección correspondiente al puerto.

Observar que los pares de registros se despliegan juntos como un registro de 16 bits. El banco de registros alternativo de Z80 aparece indicado con un sufijo “x” en el nombre del registro.

```
(qemu-gdb)stepi
16              LD HL, dato
(qemu-gdb)stepi
17              LD (HL), 0
(qemu-gdb)stepi
19      Loop:   IN A, (SW)          ; Leo los switches
(qemu-gdb)stepi
20              OUT (Leds), A       ; Prendo los Leds
```

```
(qemu-gdb)info registers
a               0xf1     241
f               0xff     255
bc              0xfe00   65024
de              0x0      0
hl              0xb016   45078
ix              0x0      0
iy              0x0      0
sp              0xffff   65535
i               0x0      0
r               0x0      0
ax              0x0      0
fx              0x0      0
bcx             0x0      0
dex             0x0      0
hlx             0x0      0
pc              0xb009   45065
(qemu-gdb)
```

Para agilizar la ejecución de las instrucciones, agregar un *breakpoint* en la dirección apuntada por `Loop`, que corresponde al `IN A,(SW)`, y luego se ejecuta el comando `cont` para que el programa continúe la ejecución hasta el próximo *breakpoint*, que en este caso es el agregado en `Loop`. (**Nota**: el debugger diferencia entre mayúsculas y minúsculas en los símbolos).

Si repetimos el comando cont podemos observar como cambian los puertos de salida. En cada iteración del loop se escriben los puertos correspondientes a HEX1 y HEX0. Si estamos trabajando con la placa en HEX0 se puede ver claramente la rotación del registro B, mientras que en HEX1 se verá que los segmentos encendidos parecen no tener sentido pero corresponden al incremento realizado sobre el registro C. Si en cambio estamos trabajando sobre el simulador, podemos observar los cambios en el renglón correspondiente a los puertos `0x80` y `0x81` en el archivo `outfile.txt` o en la ventana desde la que se lanzó QEMU. En este último caso para modificar el valor leído en el puerto de entrada se puede editar el valor especificado en el archivo `in.txt` y salvarlo mientras la ejecución está detenida en el breakpoint. En la siguiente iteración del loop se leerá el nuevo valor que deberá verse en el puerto de salida correspondiente a la dirección `0x84`.

```
(qemu-gdb)break Loop
Breakpoint 1 at 0xb007: file test.s, line 19.
(qemu-gdb)cont
Continuing.

Breakpoint 1, Loop () at test.s:19
```

```
19      Loop:   IN A, (SW)        ; Leo los switches
(qemu-gdb)
```

Si se sacan los *breakpoints* y se ejecuta el comando `cont`, gdb pierde el control sobre la ejecución del programa, siendo imposible volver a controlar el flujo de ejecución.

En caso de estar trabajando con la placa DE0 esto se debe a que el procesador deja de ejecutar el programa monitor encargado de la interacción con gdb y ejecuta solamente el programa cargado, que es incapaz de interactuar con gdb.

Borrar los *breakpoints* y ejecutar `cont` es una forma de probar la ejecución en tiempo real del programa cargado en el Z80. Observar cómo la salida en los displays HEX1 y HEX0 cambia tan rápido que no es posible distinguir nada (se ven todos los segmentos encendidos) y que si modificamos el valor de una entrada SW se modifica el estado del LED verde correspondiente.

Para poder volver a recuperar el control sobre el flujo de ejecución, es necesario llevar el SW[9] a su posición baja, de esta forma el sistema vuelve a ejecutar el monitor con lo que se recupera la comunicación con gdb. Esto se hace a través de la entrada NMI del Z80 que se verá más adelante en el curso.

En caso de estar trabajando con el simulador, si queda el programa bajo prueba corriendo sin ningún breakpoint y sin ningún mecanismo de finalización entonces para detener la ejecución debemos cambiarnos a la ventana de QEMU y presionar Ctrl+C.

```
(qemu-gdb)info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0000b007 test.s:19
        breakpoint already hit 1 time
(qemu-gdb)dele 1
(qemu-gdb)c
Continuing.

Program received signal SIGINT, Interrupt.
Loop () at test.s:19
19      Loop:   IN A, (SW)        ; Leo los switches
(qemu-gdb)
```

Se recomienda probar la secuencia anterior de comandos y otras elegidas por el usuario para familiarizarse con el funcionamiento del *debugger*.

Anexo 1 - Directivas y sintaxis ensamblador

A continuación daremos una breve descripción del formato a seguir para utilizar algunas de las directivas del compilador, por más información consultar la documentación del assembler.

Sintaxis directiva **EQU:** "*symbol* **equ** *expression*"

Sintaxis directiva **DB:** "**db** *expression*"

Sintaxis directiva **ORG:** "**.org** *offset*"

El valor especificado en la directiva ORG es relativo al comienzo de la sección *text* definida con la línea .text.

P. ej.:

```
.org 0x0100
tabla: db 0x00
```

El símbolo tabla queda con el valor `0xB100` obtenido de sumar el valor de comienzo de la sección .text (`0xB000` especificado al invocar al linker) con el valor `0x100` especificado en la directiva ORG.

Constantes numéricas:

Binario: cadena de 0's y 1's que empiezan con 0 y terminan con B, ejem: `010101b`.

Octal: cadena de dígitos octales que empieza con 0, ejemplo: `073647`.

Decimal: cadena de dígitos decimales que no empieza con 0, ejemplo: `384902`.

Hexadecimal: cadena de dígitos hexadecimales que empiezan con `0x`, ej: `0xAF51`.

Anexo 2 - Links de interés

- Documentación assembler: http://sourceware.org/binutils/docs-2.21/as/
- Documentación linker: http://sourceware.org/binutils/docs-2.21/ld/index.html
- Referencia rápida gdb: https://www.appservgrid.com/refcards/refcards/UnixLinux/GDB%20Quick%20Reference.pdf

Anexo 3 – Descripción del Hardware del sistema (procesador Z80 + memoria + puertos) cargado en la placa DE0.

El sistema que se se carga en la placa tiene las siguientes características:

Memoria:

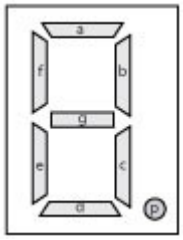
8KB	ROM	0x0000-0x1FFF
32KB	RAM	0x8000-0xFFFF

- 8Kb de ROM (0x0000-0x1FFF)
- 32Kb de RAM (0x8000-0xFFFF)

Mapa de E/S

SW[7..0]	in	Switches	0x80
BUTTON[2..0]	in	Pulsadores	0x81
HEX0[7..0]	out	Display	0x80
HEX1[7..0]	out	Display	0x81
HEX2[7..0]	out	Display	0x82
HEX3[7..0]	out	Display	0x83
LEDSG[7..0]	out	Leds	0x84

segmento	p	g	f	e	d	c	b	a
bit	7	6	5	4	3	2	1	0



El sistema cuenta además con un botón de reset en el Button 0, una entrada conectada al SW[9] para forzar una interrupción no enmascarable y un puerto de comunicación serie, que será utilizado para cargar y depurar los programas en assembler.

Anexo 4 – Scripts invocados por los macros de Notepad++

En todos los scripts el texto `$(NAME_PART)` se sustituye por el nombre del archivo (sin extensión) abierto en la solapa activa de Notepad++. El texto `$(CURRENT_DIRECTORY)` es el directorio y `$(FULL_CURRENT_PATH)` es el nombre completo (incluyendo directorio y extensión) de dicho archivo. Si estamos editando el archivo `test.s` en el directorio `c:\z80-tools\test\`, entonces `$(NAME_PART)` es `test`, `$(CURRENT_DIRECTORY)` vale `c:\z80-tools\test\` y `$(FULL_CURRENT_PATH)` será `c:\z80-tools\test\test.s`

Inicialización al cargar Notepad++

Al arrancar Notepad++ se inicializan variables utilizadas en los otros macros para determinar la ubicación de las herramientas en disco.

```
//**************************************************************
// Macro para configurar los directorios donde estan las
// herramientas del curso
//**************************************************************
// Directorios de Herramientas
SET Z80TOOLS_DIR = e:\tmp\z80-tools
SET QUARTUS_PATH = C:\altera\91\quartus

SET BINUTILS_DIR = $(Z80TOOLS_DIR)\binutils-z80
SET QEMU_DIR = $(Z80TOOLS_DIR)\qemu-z80
SET GDB_DIR = $(Z80TOOLS_DIR)\gdb-z80
SET JTAG_DIR = $(Z80TOOLS_DIR)\jtagcon
SET SREC_DIR = $(Z80TOOLS_DIR)\srecord
SET SCRIPT_DIR = $(Z80TOOLS_DIR)\scripts

ENV_SET PATH = $(QUARTUS_PATH)\bin\cygwin\bin;$(sys.PATH)

// Limpio la pantalla
CLS
// Oculto la consola
NPP_CONSOLE 0
```

Compilar

```
::Build
//*****************************************************************
// Macro para ensamblar usando las herramientas del curso
//*****************************************************************

// Voy al directorio actual
CD $(CURRENT_DIRECTORY)
// Limpio la pantalla
CLS
// Salvo el archivo actual
NPP_SAVE
// Borro los anteriores
CMD /C del $(NAME_PART) && del $(NAME_PART).o

// Voy al directorio donde estan las herramientas y compilo
$(BINUTILS_DIR)\z80-coff-as -z80 -g -as -gstabs+ -o $(NAME_PART).o $
(NAME_PART).s
$(BINUTILS_DIR)\z80-coff-ld --oformat coff-z80 -e 0xB000 -Ttext
0xB000 -o $(NAME_PART) $(NAME_PART).o

$(BINUTILS_DIR)\z80-coff-objdump -j .text -j .data -t $(NAME_PART)
```

El ensamblador `z80-coff-as` procesa el archivo `$(NAME_PART).s` y genera el archivo con el mismo nombre y extensión `.o` (parámetro `-o $(NAME_PART).o`).

El linker `z80-coff-ld` toma el archivo `$(NAME_PART).o` generado por el ensamblador y genera un archivo con el mismo nombre y sin extensión, en formato coff-z80. Notar que al linker se le indica que el programa va a estar en memoria a partir de la dirección `0xB000` (-Ttext 0xB000) y que la primera instrucción a ejecutar del programa está en la misma dirección (-e 0xB000). Además de el código de máquina, el archivo en formato coff-z80 contiene la tabla de símbolos e información adicional utilizada por el cargador y el debugger para saber en qué dirección de memoria cargar el programa y a qué dirección saltar cuando se solicita ejecutar nuestro programa.

Por último el comando `z80-coff-objdump` despliega en la consola los símbolos definidos en las secciones .text y .data.

Qemu+Gdb

```
//*****************************************************************
// Macro para depurar usando qemu y gdb
//*****************************************************************
// Voy al directorio actual
CD $(CURRENT_DIRECTORY)
// Limpio la pantalla
CLS
//Pregunto por el archivo a cargar
//el nombre del archivo seleccionado queda en $(INPUT)
INPUTBOX "Archivo a Cargar:" : $(NAME_PART)


//Arranco el QEMU
NPP_RUN $(QEMU_DIR)\qemu-system-z80.exe -io-output-file out.txt -io-
input-file in.txt -monitor file:temp -io-output-log-file output.log
-nographic -serial null -S -gdb tcp:127.0.0.1:8010


//Ejecuto el GDB en una ventana nueva
NPP_RUN $(GDB_DIR)\gdb -ex "set prompt (qemu-gdb)" -ex "target
extended-remote localhost:8010" -ex "file $(INPUT)" -ex "load"
```

El comando `qemu-system-z80` arranca el simulador. En la línea de comando se le indican los nombres de los archivos usados para emular los puertos de entrada/salida (`-io-output-file out.txt -io-input-file in.txt -io-output-log-file output.log`). Se le indica además en qué puerto tcp queda a la escucha de comandos del debugger.

El comando `gdb` arranca el *debugger*. En la línea de comando se indican los primeros comandos que deben ejecutarse dentro de gdb:

- `-ex "set prompt (qemu-gdb)"` : modifica el prompt.
- `-ex "target extended-remote localhost:8010"` : conectarse al dispositivo destino (donde se ejectutarán los programas) en el puerto tcp donde quedó escuchando el quemu.
- `-ex "file $(INPUT)"` : seleccionar archivo y cargar tabla de símbolos.
- `-ex "load"` : cargar el programa en el sistema destino.

Jtagcon+Gdb

```
//*************************************************************
// Macro para depurar con el monitor cargado en la placa y gdb
//*************************************************************
// Voy al directorio actual
CD $(CURRENT_DIRECTORY)
// Limpio la pantalla
CLS
//Pregunto por el archivo a cargar
INPUTBOX "Archivo a Cargar:" : $(NAME_PART)

// Ejecuto JTAGCON
NPP_RUN $(JTAG_DIR)\jtagcon.exe

//Ejecuto el GDB en una ventana nueva
NPP_RUN $(GDB_DIR)\gdb -ex "target extended-remote localhost:8000" -
ex "file $(INPUT)" -ex "load"
```

Similar al script *Qemu + Gdb* pero se invoca el comando `jtagcon` en lugar del qemu.

El comando `jtagcon` establece a través de la conexión USB con la placa DE0 una comunicación con el programa `monitor.hex` que corre en el procesador Z80 dentro del chip FPGA. Queda escuchando en el puerto tcp `8000` a la espera de comandos del debugger para retransmitirlos hacia el Z80.

El comando `gdb` arranca el debugger. En la misma línea de comando se le indican los primeros comandos que deben ejecutarse dentro de gdb:

- `-ex "target extended-remote localhost:8000"` : conectarse al dispositivo destino (donde se ejectutarán los programas) en el puerto tcp `8000`.
- `-ex "file $(INPUT)"` : selecciona archivo y carga tabla de símbolos.
- `-ex "load"` : carga el programa en el sistema destino.

Grabar sistema en FPGA

```
::Quartus_pgm
//*************************************************************
// Macro para programar el fpga
//*************************************************************

// Limpio la pantalla
CLS

// Programo la placa
CMD /C "$(QUARTUS_PATH)\bin\quartus_pgm" -c1 $(FULL_CURRENT_PATH)
```

Se utiliza el comando `quartus_pgm` para grabar en el FPGA de la placa DE0 el circuito con el sistema completo (procesador + memorias + puertos).

Se le pasa como parámetro el nombre del archivo abierto en la ventana activa de Notepad++ (`$(FULL_CURRENT_PATH)`), que debe ser un archivo `.cdf` generado con Quartus de Altera.

El archivo `.cdf` apunta a un archivo `.sof` con el circuito sintetizado. Este circuito incluye a la ROM del sistema, que tiene precargado el programa `monitor.hex` que se encarga de dialogar con el gdb para recibir y dar curso a los comandos del debugger.

Compilar para ROM

```
::Compilar_ROM
//*************************************************************
// Macro para compilar un .s para la ROM
//*************************************************************
// Voy al directorio actual
CD $(CURRENT_DIRECTORY)
// Limpio la pantalla
CLS
// Salvo el archivo actual
NPP_SAVE
// Borro los anteriores
CMD /C del $(NAME_PART) && del $(NAME_PART).o && del $
(NAME_PART).srec && del $(NAME_PART).hex


$(BINUTILS_DIR)\z80-coff-as -z80 -g -as -gstabs+ -o $(NAME_PART).o $
(NAME_PART).s
$(BINUTILS_DIR)\z80-coff-ld --oformat coff-z80 -e 0x0000 -Ttext
0x0000 -Tdata 0xB000 -o $(NAME_PART) $(NAME_PART).o
$(BINUTILS_DIR)\z80-coff-objdump -j .text -j .data -t $(NAME_PART)


$(BINUTILS_DIR)\z80-coff-objcopy.exe -j .text -O srec $(NAME_PART) $
(NAME_PART).srec
$(SREC_DIR)\srec_cat $(NAME_PART).srec -data-only -o $
(NAME_PART).hex -intel -obs=1 -address-length=2 -enable=footer
```

Similar al comando Compilar, pero con las siguientes diferencias.

Al linker `z80-coff-ld` se le indica que la sección `.text` se cargará a partir de la dirección `0x0000` (`-Ttext 0x0000`)y la sección .data a partir de la dirección `0xB000` (`-Tdata 0xB000`).

Los comandos z80-coff-objcopy y srec_cat convierten el archivo en formato coff-z80 que genera el linker (archivo $(NAME_PART) sin extensión) al formato hex de Intel. El resultado final es el archivo $(NAME_PART).hex que puede ser cargado en la ROM usando el comando “Cargar en ROM”.

Cargar en ROM

```
::Cargar_ROM
//*************************************************************
// Macro para cargar un .hex en la ROM
//*************************************************************
// Voy al directorio actual
CD $(CURRENT_DIRECTORY)
// Limpio la pantalla
CLS
// Salvo el archivo actual
NPP_SAVE

// Cargo el .hex en la ROM
quartus_stp -t $(SCRIPT_DIR)\update_in_system_memory.tcl $
(NAME_PART).hex
```

Se utiliza el programa `quartus_stp` de Altera para ejecutar los comandos especificados en el archivo `update_in_system_memory.tcl`

En ese archivo están los comandos a las herramientas de Altera para grabar en la ROM del sistema el archivo `$(NAME_PART).hex`

Previamente la placa debe estar encendida y conectada al puerto USB del PC, y se debe haber grabado el circuito en el FPGA mediante el comando *"Grabar sistema en FPGA"*.

Control de Versiones

Version	Fecha	Autor(es)	Comentario
3.3	2013-03-11		
3.4	2022-03-08	JOF, JPM, JPA	Agrega Links a documentos. Agrega explicación de como encontrar AppData. Actualiza link a referencia rápida gdb. Actualización de nombres y versiones.