Título

UN EMULADOR DEL LADRILLO RCX

Autor

JOSÉ ROBERTO GARCÍA CHICO

Tutor

MIGUEL ÁNGEL CAZORLA QUEVEDO

Departamento

DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

Curso

2002-2003

ÍNDICE

INTRODUCCIÓN 3

CONTENIDO 3

- Descripción del sistema a emular 3
- El ladrillo RCX 3
- Puesta en marcha del ladrillo 4
- El emulador 5
- Estructura del emulador 5
- La unidad aritmético-lógica 6
- El banco de registros 6
- El manejador/administrador de memoria 6
- La interfaz exterior 8
- Las instrucciones 8
- Ejecución del programa almacenado 9
- Reset del sistema 11
- Configuración del mapa de memoria según modelo 11
- TH8_3292, una particularización del emulador 11
- El emulador del ladrillo 12
- ¿Qué pone en marcha la emulación? 13

RESULTADOS Y CONCLUSIONES 14

BIBLIOGRAFÍA 15

INTRODUCCIÓN

Lego MindStorms Robotics Invention System es un producto comercial de la juguetera *Lego* dedicada a los juegos de construcción mediante ladrillos de plástico.

Este juego en concreto trata de la construcción y programación de robots donde la pieza fundamental es el RCX que contiene el sistema de comandos para los robots. El RCX es una minicomputadora integrada en un ladrillo y que se puede programar desde un ordenador que tuviera puerto serie *RS-232* o utilizar los cinco programas que ya vienen con el ladrillo.

El RCX utiliza sensores táctiles y de luz para recibir información de su entorno. A continuación, procesa los datos necesarios para hacer que se enciendan o se apaguen los motores.

Con este sistema, se pueden construir robots que interactúen con el entorno. Primero, se construiría el robot utilizando el RCX junto a otros ladrillos a partir del manual de instrucciones que te viene con el juego o cualquier otro. Después se realizaría un programa utilizando una herramienta que se adjunta con el juego para después transmitir ese programa al ladrillo RCX para que pudiera funcionar como queramos.

CONTENIDO

Descripción del sistema a emular

Este proyecto forma parte de otro más grande que pretende realizar un banco completo de herramientas para el desarrollo de micro robots. El proyecto se llamaba *MindStorms VDK*.

Lego MindStorm consiste en un ladrillo que actúa como el cerebro del robot y otros tantos ladrillos son los sensores y los motores con los que puede interactuar con el entorno. El ladrillo que hace de cerebro puede ser programado añadiéndole primero un *firmware* que hará correr programas que escribamos utilizando lenguajes como el *NQC* (un lenguaje basado en C) u otro parecido al *Java* o el creado por los propios fabricantes, de *Lego*, que consiste en hacer que las instrucciones sean comandos que se enlacen unos con otros como si fueran ladrillos.

Antes he mencionado que se debe de cargar un *firmware* antes de cargar los programas. Este *firmware* es en realidad un intérprete que interactúa entre la BIOS residente en la ROM del ladrillo y el programa que define cómo se comportará la máquina. Dependiendo de la configuración de sensores y motores, podremos montar distintos tipos de maquinaria no solamente robots ya que, además, incorpora un pequeño altavoz con el que podremos programar el *RCX* para que haga sonar música.

Podemos saber el estado del sistema observando una pequeña pantalla *LCD* que se sitúa entre los puertos donde se conectan los sensores y donde se conectan los motores.

El ladrillo *RCX*

El ladrillo *RCX* real está compuesto por un microcontrolador *Hitachi H8* de la familia *329x*, en concreto, se trata de *3292*. El mapa de memoria de esta familia puede configurarse a través de hardware y hay tres configuraciones diferentes aunque

comparten zonas comunes como son los registros referentes a los periféricos. Por lo tanto, el control de los periféricos está mapeado en memoria y no se precisa de instrucciones adicionales para enviar y recibir datos por los puertos.

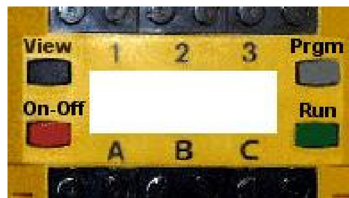
Al ladrillo *RCX* podemos conectar muchos sensores y de varios tipos como pueden ser fotoeléctricos o pulsadores pero sólo hay tres conexiones por lo que sólo puede gestionar como mucho tres sensores. También se le pueden conectar motores, hay tres zócalos habilitados a tal efecto y aunque se pueden conectar más de uno, hay que tener en cuenta que estaríamos forzando a los controladores de los motores y podríamos llegar a consumir rápidamente las baterías o quemar los puertos.

A través de la pantalla *LCD* podremos controlar el estado del sistema. Aquí podremos ver el programa seleccionado y si está en marcha o no, además de saber qué motores están encendidos y qué sensores están actuando sobre el sistema. Cuando hay se instala el *firmware* que proporciona el fabricante, podemos ver también cuánto tiempo lleva en marcha el ladrillo.

La comunicación con el *PC* se realiza a través de un puerto de infrarrojos que se trata en realidad de un puerto de comunicaciones tipo *RS-232* disponibles en todos los actuales *PC*. Junto con el ladrillo viene una torre que permite enviar y recibir datos del *RCX* al encargarse este de la comunicación por infrarrojos por parte del *PC*.

Puesta en marcha del ladrillo

La primera vez que se pone el ladrillo en marcha es cuando le ponemos las seis pilas de *tipo AA* y, aunque no apreciemos nada, la *CPU* empieza a ejecutarse para ir colocando los valores por defecto de los puertos y prepararse para estar en modo suspendido hasta que el usuario pulse el botón de *On-Off*.



Cuando se pulsa este botón, suena un par de pitidos a la vez que se muestra en la pantalla *LCD* un pequeño hombrecillo y un número que indica el programa seleccionado. Si pulsamos en botón de *Run*, haremos que el hombrecillo se ponga a caminar ya que esto significa que el programa seleccionado e identificado por el número que aparece a la derecha del hombrecillo está ejecutándose. Cuando queramos parar la ejecución, pulsaremos de nuevo el botón de *Run*.

El ladrillo puede albergar hasta cinco programas distintos y estos pueden ser seleccionados si pulsamos el botón de *Prgm* sólo si el sistema no está interpretando ya un programa.

Podemos cargar mediante la torre que se conecta al *PC* un *firmware* que no es otra cosa que un intérprete que se encarga de traducir las instrucciones compiladas siguiendo algún lenguaje de programación como el propio de *Lego*, *NQC* o *Java*. Este *firmware* estará residente en la memoria *RAM* del *RCX* hasta que se vuelva a carga otro *firmware* o a que se le quiten o agoten las pilas. El *firmware* está contenido en un fichero en formato *S-Record* propuesto por *Motorola* (no es objetivo de este texto explicar este formato así que está situado junto a los ficheros del proyecto).

A través de la misma torre enviaremos los programas que queramos que el ladrillo interprete y tendremos que procurar, si estamos utilizando varios *firmware*, que enviemos el programa para el que tengamos en el ladrillo o podría no funcionar. Normalmente, todos los programas son traducidos a un lenguaje concreto y son guardados en ficheros con extensión *rcx*.

Mientras se esté cargando el *firmware* o el programa, en la pantalla *LCD* del ladrillo se mostrará este estado mediante una sucesión de puntos. Una vez finalizada la carga y si es un programa, podremos seleccionarlo pulsando el botón de *Prgm*. Pulsando este, iremos pasando por los distintos programas almacenados en el ladrillo, identificados por un número. Sólo puede tener cinco programas como mucho. Pulsaremos el botón de *Run* para ponerlo en marcha y pararlo. Durante la ejecución del programa, podremos pulsar el botón *View* para ver el estado de los sensores.

El ladrillo *RCX* se apagará pulsando el botón *On-Off* o dejándolo cinco minutos sin operar.

El emulador

El emulador trata de un programa que intenta imitar al completo el funcionamiento de un sistema cualquiera. En este caso, se trata de interpretar el ladrillo *RCX* proporcionando el mismo interfaz que este o uno muy parecido. Internamente, puede o no funcionar exactamente como el sistema real pero lo que sí que debe de hacer es tener el mismo comportamiento que el sistema real.

El emulador se ha propuesto como herramienta educacional ya que proporcionará ayuda en asignaturas de micro robótica que utilicen este sistema para el desarrollo de la asignatura. En ningún momento se ha planteado para hacerle la competencia al *MindStorm* de *Lego* sino como un complemento que les ayude a los programadores de este sistema obtener más información sobre el funcionamiento de los futuros programas que funcionarán en el ladrillo *RCX*.

Está programado en *Java* sólo por la razón de hacerlo funcionar con independencia de la plataforma hardware y software. El programa no funcionará a tiempo real debido a que *Java* es un lenguaje interpretado y se requiere mucha potencia de proceso. Durante las pruebas, se pudo comprobar que poniéndole retardos para que funcionase a tiempo real, a 16 Mhz, se obtenía un número de ciclos por segundo mil veces inferior a si funciona sin retardo.

Estructura del emulador

El emulador está compuesto por varios objetos en los que hay uno que representa a la *ALU* de una manera general. Hay otro que representa el administrador de memoria que puede ser sobrecargado cambiar el mapa para representar cualquier modelo de la familia de microcontroladores *H8 329x*. También se ha definido la interfaz externa para todas las instancias de la clase *TH8_329x* como pines que pueden tener 1024 estados distintos. Se puede instalar un módulo que representa a la memoria externa y que será referenciado por el administrador de memoria.

Esta parte central se ha desarrollado con independencia del uso que se fuera a dar.

Después se ha puesto otra capa que representa al microcontrolador *H8 3292* y que hereda de la clase *TH8_329x*. Objetos instanciados de la clase *VisualVirtualRCX* tendrán un objeto de la clase *TH8_3292* además de poder obtener la imagen del

ladrillo *RCX* donde se le pueden pulsar los botones y los enchufes para los sensores y los motores que le podamos poner.

La unidad aritmético-lógica

Representada por el objeto *alu* instancia de la clase *TAlu*, en su interior se encuentran los *flags* resultados de la última operación realizada por el objeto. Los *flags* representan si se permiten interrupciones, si ha habido acarreo y medio-acarreo en la última operación, desbordamiento, si el resultado es cero o un número negativo y dos *flags* para el usuario. Estos *flags* son luego empaquetados en un registro llamado *CCR* y que es devuelto en un entero de 32 bits donde los *flags* se hayan en los 16 bits más bajos. El registro *CCR* es de 8 bits y se encuentra en los 8 bits más bajos de ese registro de 32 bits, los otros 8 es una copia para poder obtener el registro como una palabra de 16 bits.

Interiormente, los *flags* están almacenados en variables enteras de 32 bits para evitar estar utilizando operaciones lógicas para poner a 1 o a 0 cierto *flag*. Para algunas operaciones, se empaquetan en los 8 bits más de una variable de 32 bits y después de la operación, los *flags* son desempaquetados en sus variables correspondientes.

Las operaciones unarias y binarias aceptan un número entero de 32 bits (tipo *int*) pero que después, dependiendo del tipo de operación, se utilizará sólo el byte más bajo si se trata de una operación a nivel byte o la palabra de 16 bits más baja para el tipo correspondiente. La mayoría de las funciones devuelven el resultado con la llamada y no se tiene que consultar ninguna otra variable.

La razón de haber utilizado el tipo de 32 bits es porque ciertos microprocesadores donde se puede ejecutar el emulador pueden manejar con más eficiencia y velocidad las palabras de 32 bits que las de 16 bits y 8 bits. Además de evitar conversiones por parte de Java ya que suele dar muchos mensajes de posible pérdida de precisión.

El banco de registros

Este es el almacén que representa los registros de propósito general de la *CPU*. Hay 16 registros de 8 bits agrupados en 8 registros de 16 bits. El registro *0H* (registro de 8 bits) representa un registro mapeado en memoria para mejor manejo de los periféricos. Además, también contiene el contador de programa *PC* y la cabeza de la pila *SP* que es sólo un enlace al registro 7 (*7H* y *7L* agrupados como registro de 16 bits). El objeto que representa este banco de registros es *regs* que hereda de *TRegs*.

El objeto dispone de un método que carga la dirección del registro del periférico antes de ser accedido por el registro *0H*.

El manejador/administrador de memoria

Instanciado por el objeto *mm* de la clase *TMM_H8_329x*, es creado mediante un método sobrecargable para poder utilizar una configuración que nosotros creemos propio o existente como es el mapa del *H8-3292* y el objeto devuelto es del tipo *TMM_H8_3292* que es heredero de la clase *TMM_H8_329x* .

Dispone de un array de 65536 de enteros de 32 bits que representa la memoria del microcontrolador llamado *mem*. La memoria real es de 65536 bytes pero se ha elegido el formato de 32 bits para evitar estar tratando con conversiones de tipo,

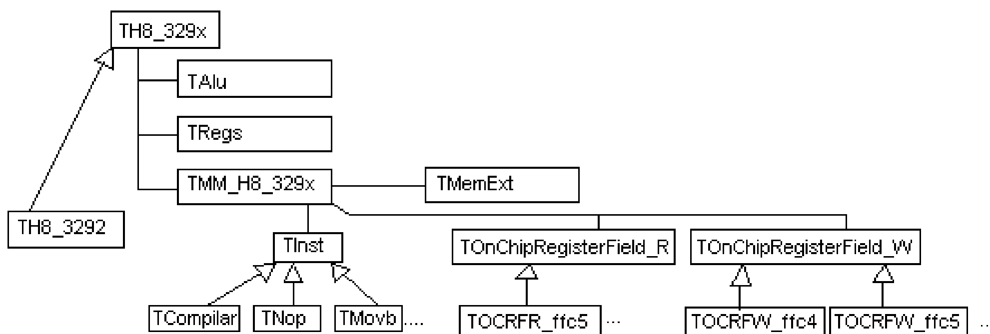
porque hay sistemas que tienen mejor rendimiento tratando con palabras de 32 bits que de 8 bits y porque el compilador de Java da error de pérdida de precisión cuando se convierten enteros más grandes a más pequeños.

Existe otro array de 32768 posiciones que representan a las instrucciones. Este array se llama *inst*. Cada posición está compuesta por un objeto del tipo *TInst* que representa a la instrucción a ejecutar. La clase *TH8_329x* define varias clases que heredan de la clase *TInst* y que cambian el comportamiento de los métodos *exe* y *debug* definidos en la clase *TInst*. Hay otro array que representa el acceso a los registros de estado y de datos de los periféricos y que hay que tener en cuenta a la hora de crear clases que hereden de *TMM_H8_329x*. Todos los objetos son creados inicialmente como del tipo *TCompilar* que heredan de *TInst* y que realizan la primera compilación antes de ejecutar la instrucción correspondiente.

Existen métodos para lectura de palabras de 16 bits como de 8 bits. Las palabras de 16 bits son del tipo *Big Endian*. Los métodos de acceso a esas palabras de 16 bits, en realidad, son dos accesos a los métodos de 8 bits. También se han creado métodos para instalar objetos del tipo *TMemExt* que representan una memoria externa y los métodos para leer y escribir en esa memoria externa además de poner los pines de dirección y de datos en el estado correspondiente para utilizar objetos que puedan usar esta interfaz. De todos modos, estos métodos ya utilizan los métodos de lectura y escritura proporcionados por el objeto del tipo *TMemExt* para un acceso más rápido.

También hay dos arrays que representan el comportamiento al acceder en modo lectura o de escritura a ciertos registros de los periféricos del microcontrolador. Además, hay dos variables que indican en qué modo del modelo de memoria está configurado el microcontrolador y que es establecido por hardware y si la memoria está en modo accesible o no. Esta misma información está repetida, representada por ciertos bits de ciertos registros. Se ha repetido esta información para poder obtener mejor rendimiento a la hora de consultar su valor cuando se lee o se escribe en la memoria.

Los registros controladores de los periféricos son objetos instanciados de clases que heredan de cuando lo que se pretende es leer el valor de ese registro y objetos para escribir en los registros son instancias de clases que heredan de *TOnChipRegisterField_W*. Cada clase tiene que definir los métodos exe declarados en las clases abstractas *TOnChipRegisterField_R* y *TOnChipRegisterField_W*. Algunos de estos objetos utilizan otros objetos instancia de clases que heredan de *TPin* que representan los pines del microcontrolador. Los objetos derivados de estas clases se guardan en el array *OnChipRegisterField_R* y *OnChipRegisterField_W* respectivamente.



La memoria de datos tiene mucho que ver con el array de instrucciones. Tanto que el hecho de escribir un dato en la memoria, provocará que la instrucción que corresponde a la dirección de memoria modificada ya no sea la misma y que se tenga que volver a recompilar la instrucción de esa dirección.

La interfaz exterior

La interfaz, como en el microcontrolador real, está compuesta por un conjunto de pines que cada cual tiene un comportamiento definido. Estos pines son instancias de clases que heredan de la clase *TPin* que tiene un estado consistente en un entero de 32 bits lo que permite un rango muy grande de valores. Hay cinco estados predefinidos que son los más utilizados: alta impedancia, nivel alto, flanco de bajada, nivel bajo y flanco de subida.

La razón de haber utilizado un entero de 32 bits para guardar el estado de los pines es debido a la existencia de un periférico que representa un conversor analógico-digital que tiene una precisión de 10 bits.

Las clases que hereden de *TPin*, deberán definir el método *PonEstado*. Este método, por defecto, cambia el estado de todos los pines a los previamente se hayan conectado al objeto instanciado de esta clase. Para conectar un pin, se utilizará el método *Conectar*. Se podrá obtener el estado de un pin consultando el método *Estado*.

Las instrucciones

Las instrucciones tienen como base la clase *TInst* en el cual se guardan datos tales como la dirección de esa instrucción, la dirección a la siguiente instrucción a ejecutar, la dirección de la siguiente instrucción si no hubiera salto y la dirección de la siguiente instrucción si hubiera salto. Además, guarda información sobre los registros que esta instrucción va a utilizar como fuente y destino y dos tipos de valores fijos que aunque tienen el mismo significado, para poder seguir mejor la especificación de las instrucciones donde se distingue entre dirección absoluta y un valor inmediato que no son otra cosa que un número entero fijo.

Además, se proporcionan varios métodos muy comunes para poder realizar el comportamiento del método *debug* que nos devuelve una cadena con la instrucción compilada. Estos devuelven una cadena donde se muestran el origen y el destino de los operandos.

El método *clk* se debe sobrecargar devolviendo en número de ciclos mínimos que se van a necesitar para considerar que la instrucción ha sido ejecutada.

Redefiniendo el método exe podremos crear instrucciones que operen con el objeto *alu* y *mm*.

Existe un tipo de objeto llamado *TCompilar* que se encarga de recompilar la instrucción. Este obtiene la palabra de 16 bits situada en el array que representan los datos guardado en el objeto *mm*.

En la clase *TH8_329x* se ha definido un array *DinaRec* para ayudar en la compilación de las instrucciones. Este array tiene objetos de clases derivadas de *TOperationCodeMap* que al llamar el método *comp*, obtendremos una nueva instrucción. A ese método se le pasará el byte más bajo de la palabra de 16 bits que representa a la instrucción así como la dirección de memoria donde está almacenada

y la dirección a donde tendría que ir después de que esa instrucción fuera ejecutada si no se produjera un salto. Cada objeto del array compila una instrucción que depende del byte más alto de esa palabra de 16 bits que define la instrucción. Ese byte es un índice, por lo que se ejecutará el método *comp* del objeto indicado por este índice.

Los objetos instanciados de *TCompilar* envían un mensaje al objeto indicado por ese byte índice para obtener la nueva instrucción pero no ejecuta una vez obtenida la instrucción. El nuevo objeto es almacenado en el array de instrucciones en la misma posición donde antes estaba el objeto de tipo *TCompilar*.

Como se ha indicado antes, las instrucciones son de 16 bits. La memoria puede direccionar hasta 65536 bytes, es decir, podemos tener como mucho 32768 objetos del tipo *TInst*. Podemos saber que una instrucción es del tipo *TCompilar* porque el método *clk* devuelve -1.

Todo esto es así debido a que el intérprete es del tipo recompilador dinámico. Es decir, las instrucciones se compilan a medida que se va ejecutando el programa emulado. Se distinguen dos fases independientes en la interpretación de las instrucciones. La primera de ellas es la de descodificación de la instrucción. Para evitar estar descodificando una instrucción invariable que puede estar almacenada en un área ROM cada vez que se ejecute, se guarda el objeto que representa su comportamiento. Por ejemplo, si pensamos en que el compilador puede necesitar 100 ciclos para obtener la descodificación de una instrucción en un bucle consistente en una instrucción que incrementa y otra que salta, no tiene sentido descodificar ambas instrucciones siempre, por lo que si la primera vez que pasamos por esas instrucciones compilamos y ya no volvemos a compilar, ganaremos en velocidad para la emulación del sistema. La fase de ejecución siempre será fija para esa posición de memoria a no ser que el manejador de memoria reciba el mensaje de modificar

Ejecución del programa almacenado

Como la CPU real, las instrucciones se van ejecutando conforme se introducen ondas cuadradas a través de los pines del reloj. Para utilizar la CPU, es necesario encenderla antes. Para hacer esto, es necesario poner el pin *Vcc* a nivel alto y *Vss* a nivel bajo. Esta es la condición necesaria para considerar que la CPU está encendida.

Como el comportamiento del pin *XTAL* es igual al pin *EXTAL*, es decir, unos de los pines del circuito oscilador se conectaría al primero y el otro pin al segundo, para simplificar, se ha ignorado el pin *XTAL* y se le ha dado toda la funcionalidad al pin *EXTAL*.

Un circuito externo se debería de encargar de llamar al método *PonEstado* de ese pin con la siguiente secuencia para emular una onda cuadrada.

La funcionalidad se ha puesto durante el flanco de bajada, aunque se podría haber puesto en cualquier otro estado, he considerado ese para intentar forzar a que el encargado de enviarle los mensajes a ese pin tenga que realizar un nivel alto antes. Más que nada, para intentar no hacer trampas porque, perfectamente, se podría enviar el mismo mensaje para que ponga el flanco de bajada y funcionaría pero se evitarían estados.

Una vez que el pin es colocado con el estado de flanco de bajada y el microprocesador está encendido, si el microcontrolador ha ejecutado una instrucción del tipo *Sleep*, se encontrará durmiendo. En ese estado, existe la posibilidad de que se haya programado el reloj que se encarga de sacar de ese estado a la CPU. Cuando

está en ese estado, se incrementa un contador que llegado a un cierto límite programable de ciclos, saca al microcontrolador de ese estado.

Una instrucción se considera ejecutada cuando han pasado un cierto número de ciclos que vienen dados por el método *clk* de cada objeto del tipo *TInst*. Una vez que se ha alcanzado el número de ciclos mínimo para considerar que el resultado ya está disponible, se llama al método *exe* de ese objeto. Antes de realizar eso, decrementa un contador interno de la *ALU* que se encarga de restaurar el *flag I* a 1 cuando hace dos instrucciones antes ha sido modificado.

Después de ejecutar la instrucción, se comprueba si hay interrupciones pendientes de servir. Las interrupciones son servidas conforme a una prioridad siendo la del *reset* la más prioritaria e inevitable, no se puede deshabilitar. Si hubiera una interrupción pendiente, se almacena en una variable interna el identificador de la interrupción además de la dirección de salto y se elimina esa interrupción pendiente. También despertaría a la CPU en el caso de que estuviera durmiendo.

Si la CPU no está durmiendo y tenemos una dirección de salto por interrupción, se guarda el contador de programa en la pila y un byte que representa los *flags* por duplicado, ya que la pila es de 16 bits. Se desactiva el *flag I* de la ALU para indicar que se está en una interrupción y se cambia el contador del programa por el valor adquirido antes. La variable que contenía ese valor se limpia para evitar que a la próxima instrucción se vuelva a saltar al manejador de la interrupción otra vez.

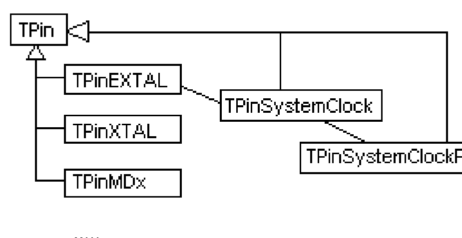
Después de todo esto, se calcula a partir de qué ciclo se ejecutará la siguiente instrucción. El valor vendrá dado por la suma de la cuenta *clk* actual más la duración de la instrucción actual más el número de ciclos de espera programable para sincronizar la CPU con los periféricos lentos.

Para poder realizar un posterior depurador, hay una variable que indica si en este ciclo se ha ejecutado la instrucción actual o no.

Por último, se incrementar el contador de ciclos ejecutados *clk*.

Este es el comportamiento del pin *EXTAL* pero conectado a este se encuentra el pin *pinSystemClock* encargado del envío de pulsos a los periféricos y de realizar una división de la frecuencia si así está programado. Si no está habilitada esta división, se le envía el estado actual al *pinSystemClockP* que es el reloj de los periféricos. Si está habilitada esta característica, el estado sólo cambiará pasado dos ciclos.

Como ya se ha comentado arriba, el pin *pinSystemClockP* es el responsable de realizar la ejecución de las tareas de los periféricos que dependan de este reloj como son los temporizadores de 8 bits, el de 16 bits, el *Watch Dog* y el conversor analógico-digital.



El primero se puso durante el flanco de bajada, mientras que los primeros se puso durante el flanco de subida.

El temporizador del *Watch Dog*, pone su bit de interrupción pendiente si se pasa un número determinado de ciclos. Esa interrupción podría ser la del *NMI* o la del *reset* según se le haya programado. Este temporizador viene bien para sacar a los programas que se quedan colgados en algún punto y reiniciar el programa entero. El número de ciclos a realizar se ha calculado según una tabla encontrada en la especificación que indicaba cuántos milisegundos tardaría en producirse esa interrupción. Como el emulador no entiende el tiempo como segundos, ya que la velocidad de emulación podría no ser a tiempo real, se ha convertido esos milisegundos a ciclos.

En el caso del temporizador de 16 bits, se incrementa su contador cambiando el estado de los pines de comparación según se haya programado y dejando pendientes las interrupciones asociadas a cada evento.

Con el temporizador de 8 bits pasa igual que con el temporizador de 16 bits.

En el caso del conversor analógico-digital, se obtiene el estado de los pines asociados cuando se rebasa el contador. Los pines usados en el emulador ya vienen preparados para poder obtener los 10 bits de precisión que puede captar el conversor real.

Reset del sistema

Cuando encendamos por primera vez el emulador y esté el circuito oscilador encendido, los registros y el PC (contador del programa) contendrán cualquier valor. *Java*, por defecto, los pone a 0, pero se hubiera querido que contuvieran basura.

Reiniciar el sistema es necesario para que se cargue en el contador del programa el vector de inicio que lo proporciona el *reset*. Para realizar el reset, hay que mantener el pin a nivel bajo (nivel alto al estar la entrada negada) durante 518 ciclos para después invertirle el nivel. Hacer esto a niveles superiores parece que se vaya a necesitar mucha información por lo que se le ha proporcionado al emulador la posibilidad de crear un hilo que se encargue de realizar este *reset*.

Cuando se realiza el reset después de que hayan pasado los 518 ciclos, se despierta a la CPU, se reinician los contadores internos menos el *clk*, se reinicia el manejador de memoria, la ALU, se eliminan las interrupciones pendientes y se indica que se ha reiniciado por *reset*.

Configuración del mapa de memoria según modelo

Esta configuración viene dada por los pines *MDx* y es obligatorio decidir qué tipo de mapa se pretende que el manejador de memoria emule. Además, esto también afectará a las conexiones internas entre los pines de los puertos de datos y los periféricos.

TH8_3292, una particularización del emulador

La clase *TH8_329x* puede emular cualquier microcontrolador de la familia *Hitachi H8 329x* donde la única diferencia entre un modelo y otro el el mapa de memoria.

El manejador de memoria que viene por defecto con la clase *TH8_329x* sólo escribe y lee en la memoria interna, ni siquiera se comunica con los periféricos para poder dar mayor libertad a los modelos derivados de este microcontrolador.

Un objeto que se instancie de la clase *TH8_3292* deberá indicar en un array de cadenas qué roms se cargarán en memoria antes de iniciar la emulación, así como declarar el objeto *mm* y el comportamiento ante lecturas y escrituras de bytes en la memoria. Ese comportamiento se ha definido tal y como se especifica. A partir de este ejemplo, se podrían crear los emuladores de los otros modelos.

El emulador del ladrillo

El objetivo final de la implementación de la clase *TH8_3292* es la de poder hacer el emulador del ladrillo RCX. Pero este ladrillo no sólo consta de ese microcontrolador, también dispone de una pantalla LCD, de un pequeño altavoz, de tres zócalos para conectar sensores, otros tres para los motores, luego tiene cuatro botones, en sensor de batería baja y un puerto infrarrojos.

La pantalla LCD, su conexión y la conexión de los cuatros (*On-Off*, *Run*, *Prgrm* y *View*) con la *CPU H8/3292* vienen integrados en la clase *TLCD* que hereda de *JLabel*. La razón de haberlo hecho así es la de poder utilizar esa interfaz gráfica en otras aplicaciones como *SimuWorld* del proyecto original.

Esta interfaz, por sí sola, no tiene funcionamiento para tratar de mantenerla independiente del proyecto. La conexión se realiza a través de otra clase llamada *VisualVirtualRCX* que es la que engloba todo el emulador.

Esta clase no solamente se encarga de crear un objeto del tipo *TLCD* sino que se encarga también de crear un objeto del tipo *TAltavoz*. Esta última clase permite al emulador no solamente producir los pitidos típicos de inicio de un programa, su finalización o el de pulsar botones. En realidad, interpreta la salida de uno de los relojes de 8 bits que tiene el emulador. La onda cuadra que genera es utilizada como onda PCM para producir sonidos de distintas frecuencias pudiendo poder programar un programa musical que funcionara tanto para el ladrillo real como para el emulado.

Sin embargo, la realidad es otra. Debido a que Java es un lenguaje interpretado que necesita mucha potencia, el número de ciclos por segundo al que puede emular es muy inferior (del orden de 1/35 de los 16Mhz a los que debería funcionar en la realidad en un AMD XP 1700 con Windows XP y 384 Megabytes de 266 Mhz DDR) por lo que el sonido es prácticamente inaudible por tener una frecuencia muy baja. El comportamiento de este objeto es el de capturar a cada ciclo el estado del pin de salida del reloj y escribir ese valor en un buffer. Cuando el buffer está lleno, es enviado a la tarjeta de sonido. Esto está hecho así para intentar tener el sonido sincronizado con el emulador y a que, por haber hecho anteriormente pruebas, la creación de un hilo independiente que se encargara de la captura del estado de dicho pin sobrecargaba el procesador haciendo la emulación aun más lenta.

Volviendo al objeto del tipo *TLCD*, ese objeto se conecta mediante dos pines a microcontrolador para poder dibujar los segmentos. Uno de los pines sirve como datos mientras que el otro sirve como reloj. Los segmentos se van coloreando o no dependiendo del pin de datos y siempre con el mismo orden. El algoritmo que se dispone sobre cómo son enviados los datos a la pantalla LCD no es del todo correcto por lo que en el emulador se produce una desincronización pudiendo observar segmentos que no deberían colorearse como activados y otros desactivados. Sobre el objeto de tipo *TLCD* se le ha añadido funcionalidad para ser usados los botones de la

interfaz gráfica mediante el ratón, produciendo un nivel alto cuando es pulsado el botón izquierdo y un nivel bajo cuando se suelta el botón del ratón. Los botones están conectados al microcontrolador emulado. El de *On-Off* está conectado al *Reset*, el de *Run* al *IRQ1* y los otros dos están conectados a dos puertos del conversor analógico-digital.

Otros objetos creados por los objetos instanciados de la clase *VisualVirtualRCX* son tres sensores y tres motores que inicialmente están configurados como en blanco. Los objetos de los sensores son creados a partir de la clase *TSensor* y los de los motores a partir de la clase *TMotor*. La razón de crear los objetos al principio y dejarlos como desactivados tienen más que ver con la forma en que se presentan en la clase *VisualVirtualRCX*. Ambas clases heredan de *JLabel* también, como *TLCD*, para poder mostrar una imagen con el sensor elegido o con motor o en blanco. Los sensores sólo pueden ser como mucho de dos tipos, que producen un cambio de nivel o que la caída de potencial varíe según un criterio como los sensores de luz que varían el voltaje según la cantidad de luz recibida. Además, también pueden ser proporcionados directamente a las aplicaciones que utilicen el objeto *VisualVirtualRCX* para la comunicación con esa aplicación.

¿Qué pone en marcha la emulación?

La respuesta a esa pregunta es fácil de responder si se sabe que antes se ha dicho que las instrucciones de la CPU y sus periféricos son ejecutados a través del pin *EXTAL* que acepta ondas cuadradas.

Existe una clase *TTimer* que se encarga de simular esa onda cuadrada. Al principio se le introdujo un retardo para poder realizar la emulación a 16 Mhz. *TTimer* hereda de *Thread* lo que proporciona ya una ejecución independiente y nos permite utilizar la instrucción *sleep* ya que *wait*, método de *Object*, está en desuso. La instrucción *sleep* era utilizada para realizar ese retardo para poder poner el emulador a 16 Mhz pero pruebas posteriores demostraron que no alcanzaban ni los 500 ciclos por segundo ya que es muy costoso para el intérprete de *Java* realizar el funcionamiento de la instrucción *sleep*. Esta es la razón por la que se optó por quitarla de la clase *TTimer*.

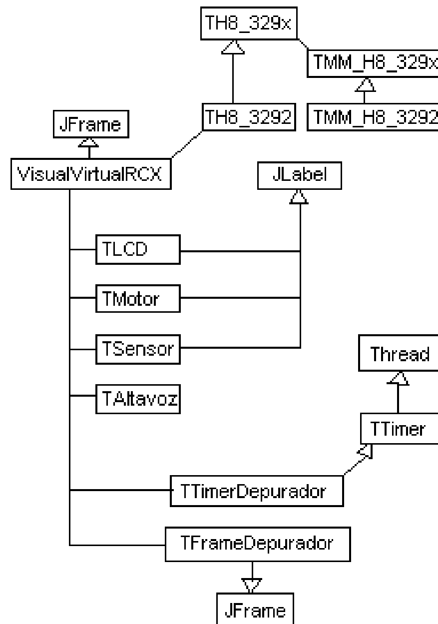
Dentro de la clase *VisualVirtualRCX* se ha creado otra clase llamada *TTimerDepurador* que hereda de *TTimer* y que proporciona varios métodos para deterner la ejecución de la CPU y poder realizar operaciones como ejecución paso a paso, ejecutar animadamente (donde cada instrucción se ejecuta después de un número determinado de milisegundos), poner *break points* (muy útiles para corregir el emulador, en un principio), parar totalmente la emulación y ponerla en marcha.

Estas utilidades son explotadas por un objeto que hereda de la clase *TFrameDepurador* que nos genera una interfaz gráfica donde podemos ver tanto las instrucciones que hay en la memoria del sistema, como los registros de la CPU y los registros de los periféricos. También podremos ver el número de ciclos ejecutados hasta el momento.

El cuadro de diálogo del depurador (objeto instanciado de la clase *TFrameDepurador*) carga de unos ficheros de datos la identificación de las áreas de memoria de la ROM según la especificación de *Kekoa*. Esto es muy útil para no perderse entre la cantidad de instrucciones en lenguaje ensamblador que descodifica el emulador.

Un hilo que se despierta cada cierto tiempo es el encargado de intentar mantener los datos actualizados. No se ha hecho que se refresque en el momento justo del cambio porque así lo único que se conseguiría es que el emulador fuera a la velocidad que tuviera Java de refrescar la pantalla.

En la columna de la izquierda podremos ver las instrucciones que ya han sido compiladas y el cursor indica que esa es la próxima instrucción a ejecutar. En la columna de en medio podremos ver el estado de los registros de la CPU así como los *flags* de la ALU y el registro especial que se accede a través del registro *0H*. La tercera columna, un poco más críptica, nos da información sobre el estado de los registros de los periféricos del microcontrolador.



RESULTADOS Y CONCLUSIONES

La emulación es un tema que viene desde muy antiguo, ya las CPUs son intérpretes y hay emuladores hardware como el *IBM System 360* que realizaba la emulación de máquinas de familias anteriores para mantener la compatibilidad o como los microprocesadores *Intel 80x86* que a partir del *80286* se emulaba el comportamiento del antiguo microprocesador de 16 bits *8086* llamado modo real para poder seguir manteniendo todo el software para las empresas desarrollado para ese microprocesador que tanto éxito tuvo en su tiempo y que aun perdura esa compatibilidad en los Pentiums actuales.

Realizar el comportamiento de las instrucciones es algo relativamente fácil si se comprenden y en varias tardes se pueden implementar ya que al final se trata de un mecanismo muy repetitivo. Sin embargo, ya no son tan fáciles de implementar los periféricos ni los puertos al ser muy configurables y al poder detectar varios niveles de caída de potencial como es el puerto analógico-digital que permite una resolución de *1024 niveles*. Esto ha hecho que se tuviera que diseñar un mecanismo de comunicación entre los distintos componentes de la circuitería como es la creación de la clase *TPin* que permite enviar pulsos o cualquier valor encapsulado en un entero y que es propagado entre todos los componentes que estuvieran conectados a él.

Ejemplos de estas conexiones los tenemos en que la mayoría de los pines del microcontrolador son puertos de carácter general pero que a estos están conectados

los específicos de cada periférico. Un cambio en el estado de los pines de carácter general, producirán envíos de mensajes a los periféricos que estuvieran conectados a él (puerto de entrada) o al revés, que un periférico cambiara el estado de su pin y se propagara al pin de carácter general y desde ahí a los que estuvieran conectados a él. Este mecanismo puede producir una ralentización del sistema pero es una manera de mantener todo sincronizado y hacerlo lo más real posible.

Además, sobre la lentitud del sistema en general, también influye el objeto *TAltavoz* que tiene que enviar datos a la tarjeta de sonido cuando el buffer está lleno, influye también que la pantalla LCD tiene que estar constantemente refrescada (al menos, en la vida real) y a que se tienen que controlar muchos periféricos por cada ciclo como también es detectar si se ha producido una interrupción. El intérprete de Java no es suficiente para poder controlar todo esto y hacerlo lo más aproximado a una ejecución a tiempo real. A pesar de haber hecho la emulación de las instrucciones con un recompilador dinámico (las instrucciones constantes son descodificadas en tiempo de ejecución para evitar tener que volverlas a descodificar), se sigue sin alcanzar una buena velocidad. Sobre las instrucciones influye mucho el estado de la ALU. *Java* no tiene instrucciones para recoger los *flags* de la CPU donde se está ejecutando el emulador, por lo que hay que analizar tanto las operaciones como los operandos para poder obtener los *flags* en lugar de realizar esa ejecución en la CPU real y recoger el resultado directamente. Sin embargo, se eligió Java para poder portar el emulador sin tener que compilar para cada sistema.

Para finalizar, quisiera aclarar que este proyecto tiene como fin el poder realizar una herramienta de apoyo para los programados entusiastas del sistema *Lego MindStorm* así como ayudar a los estudiantes de asignaturas de micro robótica para ganar en tiempo, en un principio, al probar sus programas en el emulador y poder comprobar el comportamiento de sus programas antes de montar el sistema completo y que pudiera romperse alguna pieza por algún error de programación que tiene todo el mundo hasta los más expertos.

BIBLIOGRAFÍA

RCX Internals

Página de Kekoa Proudfoot donde se analiza el sistema incluso la ROM y el Firmware del RCX original.
http://graphics.stanford.edu/~kekoa/rcx/

H8/300 Programming Manual

Manual sobre las instrucciones de la familia H8 de Hitachi y que se encuentra en Internet como libre distribución.

Hitachi Single-Chip Microcomputer – Hardware Manual (3ª edición)

Especificación sobre la familia H8/329x de Hitachi y que también se encuentra como libre distribución.

leJOS, a Java system for the Lego Mindstorms RCX

Sistema operativo que viene a sustituir el firmware de Lego y que se basa en el lenguaje Java para la programación de los autómatas.
http://lejos.sourceforge.net/

legOS

Otro sistema operativo basado programado en un leguaje parecido al C, llamado NQC, siguiendo los estándares POSIX.
http://www.noga.de/legOS/