Migración del GCC a un procesador RISC

Nicolás Alvarez, Leonardo D. Alabart, Miguel A. Sagreras, Alberto Dams.
nalvare@cactus.fi.uba.ar, lalabar@cactus.fi.uba.ar, msagre@cactus.fi.uba.ar,
adams@cactus.fi.uba.ar

Laboratorio de Sistemas Digitales, Departamento de Electrónica, Facultad de Ingeniería
Universidad de Buenos Aires
Av. Paseo Colón 850, (1063) Ciudad de Buenos Aires.
Tel.: 4342-9184, int. 297

Resumen

Los sistemas embebidos requieren procesadores con arquitecturas específicas para las tareas que desarrollan. Sin embargo, una vez definida la arquitectura se requiere de un compilador para que la misma sea utilizable.
El gcc es un compilador pensado para ser migrado a diferentes arquitecturas. A pesar de esto, se encuentran dificultades al realizar una migración específica.
En este trabajo se detallan los puntos clave y los principales obstáculos que se presentaron al migrar el GCC a un nuevo conjunto de instrucciones para una arquitectura con características RISC.

1. Introducción

- **El mecanismo de migración**

Cuando se migra el GCC a una nueva arquitectura, ésta debe ser descripta en detalle. Esto es realizado en parte con macros de C, declarando valores de propiedades generales, y detallando las instrucciones y los tipos de operandos.
Esta descripción se encuentra mayormente en tres archivos: un archivo ".h", otro archivo ".c" (ambos archivos con sintáxis de C) y el archivo de descripción de la máquina ".md".
La descripción de la máquina se realiza en un lenguaje intermedio llamado RTL (Register Transfer Language). El archivo '.md' contiene un patrón para cada instrucción de la arquitectura (o al menos cada instrucción que tiene importancia que el compilador conozca). Además contiene la definición de los distintos atributos. Muchas de las funciones utilizadas en este archivo se definen en el archivo '.c'.

Cada patrón de instrucción contiene una expresión RTL incompleta, con partes que deben ser llenadas más tarde, restricciones para los operandos, y un patrón de salida (o un código en C para generar la salida). A la derecha se puede observar el formato a cumplir:

(define_tipo-de-patrón "(opcional) nombre-del-patrón"
[(set (target)
(operando))
opcional expresiones]
"opcional condición"
"output template"
(opcional [atributos]))

En el archivo '.h' se encuentra casi todo el trabajo inicial de la portación, realizado para describir la arquitectura y la ABI (Interfaz Binaria de Aplicación).
Los diferentes nodos que componen este archivo son: control de la compilación, especificación de parámetros de la máquina en tiempo de ejecución, almacenamiento, tipos de datos utilizados en el lenguaje, uso de los registros, clases de registros, estructura de la pila y convenciones de llamada a

función, modos de direccionamiento, descripción de los costos relativos de las operaciones, división de la salida en secciones, definición del lenguaje ensamblador de salida, salida de datos, salida de variables no inicializadas, generación y salida de etiquetas, salida de instrucciones en lenguaje ensamblador, dispatch tables, alineamiento, formato de la información para depuración, y parámetros varios.
El archivo '.c' no es más que una extensión del archivo '.h' y algo del '.md'. No tiene asociada específicamente ninguna fase de desarrollo. Cuando una macro en el archivo '.h' es demasiado complicada, se construye en el archivo '.c' una función con el mismo nombre de la macro, pero en minúsculas. Esto implica una importante mejora en la legibilidad y una ayuda en la tarea de depuración.

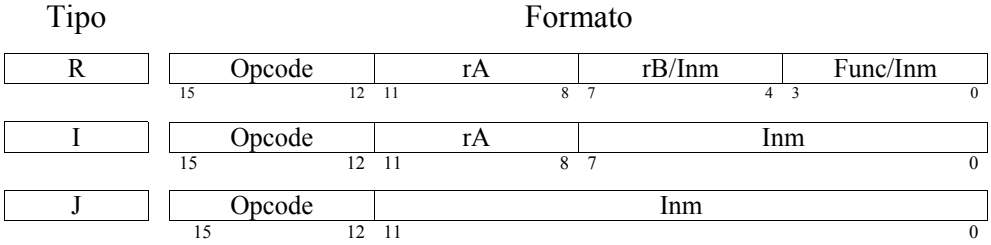
Debido a que el trabajo de portación no es una actividad demasiado difundida, no existe una forma única de proceder. De todos modos, a partir de la información obtenida de otras portaciones y de las experiencias vividas durante el desarrollo del trabajo en cuestión, se pueden establecer ciertas pautas a tener en cuenta.
Primeramente se debe modelizar la ABI (Interfaz Binaria de Aplicación). Si existe una ABI que presente características similares a las que se pretende tener se la puede utilizar, y sino se deberá comenzar desde cero.
Se deben determinar las características de la máquina sobre la cual se pretende correr el código generado por el compilador, como ser los tipos básicos y la estructura de la memoria. Se debe definir la forma en que las funciones son llamadas y de que manera devuelven valores.
Seguidamente se debe escribir el archivo '.md' de una manera simplificada, con sólo las instrucciones básicas que son necesarias para lograr un mínimo funcionamiento. Del mismo modo se deben escribir los archivos '.h' y '.c'. A medida que se logran resultados positivos se debe ir trabajando sobre los detalles no tenidos en cuenta en un primer momento, hasta llegar a cubrir todas las características de la arquitectura.
Después de haber sometido a pruebas al compilador mediante una serie de programas de prueba (comprobación detallada de las diferentes características) se debe utilizar un programa 'real' para tener cabal dimensión del funcionamiento del compilador; algunos errores pueden surgir cuando ya se presenta una cierta interacción entre las distintas partes de un programa.

- **Breve descripción de la arquitectura del procesador**

Arquitectura Load-Store.
Arquitectura totalmente abierta.
Cumple con la filosofía RISC (Reduce Instruction Set Computer).
Contine un "pipelining" de 5 etapas diseñado para ser integrado en FPGAs.
Cantidad de registros físicos: 16 (r0 ... r15).
Tamaño de registro: 32 bits.
Registros ortogonales (excepto por el r0 que se utiliza para la dirección de retorno en las instrucciones JAL y JRAL).
Tamaño de datos: 8, 16, y 32 bits.
Longitud de las instrucciones: 16 bits (tamaño de opcode fijos).
Modos de direccionamiento: Inmediato, Registro, Indirecto + Desplazamiento.
Posee carga de valores inmediatos de 8 bits.
Manejo de punto flotante: Por software.
Manejo de mult. y div.: Por software.
Implementación de la pila: Por software.
Branch delay slot para mantener al "pipeline" lo más lleno posible.
Las instrucciones tienen 2 operandos como máximo.
El resultado es colocado en uno de los registros operando.

Saltos condicionales con desplazamiento inmediato de 8 bits.
Implementación de saltos relativos al PC lo que permite la generación de código independiente de la posición (PIC - Position Independent Code).
Bits de estado: bit C (Carry), bit N (Negative), bit T (Test).

A continuación se muestra el formato de los tres tipos de instrucciones:



2. El archivo 'carp.md': Descripción de la máquina

Una de las primeras cosas que se definieron en este archivo fueron los atributos (*define_attr*) "*type"* y "*length"*. El primero de ellos se utilizó para poder diferenciar los distintos tipos de instrucciones (*unknown*, *load*, *store*, *move*, *extend*, *logic*, *add*, *mul*, *shift*, *compare*, *branch*, *jump*, *fp*) y el segundo para poder especificar la cantidad de instrucciones, en lenguaje ensamblador, que se necesitan para realizar una determinada operación.
Seguidamente se fueron confeccionando las diferentes operaciones (algunas de las cuales son obligatorias, sin ellas el compilador no podría funcionar).
Una de las principales es la operación *mov*, ya que su ausencia implicaría la imposibilidad de realizar transferencia de datos dentro de la máquina. Se definieron tres diferentes *insns* para este tipo de operación, una para cada modo de máquina (SI, HI, QI), lo que representa el movimiento de datos de distintos tamaños. Este movimiento se puede dar de las siguientes tres maneras: de registro a memoria, de memoria a registro y de inmediato a registro. Todo esto está contemplado con el uso de las diferentes letras de restricción (definidas en el archivo carp.h).

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,r,m,r,r,r")
                (match_operand:SI 1 "general_operand"   "r,m,r,I,K,i"))]
  ""
  "*
 {
  int x1, lo, hi;
  switch(which_alternative)
   {
     case 0:
       return \"mov  %0,%1\\t # movsi: reg to reg.\";
     case 1:
       return \"lw   %0,%1\\t # movsi: load.\";
     case 2:
       return \"sw   %0,%1\\t # movsi: store.\";
     case 3:
       x1 = (unsigned) INTVAL (operands[1]); lo = x1 & 0x000000FF; hi = x1 & 0x0000FF00;
       hi = hi >> 8;
       if (INTVAL (operands[1]) > -1)
         { ... }
       else
         { ... }
     case 4:
```

```
x1 = (unsigned) INTVAL (operands[1]); lo = x1 & 0x000000FF; hi = x1 & 0x0000FF00;
hi = hi >> 8;
if (INTVAL (operands[1]) != 0)
  {
   output_asm_insn (\"xor  %0,%0\\t # movsi: 16  bits unsigned imm.\", operands);
   operands[1] = GEN_INT (hi); output_asm_insn (\"lih  %0,%1\", operands);
   operands[1] = GEN_INT (lo); return \"ori  %0,%1\";
            }
else
  return \"xor  %0,%0\\t # movsi: 16  bits unsigned imm (zero).\";
case 5:
 output_asm_insn (\"xor  %0,%0\\t # movsi: 32 bits symbolic constants.\", operands);
 output_asm_insn (\"lih  %0,hh(%1)\", operands);
 output_asm_insn (\"ori  %0,hl(%1)\", operands); output_asm_insn (\"llsi %0,0\", operands);
```

En este patrón se puede observar una resolución directa de los primeros tres casos, en los que se emite simplemente el assembler correspondiente a los movimientos de datos entre registros y memoria-registro (incluye a las instrucciones de máquina *mov*, *lw* y *sw*). Los siguientes casos tienen en cuenta los valores de datos inmediatos y las constantes simbólicas. Ya que la arquitectura cuenta con instrucciones que hacen posible la transferencia de valores constantes de sólo 8 bits se debió hacer uso de más de una instrucción para lograr movimientos de datos mayores. Para el caso de las constantes simbólicas (que no son resueltas por el compilador) se utilizaron las pseudo instrucciones *ll()*, *lh()*, *hl()* y *hh()*, que sirven para gobernar los cuatro octetos que conforman el dato. Debido a esto el ensamblador debió ser adaptado para poder reconocerlas como instrucciones válidas.
Las operaciones de extensión de signo y de extensión de cero se han simulado con sendas secuencias de desplazamientos, tanto aritméticos como lógicos.
Las operaciones de desplazamiento permiten un inmediato de sólo 4 bits, lo cual se refleja con la letra de restricción L.
Para la construcción de los saltos condicionales (*branches*) se hizo uso de una función desarrollada en el archivo de portación ".c" llamada *gen_cond_branch*, la cual recibe como parámetros los operandos y el tipo de comparación que se desea realizar (ej: *gen_cond_branch (operands, EQ)*).
Las insns "*nop"* y "*one_cmplsi2"* se simularon con las instrucciones "*and r0,r0"* y "*nor"*, respectivamente.
Debido a que no se cuenta con instrucciones de máquina *jump* sin linkeo, la insn "*jump"* se implementó con la instrucción de máquina *jal*, y la insn "*indirect_jump"* con *jral*.

3. El archivo 'carp.h'

La siguiente explicación se centra en los aspectos de la migración más importantes introducidos por medio de este archivo.
Para el pasaje de parámetros a una función se estableció que los cuatro primeros fueran pasados a través de registros; esto se establece por medio de la macro *FUNCTION_ARG_REGNO_P(N).* A este número se llegó tomando en cuenta las estadísticas existentes, pruebas realizadas en nuestra migración, y por la cantidad reducida de registros con que cuenta la arquitectura. A partir del quinto parámetro, inclusive, todos son pasados por la pila.
Se definieron las siguientes correspondencias existentes entre los tipos de datos de C y los que posee el compilador migrado: El tipo de datos char (signado o sin signo) es un byte signado o sin signo (8 bits), el tipo de datos short int (signado o sin signo) es una media palabra signada o sin signo (16 bits), los tipos de datos int y long (signados o sin signo) son una palabra signada o sin signo (32 bits), los punteros, a cualquier tipo, son representados como enteros sin signo de 32 bits.
Los tipos *structure*, *array* y *union* asumen el alineamiento de su elemento más restrictivo (en cuanto

a alineamiento se trata), el tipo *array* utiliza el alineamiento de sus elementos, los tipos *structure* y *union* pueden requerir dejar posiciones de memoria sin utilizar (padding) para lograr las restricciones de alineamiento (cada elemento es asignado a la dirección alineada más baja).
En lo referente al retorno de valores (macro *GP_ARG_RETURN*), una función lo efectúa (ya sea un valor entero o un puntero) colocándolo en el registro de uso general r8 (originariamente se había elegido el r4, el mismo para pasaje del primer parámetro, pero se lo tuvo que cambiar debido a que el registro utilizado para devolver valores de una función es el único que se puede utilizar como auxiliar en la implementación de la macro *PROLOGUE*, y como resulta evidente, si este fuera el r4 se estaría destruyendo el valor del primer parámetro, en caso de que existiera). Esta última macro es responsable de preparar la pila, inicializando el registro utilizado para el frame pointer, salvando los registros que deben ser salvados, y reservando lugar de almacenamiento para las variables locales.

4. El archivo 'carp.c'

Una de las funciones definidas aquí es la encargada de emitir el código en assmebler, que es llamada desde el 'carp.h' de la siguiente manera:

#define PRINT_OPERAND(FILE, X, CODE) print_operand(FILE, X, CODE)

Las demás funciones que introdujimos en este archivo son:
carp_output_function_epilogue, carp_output_function_prologue, print_operand_punct_valid_p, print_operand_address, gen_cond_branch.
Esta última función es la encargada de generar una comparación y luego un salto (ambas expresiones rtx).

5. Conclusiones y estado de desarrollo

La migración expuesta en este escrito ha sido probada con el software denominado Dhrystone, obteniéndose resultados aceptables. El compilador sigue siendo probado con otros programas de modo tal de contar con una comprobación más exhaustiva. También se están ensayando modificaciones en los parámetros de la portación con el objeto de lograr la emisión de un código más reducido.
Asimismo se tiene como una próxima etapa la realización de optimizaciones de bajo nivel, las cuales se desarrollan en la etapa que genera el código RTL.

6. Referencias

[1] Stallman, R. M., *Using the GNU Compiler Collection. For version 3.1*, Free Software Foundation Inc., Boston, U.S.A., 1998.

[2] Patterson, D. A., Hennessy, H. L., con distribución de D. Goldberg. *Computer Architecture: a quantitative approach*, Morgan Kaufmann Publishers, San Francisco, U.S.A., 1996.

[3] OpenCores.org and authors*, OpenRISC 1000 Architecture Manual*, 2003.

[4] Da Silva Gillig, J. U., Sagreras M. A., Dams A., *6617: Arquitectura RISC de ancho de palabra de datos parametrizable para implementaciones sobre tecnología FPGA*, Buenos Aires, Argentina, 2003.