Procesadores Comerciales (3º Grado Informática)

Arq**T**ec**C**om - Dpto. Informática e Ingeniería de Sistemas.

CPS - UniZar

Práctica 1: Simulador de un procesador segmentado

Los objetivos de esta práctica son: i) conocer un simulador que modela un procesador segmentado de cinco etapas, ii) modificar el simulador para modelar operaciones multiciclo, iii) estudiar el comportamiento y cuantificar las prestaciones del procesador.

Descripción del simulador de partida

En esta práctica se estudia y posteriormente se modifica un simulador sencillo de un procesador segmentado. El simulador consta de varios ficheros en C que se encuentran en:

```
pilgor:/export/home/alumnos/a01/segmentado/
```

En esa misma ubicación también se encuentra un fichero `makefile` para compilar los ficheros y crear el ejecutable (`make`).

El simulador proporcionado modela el comportamiento temporal (ciclo a ciclo) del procesador. Obtiene el número de instrucciones ejecutadas por un programa y el número de ciclos consumidos para su ejecución. El núcleo del simulador consiste en un bucle que ejecuta una iteración por cada ciclo simulado. Cada iteración simula el comportamiento temporal de cada una de las etapas del segmentado, empezando por las etapas del final y terminando por las del principio.

Cada etapa se simula mediante una variable (algo parecido al IR) que contiene la información relativa a la instrucción a ejecutar. Su definición se encuentra en el fichero cabecera.h:

```
typedef struct {
  unsigned long long pc;      /* PC de la instrucción */
  unsigned long long ea;      /* dirección efectiva (load/store/salto) */
  unsigned long iw;           /* palabra de instrucción */
  char rd, rs1, rs2, co;      /* registros usados y codigo de operación */
                              /* registro = -1 indica registro no usado */
                              /* 0 <= registro <= 31 indica registro entero */
                              /* 32 <= registro <= 63 indica registro pf */
  char an, nreg, taken;       /* otros */
} IREG;
```

En cada ciclo, el simulador procesa la instrucción de cada etapa (almacenadas en variables de tipo IREG: etapa_Bin, etapa_Din, etapa_Ain, etapa_Min y etapa_Ein), y genera una salida que se almacena en otra variable (etapa_Bout, etapa_Dout, ...). Cada etapa genera también la señal de carga de su registro de entrada, que valdrá "1" cuando la etapa queda libre y valdra "0" cuando la etapa queda ocupada. La salida de una etapa será la entrada de la etapa siguiente durante el ciclo siguiente.

El código de partida que se proporciona simula un procesador segmentado lineal como el estudiado en las clases teóricas: cinco etapas con todos los cortocircuitos posibles con destino a la etapa decode, y memoria ideal. Todos los saltos se resuelven en la etapa de decodificación y son retardados, por lo que no provocan detención. El modelo de procesador que vamos a usar como base sólo se detiene ante un riesgo RAW entre una instrucción LOAD productora y su consumidora a distancia uno. En el fichero cpu.c , en la función principal -sim()- se encuentra el bucle que simula este modelo (CPI >1):

```
while (quedan_inst)
{
  /* etapa Escritura en BR, nada que simular en esta etapa */
  etapa_Eout = etapa_Ein;
  carga_E = 1;    /* la etapa E siempre queda libre */

  /* etapa Memoria, nada que simular en esta etapa */
  etapa_Mout = etapa_Min;
  carga_M = 1;    /* la etapa M siempre queda libre */

  /* etapa Alu, nada que simular en esta etapa */
  etapa_Aout = etapa_Ain;
  carga_A = 1;    /* la etapa A siempre queda libre */

  /* etapa Decode, deteccion de riesgos y parada en caso de RAW,
     si no hay problemas la instruccion pasa a A */
  if (etapa_Ain.co == LOAD &&
       (etapa_Ain.rd == etapa_Din.rs1 || etapa_Ain.rd == etapa_Din.rs2)) {
       etapa_Dout = inula;    /* la instruccion no progresa */
       carga_D = 0;           /* la etapa D NO ha quedado libre */
  }
  else {
       etapa_Dout = etapa_Din;
       carga_D = 1;           /* la etapa D SI ha quedado libre */
  }

  /* etapa Busqueda, nada que simular en esta etapa */
  etapa_Bout = etapa_Bin;
  carga_B = carga_D;  /* la instrucción de la etapa B solo progresa si
                              la etapa D ha quedado libre */

  /* si la etapa B ha quedado libre,
       leemos otra instruccion de la traza */
  if (carga_B) quedan_inst = get_instr(&etapa_Bin);

  reloj();
}
```

La función `reloj()` se encarga de copiar el registro de salida de cada etapa sobre el de entrada de la etapa siguiente:

```
void reloj()
{
    if(carga_D) etapa_Din = etapa_Bout;
    if(carga_A) etapa_Ain = etapa_Dout;
    if(carga_M) etapa_Min = etapa_Aout;
    if(carga_E) etapa_Ein = etapa_Mout;
    tiempo++;
}
```

Las definiciones de los códigos de operación se encuentran en cabecera.h:

```
/* códigos de operación definidos (campo .co de IREG) */
#define NOP 0
#define LOAD 1
#define STORE 2
#define ARITM 3
#define BRCON 4 /* salto condicional */
#define BRINC 5 /* salto incondicional */
#define FLOAT 6
#define OTROS 7
```

Las funciones inicpu() y fincpu() del fichero cpu.c se ejecutan al principio y al final de la simulación respectivamente y sirven para inicializar estructuras de datos la primera y para escribir resultados de simulación la segunda.

El simulador lee una traza generada por una herramienta de SUN Microsystems llamada shade. Para cada instrucción, la traza contiene información relativa a su ejecución. Shade permite parametrizar tanto las instrucciones para las que se desea generar información (ej. sólo los saltos, sólo los loads y stores, o todas las instrucciones) como la información que se desea de cada una de ellas (ej. contador de programa, palabra de instrucción, valor de los registros fuente y destino, dirección efectiva de memoria o de salto, etc.). Si alguien tiene curiosidad, puede consultar más información acerca de shade en la referencia [1].

Compilación y ejecución

Ejecutando la aplicación make en el directorio donde habéis copiado los fuentes del simulador se crea un ejecutable llamado simseg.

Las simulaciones se lanzan de la siguiente forma:

```
./simseg -- benchmark
```

Donde benchmark es el programa cuya ejecución queremos simular. Como ejemplo, la siguiente línea de comando lanza nuestro simulador ejecutando el comando "ls -l":

```
./simseg -- ls -l
```

El tiempo necesario para simular la ejecución de un benchmark es varias decenas de veces mayor que el tiempo sobre la máquina real, por lo que se analizarán programas sencillos, en concreto los programas matrizfi.c y matrizc.c. Puedes encontrar su código fuente en:

```
pilgor:/export/home/alumnos/a01/bench/
```

Usa la opción de optimización -xO2 del compilador para obtener el ejecutable. De esta forma todos tendremos el mismo código y nuestros resultados serán comparables. L línea de compilación sería la siguiente:

```
cc -xO2 matrizc.c -o matrizc
```

Puedes obtener el código ensamblador de tu programa compilando con la opción -S. Como ejemplo:

```
cc -xO2 -S matrizc.c -o matrizc.s
```

Trabajo a realizar

Estudio del uso de los cortocircuitos del simulador base

Se trata de calcular el porcentaje de registros suministrados por el banco de registros, y por los cortocircuitos a distancias 1, 2 y 3 (suponiendo banco de registros con escritura de ciclo completo). Recordar que una instrucción puede tener 0, 1 ó 2 registros de entrada. Los campos rs1 y rs2 contienen el identificador de registro fuente utilizado o el valor -1 en caso de que la instrucción no utilice dicho registro. El campo rd indica el registro destino usado o el valor -1 en caso de que la instrucción no tenga registro destino. Podéis utilizar el programa scortos para verificar la corrección de vuestro código.

Modelado de unidades funcionales multiciclo con un solo camino de ejecución

En esta tarea vais a modificar el simulador para que considere la ejecución de instrucciones multiciclo en un modelo con un solo camino de ejecución. En concreto vamos a suponer que las instrucciones del tipo FLOAT tienen latencia de inicio y latencia de ejecución de 5 ciclos.

Modelado de unidades funcionales multiciclo con varios caminos de ejecución

El siguiente paso consiste en pasar de un segmentado con un solo camino de ejecución a otro con varios caminos. En concreto modelaremos tres caminos: 1) instrucciones de memoria, 2) instrucciones FLOAT, y 3) resto de instrucciones. Seguimos considerando que las instrucciones del tipo FLOAT tienen latencia de inicio y latencia de ejecución de 5 ciclos. En este apartado sólo modelaremos las paradas por riesgo estructural en la unidad de FLOAT y en la escritura en el banco de registros. No extenderemos el control de riesgos RAW y WAW más allá de lo que ya está implementado, la instrucción Load seguida de consumidora a distancia 1.

Experimentación y análisis de resultados

Para el benchmark matrizc, medir tiempos de ejecución para los distintos modelos. Descomponer los ciclos de detención en función de quien provocó su pérdida: bloqueo por RAW después de load, por riesgo estructural en la UF de float, ...

Referencias

Introduction to Shade.