Alonso Rodríguez Iglesias :: CHS 1920 – Examen 01-06-2020

1. Un ASIP es un procesador de aplicación específica que se caracteriza por la inclusión de instrucciones personalizadas para el dominio de aplicación (por ejemplo para routers, sistemas multimedia como decodificadores, etc).
Una característica básica de estos y que los diferencia de los ASIC (entre otras) es su filosofía: No se desarrollan desde cero, sino que se parte de una base sólida, como un procesador RISC, VLIW, etc, al que se le añaden instrucciones, componentes, etc.
Estos procesadores son sintetizados a medida y automáticamente mediante un software de diseño, por ejemplo algún software basado en LISA.
Así, podremos personalizar parámetros como la cantidad de registros, los módulos del procesador (por ejemplo una unidad de cálculo de FFTs…), etapas, etc.
El software de diseño, a parte del diseño del procesador, generará automáticamente herramientas como compilador de C, debugger, profiler, linker, ensamblador, simulador… En definitiva, herramientas para poder trabajar con el ASIP.

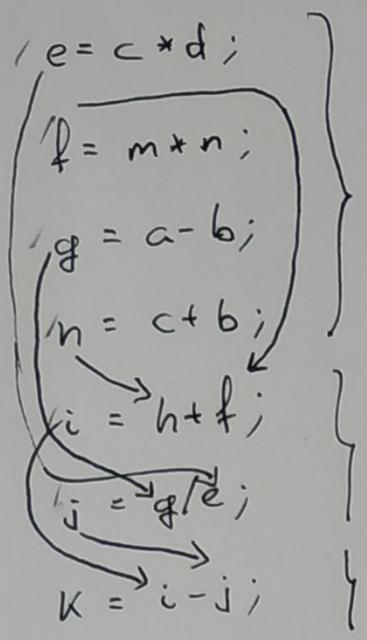
Entre los principales detalles que los diferencian de otras opciones tecnológicas podemos encontrar:

- Diseño modular y automatizado -> El procesador y las herramientas se crean de forma automática, ahorrando mucho tiempo de diseño y desarrollo, a diferencia de los ASIC, en los que se suele diseñar de cero y siguiendo una filosofía de grano mucho más fino.
- No es exactamente un procesador con coprocesadores y/o aceleradores -> Podemos personalizar muchos más parámetros, como el ancho de palabra, número de registros, etc, para ajustarlo a nuestras necesidades, tanto de potencia de cómputo como de potencia eléctrica.
- En todos los casos anteriores, es conveniente comenzar la optimización por las partes del código que requieren más tiempo de CPU, diseñando hardware específico para ellos, por ejemplo. Sin embargo en los ASIP, esto se puede realizar de forma mucho más automatizada y modular, por ejemplo, en el código LISA, añadiendo una unidad de procesado de FFTs, etc, con un muy buen acoplamiento.

2. No entiendo muy bien a qué se refiere con “el nivel de granularidad”, es decir, el nivel de granularidad de un programa con precisión de ciclo de reloj, es el ciclo, y debemos tener en cuenta el número de ciclos del método. Es un método analizado más a grano fino que por ejemplo un método implementado siguiendo el modelo dataflow. Creo que también se puede referir a qué tipo de método sería:
Un método sensible únicamente al ciclo de reloj, es un método cuyas señales están registradas, ya que solamente cambian al tic del reloj.
Sin embargo, si es sensible a más señales, tendríamos un método parcialmente asíncrono, como, por poner un ejemplo el banco de registros que implementamos mi compañero y yo en la parte de VHDL del proyecto, donde los registros funcionan de forma síncrona (no es como que pudiesen funcionar de otra manera) y los muxes de forma asíncrona.

3. Se ilustra a continuación:

Concurrencia



⇒ { e, f, g, h }
{ i, j }
{ k }

Paralelismo

| 2 sum / rest
| 1 mul / div

1- e = c * d ; g = a - b ; h = c + b ;

2- ~~g = a - b ;~~ f = m * n ;

3- i = h + f ; j = g / e ;

4 - k = i - j ;

=D { e, g, h }
{ f }
{ i, j }
{ k }

4. Primeramente, comentar que una optimización muy tonta que se nos pasó en el momento de programar el código es que hacemos lo siguiente:

```
addi $t0, $s3, -8
bne $t0, $0, for_n_1
```

cuando consume menos ciclos y es más sencillo poner un registro a al valor que restamos en el addi, y hacer un beq.

A parte de esto, optimizaciones al código podrían ser:

Dado que el código es bastante sencillito a nivel de que son únicamente sumas, desplazamientos, y operaciones bitwise, sin multiplicaciones ni operaciones similares, probablemente tenga un CPI bastante bueno. Ahora bien, dentro de la simpleza de las operaciones, tenemos partes del código que son más demandantes que otras: los for.

Y es que cada for interno se ejecuta 8 veces, es decir 16 iteraciones por 128 iteraciones del for externo.
Una opción sería, en el diseño, incluir un coprocesador que realizase cada uno de los for, ya que la operación que realizan es idéntica.
Como esta es una solución muy general, que no aporta detalle de cómo se realizaría, vamos a fijarnos en cómo se podría hacer:

```
sll $s2, $s2, 1
andi $t0, $s1, 1
or $s2, $s2, $t0
srl $s1, $s1, 1
```

Se puede realizar en un coprocesador con una instrucción. Esta se podría llamar: “sols”, de Shift left, And-Little, Shift right, y realizaría esas tres operaciones en un solo ciclo. Con ello conseguiríamos una mejora.

De esta forma, dependiendo de la velocidad, latencia, etc, del coprocesador, podría ser más conveniente que el coprocesador ejecutase directamente las 8 iteraciones del bucle for, o que se le llamase 8 veces desde el programa principal, aunque probablemente lo más beneficioso será la segunda opción.

Otra instrucción nueva que podríamos crear sería una que realice estas cuatro instrucciones en paralelo:

```
add $t5, $t1, $t3
sub $t6, $t1, $t3
add $t7, $t2, $t4
sub $t8, $t2, $t4
```

Y esto nos lleva al siguiente punto. Hasta el momento estas instrucciones son SIMD, es decir, creamos una sola instrucción que realice varias acciones sobre

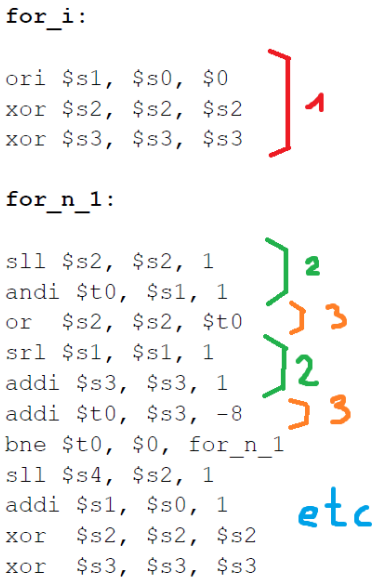
múltiples datos, que se ejecutan en un coprocesador, pero también tenemos la alternativa VLIW.

De esta manera podríamos tener un ASIC VLIW para operaciones en paralelo, al que mandaremos, por ejemplo, instrucciones independientes de cuatro en cuatro, o incluso instrucciones VLIW de tamaño flexible.

Para el caso anterior seria algo del estilo:

`add $t5, $t1, $t3`	`sub $t6, $t1, $t3`	`add $t7, $t2, $t4`	`sub $t8, $t2, $t4`

De esta forma, como tenemos un montón de operaciones independientes, si nos basamos en un chip VLIW podemos obtener una gran mejoría. Así, podríamos agrupar las operaciones independientes tal que, por ejemplo, para 4 instrucciones por instrucción VLIW:



Una gran ventaja de los VLIW es que no son tan complicados de implementar como los chips superescalares, por lo que tenemos un importante ahorro de coste ahí.

Además, al emplear un ASIP VLIW se da por hecho que ya no es un MIPS como tal, pero por si acaso listo los componentes a modificar o eliminar:

i. Eliminar coprocesador de multiplicación y punto flotante = ↓ coste
ii. Podríamos reducir el tamaño del banco de registros = ↓ consumo
iii. Las ALU son componentes estándar, así que no se podría “simplificar” de forma tan sencilla, pero lo que es cierto es que necesitaríamos una ALU muy sencilla. Además, deberemos recurrir a una segmentación muy agresiva, o lo que es más probable, replicación de componentes.
iv. Otra ventaja es que no tendríamos que reprogramar casi nada, puesto que el compilador generado automáticamente se encargaría de optimizar el código de la mejor manera que considere.