Planificación dinámica y el algoritmo de Tomasulo

Tomás Nahuel Olivera, Facundo Nahuel Fernández

Abstract—Este informe describe los principios de la planificación dinámica para la ejecución de instrucciones, sus problemáticas y sus beneficios. Se hará un breve recorrido histórico sobre computadoras que implementaron esto y se analizarán dos algoritmos que usan esta planificación: el algoritmo del marcador y, principalmente, el de Tomasulo. Finalmente, se habla sobre la importancia que tuvo y tiene la implementación del algoritmo de Tomasulo en los procesadores actuales.

Index Terms—Pipelining, Planificación dinámica, Algoritmo de Marcador, Scoreboard, Algoritmo de Tomasulo.

1 Introducción

A principios de los años 60, se intentaba buscar la mejor manera de implementar el *pipelining* para ejecutar múltiples instrucciones paralelamente. En un principio, las implementaciones se basaban en que se buscaban y emitían las instrucciones en orden, pero surgía el problema de que, si una instrucción dependía de los datos de otra, se detenía este proceso, por lo que se debía planificar el envío de instrucciones mediante software para intenta minimizar las detenciones. Este enfoque es denominado *planificación estática*. Para evitar las interrupciones, durante los mismos años se desarrolló la ***planificación dinámica***.

2 La planificación dinámica

En la planificación dinámica, a diferencia de la estática, el hardware determina qué instrucciones se ejecutan. En esencia, el procesador ejecuta instrucciones **fuera de orden**. El orden de las instrucciones sí importa, pero se ejecutarán dependiendo de la disponibilidad de los operandos que utilicen, considerando obviamente las limitaciones del procesador (como la disponibilidad de unidades funcionales para ejecutar cada operación).

Si, por ejemplo, tenemos la siguiente secuencia de código:

```
DIV R1, R3, R2
SUB R9, R9, R6
ADD R7, R7, R5
```

con una implementación de planificación estática, debemos esperar que la operación DIV (de larga ejecución) se complete para poder ejecutar SUB y ADD, pero estas podrían completarse antes que la primera termine al no existir dependencias de datos. Este es el principal problema que la planificación dinámica soluciona. Además, permite que códigos compilados pensados para un determinado tipo de pipelining se ejecuten de manera eficiente en un procesador con otro tipo. Como desventajas se encuentran el significativo aumento de complejidad del hardware y la aparición de riesgos por dependencias de datos (data hazards) al ejecutar instrucciones fuera de orden.

Los dos algoritmos más conocidos que utilizan la planificación dinámica son el de Marcador (Scoreboard), utilizado por primera vez en la computadora CDC 6600 en el año 1964 [3]; y el de Tomasulo, implementado por Robert M. Tomasulo, por primera vez en el procesador IBM360/91 en el año 1966 [1], [5].

3 Riesgos por dependencia de datos

Los riesgos por dependencia de datos en pipelining ocurren cuando una instrucción depende de los resultados de otra instrucción que la precede, si estos resultados aún no han sido calculados.

Hay tres tipos de estos riesgos:

- **RAW**: Read after Write, o lectura después de escritura, ocurre cuando una instrucción trata de leer el resultado de una instrucción que aún no ha sido escrito. Es el riesgo más común de los tres. Considere las dos instrucciones siguientes como ejemplo:

  I1. R2 ← R5 + R3
  I2. R4 ← R2 + R3

  La primera instrucción guarda un valor en R2 y la segunda usa este como operando para una suma. Si el resultado de la primera operación no fue guardado, la segunda no debe ejecutarse (I2 depende de I1).

- **WAR**: Write after Read, o escritura después de lectura, ocurre cuando un registro que está siendo utilizado para una operación se utiliza para escribir en una instrucción posterior. Considere las dos instrucciones siguientes como ejemplo:

  I1. R4 ← R1 + R5
  I2. R5 ← R1 + R2

Cuando existe una posibilidad de que la segunda instrucción termine antes que la primera, se debe asegurar que el resultado del registro R5 no se guarde antes de que la primera instrucción haya obtenido los operandos.

- **WAW**: Write after Write, o escritura después de escritura, ocurre cuando un registro es utilizado para escribir en dos instrucciones consecutivas. Considere las dos instrucciones siguientes como ejemplo:

  I1. R2 ← R4 + R7
  I2. R2 ← R1 + R3

  La escritura de I2 debe ser pospuesta hasta que I1 haya completado su ejecución.

4 ALGORITMO DE MARCADOR

En el algoritmo de marcador, las instrucciones siguen emitiéndose en orden, debido a que pueden existir riesgos estructurales. A pesar de esto, y a diferencia de una planificación estática, una instrucción comienza su ejecución ni bien sus operandos estén disponibles. Al ejecutar fuera de orden, pueden existir los riesgos WAR y WAW. El responsable de emitir y/o ejecutar instrucciones que no dependan de otras será un marcador que detectará todos los riesgos.

Para que la ejecución fuera de orden sea mejor, se debe contar con múltiples unidades funcionales para ejecutar varias instrucciones a la vez. Es por esto que el CDC 6600 tenía 4 unidades de punto flotante, 5 para referencias a memoria y 7 para operaciones enteras.

El algoritmo cuenta con cuatro pasos (no se consideran pasos para accesos a memoria, nos centraremos en operaciones de punto flotante):

- Emisión: Si una unidad funcional está libre (se chequean riesgos estructurales) y no hay otra instrucción ejecutándose que tenga el mismo registro destino, el marcador le pasa la instrucción a la unidad funcional. Notar que se evitan riesgos WAW al verificar el registro destino.
- Lectura de operandos: El marcador comprueba si los operandos de entrada están disponibles. Un operando está disponible si no existe alguna instrucción activa, ya emitida, que vaya a escribir en tal operando. Una vez que se lleva a cabo esta verificación, la unidad funcional puede leer los operandos de los registros correspondientes para iniciar la ejecución. De este modo, el marcador resuelve dinámicamente los riesgos RAW.
- Ejecución: La unidad funcional ejecuta la instrucción usando los operandos y, al finalizar, se notifica al marcador que el resultado está disponible.
- Escritura de resultados: El marcador comprueba los riesgos WAR al finalizar de ejecutarse una instrucción. Si existe el riesgo, se detiene la escritura hasta que este desaparezca. Cuando no existe, se indica a la unidad funcional que escriba el resultado en el registro destino.

No nos centraremos en las partes que tiene el marcador y cómo es que se conecta con las unidades funcionales para llevar a cabo la emisión y ejecución de instrucciones. Sólo diremos que lo más costoso del CDC 6600 fue que tenía un conjunto de buses que se conectaban con cuatro grupos de unidades funcionales, y solamente una unidad de un grupo podía escribir/leer operandos durante un ciclo. Esto se hizo para solucionar que, si existía más de una unidad funcional que podía llevar a cabo los pasos de lectura o escritura, no podían ser más estas unidades que la cantidad de buses disponibles.

Además de la limitación mencionada anteriormente, hay un claro problema con los riesgos WAR y WAW. Para ambos casos, aunque se dan en diferentes etapas, se detiene la emisión de instrucciones hasta que los riesgos no existan, causando que no se ejecuten tantas instrucciones al mismo tiempo y haciendo que el algoritmo no sea bueno resolviendo estos casos.

5 ALGORITMO DE TOMASULO

IBM, compitiendo con CDC, tenía el objetivo de conseguir un alto rendimiento en operaciones de punto flotante debido a los grandes retardos que tenía, en un principio, el IBM 360/91 en este aspecto. Para esto, unos años después del lanzamiento del CDC 6600, diseñaron el algoritmo de Tomasulo y lo implementaron en su computadora, consiguiendo muy buenos resultados.

A diferencia del algoritmo de marcador, el de Tomasulo no detecta los riesgos centralizadamente sino que el control de ejecución está distribuido en **estaciones de reserva** ubicadas en cada unidad funcional; los datos van a las unidades funcionales directamente en vez de ir desde registros gracias a la implementación de un **bus común de datos**, que permitió que todas las unidades esperen simultáneamente que un operando se cargue; y el uso de **renombrado de registros** que permitirá evitar los riesgos WAR y WAW sin tener que frenar la emisión de instrucciones.

5.1 Unidad de Punto Flotante

En Fig. 1 se observa una unidad de punto flotante de Tomasulo, que está formada por los siguiente elementos:

- Una cola de instrucciones conectada a la unidad de instrucciones. Estas se emiten y son guardadas en las estaciones de reserva. Llamado FLOS (Floating Operation Stack) por Tomasulo.

- Dos buffers: uno de carga y uno de almacenamiento. Guardan datos que van y vienen de memoria. Ambos tienen campos para emplear el control de riesgos.

- Registros de punto flotante: guardan valores, están conectados por un par de buses a las unidades funcionales y por un único bus a los buffers. Tienen un bit que marca si están ocupados o no (indica si alguna unidad lo está utilizando).

- Unidades funcionales: Hay unidades funcionales que realizan operaciones de suma y otras de multiplicación/división. En la figura vemos una unidad de cada tipo pero pueden haber múltiples de estas para realizar operaciones simultáneamente. Cada unidad funcional tiene una estación de reserva.

- Bus Común de Datos: Todos los resultados de las unidades funcionales y de memoria se envían a este bus, que va a todos los sitios.

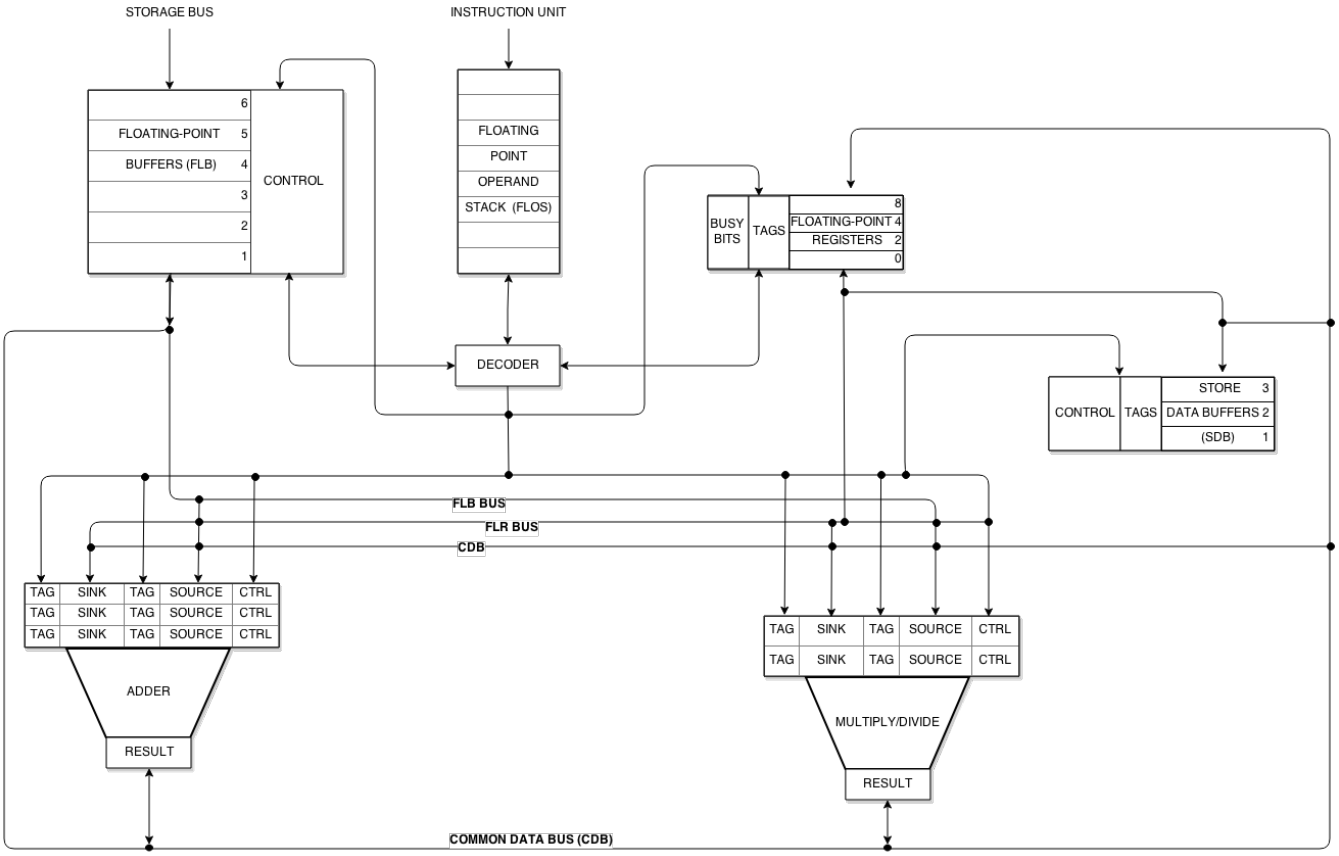

Fig. 1. Unidad de Punto Flotante de Tomasulo

- Estaciones de reserva: Cada estación de reserva tiene seis campos.

- OP: la operación a realizar entre los operandos (por ejemplo: ADD).
- Q_j, Q_k: Son las estaciones de reserva que producirán el correspondiente operando. Si el valor es 0, significa que el operando está disponible en V_j o V_k.
- V_j, V_k: Es el valor de los operandos. Si guarda un cero, significa que se espera el valor desde donde indica el correspondiente Q.
- Busy: Indica que la estación de reserva está ocupada al igual que su unidad funcional.

Notar que si Q es 0, V estará disponible y viceversa.

Tanto los buffers de almacenamiento como los registros tienen un campo Q_i, que es el número de la unidad funcional que producirá un valor para que se almacene en este registro o en memoria. Si es cero, ninguna instrucción activa está calculando un resultado para este registro o buffer.

Excepto los buffers de carga, todas las partes que guardan valores contienen un campo de etiqueta. Este campo es un valor que sirve para denotar una de las estaciones de reserva o uno de los buffers de carga. Describe la unidad funcional (o buffer de carga) que producirá un resultado necesario como un operando. Valores no usados, como cero, indican que el operando está disponible.

5.2 Etapas del algoritmo

Similarmente al algoritmo del marcador, en el algoritmo de Tomasulo pueden identificarse tres etapas para las instrucciones:

- Emisión: Se emite una instrucción desde la cola de instrucciones. En primer lugar, se chequea si no hay riesgos estructurales, es decir, si hay buffers o estaciones de reserva libres, según corresponda. En caso de no haber riesgos, existen las dos opciones que se detallan:
  - Si es una operación de punto flotante, va a una estación de reserva vacía, guardándose el tipo de operación a realizar y el valor (V) de los operandos (si no están disponibles, la estación o buffer que lo producirá (Q)).
  - Si es una operación de carga o almacenamiento, simplemente se emite la instrucción.
- Ejecución: Si los operandos no están disponibles, se vigila el bus común de datos, esperando que el registro se compute. La unidad funcional no debe estar ocupada ejecutando otra operación.
- Escritura de resultados: Cuando el resultado está disponible, se escribe en el bus común de datos y este se encarga de escribirlo en los registros y unidades funcionales que lo estén esperando.

Notar que en el segundo paso se evitan los riesgos RAW, ya que un operando no estará disponible para lectura

antes de que una instrucción anterior lo escriba. Además, el operando se guarda desde el bus común de datos y no se espera que se actualice el registro para luego obtener su valor. Esto permite que muchas instrucciones que esperan un resultado puedan liberarse y empezarse a ejecutar simultáneamente, cosa que no sucedía con el marcador.

Aunque sea similar al algoritmo del marcador, gracias al renombrado de registros que se lleva a cabo mediante el uso de Q_k, Q_j, V_k y V_j, el algoritmo eliminó los riesgos WAW y WAR sin tener que detener la emisión de instrucciones.

Para ver como se evitan los riesgos WAR, podemos considerar la siguiente secuencia de código como ejemplo:

```
MUL F7, F0, F5
ADD F5, F9, F1
```

Claramente tenemos un riesgo WAR con el registro F5, el cual es eliminado de dos maneras: si finalizó la ejecución de la instrucción que le da el valor a F5, se guarda en V_k el resultado y MUL puede ejecutarse independientemente de ADD. Por otro lado, si no se completó la ejecución de esa instrucción, Q_k tendrá señalada la operación de la que espera resultado y, nuevamente, MUL será independiente de ADD, que podrá ejecutarse aunque MUL esté a la espera. En ambos casos, MUL puede ejecutarse sin que exista el riesgo de que la instrucción siguiente modifique el registro y cause problemas.

Para los riesgos WAW, podemos pensar en la siguiente secuencia:

```
MUL F7, F1, F5
LD F0, F7
ADD F7, F9, F2
LD F2, F7
```

Si bien hay dos operaciones que quieren escribir en el registro F7, no tendremos problema en que ADD finalice antes, ya que las operaciones de carga, en vez de esperar a que se cargue el registro, guardaran una etiqueta que indicará que están esperando a que la instrucción anterior finalice (es decir, el primer LD guardará una etiqueta que diga MUL1 y el segundo guardará una etiqueta que diga ADD1, evitando el problema con F7).

5.3 Otras cuestiones importantes

Si bien se explicó sobre el algoritmo en una unidad de punto flotante, es importante aclarar que la aplicación de este no está limitada únicamente a operaciones aritméticas de este tipo o para una arquitectura como la de el computador de IBM que lo utilizó por primera vez. Puede ser usado en cualquier computadora con un procesador que tenga múltiples unidades de ejecución.

A continuación se presentará un ejemplo del algoritmo de Tomasulo aplicado a una secuencia de instrucciones. Algo que consideramos importante mencionar y que no se ve reflejado en el ejemplo elegido, es cómo el algoritmo actúa ante un loop: si se tiene una buena estrategia de salto condicional, el uso de las estaciones de reserva va a permitir que múltiples ejecuciones de un bucle se lleven a cabo al mismo tiempo. Si, por ejemplo, un bucle tiene una operación de multiplicación adentro, podremos tener en las unidades funcionales más de una multiplicación ejecutándose, por más que correspondan a diferentes iteraciones

5.4 Ejemplo práctico

Para el siguiente ejemplo partimos de las siguientes suposiciones para facilitar el entendimiento del algoritmo de Tomasulo:

- Hay dos unidades funcionales de suma y dos de multiplicación/división.
- Ya hay guardado en F4 el valor 2.2.
- Se toman los siguientes ciclos de reloj para cada operación: Load: 2 Add: 2 Substract: 2 Multiply: 10 Divide: 40
- Como etiquetas de las estaciones de reserva se guardarán las operaciones que se están esperando. Por ejemplo, si una instrucción espera como valor de un operando al resultado de la cuarta operación de suma que hay en la secuencia de instrucciones, se guardará la etiqueta ADD4.

Las siguientes instrucciones representan un programa que será ejecutado mediante el algoritmo de Tomasulo y se detallará qué es lo que sucede en cada ciclo de reloj, incluyendo una tabla que indica el estado de las estaciones de reservas.

```
I1. LD F1, 76
I2. LD F2, 90
I3. MUL F3, F2, F4
I4. SUB F5, F1, F2
I5. DIV F0, F3, F1
I6. ADD F1, F5, F2
```

- Ciclo 1: Se emite I1 (instrucción de carga). El operando de carga se envía junto con el valor inmediato 76 y se coloca en el buffer de carga. El registro de punto flotante F1 pasa a estar ocupado (el campo Q_i guardará una etiqueta LD1).

- Ciclo 2: Se emite otra instrucción de carga, I2, y funciona prácticamente igual que la instrucción anterior, con el operando de carga siendo enviado junto con el valor inmediato 90 para colocarse en el buffer de carga. También marcará el registro de destino F2 como ocupado de la misma manera que antes (el campo Q_i guardará una etiqueta LD2).

- Ciclo 3: Ya pasaron dos ciclos desde que la primera instrucción de carga (I1) se emitió, por lo que está a punto de terminar. El bus de datos común "se ilumina", es decir, los datos pueden difundirse a los suscriptores que los están esperando, en este caso es el registro de destino F1.
  Todavía hay una instrucción de multiplicación que debe ser emitida, así que primero la operación se guarda en una de las estaciones de reserva de la unidad funcional de multiplicación y esta se marca como ocupada. El primer operando, que viene de F4, está listo para carga (suposición inicial), por lo que se guardará el valor V_k. Sin embargo, el segundo operando F2 aún no está listo porque todavía está

en estado ocupado y eso es exactamente lo que esperaríamos que sucediera (está siendo utilizado por I2 para escritura, hay un riesgo RAW), entonces se guardará en Q_j la etiqueta LD2. Esta instrucción no estará lista para ejecutarse hasta que el resultado de la carga esté disponible.
El registro de destino F3 es marcado como ocupado.

Estaciones de reserva						
Nombre	Ocupada	OP	V_j	V_k	Q_j	Q_k
Add1	No					
Add2	No					
Mul1	Si	MUL		(F4)	LD2	
Mul2	No					

- Ciclo 4: La segunda instrucción de carga ya tuvo tiempo suficiente para terminar de ejecutarse y está listo para guardarse en el registro F2. Este valor no sólo está siendo escrito en el registro, ya que este no es el único suscriptor de la información, también lo necesitamos como un operando en la estación de reserva que guarda la multiplicación. El bus común de datos transfiere el valor directamente a ambos suscriptores.
  Además, tenemos una instrucción de resta que se emitirá. Funcionará de forma similar a la instrucción de multiplicación, aunque esta vez vamos a usar una de las estaciones de reserva de la unidad funcional de suma. Se puede guardar el valor de F1 en V_j, pero el segundo operando deberá suscribirse al bus de datos común para poder recibir el valor en el siguiente clock, se guardará la etiqueta LD2 en Q_k. Si bien la segunda instrucción de carga finalizó, F2 no fue cargado sino que está en el bus de datos. Ante este caso existen opciones: tomar el valor directamente o esperar al siguiente ciclo a que F2 está desocupado. Optaremos por el segundo, pero es cuestión de cómo esté diseñado el procesador y no depende del algoritmo de Tomasulo.
  Por último, se marca el registro F5 como ocupado.

Estaciones de reserva						
Nombre	Ocupada	OP	V_j	V_k	Q_j	Q_k
Add1	Si	SUB	(LD1)			LD2
Add2	No					
Mul1	Si	MUL		(F4)	LD2	
Mul2	No					

- Ciclo 5: F2 se ha escrito y ya no está ocupado. Gracias a esto podemos actualizar la estación de reserva que guarda la resta con el operando. Ahora ambas instrucciones están listas para ejecutarse, los datos se mueven de las estaciones de reserva a sus correspondientes unidades funcionales, y las operaciones de multiplicación y sustracción comienzan allí.
  Se emite la instrucción de división. Como estamos en el mismo ciclo que cuando la primera estación de reserva inició la multiplicación, no está libre todavía, así que se ocupará la segunda estación. Si sólo hubiera una estación de reserva disponible, la emisión de instrucciones se habría detenido, pero como hay más no tenemos conflicto. F1 está disponible, por lo que en V_k se guarda su valor. La estación esperará a que esté disponible la operación que escriba a F3, se guarda en Q_j la etiqueta MUL1.
  Finalmente, se cambia el estado de F2 a no ocupado.

Estaciones de reserva						
Nombre	Ocupada	OP	V_j	V_k	Q_j	Q_k
Add1	Si	SUB	(LD1)			LD2
Add2	No					
Mul1	Si	MUL		(F4)	LD2	
Mul2	Si	DIV		(LD1)	MUL1	

- Ciclo 6: Se inicia la ejecución de la sustracción y la multiplicación ya que tienen los valores de ambos operandos disponibles. Nuestra siguiente instrucción es una suma y funciona como las instrucciones aritméticas anteriores. Esta vez se utiliza la segunda estación de reserva para operaciones de suma.
  Una vez más, sólo uno de los dos operandos está disponible. En Q_j se guarda SUB1 y en V_k se guarda el valor de F2.

Estaciones de reserva						
Nombre	Ocupada	OP	V_j	V_k	Q_j	Q_k
Add1	Sí	SUB	(LD1)	(LD2)		
Add2	Sí	ADD		(LD2)	SUB1	
Mul1	Si	MUL	(LD2)	(F4)		
Mul2	Si	DIV		(LD1)	MUL1	

- Ciclo 7: No hay más instrucciones en la cola.
  Se inició una operación de resta en el ciclo cinco, hace apenas dos ciclos. Esta operación sólo tarda dos ciclos de reloj en completarse, por lo tanto, el resultado sale del sumador y se transmite al bus de datos común. Desde allí será escrito en dos lugares, concretamente a los dos suscriptores, el registro F5 y la estación de reserva del ADD (I6).

Estaciones de reserva						
Nombre	Ocupada	OP	V_j	V_k	Q_j	Q_k
Add1	No					
Add2	Sí	ADD	(SUB1)	(LD2)		
Mul1	Si	MUL	(LD2)	(F4)		
Mul2	Si	DIV		(LD1)	MUL1	

  Notar que la operación ADD será capaz de iniciar la ejecución en el octavo ciclo, lo que es significativo porque la instrucción ADD (I6) llegó después que la instrucción DIV (I5) en el programa, por lo que se estará llevando a cabo una ejecución fuera orden.
  Las instrucciones que estaban ejecutándose seguirán haciéndolo por la cantidad de ciclos que estas tarden en completarse.

Estado de las instrucciones			
Instrucción	Emisión	Inicio	Fin
LD F1, 76	1	1	3
LD F2, 90	2	2	4
MUL F3, F2, F4	3	5	15
SUB F5, F2, F4	4	5	7
DIV F0, F3, F1	5	16	56
ADD F1, F5, F2	6	8	10

En la tabla anterior, se indica el número de ciclo en que se emitió, inició y finalizó la ejecución para cada instrucción.

6 Aplicaciones actuales del algoritmo de Tomasulo

La planificación dinámica de instrucciones es algo altamente necesario hoy en día para obtener una mayor velocidad en la ejecución. El algoritmo de Tomasulo demostró ser altamente eficiente, tanto que en la actualidad gran parte de los procesadores hacen uso de variaciones de este. Fuera de IBM, no se utilizó durante varios años tras su implementación en la arquitectura System/360 Modelo 91. Sin embargo, su uso aumentó enormemente durante la década de 1990 por tres razones:

- Una vez que las cachés se hicieron comunes, la capacidad del algoritmo Tomasulo para mantener la concurrencia durante tiempos de carga impredecibles causados por fallos de caché se hizo valiosa en los procesadores.
- La programación dinámica del algoritmo mejoraba el rendimiento a medida que los procesadores emitían cada vez más instrucciones.
- El aumento de la producción de software de gran consumo hizo que los programadores no quisieran diseñar software para una estructura de pipeline específica.

Muchos procesadores modernos implementan esquemas de programación dinámica derivados del algoritmo original de Tomasulo, como los populares chips Intel x86-64 [6], [7].

En la microarquitectura del núcleo de Intel, cada instrucción es separada en diversas micro-operaciones y la unidad del núcleo de ejecución renombra los registros a un conjunto mayor de micro-registros. Durante la etapa de emisión, los registros se asignan a este conjunto ampliado de micro-registros. Los riesgos derivados de las dependencias de nombres se evitan asignando un nuevo registro no utilizado como registro de destino y cambiando el nombre de este registro. Además, existe un buffer de ordenación de memoria (MOB) independiente que gestiona los peligros debidos a las dependencias de nombres que se producen en las instrucciones de almacenamiento y carga de memoria. Este buffer mejora respecto a Tomasulo los accesos a memoria contigua y la gestión de cargas en operaciones de almacenamiento.

El algoritmo de Tomasulo no sólo es útil para procesadores éstandar, sino que sirve para sistemas heterogéneos. Existe una variación del algoritmo, llamada MP-Tomasulo, que implementa la ejecución de tareas paralelizada para programas secuenciales. Es una aplicación de Tomasulo en MPSoCs (MultiProcessor System on Chip) [8]. En este enfoque, se consideran procesadores y núcleos como unidades funcionales, y las tareas que deben realizar son el análogo a las instrucciones. Además, se envían tareas a los diferentes núcleos según el tiempo de ejecución estimado (si una unidad tarda menos que las otras en ejecutar una tarea, se le asigna a esta). Los resultados experimentales demuestran que MP-Tomasulo puede ejecutar las tareas fuera de orden para alcanzar entre el 93% y el 97% de la velocidad máxima ideal teórica.

7 Conclusión

A lo largo de este informe, hemos expuesto las ventajas de la planificación para la ejecución de instrucciones por sobre las planificación estática, y los riesgos de dependencia de datos que surgen al ejecutar fuera de orden. Vimos como está diseñado el algoritmo del marcador, el cual solucionaba los riesgos RAW dinámicamente, pero la emisión de instrucciones en este se detenía ante la posibilidad de riesgos WAW o WAR. El algoritmo de Tomasulo implementó el renombramiento de registros mediante el uso de estaciones de reserva para solucionar este problema. Además, innovó con el uso de un bus común de datos mediante el cual se difundían los resultados de una operación a todas las unidades que los requerían, evitando así el uso de múltiples buses que conectaban a los registros, ahorrando ciclos en la ejecución de una instrucción. Hoy en día se utilizan variaciones del algoritmo de Tomasulo tanto en la mayoría de procesadores modernos como en sistemas heterogéneos.

References

[1] Tomasulo, Robert Marco (Jan 1967). "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". IBM Journal of Research and Development. IBM. 11 (1): 25–33

[2] Hennessy, John L.; Patterson, David A. (2012). Computer Architecture: A Quantitative Approach. Waltham, MA: Elsevier.

[3] Thornton, James E. (1965). "Parallel operation in the control data 6600". Proceedings of the October 27–29, 1964, fall joint computer conference, part II: very high speed computer systems. AFIPS '64. San Francisco, California: ACM. pp. 33–40.

[4] Tong, Allan, "Dynamich Schedulling", 2001. [Online]. Disponible: https://web.archive.org/web/20120919151240/http://www.cs.umd.edu/class/fall2001/cmsc411/projects/dynamic/intro.html

[5] "IBM System/360, Model 91 (console)", 2001. [Online] Disponible: https://ed-thelen.org/comp-hist/vs-ibm-360-91.html

[6] Intel 64 and IA-32 Architectures Software Developer's Manual (Report). Intel. [Online] Disponible: https://www.intel.com/content/www/us/en/content-details/782158/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html?wapkw=intel

[7] Yoga, Adarsh. "Differences between Tomasulo's algorithm and dynamic scheduling in Intel Core microarchitecture", 2010. The boozier. [Online] Disponible: https://adusan.blogspot.com/2010/11/differences-between-tomasulos-algorithm.html

[8] Chao Wang, Xi Li, Junneng Zhang, Xueahi Zhou, Xiaoning Nie (May 2013). "MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs". ACM Transactions on Architecture and Code Optimization, Vol. 10, No. 2, Article 9.