66.20 - Organización de Computadores
Trabajo Práctico Final
Predición de Saltos en el camino de datos MIPS 32

Andrew Parlane

18 de febrero de 2019

Índice

Índice de figuras

1. Objetivo

El objetivo de este trabajo es diseñar, implementar y caracterizar varios predictores de saltos, usando el programa DrMIPS. DrMIPS es un programa de fuente abierto escrito en JAVA, que permite simular el camino de datos de un CPU MIPS 32. El camino de datos y el conjunto de instrucciones son implementado mediante archivos JSON.

2. Introducción

En un CPU monociclo cada instrucción es ejecutado completamente antes de la siguiente comienza. En el caso de un salto condicional esto significa que, es conocido si el salto estará tomado o no, antes de cuando es necesario leer la siguiente instrucción. Mientras que en un CPU con un pipeline, dos o más instrucciones pueden estar ejecutando al mismo tiempo. Así puede ser necesario comenzar leer la siguiente instrucción antes de conocer si un salto previo estará tomado o no. Esto es un riesgo de control.

En un CPU MIPS 32 básico el pipeline tiene cinco etapas:

- IF - Lea la instrucción desde la memoria de instrucciones.
- ID - Descodifica la instrucción y lea los registros.
- EX - Ejecuta operaciones ALU.
- MEM - Escribe / Lea la memoria de datos.
- WB - Escribe el resultado al registro.

Por un salto condicional el CPU solo sabe si debería tomarlo o no después de la operación ALU. Normalmente resuelve esto en la etapa MEM, así hay tres ciclos donde el CPU no sabe cual instrucciones leer.

Una solución de esto es introducir un “stall” en el pipeline por cada ciclo hasta el resultado del salto es conocido. Por ejemplo:

```
    AND R1, R2, R3
    ADD R4, R5, R6
    BEQ R1, R2, FOO
    ADD R7, R8, R9
    ADD R10, R11, R12
    ADD R13, R14, R15
FOO:
    OR R1, R2, R3
    OR R4, R5, R6
```

Ciclo		IF	ID	EX	MEM	WB
1		AND R1, R2, R3	-	-	-	-
2		ADD R4, R5, R6	AND R1, R2, R3	-	-	-
3		BEQ R1, R2, FOO	ADD R4, R5, R6	AND R1, R2, R3	-	-
4		STALL	BEQ R1, R2, FOO	ADD R4, R5, R6	AND R1, R2, R3	-
5		STALL	STALL	BEQ R1, R2, FOO	ADD R4, R5, R6	AND R1, R2, R3
6		STALL	STALL	STALL	BEQ R1, R2, FOO	ADD R4, R5, R6
7		OR R1, R2, R3	STALL	STALL	STALL	BEQ R1, R2, FOO
8		OR R4, R5, R6	OR R1, R2, R3	STALL	STALL	STALL
9			OR R4, R5, R6	OR R1, R2, R3	STALL	STALL
10				OR R4, R5, R6	OR R1, R2, R3	STALL
11					OR R4, R5, R6	OR R1, R2, R3
12						OR R4, R5, R6

Figura 1: Evitar riesgos de control mediante stalls.

El problema con este método es que no es muy eficiente. Cada salto condicional toma cuatro ciclos para ejecutar. Un método mejor sería intentar predecir si el salto será tomado o no, y si se equivoca, vaciar las primeras tres etapas del pipeline. En este caso el truco es diseñar un predictor muy preciso sin usando demasiados recursos.

También hay otros métodos para mejorar el desempeño como “Branch Delay Slots” o resolviendo el salto en una etapa anterior, pero esos son afuera del tema de este proyecto.

3. Predictores de Saltos

3.1. Predictores Estáticos

Un predictor de saltos estático elige la misma acción por cada salto. Hay dos posibilidades: siempre no tomado y siempre tomado.

En el caso de siempre no tomado el procesador sigue leyendo las siguientes instrucciones (PC + 4). En la etapa de MEM si el salto no está tomado, no tiene que hacer nada. Pero si el salto está tomado, tiene que vaciar las primeras tres etapas del pipeline (IF, ID, EX) y ajusta el PC a la dirección nueva. Figura 2 muestra que pasa cuando un salto es tomado.

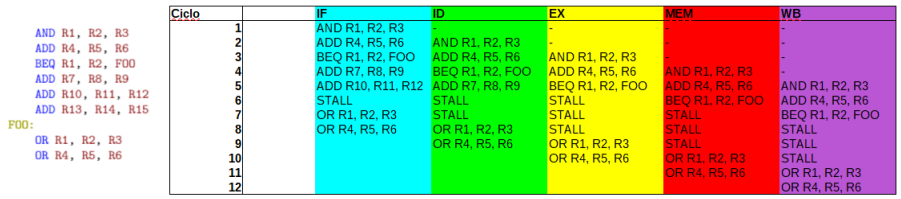

Figura 2: Vaciando el pipeline después de una predicción equivocado de salto no tomado.

La otra opción es siempre suponer que el salto será tomado. El problema con este método es que no sabe que es un salto hasta la segunda etapa (ID), así siempre tiene que vaciar la primera etapa (IF). Si el salto actualmente es tomado, no hay nada más hacer, pero en el caso que no es tomado, tiene que vaciar las primeras tres etapas de nuevo. Figura 3 muestra los dos casos.

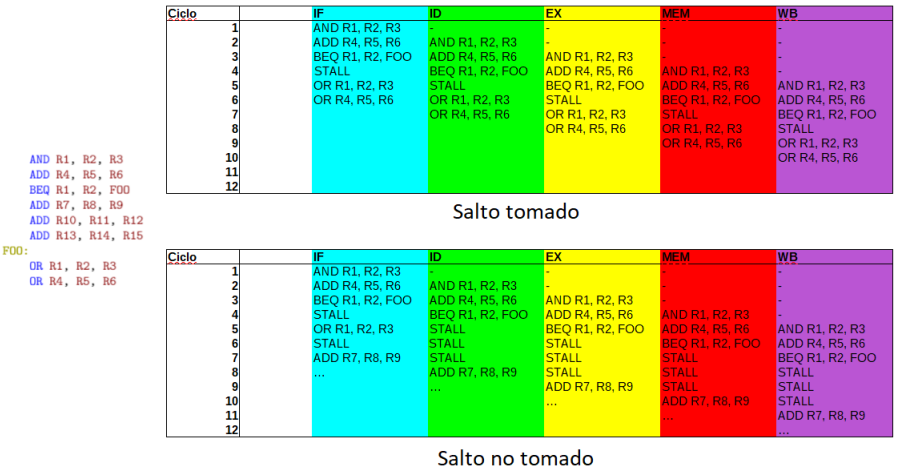

Figura 3: El flujo de ejecución con un predictor de saltos siempre tomados.

3.2. Predictores Dinámicos

Predictores de saltos dinámicos intentan predecir si el salto será tomado o no basado en la historia del salto (historia local) o de todos los saltos (historia global).

3.2.1. Predictor de Historia Local

Un predictor de saltos de historia local usa 2^N contadores que saturan, de M bits cada uno. En la etapa ID un contador es direccionado con N bits del PC. Si el valor corriente del contador tiene el bit más significativo encendido predice que el salto estará tomado. En la etapa MEM el contador es actualizado, si el salto actualmente está tomado, se incrementa por uno, y si no, se decrementa por uno. También en la etapa MEM si la predicción estuvo equivocado, vacía las primeras tres etapas del pipeline y ajusta el PC al valor correcto. Figura 4 muestra un predictor de historia local.

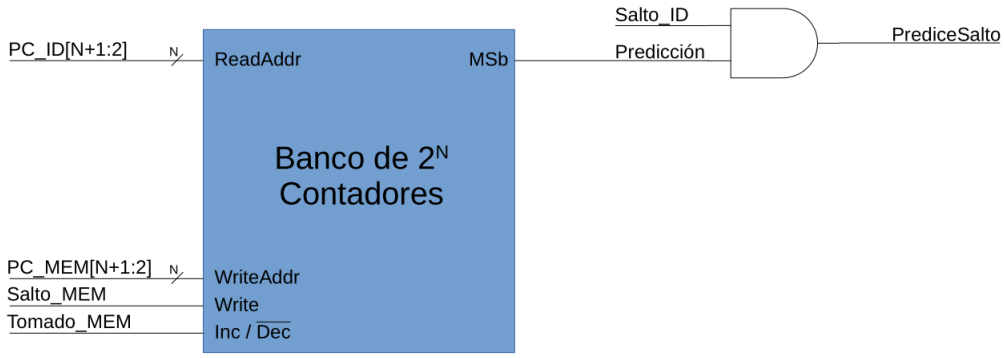

Figura 4: Un predictor de historia local.

La idea de esto es que si un salto es tomado más frecuentemente que no, el valor del contador normalmente estará cerca el máximo y predice tomado, y viceversa.

Porque hay 2^N contadores, usamos N bits del PC para direccionar los contadores. En MIPS32 cada instrucción es cuatro bytes, así los dos bits más bajos del PC siempre son ceros. Normalmente usan los siguiente N bits menos significativos. Interferencia ocurre cuando dos saltos direcciona el mismo contador. Eligiendo N es un intercambio entre menos interferencia y la cantidad de memoria necesario para guardar los contadores.

Cada contador tiene M bits, si M=1 el contador solo puede ser 0 o 1, así cada vez la predicción es que este salto hizo la última vez. El problema con esto es que no hay histéresis. El siguiente código demuestra esto.

```
while (1)
{
    for (int i = 0; i < 3; i++)
    {
        ...
    }
}
```

	M=1			M=2		
i	contador	predicción	Hit / Miss	contador	predicción	Hit / Miss
Primer lazo						
1	0	N	Miss	0	N	Miss
2	1	T	Hit	1	N	Miss
3	1	T	Miss	2	T	Miss
Segundo lazo						
1	0	N	Miss	1	N	Miss
2	1	T	Hit	2	T	Hit
3	1	T	Miss	3	T	Miss
Tercero lazo						
1	0	N	Miss	2	T	Hit
2	1	T	Hit	3	T	Hit
3	1	T	Miss	3	T	Miss
Todos los demás						
1	0	N	Miss	2	T	Hit
2	1	T	Hit	3	T	Hit
3	1	T	Miss	3	T	Miss

Si ejecuta este lazo suficiente vecez que los primeras dos iteraciones están despreciable, la tasa del hit para M = 1 es 33 %, y para M = 2 es 67 %. Sin embargo, si M es demasiado grande dura mucho tiempo para llegar al estado de equilibrio.

3.2.2. Predictor de Historia Global

Si dos saltos están correlacionados el resultado de uno puede ser adivinado desde el resultado del otro. Por ejemplo, en el código siguiente, si los dos primeros saltos no están tomados (a == 0 y b == 0) entonces el tercer salto no estará tomado tan poco, siempre que a ni b están modificados adentro de los bloques.

```
if (a == 0)
{
    ...
}
if (b == 0)
{
    ...
}
if (a == b)
{
    ...
}
```

Para implementar un predictor de historia global, se usa un registro de desplazamiento de N bits para direccionar 2^N contadores que saturan, de M bits cada uno. El resultado actual de cada salto entra el registro, guardando una historia global de los saltos. Figura 5 muestra un predictor de historia global.

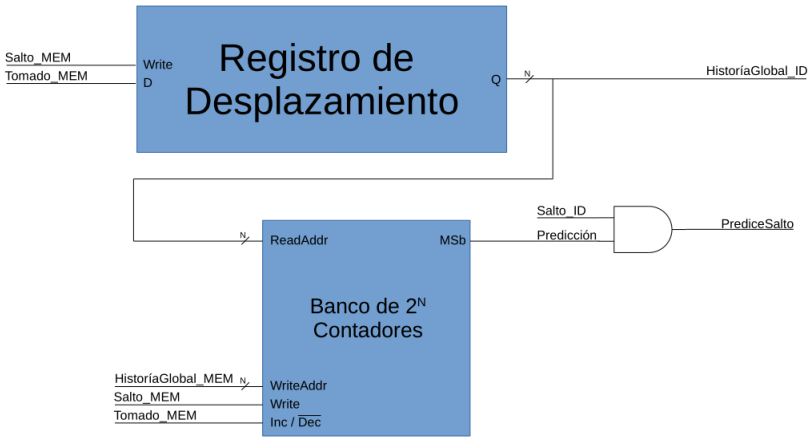

Figura 5: Un predictor de historia global.

3.2.3. Predictor de Dos Niveles

Un predictor de dos niveles usa ambas de las técnicas anteriores. Hay un registro de desplazamiento de P bits, y 2^P bancos de 2^N contadores con M bits cada uno. El banco es direccionado con el registro de desplazamiento y el contador es direccionado por N bits del PC. Figura 6 muestra un predictor de dos niveles.

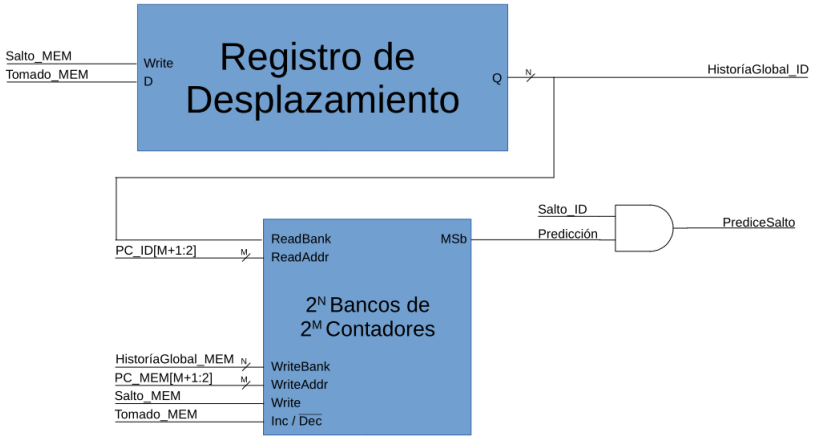

Figura 6: Un predictor de dos niveles.

3.3. CPI

CPI (Ciclos por instrucción) es una medida de desempeño. Comparando los CPIs por las instruucciones de saltos por los varios predictores da una idea de cual predictor será mejor. Sin embargo no es definitivo, si un predictor tiene un CPI más bajo que otro, pero la implementación significa que la frecuencia del reloj tiene que ser más lento, entonces puede ser menos eficiente.

	CPI			
	Predice Tomado		Predice No Tomado	
Predictor	Tomado	No Tomado	Tomado	No Tomado
Siempre Stall	4	4	4	4
Supone No Tomado	-	-	4	1
Supone Tomado	2	4	-	-
Dinámicos	2	4	4	1

El CPI Promedio ($\overline{CPI}$) depende del Hit Rate y del ratio de tomado a no tomado en el programa. Pero es claro que equivocando cuesta mucho, así el diseño del predictor es muy importante.

4. Implementación en DrMIPS

4.1. Componentes JAVA

Tuve que implementar unos componentes en JAVA para uso en mis caminos de datos:

- Banco de Contadores que Saturan, con el ancho de los contadores y el número de contadores configurable.
- Registro de desplazamiento, con el ancho configurable.
- Demultiplexador de tamaño configurable.

No estoy muy familiar con JAVA pero pude implementar estos componentes sin dificultades, usando el componente del Register File como plantilla para los primeros dos, y del multiplexador para el último.

DrMIPS tiene bancos de pruebas para cada componente los que están ejecutados como parte del proceso de compilación. He escrito un banco de prueba para cada uno de mis componentes para probar que funcionan como deseado.

4.2. Camino de Datos

Empecé con el camino de datas "Pipeline Extended" lo que viene con DrMIPS. Este camino de datos supone que saltos no estarán tomados. Usando eso como base implementé los siguientes diseños:

- Siempre Stall.
- Supone Tomado.
- Historia Local.
- Historia Global.
- Dos Niveles.

En los casos de los predictores dinámicos inicializo todos los contadores a $2^{M-1} - 1$, así están en el medio del rango en el lado de predecir no tomado.

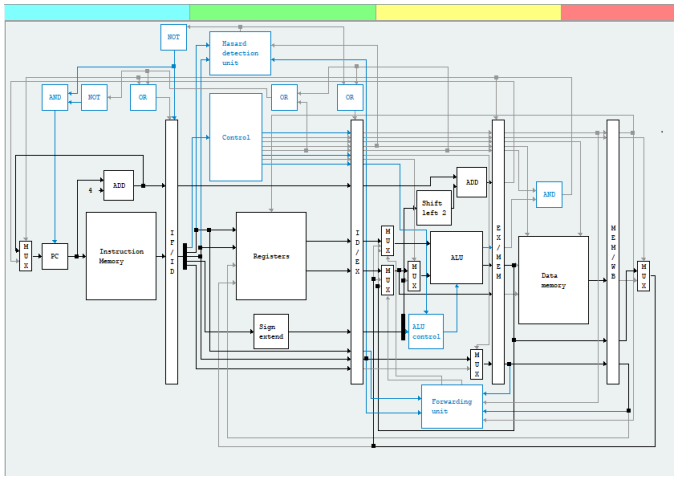

Figura 7: El camino de datos que siempre stalls.

4.2.1. Siempre Stall

En este camino de datos, cuando un Salto es detectado en la etapa de ID, EX o MEM, las etapas anteriores están vaciado. En la etapa MEM actualizamos el PC. Figura 7 muestra el diseño final.

4.2.2. Supone Tomado

En este camino de datos, tuve que mover la calculación del target del salto a la etapa ID. Añadí un mux para elegir como actualizar el PC (PC+2 / target en ID / target correcto en MEM). Vacío IF cuando hay un salto en ID Y finalmente en vez de vaciando las etapas anteriores en MEM cuando el salto actualmente es tomado, hago cuando el salto no es tomado. Figura 8 muestra el diseño final.

4.2.3. Historia Local

En este camino de datos he usado mi banco de contadores para predecir el salto en la etapa ID, y actualizarle en la etapa MEM. Como siemper hay lógica para vaciar las etapas IF, ID y EX cuando es necesario, el único diferencia es que solo vacia IF desde ID si predice un salto. Figura 9 muestra el diseño final.

4.2.4. Historia Global

En este camino de datos he usado mi registro de desplazamiento para direccionar el banco de contadores. En la etapa MEM actualizo el registro de desplazamiento tanto como el contador. Figura 10 muestra el diseño final.

4.2.5. Dos Niveles

Finalmente hay el predictor de dos niveles. Funciona parecido al predictor de historia local, pero en vez de tener un banco de contadores hay cuatro direccionado con la historia global. El única otra cosa interesante es el uso de mi demultiplexor para elegir cual banco actualizar en la etapa MEM. Figura 11 muestra el diseño final.

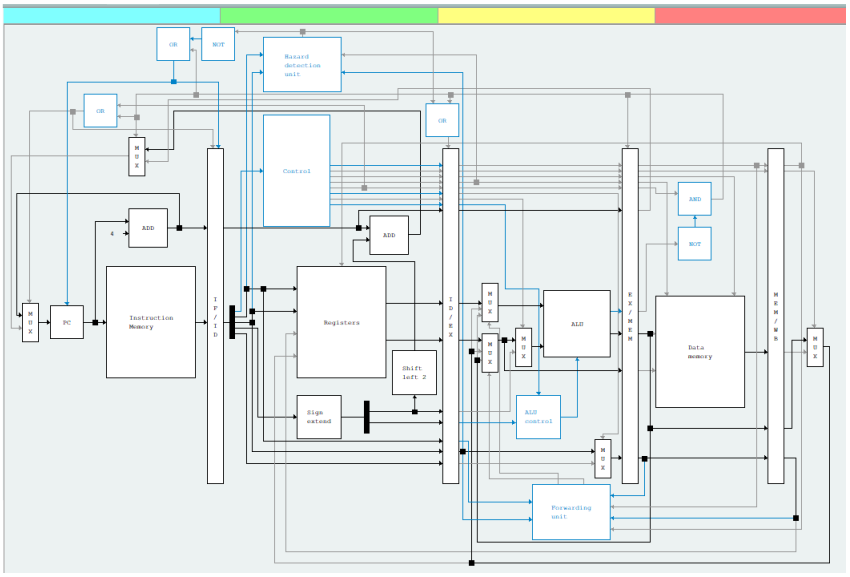

Figura 8: El camino de datos con un predictor estático de suponer tomado.

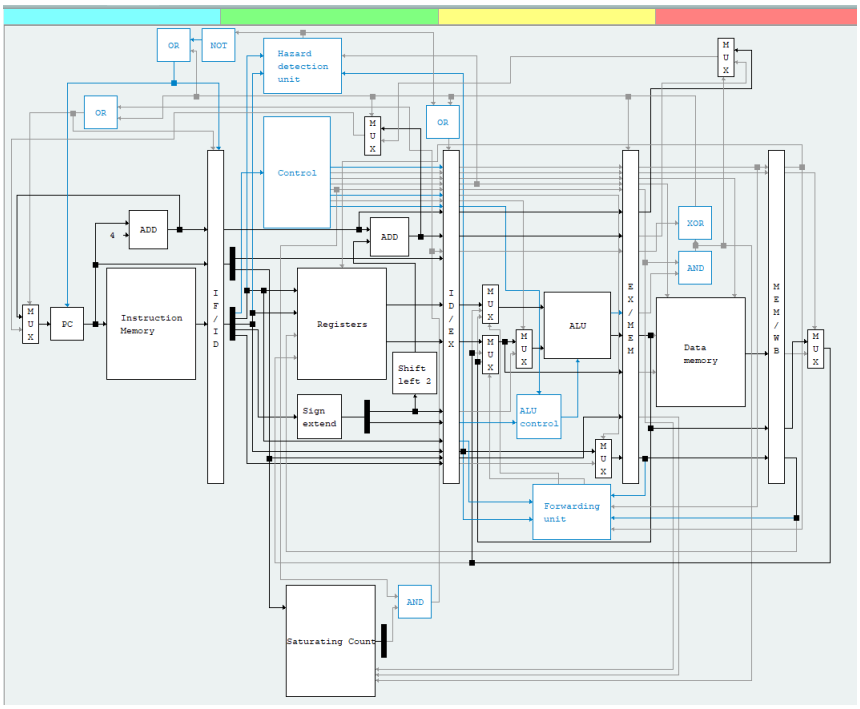

Figura 9: El camino de datos con un predictor de historia local.

En el camino de datos de historia local y historia global no es tan difícil cambiar el número de contadores, sus anchos y el ancho del registro de desplazamiento. Pero en este camino de datos es mucho más difícil cambiar el número de banco de contadores. Si querrías aumentar el ancho

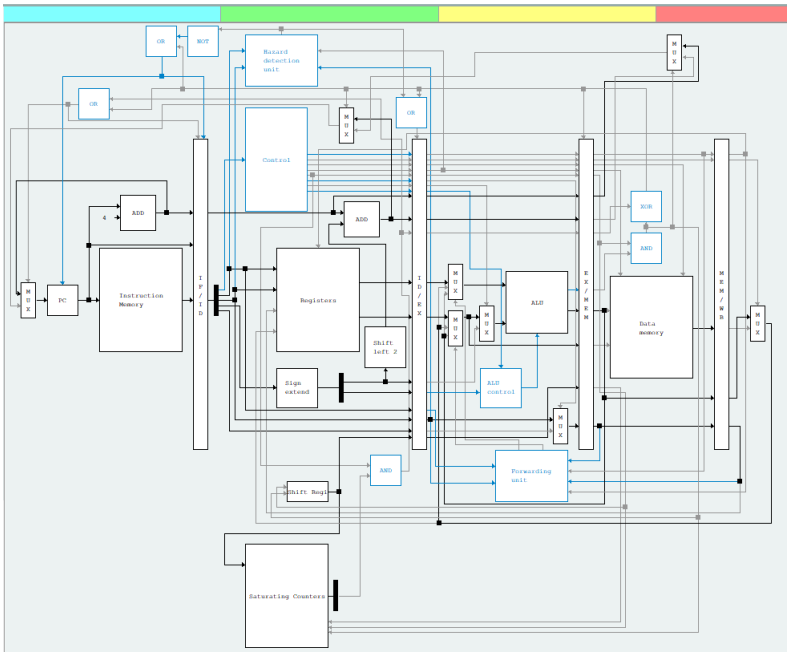

Figura 10: El camino de datos con un predictor de historia global.

del registro de desplazamiento un bit, tienes que manualmente añadir y conectar cuatro más bancos de contadores mediante la manipulación de archivos JSON, que es muy laborioso. Una optimización futuro será añadir un nuveo componente JAVA que tiene un conjunto de bancos de contadores, como lo que es mostrado en Figura 6.

4.3. Depurando

Durante la implementación del algoritmo quicksort para analizar el desempeño de los predictores, encontré un bug donde mi algoritmo no estuvo ordenando los valores correctamente. Mientras estuve depurandolo, encontré que el comportamiento estuvo diferente entre los caminos de datos "Pipeline Extended" y "Unicyle Extended". El bug estuvo en el diseño del camino de datos "Pipeline Extended" que viene por defecto con DrMIPS y no en mi algoritmo.

Para encontrar este problema tuve que ir paso por paso por miles de ciclos que estuvo muuy lento. Para acelerar esto implementé breakpoints en DrMIPS, así que pude ejecutar mi algoritmo hasta una instrucción.

El bug estuvo que cuando la Hazzord Detection Unit encuentra un hazzard desactiva escritos al PC. Si en el mismo cíclo la etapa MEM está actualizando el PC debido a un salto predicho equivocadamente, el escrito está ignorado y el flujo sigue por la rama equivocado. Arreglé este bug en dos de los pipelines que vienen con DrMIPS y en todos mis pipelines. Generé un pull request al repositorio upsteam que estuvo acceptado. También generé un pull request para la implementación de los breakpoints.

4.4. Scripts

Solo realicé una implementación de cada camino de datos, pero quise una forma analizarles con un rango del número de contadores (2^N) y sus anchos M. Para hacer esto, escribí unos scripts

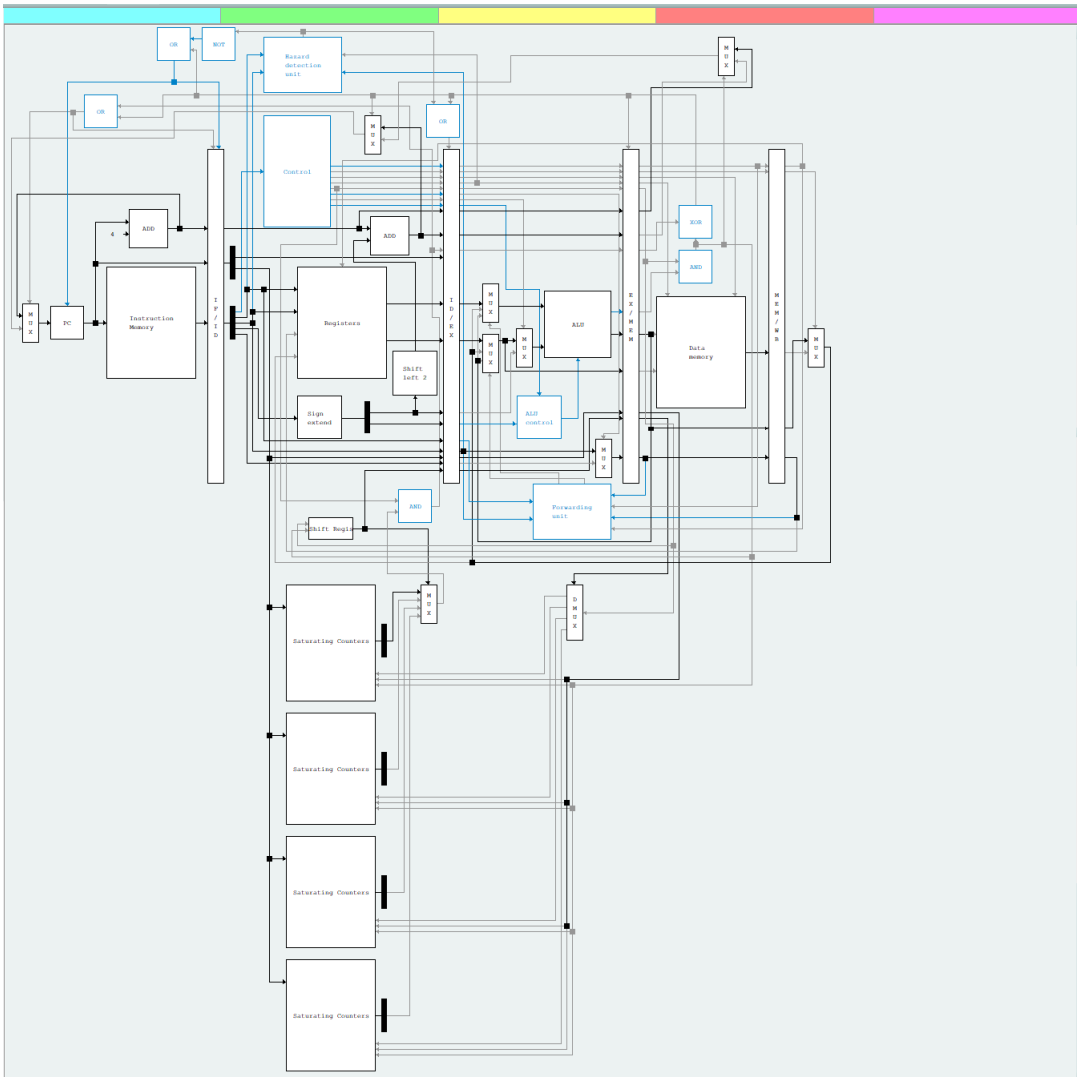

Figura 11: El camino de datos con un predictor de dos niveels.

en python para modificar los archivos JSON.

También escribí otro script en python para dibujar gráficas de los "Hit Rates" de los predictores y los CPIs promedios por todos los parámetros.

5. Algoritmos de Prueba

Cuando llegé a la hora de escribir código para analizar el desempeño de cada camino de datos, encontré unas limitaciones en DrMIPS:

- Por defecto solo ejecuta mil ciclos. Incrementé este limite a ochenta mil, que es mejor pero no es ideal.

- Si ejecuta más de ochenta mil ciclos DrMIPS falla por falta de memoria.
- Las instrucciones implementado por defectos están limitados. La falta de JR es espacialmente molesto porque no puedes volver desde una función. Podría haber implementado la instrucción JR pero habría sido necesario implementarle por cada camino de datos, y aun haciéndolo una vez es bastante laborioso.
- La memoria de datos es de solo cuatro cientos bytes.

Por estos razones estuve limitado en los algoritmos que pude implementar. Decidí no seguir un ABI en mi código, uso cualquier registros y no tengo un stack (excepto de en la prueba de quicksort donde he implementado uno personalizado).

5.1. Lazo

```
li $t0, 0
li $t1, 99
start:
        addi $t0, $t0, 1
        ble $t0, $t1, start
```

Este algoritmo muestra como los predictores diferentes funcionan con un salto que casi siempre es tomado.

5.2. Mayor Denominador Común (GCD)

En esta prueba calculo el GCD para todos los pares de números en los rango de 50-59 y 90-99.

```
    li $s2, 99                      ; maxA = 99
    li $s3, 59                      ; maxB = 59;
    li $s4, 0                       ; idx = 0;

    li $s0, 90                      ; a = 90;
a_loop_start:                       ; do {
    li $s1, 50                      ;   b = 50;
b_loop_start:                       ;   do {
    move $t0, $s0
    move $t1, $s1
    b gcd_loop_start                ;     res = gcd(a,b);
gcd_return:
    sw $t2, 0($s4)                  ;     results[idx] = res;
    addi $s4, $s4, 4                ;     idx++;
    addi $s1, $s1, 1                ;     b++
    ble $s1, $s3, b_loop_start      ;   } while (b <= maxB)
    addi $s0, $s0, 1                ;   a++
    ble $s0, $s2, a_loop_start      ; } while (a <= maxA)
    b endProg                       ; return;

gcd_loop_start:                     ; while (1) {
    beq $t0, $zero, ret_b           ;   if (a == 0) return b;
    beq $t1, $zero, ret_a           ;   if (b == 0) return a;
```

```
    beq $t0, $t1, ret_a             ;   if (a == b) return a;
    bge $t0, $t1, a_greater_than_b  ;   if (a < b) {
    sub $t1, $t1, $t0               ;     b -= a;
    b gcd_loop_start                ;   }
a_greater_than_b:                   ;   else {
    sub $t0, $t0, $t1               ;     a -= b;
    b gcd_loop_start                ;   }
                                    ; }

ret_a:
    move $t2, $t0
    b gcd_return
ret_b:
    move $t2, $t1
    b gcd_return

endProg:
```

5.3. Bubble Sort

```
; Cargar data generado por un generador de números random.
; Uso 50 valores entre 0 y 100.
li $t0, 94
li $t1, 0
sw $t0, 0($t1)
li $t0, 29
addi $t1, $t1, 4
sw $t0, 0($t1)
li $t0, 87
addi $t1, $t1, 4
sw $t0, 0($t1)
...

; t1 dirección de el último valor
; t2 cambiado? flag
; t3 dirección de valor corriente
; t4 valor1
; t5 valor2
startOuter:                         ; do
                                    ; {
    li $t2, 0                       ;   bool cambiado = 0;
    li $t3, 0                       ;   uint8_t *ptr = 0; // primer valor es a dirección 0

startInner:                         ;   do
                                    ;   {
        lw $t4, 0($t3)              ;     valor1 = ptr[0];
        lw $t5, 4($t3)              ;     valor2 = ptr[1];
        ble $t4, $t5, noSwap        ;     if (valor1 > valor2)
                                    ;     {
```

```
        sw $t4, 4($t3)              ;       ptr[1] = valor1;
        sw $t5, 0($t3)              ;       ptr[0] = valor2;
        li $t2, 1                   ;       cambiado = 1;
                                    ;     }
noSwap:
        addi $t3, $t3, 4            ;     ptr++;
        beq $t3, $t1, endLoop
        b startInner                ;   } while (ptr != último)

endLoop:
        li $t6, 1
        beq $t2, $t6, startOuter    ; } while (cambiado)
```

5.4. Quick Sort

El algoritmo básico de quicksort es:

```
void quickSort(int *first, int *last)
{
    if (low < high)
    {
        int *j = ...;

        quickSort(first, j - 1);
        quickSort(j + 1, last);
    }
}
```

Es una función recursivo, así es necesario tener un stack. El stack frame necesita guardar tres valores:

- La dirección a volver ($RA).
- El pivot (j).
- La dirección del último valor (last).

Implementé un stack que crece hacia abajo desde el fin de memoria. Porque solo hay 400 bytes de memoria de datos y usa los primeros 200 bytes por la lista de valores a ordenar, solo hay 200 bytes disponibles por el stack. Mi implementación en C me muestra que la profundidad máxima es 13 llamadas por la misma lista de datos. 13 stack frames de 12 bytes cada uno significa que necesita 156 bytes por el stack, y tiene 200 bytes, así funciona.

Por la falta de la instruucción JR no puedo solo guardar la dirección a volver como normal. Tuve que usar un valor para indicar cual etiqueta usar. Aquí es el código relevante.

```
qsort:
    ...

    ; guarda los valores en el stack
    subi $sp, $sp, 12
    sw $ra, 8($sp)
```

```
    sw $t2, 4($sp)
    sw $a1, 0($sp)

    ; queremos volver a retRecursive1
    li $ra, 1
    ...
    b qsort                 ;   qsort(first, j-1);
retRecursive1:

    ; queremos volver a retRecursive2
    li $ra, 2
    ...
    b qsort                 ;   qsort(j+1,last);
retRecursive2:

    ; recuperar RA del stack
    lw $ra, 8($sp)

    ; volver al último stack frame.
    addi $sp, $sp, 12

    ; return
    beq $ra, $zero, retRecursive1   ; if (RA == 0) return to retRecursive1
    li $s0, 1
    beq $ra, $s0, retRecursive2     ; else if (RA == 1) return to retRecursive2

endProg:                            ; else exit()
```

6. Resultados

6.1. Predictores Estáticos

Ejecuté cada prueba para los tres predictores estáticos. Los resultados están:

		Siempre Stall		Predecir NT		Predecir T	
Prueba	Saltos	CPI	HR %	CPI	HT %	CPI	HR %
Loop	100	2.0	0.0	2.0	1.0	1.3	99.0
GCD	6256	2.6	0.0	1.7	64.2	2.6	35.8
Bubble Sort	6909	2.1	0.0	1.8	41.7	1.8	58.3
Quick Sort	1411	1.9	0.0	1.6	51.6	1.7	48.4

Es claro que la efectividad del predictor tiene un impacto grande en el desempeño del programa. En la prueba de bubble sort el predictor T tiene un mejor hit rate, pero el CPI promedio es igual al predictor NT, esto es porque una predicción correcta de T cuesta dos ciclos mientras una predicción correcta de NT cuesta solo un ciclo. En el caso del lazo el predictor T es mucho mejor que el predictor NT, pero en general el predictor NT es mejor.

6.2. Predictores Dinámicos

Ejecuté cada prueba con cada predictor por N entre 1 y 6, y M entre 1 y 6. Todo el dato crudo puede estar encontrado en mi github en el archivo data/analysis.ods, aquí solo pongo los

pedazos interesantes.

El diseñador de un CPU quiere que su predictor de saltos tiene el mejor desempeño posible por un mínimo de recursos. Los recursos principales son área y potencia, aquí solo trato de área porque potencia es difícil medir y depende mucho del tecnología de fabricación, mientras área es directamente vinculado con el número de bits usados que es una métrica mucho más fácil medir.

Calculé cuantos bits cada predictor usa por cada par de N y M:

- Historia Local - Hay 2^N contadores, cada uno tiene M bits. Así hay $M * 2^N$ bits total.
- Historia Global - Hay 2^N contadores, cada uno tiene M bits, y hay un registro de desplazamiento de N bits. Así hay $M * 2^N + N$ bits total.
- Dos Niveles - Hay 4 bancos de 2^N contadores, cada uno tiene M bits, y hay un registro de desplazamiento de 2 bits. Así hay $4M * 2^N + 2$ bits total.

Después usé python para dibujar unas gráficas mostrando el CPI y HR contra el número de bits usado. Nota que es posible tener más de un punto de data para el mismo número de bits. Por ejemplo en el predictor de historia local, los pares de parámetros: (N=2, M=4), (N=3, M=2), (N=4, M=1) todos usan 16 bits.

Que encontré es hasta un limite hay un fuerte relación entre número de bits y el HR y el CPI, después de ese limite, el HR y el CPI no cambian mucho y a veces empeoran. Figura 12 muestra estos resultados. Por los predictores de historia local y historia global ese limite es acerca 50 bits, pero en el caso del predictor de dos niveles el limite es acerca 150 bits. Esto indique que por mis pruebas el predictor de dos niveles necesita más bits para estar efectivo.

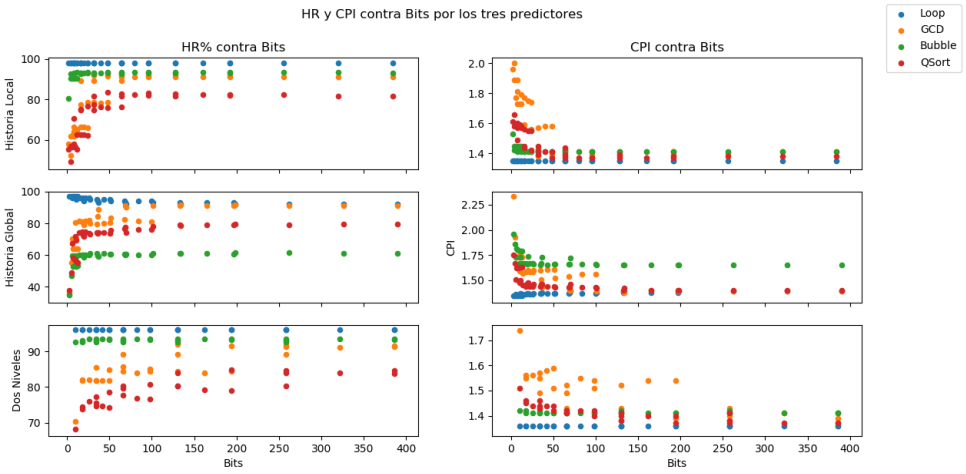

Figura 12: Gráficos de CPI y HR % contra bits por cada predictor.

Figura 13 muestra como se comparan los predictores por cada algoritmo. El predictor de historia global es el peor en todos los casos excepto de por el GCD donde es comparable a los otros. Los predictores de historia local y de dos niveles están muy parecidos pero lo de historia local está mejor por menos bits.

En Figura 13 hay un punto de historia local cerca 50 bits que tiene un muy bueno CPI por todos los algoritmos. Volviendo a mi data cruda lo encontré como (N=4, M=3). La siguiente tabla muestra como se compara con el mejor de cada predictor. Tiene el mejor CPI en cada caso excepto del GCD y en ese caso es muy cerca.

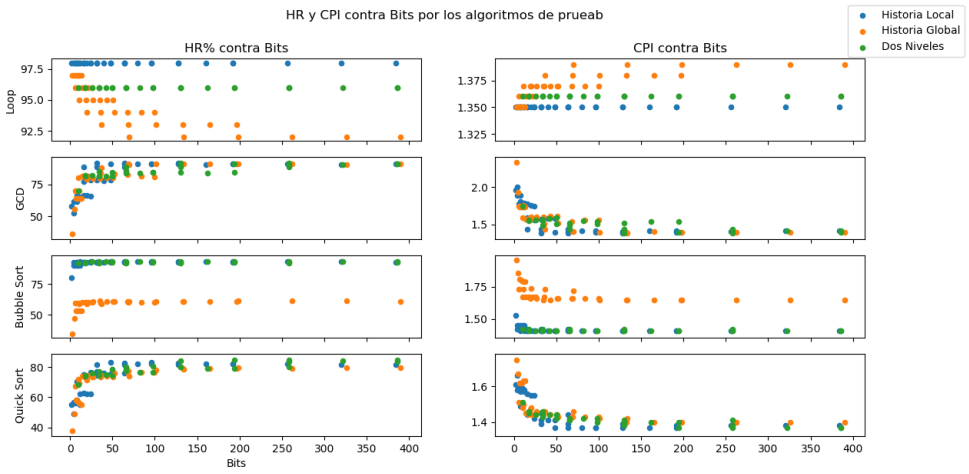

Figura 13: Gráficos de CPI y HR % contra bits por cada prueba.

	Loop		GCD		Bubble Sort		Quick Sort	
Predictor	HR %	CPI	HR %	CPI	HR %	CPI	HR %	CPI
Historia Local Mejor	98.00	1.35	91.90	1.38	93.47	1.41	83.35	1.37
Historia Global Mejor	97.00	1.35	91.62	1.38	61.63	1.65	79.52	1.40
Dos Niveles Mejor	96.00	1.36	91.86	1.38	93.56	1.41	84.76	1.37
Historia Local (N=4, M=3)	98.00	1.35	91.54	1.39	93.47	1.41	83.35	1.37

7. Conclusión

Durante este proyecto aprendí:

- Cómo funciona DrMIPS.
- Cómo leer y escribir archivos de JSON mediante JAVA, python y de mano.
- Cómo funcionan varios predictores de saltos.
- La importancia de benchmarks.
- Cómo usar python para analizar data y dibujar gráficas.

Si supongamos que mis algoritmos de pruebas están representativos del código que este CPU va a ejecutar en el mundo real, entonces el predictor de saltos más óptimo es lo de la historia local con $2^4 = 16$ contadores de 3 bits cada uno. Creo que esta suposición es valido debidos a las restricciones en las instrucciones disponibles y en el tamaño de la memoria de datos.

Los benchmarks son el parte más importante del diseño de un predictor de saltos tanto cómo del CPU entero. Si no están representativos de caso de uso real, el diseño no va a estar óptimo. Habría sido mejor tener más benchmarks incluyendo unos que están más complicados y ejecutan para más ciclos. Creo que habría comenzado encontrar que el predictor de saltos de dos niveles estuvo mejor que lo de la historia local.

Sugiero que un proyecto futuro será mejorar DrMIPS para quitar estos limites en el tamaño de memoria de datos y el número de ciclos máximos, implementar más instrucciones, especialmente JR y implementar benchmarks más complicados.

8. Código

Todo el código fuente y datos crudos son disponible en mi repositorio de GitHub: Organización de computadores y DrMIPS.