Advanced Pipelining 2

د. سيرا أستور

RB Informatics; 23/12/2024

بنيان الحاسوب ٢

درسنا في محاضرة سابقة عن التوارد و عن الأعطال التي يمكن أن تواجهنا (Hazards) في هذه التقنية و رأينا بعض حلول هذه الأعطال كاستخدام دارة إحالة (Forwarding) و استخدام توقفات Stalls وغيرهما .. ولكن واجهتنا بعض الأعطال المعقدة ، لذلك سنرى في هذه المحاضرة مفاهيم و حلول لهذه المواضيع .

Parallelism via Instructions

التوارد عبارة عن تراكب تعليمات ، كما أسميناه في محاضرة سابقة (Instruction-level parallelism) ILP

إن الـ ILP يزيد الـ Performance **، وهناك وسيلتين لزيادته :**

A : **زيادة عدد المراحل التي تمر فيها التعليمة في التوارد** (depth of pipeline) :

هذه الطريقة ترتب إمكانية وجود أعطال أكثر .

B : **تصدير تعليمات بوقت واحد** (Multiple issuing) :

ويعني دخول أكثر من تعليمة لقناة التوارد لكل مرحلة ، وهذا يعني أنه يجب زيادة مكونات المعالج .

هذا يحسن الأداء كثيراً و يجعل معدل التنفيذ يتجاوز معدل النبضة ، أي ممكن أن تنتهي أكثر من تعليمة في المرحلة أو تخرج من نبضة ساعة أكثر من تعليمة منتهية و ذلك يعني (CPI < 1) .

المعالجات متعددة الإصدارات Multiple Issue Processors

في المعالجات متعددة الإصدارات لدينا ثلاث مفاهيم :

1. Issue Slots :

وهي تحدد كم يستطيع المعالج أن يدخل تعليمات مع بعضها في كل نبضة .

مثلاً : لدي 4 تعليمات ، و نريد استخدام multiple issue **، فهنا كم تعليمة نستطيع تصديرها سوية لقناة التوارد في كل نبضة ساعة .**

1) يجب أن تكون التعليمات التي سنمررها سوية مستقلة عن بعضها .
2) لا يشترط تمرير عدد ثابت من التعليمات في كل نبضة .
3) سندرس نوعين لـ issuing وهما : Static - Dynamic
4) في الـ Static issuing و VLIW (very Long instructions word)

وهي وضع أكثر من تعليمة بتعليمة واحدة ، يتم تحديد التعليمات التي يمكن تصديرها مع بعضها و ذلك لتسهيل عملية التصدير و فك الترميز .

2. Data and Control Hazards :

وذلك مؤكد ، لأنه سيتم إدخال أكثر من تعليمة فلا بد من حصول أعطال .

3. Instructions Issue Policy :

هناك ثلاث سياسات لإصدار التعليمات :

1- In-Order Issue with In-Order Completion.

-أي أن ترتيب الخرج نفس ترتيب الدخل .

مثلاً : كان الدخل التعليمتين 1 و 2 أولاً ثم التعليمتين 3 و 4 يكون الخرج 1 و 2 ثم 3 و 4 .

2- In Order Issue with Out-of-Order Completion.

-الخرج يكون غير ترتيب الدخل .

3- Out-of-Order issue with Out-of-Order Completion.

-الدخل لم يتم ترتيبه و بالتالي الخرج أيضاً ، لكن هنا يقوم المعالج بجمع التعليمات التي يمكن أن تكون مع بعضها و يكون الخرج حسب ذلك .

- في الثلاثة سياسات السابقة عند الانتقال من الأول إلى الأخير تزداد مرونة التنفيذ و لكن تصبح أكثر تعقيداً و تصبح نسبة حدوث الأعطال أكبر .

Aim a good deal higher than your objective

Multiple issuing		
	Dynamic	Static
Time	- على مستوى run time - يتخذ المعالج خطوات الـ issuing أثناء الـ run-time	- على مستوى الـ compiler - في البرنامج ، هناك بعض الخطوات يتخذها الـ compiler قبل التنفيذ ، مثل تجميع التعليمات لتصديرها معالجة للأعطال .
Issue packet حزمة الإصدار	- يحددها المعالج أثناء وقت التنفيذ .	- وهي مجموعة من التعليمات التي سيتم تصديرها سوية في دورة واحدة ، ويمكن أن يتم تحديدها من قبل الكومبايلر . - تتم رؤيتها على أنها تعليمة طويلة بعدة معاملات .
Examples	- المعالجات العددية الفائقة - superscalar processors : تقوم بجدولة عمليات التوارد و لديها الـ Hardware الذي تحتاجه لإعادة ترتيب التعليمات لتجنب الأعطال و التأخير . - مثل : Itanuim, Itanuim-2 processors	-VLIW (very Long instructions word) -واحدة من أهم المعالجات التي اختصت بهذا النمط هي معالجات Intel . -EPIC (Explicitly Parallel Instruction Computer)

Instruction Issue Policy

- Instruction issue : **يشير إلى عملية بدء تنفيذ في وحدات المعالج تحدث الـ issuing عندما تنتقل التعليمات من مرحلة الـ decode في التوارد إلى بداية مرحلة التنفيذ .**
- Instruction issue policy : **تشير إلى البروتوكول المستخدم في تصدير التعليمات .**

الترتيبات المهمة في هذا البروتوكول :

I. ترتيب التعليمات في مرحلة الجلب .
II. ترتيب التعليمات في مرحلة التنفيذ .
III. ترتيب التعليمات في مرحلة تحديث محتوى السجلات و عناوين الذاكرة .

التخمين Speculation

يعتمد على التنبؤ (prediction) ، حيث يخّمن الكومبايلر أو المعالج التعليمات الصادرة و أولوياتها ،
لذلك فإن قدرة التنفيذ لكل تعليمة ووقت بدئها تعتمد على التعليمات التي تم تخمينها .

مثلاً :

- في تعليمات التفرع يمكن أن تنفذ تعليمات قبل أن يتم التخمين
مثل التعليمات التي تلي تعليمة التفرع يمكن أن يتم تنفيذها
ولكن بالواقع نحن لا نحتاج لتنفيذها حسب الشرط .

- إذا كانت هناك تعليمتين Load و Store لنفس السجل
(تعليمة تكتب و أخرى تقرأ أي يوجد ترابط) .
يجب علينا وصلهما بإعادة ترتيب الكود ، لكن إذا لم تكونا مترابطتين يمكن تنفيذ تعليمة Load قبل Store حيث أن الكتابة لن تؤثر على القراءة .

Speculation Difficulty?

1. يمكن أن يكون التخمين خاطئاً
2. فإذا كان خاطئاً سنحتاج لآلية تأكد و تراجع عن التعليمات التي تم تصديرها .
3. قد يرمي استثناءات exceptions .

أنواع Speculation

Hardware Speculation

- عتاد المعالج يمكن أن يقدم نفس الانتقال في وقت التنفيذ باستخدام تقنيات مختلفة .

- عندما يكون التخمين غير صحيح ، يقوم الهارد وير بتخزين مؤقت لنتائج التخمين حتى يتم التأكد بأنه ليس صحيح .

- فإذا كان التخمين صحيحًا ، يتم إتمام التعليمات حيث يتم السماح بكتابة محتوى الـ Buffers على السجلات و الذاكرة حسب الوجهة .
- إذا كانت خاطئة ، يمسح الهارد وير محتوى الـ Buffers ، ويعيد تنفيذ تسلسل التعليمات .

Software Speculation

- يمكن أن يستخدم الكومبايلر التخمين للإعادة ترتيب التعليمات .

- لتلافي تصدير التعليمات غير الصحيح ، يدخل الكومبايلر تعليمات إضافية ليتحقق من مدى صحة التخمين و يزودها بروتين ليصلح الخطأ.

- هذا النوع يتم على مستوى الـ compiler

التخمين المعتمد على Hardware Speculation يتركب من ثلاثة أفكار أساسية :

1. التوقع الديناميكي للتفرع Dynamic Branch Prediction :
   وذلك لاختيار التعليمات التي ستنفّذ .
2. Speculation :
   وذلك للسماح بتنفيذ التعليمات قبل حدوث تبعيات تحكم وحلها فنضبح غير قادرين
   على إلغاء تأثير تسلسل التعليمات الخاطئ .
3. الجدولة الديناميكية Dynamic Scheduling :
   للتعامل مع جدولة مجموعات مختلفة من الـ Blocks للتعليمات الأساسية .

جميع التقنيات المستخدمة :

- تقنيات الجدولة الديناميكية (ذكرناها)
- Reorder Buffer (ROB)
- Commit Stage : يتم كتابة النتيجة النهائية في هذه المرحلة بعد التأكد أن كل المراحل صحيحة .

مثال : Static Multiple Issue with the MIPS

بفرض لدينا معالج MIPS بحالة static two-issue حيث أنه يصدر تعليمتين سوية
الأولى يمكن أن تكون عملية ALU أو Branche و الثانية يمكن أن تكون Load أو Store .
إن تصدير تعليمتين في النبضة الواحدة يتطلب عملية fetch و decode لـ 64 bit
(حيث نعلم أن حجم التعليمة في MIPS هو 32 bit)

في العديد من المعالجات التي تعمل بتقنية static-multiple (بشكل أساسي جميع معالجات VLIW)
يكون إصدار التعليمات مفيد بمرحلة الـ decode و إصدار كل تعليمة .

يتطلب في ذلك أن تكون التعليمات مزدوجة و تمتد على 64 bits ،
و أن تكون التعليمات التي تظهر أولًا إما ALU أو Branche.
إذا لم نستطيع إيجاد تعليمة يمكن استخدامها مع الأخرى فنستبدلها بـ NOP أي نضع مع الأخرى NOP .

Instruction type	Pipe stages							
Alu or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
Alu or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
Alu or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
Alu or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

الشرح :

كما نرى في الجدول أعلاه فإنه يتم جلب تعليمتين في نفس النبضة ، ونجد هنا حالة مثالية حيث لا يوجد لتعليمة NOP ، أو Stall .

لنفكر كم قيمة CPI :

نلاحظ أنه بعد وقت معين ، خلال كل نبضة تنتهي تعليمتين :

$$CPI = \frac{1}{2} = 0.5$$

لكن ذلك يتطلب عتاد إضافي :

1. بوابات إضافية في ملف السجلات ، حيث أنه في النبضة الواحدة ربما نحتاج لقراءة سجلين للـ ALU و سجلين للتخزين و واحدة لكتابة ناتج عمليات ALU وآخر لكتابة ناتج تعليمة الـ Load .
2. جامع إضافي (Adder) ، و ذلك لحساب عنوان التعليمة التالية .

- بهذا إن أفضل أداء يمكن الحصول عليه هو ضعف الأصلي ، فكما رأينا في الحالة المثالية كانت تتم تعليمتين في نبضة واحدة .
- لكن هذا يضاعف البيانات الممررة و يزيد نسبة الإصابة بأعطال التحكم و المعطيات .

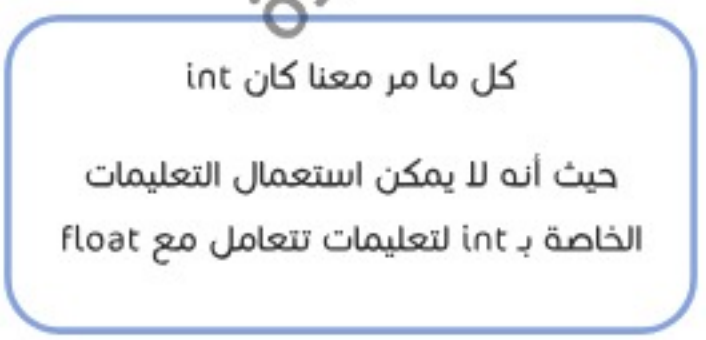
Static Multiple Issue

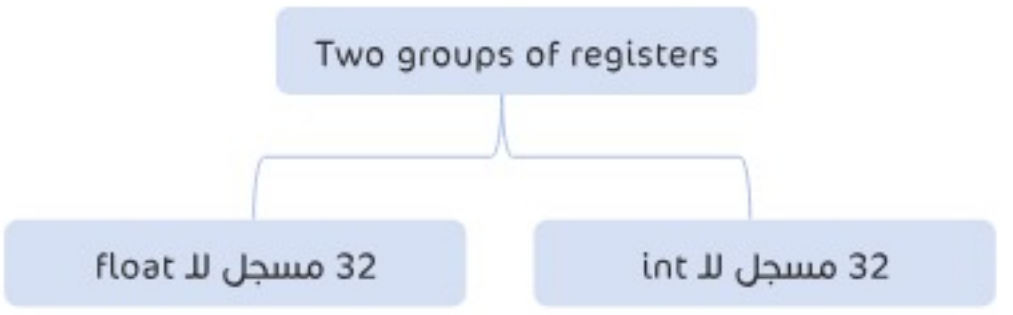
تذكرة : إن مشكلة الـ Load-use في معالج يصدر تعليمة واحدة في النبضة ، كانت تحتاج ل Stall واحدة بعد الإحالة (يتم التأخير بنبضة واحدة) .

Use latency : هي عدد النبضات التي تكون بين تعليمة **Load** وتعليمة أخرى تعتمد على ناتج كتابة الـ **Load** ، وذلك بدون إيقاف التوارد .

وفي معالج يصدر تعليمتين (**two issue processors**) فإن مقدار الـ **use latency** لتعليمات الـ **Load** هي أيضاً بنبضة واحدة .

حيث أن التعليمتين التاليتين لـ **Load** لا يمكن أن يستخدموا ناتج الـ **Load** بدون عمليات **stalls**.





Loop Unrolling

أعد ترتيب تعليمات الحلقة التالية و ذلك بحالة static two-issue pipeline for MIPS

و ذلك لتجنب الـ Stall قدر الإمكان بفرض أن التفرعات قد تم التنبؤ بها و يتم التعامل معها

من قبل Hardware ، في الكود التالي نمر على عناصر مصفوفة :

```
Loop :  1  lw    $t0   0($s1)          # $t0 = array element
        2  addu  $t0   $t0   $s2       # add scalar in $s2
        3  sw    $t0   0($s1)          # store result
        4  addi  $s1   $s1   -4        # decrement pointer
        5  bne   $s1   $zero ,Loop     # branch $s1 != 0
```

تذكرة : التعليمتين الأخيرتين لاستمرار مرور الحلقة على عناصر المصفوفة مع شرط المرور .

لاحظنا أنه لدينا 3 تبعيات 3 Data Dependences و هي من نوع True dependences

تحسن الأداء باستخدام الـ multiple issue .

-الجدول التالي يبين كيف تمت العملية :

	ALU or Branch instruction	Data transfer instruction	Clock cycle
Loop :	NOP	lw $t0 , 0($s1)	1
	addi $s1 , $s1 , -4	NOP	2
	Addu $t0 , $t0 , $s2	NOP	3
	Bne $s1 , $zero , LOOP	sw $t0 , 4($s1)	4

الشرح :

-التعليمة الأولى هي Load يجب أن تكون معها تعليمة ALU ، و لكن التعليمة التي بعدها مرتبطة بها

- لذلك تحتاج لنبضة بينهما ، فنضع Lw مع NOP في النبضة الأولى ونضع addu في النبضة الثالثة ،

- sw مرتبطة بمعامل addu الذي تكتب عليه لذلك نضعها بعدها في النبضة الرابعة ،

- addi لا ترتبط بشيء فنضعها في النبضة الثانية حيث لم نضع بها شيء و يجب أن يكون هناك نبضة

- بين Lw و addu نضع معها NOP لأنه لا يوجد تعليمة تحميل أو تخزين تناسب لتكون معها ،

-و كذلك addu ، بقيت تعليمة التفرع فنضعها في النهاية مع sw في النبضة الرابعة .

- نلاحظ أننا احتجنا لأربع نبضات لتنفيذ 5 تعليمات (دون NOP)
- $CPI = \frac{4}{5} = 0.8$
- يمكننا تحسين الأداء أكثر و ذلك باستخدام Loop unrolling و register renaming .
- هناك تقنية هامة يقوم بها الـ compiler تعطي أداء أفضل للحلقات

فمثلاً : نريد الوصول لعناصر مصفوفة عددها 20 ، نصل لعنصرين بدون حلقة ثم نكررهما عشر مرات بحلقة ، بدل من أن نصل لعنصر و نكرره 20 مرة , حيث خلال عملية الـ unrolling نغير أسماء للسجلات أو نستعين بسجلات ثانية عند تكرار التعليمات مثل ($t1 ,$t2 ,$t3) وذلك للتقليل من الـ name dependences .

	ALU or Branch instruction	Data transfer instruction	Clock cycle
Loop :	addi $s1 , $s1 , -16	lw $t0 , 0($s1)	1
	NOP	lw $t1 , 0($s1)	2
	addu $t0 , $t0 , $s2	lw $t2 , 0($s1)	3
	addu $t1 , $t1 , $s2	lw $t3 , 0($s1)	4
	addu $t2 , $t3 , $s2	lw $t0 , 0($s1)	5
	addu $t3 , $t3 , $s2	lw $t1 , 0($s1)	6
	NOP	lw $t2 , 0($s1)	7
	addu $s1 , $zero , Loop	lw $t3 , 0($s1)	8

ماذا نلاحظ ! :

- نتنبه لفكرة الـ Load use كما في الجدول الماضي ، و هنا أعدنا كل تعليمة أربع مرات عدا التعليمات الخاصة بالتحكم بالحلقة ، وبالنسبة لقيمة الـ offset في كل من lw و sw ، فإن كل عنصر يحتاج 4 byte وقد أعدناها 4 مرات أو تبعد عن العنوان الأساسي أربع خطوات (16 byte) ، و نتراجع 4 هكذا لتنتهي ، و نتنبه لاختلاف تسميات السجلات لمنع حدوث name dependences .

- نلاحظ أنها أصبحت 14 تعليمة استغرقت 8 نبضات
- $CPI = \frac{8}{14} = 0.57$

Basic Pipeline Scheduling and Loop Unrolling

- إن أفضل للتوارد عندما يتم الحفاظ على الكود دون Stalls .
- إيجاد تسلسل مناسب لتعليمات غير مترابطة لتستطيع أن تدخل منطقة التوارد .
- يجب أن يتم الفصل بين التعليمات التي تعتمد على بعضها بنبضات (بتعليمات أخرى) وذلك بمقدار الـLatency لتلك التعليمات .

مثال :

- Adding a scalar to a vector
- for (i = 999; i >= 0; i = i–1)
- x[i] = x[i] + s;

سنفرض أن R1 هو عنوان العنصر من المصفوفة ذو العنوان الأعلى (العنصر الأخير)

و F2 تحتوي على القيمة التي سنضيفها .

```
Loop:   L.D      F0 , 0(R1)        # F0= array element
        ADD.D    F4 , F0 , F2      # add scalar in F2
        S.D      F4 , 0(R1)        # store result
        DADDUI   R1 , R1 , -8      # تسيير العداد ;8 bytes (per DW) (لأنها float)
        BNE      R1 , R2 , Loop    # branch R1!=R2
```

Latencies of FP operations used

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

يوضح الجدول مقدار التأخير (latency) التي يجب أن تكون بين أنواع التعليمات

لنرى كيف يصبح الكود السابق تبعاً لهذا الجدول :

```
                                  Clock cycle issued
Loop:   L.D      F0 , 0(R1)       1
        Stall                     2
        ADD.D    F4 , F0 , F2     3
        Stall                     4
        Stall                     5
        S.D      F4 , 0(R1)       6
        DADDUI   R1 , R1 , #-8    7      # نحصل على الناتج في مرحلة الـ exe
        Stall                     8
        BNE      R1 , R2 ,Loop    9      # المقارنة تتم في الـ decode
```

الشرح :

حسب الجدول هناك نبضة تأخير واحدة بين الأولى (تعليمة load) و الثانية (ALU) لذا وضعنا Stall واحدة ،
ثم بين الثانية و الثالثة (store) حسب الجدول هناك نبضتي تأخير فتم وضع 2 Stalls ،
وأما بالنسبة لمرحلة الرابعة و الخامسة فإن نسبة التأخير التي تسبب مشاكل هي نبضة واحدة بين exe و decode
لذلك وضعنا Stall واحدة فقط .

في الكود السابق : فصلنا بين التعليمات بدون إعادة ترتيبها و نتج عن ذلك أن عدد النبضات
التي استغرقتها العملية هي 9 نبضات .

لنرى إذا غيرنا الترتيب :

Basic Scheduling

```
Loop:   L.D      F0 , 0(R1)      1
        DADDUI   R1 , R1 , #-8   2
        ADD.D    F4 , F0 , F2    3
        Stall                    4
        Stall                    5
        S.D      F4 , 8(R1)      6
        BNE      R1 , R2 ,Loop   7
```

- نلاحظ أن عدد النبضات المستغرقة أصبحت سبع نبضات.

Loop Example with Delayed Branch Slot

هنا يوجد نبضة تأخير بعد تعليمة التفرع وذلك لضمان عدم تنفيذ تعليمات خاطئة قبل التحقق من الشرط ،
لنرى الكود قبل عمل schedule :

Unscheduled code: 10 clock cycles

```
1 Loop:  L.D      F0 , 0(R1)
2        stall                          #load interlock
3        ADD.D    F4 , F0 , F2
4        stall                          #data hazard (F4)
5        stall                          #data hazard (F4)
6        S.D      F4 , 0(R1)
7        DADDUI   R1 , R1 , #-8
8        stall                           #branch interlock
9        BNE      R1 , R2 , Loop
10       stall                          #delayed branch slot
```

نلاحظ أن ذلك استغرق عشر نبضات .

Scheduled code: we need 6 cycles

```
1 Loop:   L.D      F0 , 0(R1)
2         DADDUI   R1 , R1 , #-8
3         ADD.D    F4 , F0 , F2
4         stall
5         BNE      R1 , R2 , Loop
6         S.D      F4 , 8(R1)
```

هنا استطعنا وضع S.D بعد BNE وذلك بسبب وجود نبضة تأخير بعد التفرع ، ولكن عندما لا تكون لا نستطيع لأنه يمكن أن تكون التعليمة التي بعدها لا يمكن تغييرها

#data hazard (F4)

يجب أن يكون بينهما نبضتين

0+8 = 8

- نلاحظ أن عدد النبضات المستغرقة 6 نبضات .

LOOP Unrolling Example

هنا لنشر حلقة ما ، ننسخ التعليمات الفعالة المكررة عدة مرات و نعدل عداد الحلقة بما يتناسب مع التغيير .
بهذه الطريقة يمكننا التخلص من كل الـ Stalls ، ولكن ذلك يتطلب سجلات أكثر للتنفيذ .

```
1 Loop:   L.D      F0 , 0(R1)
2         ADD.D    F4 , F0 , F2      أول تكرار
3         S.D      F4 , 0(R1)        #drop DADDUI & BNE
4         L.D      F0 , -8(R1)
5         ADD.D    F4 , F0 , F2      الثاني
6         S.D      F4 , -8(R1)       #drop DADDUI & BNE
7         L.D      F0 , -16(R1)
8         ADD.D    F4 , F0 , F2      الثالث
9         S.D      F4 , -16(R1)      #drop DADDUI & BNE
10        L.D      F0 , -24(R1)
11        ADD.D    F4 , F0 , F2      الرابع
12        S.D      F4 , -24(R1)
13        DADDUI   R1 , R1 , #-32    #-8 *4 = -32
14        BNE      R1 , R2 , LOOP
15        NOP
```

نلاحظ وجود Name dependencies لذلك يجب علينا تغيير أسماء السجلات المكررة .

يصبح الكود unrolled , unscheduled :

1 Loop:	L.D	F0 , 0(R1)	→ stall	
2	ADD.D	F4 , F0 , F2	→ 2 stall	
3	S.D	F4 , 0(R1)		
4	L.D	F6 ,-8(R1)	→ stall	# Renamed F0
5	ADD.D	F8 , F6 , F2	→ 2 stall	# Renamed F4, F0
6	S.D	F8 , -8(R1)		# Renamed F4
7	L.D	F10 , -16(R1)	→ stall	# Renamed F0
8	ADD.D	F12 , F10 , F2	→ 2 stall	# Renamed F4, F0
9	S.D	F12 , -16(R1)		# Renamed F4
10	L.D	F14 , -24(R1)	→ stall	# Renamed F0
11	ADD.D	F16 , F14 , F2	→ 2 stall	# Renamed F4, F0
12	S.D	F16 , -24(R1) # Renamed F4		
13	DADDUI	R1 , R1 , #-32	→ stall	
14	BNE	R1 , R2 , LOOP		
15	NOP		→ stall	

نلاحظ أنه أصبح هناك 28 نبضة لكل iteration 7 نبضات .

لنعيد ترتيب التعليمات unrolled , scheduled :

1 Loop:	L.D	F0 , 0(R1)	8	ADD.D	F16 , F14 , F2
2	L.D	F6 , -8(R1)	9	S.D	F4 , 0(R1)
3	L.D	F10 , -16(R1)	10	S.D	F8 , -8(R1)
4	L.D	F14 , - 24(R1)	11	DADDUI	R1 , R1,#-32
5	ADD.D	F4 , F0 , F2	12	S.D	F12 , 16(R1)
6	ADD.D	F8 , F6 , F2	13	BNE	R1 , R2 , Loop
7	ADD.D	F12 , F10 , F2	14	S.D	F16 , 8(R1)

- نلاحظ أن عدد النبضات انخفض لـ 14 نبضة .

- في تعليمتي الـ S.D الأخيرتان ، قيمة R1 تغيرت قبل ، حيث أصبحت :

. 8 = 24 - 32 (الثانية) 16 = 16- 32+ -

Compiler Perspectives

الحلقات غير المنتظمة Unrolling Loops :

- من وجهة نظر الـcompiler فإن الحد الأعلى للحلقة في عملية unrolling loops غير معروف ،
- لنفترض أنه n ونريد أن ننسخ من الحلقة k نسخة ، لن ننشأ single unrolled loop ،
- بل ننشأ زوج من الحلقات المتتالية عن طريق :

1. تنفيذ الحلقة الأصلية (n mod k) مرة . (mod : باقي قسمة)
2. تنفيذ الـ unrolled loop (n/k) مرة .
3. 3إذا كانت n كبيرة معظم التكرارات تحدث في unrolled loop .

التبعيات Dependencies :

- من وجهة نظر الـcompiler يجب عليه الحفاظ على تبعيات البيانات (data dependencies)
- و لا يجب تغيير تبعيات و ارتباطات البيانات حتى مع تغيير الأسماء أو الترتيب يجب مراعاة ذلك عن طريق :

4. تحديد فيما إذا كانت تكرارات الحلقة مستقلة .
5. إعادة تسمية السجلات خلال عملية نشر الحلقة .
6. التخلص من تعليمات التفرع .
7. ضبط إعدادات الحلقة من شروط و مقدار تسرع العداد .
8. تحديد فيما إذا كان يمكن التبادل بين تعليمات الـ Load و الـ Store .
9. جدولة و إعادة ترتيب التعليمات مع الحفاظ على التبعيات .

إعادة التسمية Renaming :

- إن التعليمات المترابطة لا يمكن تنفيذها على التوازي ،
- من السهل تحديد الترابط بالسجلات من أسمائها (يمكننا الإصلاح بتغيير الأسماء)
- ولكن سيكون ذلك أصعب بالنسبة للذاكرة .
- وهذه المشكلة تسمى (memory disambiguation) توضيح محتويات الذاكرة .

- مثلاً لدينا سجلات مع عناوين خاصة بها في الذاكرة .

كيف نعرف إذا (R6) 20 = (R4)100 ، و هل في دورات مختلفة ستكون (R6)20 = (R6)20 :

- في مثالنا السابق يجب أن يحدد الـ compiler بأنه إذا لم يتغير R1 فإن :

(R1)24- ≠ (R1)16- ≠ (R1)8- ≠ (R1)

- وفي هذه الحالة إذا أردنا إعادة الترتيب يمكن المبادلة بين تعليمات load و store .

The End