6
15
1800 sp
نظري

Pipelining & Advanced pipelining

د. سيرا أستور

11/11/2024

بنيان الحاسوب ٢

تعرفنا في المحاضرة السابقة على مفهوم الـ Pipelining وما هي الأعطال التي تواجهنا عند استخدامه.
اليوم سنتابع عرض هذه المشكلات وسنتعرف على حلولها.

3. Control Hazard (عطل التحكم)

(خطر التحكم): يحدث عندما نريد اتخاذ قرار أو عمل إجراء قبل معرفة نتيجة الحالة السابقة التي لا تزال قيد التنفيذ.
سبب ذلك يمكن أن يكون: تنفيذ خارج التسلسل (مثل تعليمات القفز والتفرع)، حيث ممكن أن تكون التعليمة أو المعامل الذي سنستخدمه ليس متاحاً بعد.

مثلا تعليمات القفز:

عندما يتم الوصول لتعليمة قفز، المشكلة أنه يتم جلب التعليمة التالية قبل معرفة نتيجة الشروط، ونعلم أنه إذا تحقق الشرط نذهب للتعليمة المطلوبة وإذا لم يتحقق نذهب للتعليمة التي تليها:

حيث أنه إما سيتم زيادةُ الـ (PC + 4) أو لا يتغير

تذكرة: العنونة بالذاكرة تكون بالبايت
وليست بالكلمة

Branch Not Taken
Branch does not change the (pc + 4)

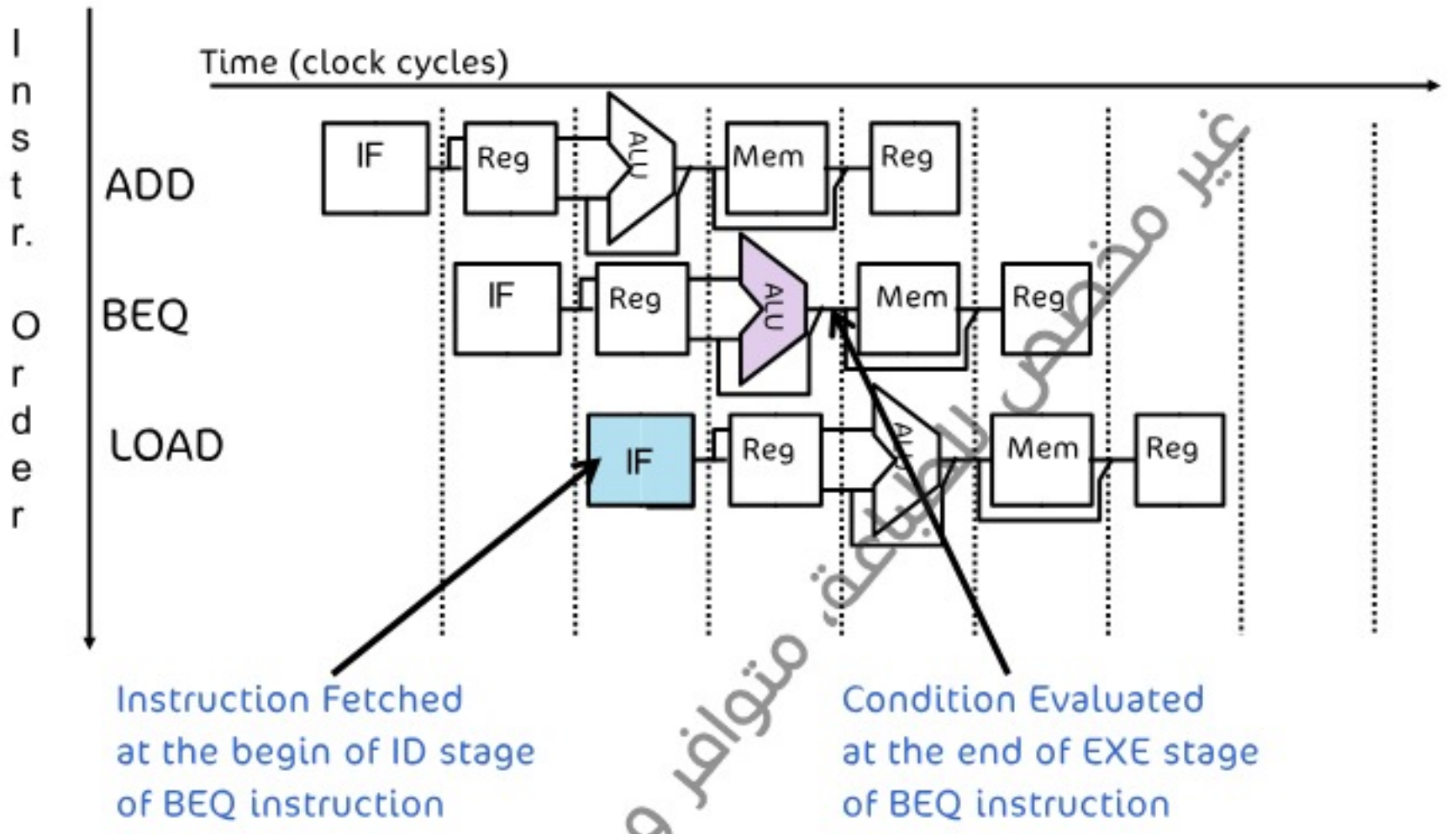
Branch Taken
Branch changes the (pc + 4) to new target

الطرق البسيطة للتعامل مع مشاكل تعليمات (Branch) هي:

1. (Freeze the pipeline) : بما معناه تجميد التعليمات بعد تعليمات التفرع (عملية stall أو bubble) ريثما تصدر نتيجة الشرط .
2. (Flush the pipeline) : حذف كل التعليمات بعد التفرع إذا كان الشرط قد تحقق و تفرعت التعليمة إلى الـ target المطلوب .

مثال يوضح سبب حدوث أعطال التحكم (Control Hazard illustration):

لدينا تعليمة BEQ (Branch if equal) والشرط محقق، عنوان التعليمة التالية يتم تحديده بعد فحص شرط التفرع في مرحلة التنفيذ EX، لكن التعليمة التالية يتم جلبها عندما تصبح هذه التعليمة في مرحلة فك الترميز ID.
أي قبل أن يتغير الـ (pc+4) إلى عنوان الـ target الجديد وذلك يؤدي لحدوث Control Hazard.



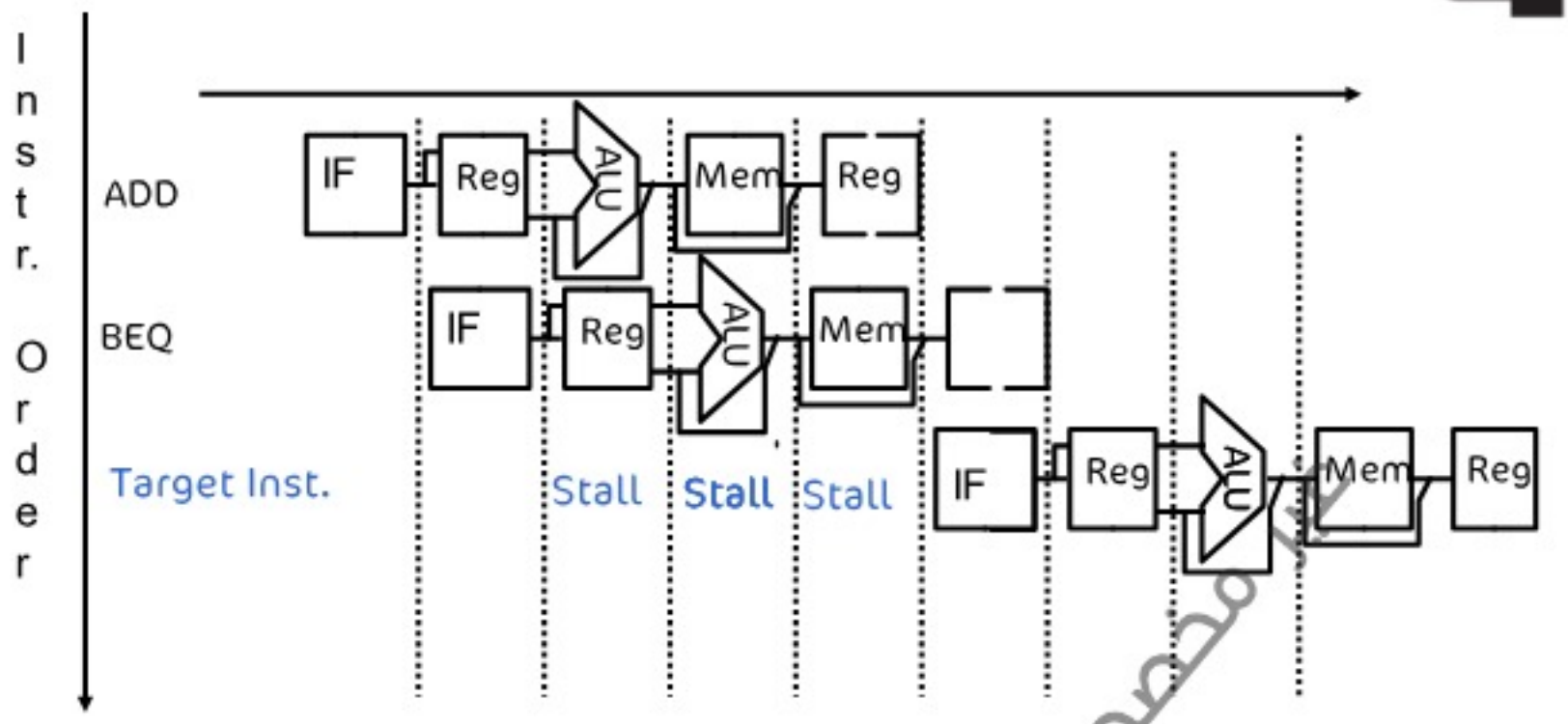
كما نلاحظ ناتج تعليمة BEQ يخرج بعد الـ ALU
لكن التعليمة التي تليها تبدأ عندما تكون تعليمة BEQ في مرحلة فك الترميز أي قبل فحص الشرط.

Branch Hazard Solutions

1. Stall.
2. Redo Fetch after branch.
3. Delayed branching.
4. Branch prediction.
5. Multiple Streams.

1. Stalls

إن نتيجة تعليمة التفرع تكون متاحة بعد مرحلة التنفيذ وعنوان الـ target يكون متاح في المرحلة التالية
أي يتم جلبه في العنوان التالي، لذلك كما نلاحظ أدناه في المخطط تم استخدام Stalls.



نعلم أنه في مرحلة ID (instruction decode) تتم قراءة السجلاتrs و rt

على الرغم من عدم معرفته نوع التعليمة بعد، هناك تعليمات لا تحتاج للسجل rt لكن تتم قراءتها

وإذا لم نستخدمها نتخلص منها، وذات الشيء في تعليمة Branch حيث يتم حساب العنوان في هذه المرحلة، وتم هنا التخلص من stall واحدة

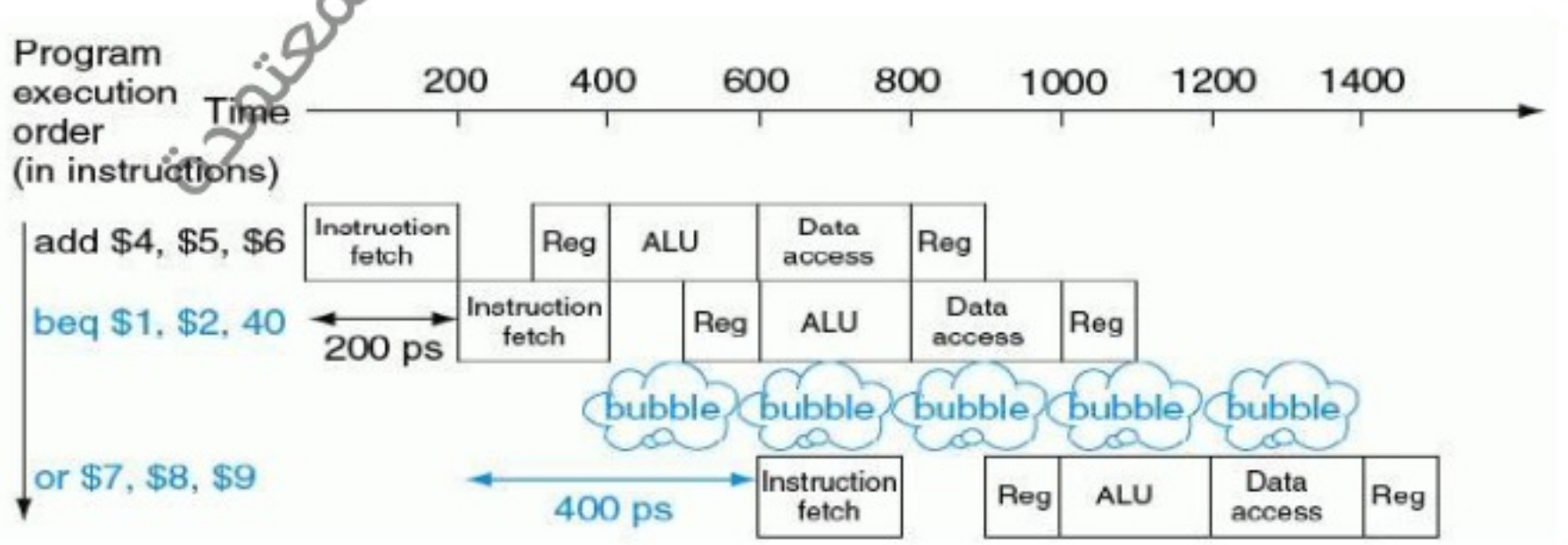
ونضيف في الـ Hardware، ADDer لتساعدنا في الحساب ولكن مع ذلك ستبقى Stall واحدة .

ستبقى Stall واحدة لأننا سننتظر نهاية ID لتعليمة Branch قبل جلب التعليمة التالية.

إذاً هنا حاولنا التقليل من حالات التوقف ومع ذلك بقي واحدة .

أضفنا قطع إضافية للـ Hardware في مرحلة الـ Decode :

1. ALU وحدة حساب منطقية إضافية لحساب عنوان التفرع
2. Comparator مقارن لتوليد الإشارة الصغرية
3. Hazard detection unit وحدة اكتشاف الـ Hazards تكتب عنوان التفرع في الـ pc



2. Redo Fetch after Branch

- إذا تم اكتشاف تعليمة تفرع خلال مرحلة قراءة التعليمة، فإن التعليمة التالية التي سيتم جلبها يجب أن يحصل لها Stall.
- إذا اعتبرنا أن شرط التفرع محقق، فإن التعليمة التي يتم جلبها في هذه الدورة سيتم تجاهلها ولن تقدم أي عمل.
- إعادة جلب تعليمة التفرع (target) سوف يزودنا هذا بالتعليمة الصحيحة
- إن لم يتحقق شرط التفرع فإن عملية الجلب التالية غير ضرورية
- يكون التأخير بنبضة واحدة فقط لتعليمة التفرع إذا لم يتحقق شرطها

شرح:

عمل fetchجديدة وعدم وضع stall، لأن الشرط يمكن أن يكون إما محقق أو غير محقق، فنقوم بجلب للتعليمة التي تليها لأن الشرط ممكن ألا يتحقق ونريد التعليمة التالية لها تماما.

3. Delayed Branch

هو حل يتم تنفيذه في الـ compiler وهو يعتمد على إعادة ترتيب التعليمات.

مثال:

- Stall on branch

```
        add $4, $5, $6
        beq $1, $2 , skip
        next instruction
        . . .
skip    or $7, $8, $9
```

- Delayed branch

```
        beq $1, $2 , skip
        add $4, $5, $6
        next instruction
        . . .
skip    or $7, $8, $9
```

- لماذا وضعنا add بعد تعليمة التفرع وكيف استطعنا ذلك؟

كما نعلم إن لم يتحقق شرط التفرع فستنفذ التعليمة التالية وإن تحقق فسيتم القفز للفرع المطلوب.
هنا سيصبح لدينا إشكال في جلب العنوان أو في بعض المعاملات، ولتجنب وضع أي Stalls
فإننا نضع بعد الـ Branchتعليمة لا تؤثر على تعليمة التفرع (أي لا تؤثر على مضمون معاملاتها)
ولا تؤثر على الـ target الذي سنقفز اليه في حال تحقق الشرط.

- إن لم نجد تعليمة مستقلة نضع تعليمة NOP.

4. Prediction

Pipeline Flush: في هذه التقنية بالنسبة لتعليمة التفرع ، نخمّن اتجاه واحد للتفرع للبداية ،

ونعود إذا كان الشرط غير محقق.

هناك توقعين محتملين:

1. Branch taken (محقق)
2. Branch not taken (غير محقق)

- ننفذ هذه الطريقة: إما باستخدام سجل ذو بت واحد، أو باستخدام سجل ذو بتين، وسنستخدم هنا مخططات FSM.
- يستخدم هذا الحل في الحلقات ويكون أكثر إفادة فيها.

للتنبؤ نوعان :

- Static

- Compile Time.

- يعتمد على الجزء البرمجي (compiler).

- أبسط مخطط هو التنبؤ بأن شرط التفرع محقق.

- نسبة التوقع الخاطئ تساوي توقع الفرع غير المحقق والذي يساوي في مقاييس الـ SPEC 34%.

- Dynamic

- Rum Time.

- يعتمد على العتاد (Hardware).

- أبسط مخطط يمكن التنبؤ به:

(branch-prediction buffer)

or (branch history table)

- حيث هناك ذاكرة صغيرة لها عناوين لتعليمات التفرع ، سنوضح ..

Branch Prediction Buffer

- يمكن تعريفها كذاكرة (cache) خاصة تصل مع عنوان التعليمة خلال مرحلة جلب التعليمة IF.
- أو يمكن اعتبارها كزوج من البتات مرفقة بكل جزء من ذاكرة التعليمة المؤقتة ويتم جلبها مع التعليمة. حيث نستخدم الـ cache لتخزين الحالة السابقة التي تحدد في كل لحظة إذا كان سيتم القفز أم لا،

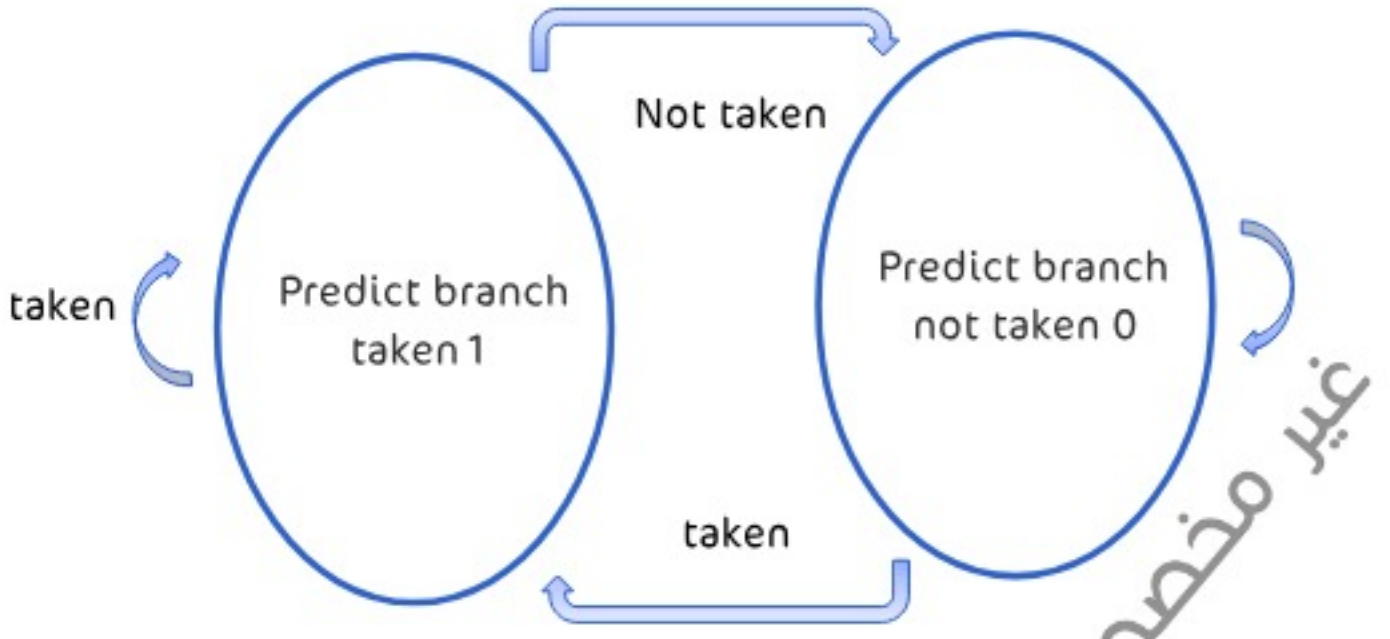
أي أنه يعتمد على الـ historical data

ولها نوعان:

- One bit prediction scheme:
  - تحتوي الذاكرة على بت واحد يخبرنا إذا كانت تعليمة التفرع محققة أم لا.
- Two bits prediction scheme:
  - تحوي الذاكرة بتين، يجب أن يتم التنبؤ مرتين حتى تتغير الحالة.
  - مثل الـ FSM machine

ستتضح الفكرة في الفقرات التالية.

- 1-bits Branch Prediction Buffer:

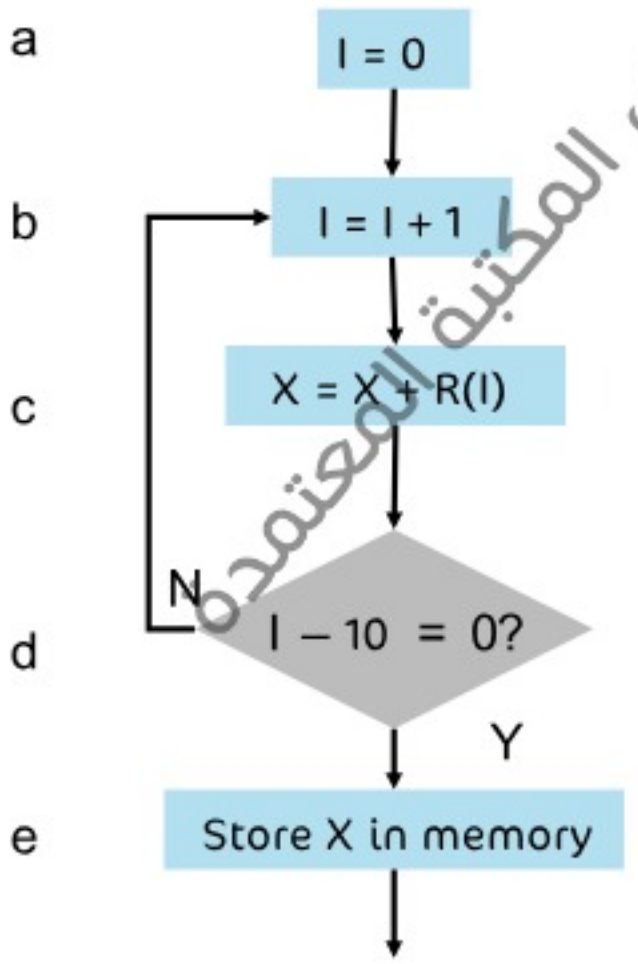


التمثيل يتم ببت واحد، ويسمى بـ (History bit)

History bit = 1, branch was taken
History bit = 0, branch was not taken

إذا كان شرط التفرع محقق أول مرة فيأخذ البت قيمة 1،
إذا تحقق مرة ثانية فيعود ليأخذ 1،
عندما نعيد الشرط ولا يتحقق تصبح قيمته 0.

Branch Prediction for a Loop



Execution of Instruction d

Execu-tion seq.	Old hist. bit	Next instr.			New hist. bit	Predic tion
		Pred.	I	Act.		
1	0	e	1	b	1	Bad
2	1	b	2	b	1	Good
3	1	b	3	b	1	Good
4	1	b	4	b	1	Good
5	1	b	5	b	1	Good
6	1	b	6	b	1	Good
7	1	b	7	b	1	Good
8	1	b	8	b	1	Good
9	1	b	9	b	1	Good
10	1	b	10	e	0	Bad

h.bit = 0 branch not taken , h.bit = 1 branch taken .

- لدينا حلقة أي لدينا شرط.

شرح الجدول:

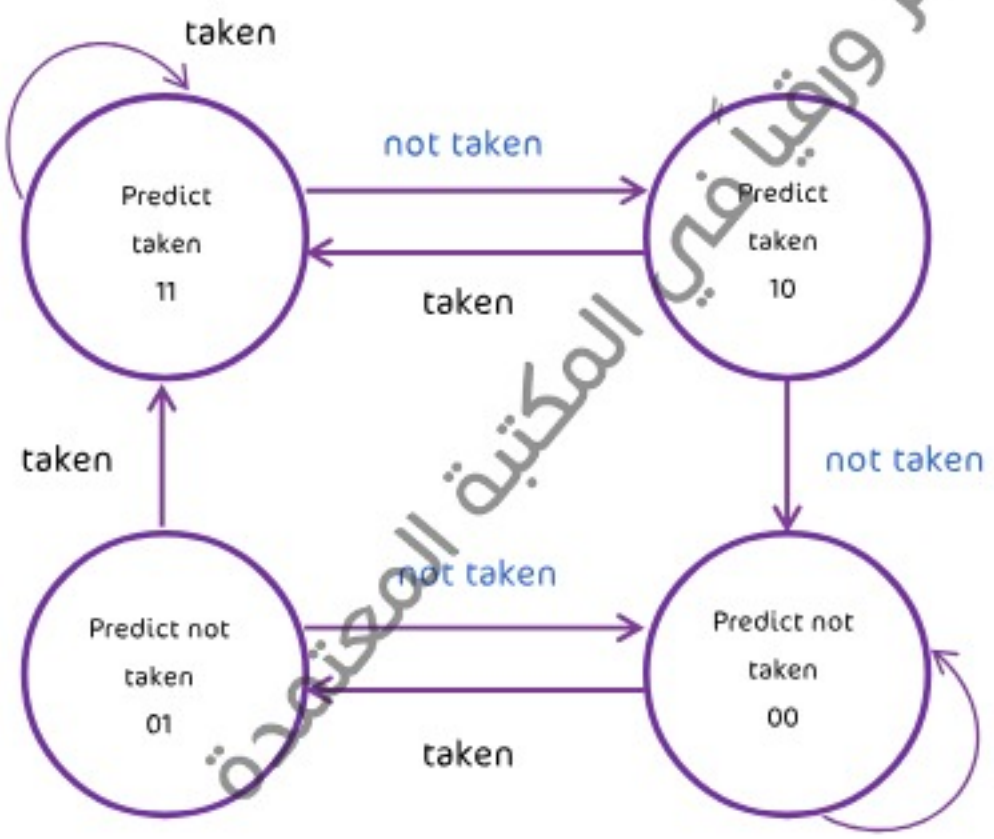
- حقل (old history bit) : يحمل الحالة السابقة إذا كانت (1) taken أو (0) not taken .
- حقل (next instruction) : وفيه ثلاث حقول :

- Prediction: يعتمد على حالة الـ history.
- Instruction: رقم التعليمة (دورة الحلقة).
- Actual: هل تنبؤ الحالة التالية صحيح أما لا.

شرح المثال:

- عند البدء بتنفيذ التعليمات، ستتنبأ أنه سينفذ الخطوة e، لكنه في الحقيقة لم يتحقق شرط الحلقة لذلك سيتم تنفيذ b، فالتنبؤ خاطئ، في التنفيذ التالي تصبح الـ next history bit هي الـ old history bit ثم يتم القفز ونبقى ضمن الحلقة حتى تنتهي، والخروج يكلف تنبؤ خاطئ.
- لذلك ببت واحد فقط وُجد تنبؤين خاطئين من أصل 10
- وهكذا يكون 80% تنبؤ صحيح، وهذه أقل نسبة نحصل عليها من التنبؤ.

2-bits Branch Prediction Buffer:



مخطط حالات الـ 2 bits باستخدام FSM:

- عند الانتقال من حالة لحالة نغير بت واحد فقط وليس بتين.
- نختار البت الذي سنغيره حسب تسلسل الترميز

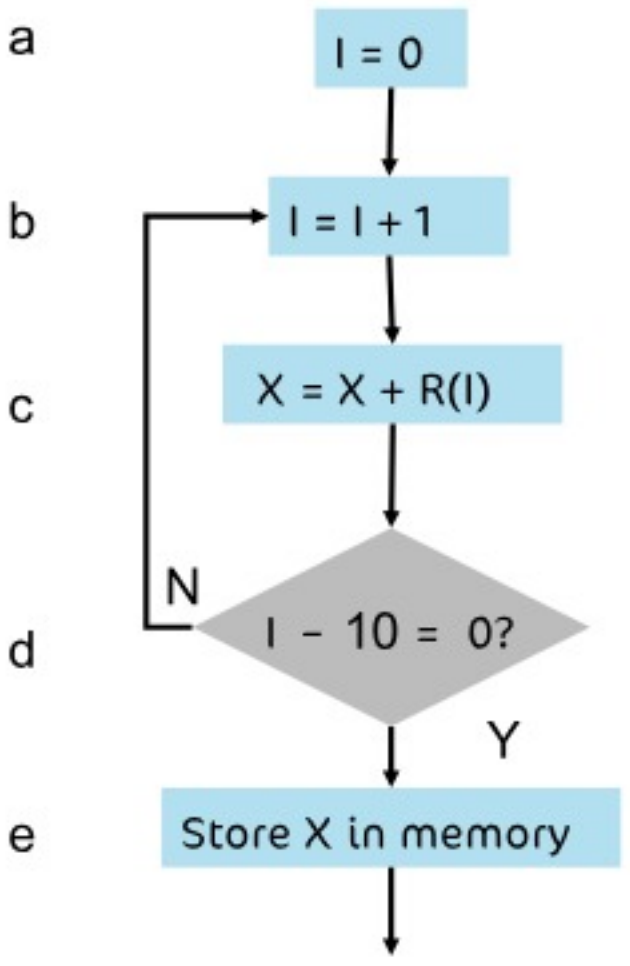
مثلاً: كنا في الحالة 10 فإذا كان التالي:

taken: تصبح البتات 11

not taken: تصبح البتات 00

ملاحظة: أن 10 هي taken وفيها خطأ واحد فقط.

Branch Prediction for a Loop



Execution of Instruction d

Execu-tion seq.	Old Pred.Buf	Next instr.			New pred. Buf	Prediction
		Pred.	I	Act.		
1	10	b	1	b	11	Good
2	11	b	2	b	11	Good
3	11	b	3	b	11	Good
4	11	b	4	b	11	Good
5	11	b	5	b	11	Good
6	11	b	6	b	11	Good
7	11	b	7	b	11	Good
8	11	b	8	b	11	Good
9	11	b	9	b	11	Good
10	11	b	10	e	10	Bad

شرح المثال:

نلاحظ في الحالة الأولى، كانت 10 (taken) وبقيت taken ولكن هذا التسلسل يؤدي إلى الترميز 11، لذلك فهي ليست خاطئة ولكن الفرق في الوصول إلى الحالة taken .

ثم نتابع في الحلقة حتى يتحقق الشرط فنخرج منها، ويكون تنبؤنا خاطئ بالنسبة للتنفيذ الأخير.

وهنا أصبحت صحة التنبؤ بنسبة 90٪.

Branch delay:

- درسنا أن الحل يعتمد على إعادة ترتيب تعليمات البرنامج.

لنتعرف على طرق إعادة الترتيب تتم هذه الحلول في الـ (Compile Time) :

a

From before

```
add $s1, $s2, $s3
if $s2 =0 then
    Delay slot
```

b

From target

```
sub $t4, $t5, $t6
. . . . .
add $s1, $s2, $s3
if $s1 = 0 then
    Delay slot
```

c

From fall

```
add $s1, $s2, $s3
if $s1 =0 then
    Delay slot
sub $t4, $t5, $t6
```

في كل حالة من الحالات السابقة يجب ان نجد تعليمة نضعها بدل مربع التأخير، لأنه يشكل stall أو NOP

```
if $s1 =0 then
    add $s1, $s2, $s3
```

```
add $s1, $s2, $s3
if $s1 =0 then
    Sub $t4, $t5 ,$t6
```

```
add $s1, $s2, $s3
if $s1 =0 then
    Sub $t4, $t5 ,$t6
```

نلاحظ أن تعليمة الطرح كانت موجودة قبل الـ target لذلك رفعناها لأعلى وبقيت نفس العمل حيث أنه إذا لم يتحقق الشرط سوف تتنفذ.

هنا يتم القفز للخلف ولكي نعبئ التأخير ننسخ الـ target ونضعه بدل التأخير فيصبح إذا تحقق الشرط أو لم يتحقق سينفذ ونلاحظ أنه لا يؤثر على البرنامج.

لأنه يوجد استقلال تام بالتعليمات، أي لا يوجد تغيير في المعاملات المشتركة بين التعليمتين، لذلك استطعنا وضعها بدل التأخير .

Keep Going ...

Advanced Pipelining

يواجه التوارد ثلاثة أخطار (أعطال) Hazards:

Structural:	Data:	Control:
كالوصول للذاكرة يكون هناك تعليمة تكتب وتعليمة ثانية تقرأ بنفس النبضة من نفس الذاكرة	وهنا يوجد مشكلة كبيرة مرت معنا سابقاً : وهي Load - use	مثل علاقات الـ Branch

الحلول الشائعة لتلك المشاكل:

Stalls (NOP)	Code recording	Add Hardware (more resources, forwarding)

وترافقها ثلاث تبعيات Dependences:

Data dependences (True data dependences) إذا كان هناك تعليمتان مرتبطتان بالبيانات فلا يمكن تنفيذهما سوية أو أن يتم تنفيذهما بشكل متداخل	Name dependences	Control dependences

1. Data dependences

مثال :

لنفرض وجود تعليمتان متتاليتان الأولى (i) والثانية (j).

التعليمة (j) تعتمد على بيانات التعليمة (i) مثلاً عن طريق الحالات التالية:

- أحد معاملات التعليمة (j) هو ناتج التعليمة السابقة لها (i).
- (j) تعتمد على (i) و (i) تعتمد على (k) (سلسلة ارتباطات).

- تتدفق (تنتقل) البيانات بين التعليمات إما من خلال :

Registers : يكون الارتباط باستخدام اسم السجل نفسه .

Memory Location : أصعب شيء يتم التعامل معه هو تدفق البيانات في الذواكر .

فمثلاً : 100(R4) و 20(R6) يمكن أن يكونا تمثيلين مختلفين لعنوانين متطابقين .

2. Name Dependent

يحدث ذلك عندما يكون هناك تعليمتان تستخدمان نفس السجل أو نفس موقع الذاكرة .

لكن لا يوجد تدفق للبيانات بين التعليمات المرتبطة بهذا الاسم.

ملاحظة: ليست من نوع true dependence

لها نوعان:

1) **Anti-dependence:**

مثلاً: تعليمة (j) تكتب على السجل بينما التعليمة (i) تقرأ من نفس السجل.

2) **Output dependence:**

مثلاً: التعليمتان (i) و (j) يكتبان على نفس السجل أو موقع الذاكرة ويجب المحافظة على ترتيب التعليمات.

الحل:

بإعادة تسمية السجلات إما باستخدام compile أي برمجياً أو باستخدام Hardware عملياً.

مثال اكتشف التبعيات:

```
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 ;   # decrement pointer  8 bytes
        BNE  R1 ,R2,Loop ;       # branch R1!=R2
```

1- **Data Dependences:**

1. التعليمتين الأولى والثانية:

الأولى تكتب ضمن السجل F0 والثانية التالية لها مباشرة تستخدمه في عملية حسابية.

2. التعليمتين الثانية والثالثة:

حيث أن الثانية تضع ناتج العملية في F4 والتعليمة التالية تقرأ من F4 وتخزن في الموقع O(R1) .

3. كذلك في التعليمتين الأخيرتين:

يتم الكتابة على R1 والقراءة منه قبل أن تتم الكتابة.

2- **Name Dependences:**

- تعليمة DADDUI تكتب على السجل R1، والتعليمة التي تسبقها تقرأ منه.

فيكون الحل هنا بإعادة تسمية السجل فتصبح آخر تعليمتين:

```
DADDUI R4, R1, #-8;
BNE R4, R2, Loop;
```

Data Hazards :

- يمنع الـ Hazard تنفيذ التعليمة التالية في تسلسل التعليمة في الوقت أو النبضة المحددة لها.
- يظهر معنا هذا النوع عندما يكون هناك ارتباط لتعليمتين متتاليتين بالبيانات أو الاسم (أسماء السجلات).
- يتم تصنيف هذا النوع لثلاث حالات.

1. RAW (Read after Write)

تعليمة تحتاج القراءة من سجل ولكن التعليمة التي قبلها تكتب على نفس السجل
ونعلم أن الكتابة بآخر مرحلة، لذلك ستقرأ خطأ إذا لم تتم التعليمة التي قبلها الكتابة عليه.
- يصنف هذا النوع كـ true data dependence

2. WAW (Write after Write)

تعليمتين متتاليتين الأولى تكتب على معامل ما والتي تليها تكتب عليه أيضاً
وهذا يعطي ترتيب خاطئ للمعلومات.
يحدث هذا النوع في الـ Pipelines التي تحتوي أكثر من مرحلة كتابة خلال تسلسل مراحل تنفيذ التعليمة
لكن هذه الحالة لا تحدث في معالجات MIPS لأن مرحلة الكتابة هي مرحلة واحدة فقط وهي الأخيرة.
- يصنف هذا النوع كـ output dependence أو not true dependence

3. WAR (Write after Read)

مثلاً (j) تعليمة تحاول الكتابة على سجل قبل أن تقرأه التعليمة التي قبلها (i)
ولكن هذا لا يحدث في معالجات MIPS أيضاً لأن مرحلة القراءة تتم في الـID أما مرحلة الكتابة هي آخر مرحلة.
- يصنف هذا النوع كـ Anti – dependence

أما بالنسبة للنوع RAR (Read After Read) فهو لا يسبب مشاكل ولا يشكل Hazards.

Last Dependent

3. Control Dependent

- يحدد الـ Control dependence ترتيب تنفيذ التعليمة (i) مع مراعاة تعليمات التفرع
- هذا ما يجعل التعليمة (i) تنفذ في الترتيب الصحيح.
- **القيود على Control dependence :**

1. لا يمكن نقل تعليمة ما يعتمد تنفيذها على تعليمة Branch و وضعها قبلها لأن تنفيذها يصبح غير مقيد بتعليمة التفرع.
2. لا يمكن نقل تعليمة لا يعتمد تنفيذها على تعليمة Branch و وضعها بعدها لأنه سيصبح تنفيذها مقيد بتعليمة التفرع.

- **تبعية التحكم و إعادة الترتيب (Control Dependent & Reordering) :**
  - لكي نحافظ على ترتيب واحد لتعليمات البرنامج التي ستنفذ يجب أن تكون الـ control dependence محفوظة أيضاً
  - أحياناً تتم مخالفة الـ control dependence في تنفيذ تعليمات لا يجب تنفيذها يمكننا عمل هذه المخالفة ولكن بشرط ألا يؤثر هذا على صحة البرنامج.
  - هناك نقطتان أساسيتان ليسير البرنامج بشكل صحيح مع المحافظة على كل من: data dependences و control dependences .

والنقطتان هما:

1. Exception Behavior : أي عملية إعادة ترتيب لتعليمات البرنامج يجب ألا تسبب أي (exception) جديد في البرنامج
2. Data Flow : إن تعليمات التفرع تجعل البيانات تسير بشكل عملي (أي في أي اتجاه) فلا يكفي فقط الحفاظ على الـ data dependence وإنما ترتيب البرنامج هو الذي يحدده أي جزء سوف يسلم البيانات للتعليمة.

Examples ..

Examples: Program Correctness

1) مثال عن مشكلة الاستثناءات (Exception)

```
DADDU R2, R3, R4
BEQZ  R2, L1
LW    R1, O(R2)
L1: . . .
```

كما نلاحظ فإنه يفحص شرط توافق الـ R2 مع الصفر وذلك قبل أن تتم الكتابة
على R2، ولو غيرنا الترتيب سيغير ذلك النتيجة لأنها Data Dependent
فلو غيرنا الترتيب ماذا لو كانت R2 = 0 ونريد الوصول للعنوان (0) والتعديل عليه
وهذا غير صحيح لأن العناوين لدينا تحتوي على نظام التشغيل.
وإن تعليمتي التفرع والتحميل أيضا لا يمكن التبديل بينهما لأنهما control dependent

والحل هنا هو: Hardware Speculation وهذا يرمي استثناء memory .

2) مثال عن تدفق المعطيات (Data Flow)

```
DADDU  R1, R2, R3
BEQZ   R4, L
DSUBU  R1, R5, R6
L: . . .
OR     R7, R1, R8
```

نستطيع الحفاظ على الـ Data dependent :
بالمحافظة على تسلسل التعليمتين: DADDU و DSUBU
ولكن جمع هذا التسلسل ونقله قبل أو بعد التفرع ليس حل
وذلك بسبب وجود control dependent وبسبب استخدام R1 في تعليمة OR
وتغيير ترتيبهم يؤدي لخرج خاطئ.

ملاحظة: إذا وجد Exception أو تغيير في الـ control statement لا نعدل على الترتيب
ولكن إذا لم يؤثر ذلك على البرنامج فيمكن عمل reorder للتعليمات.

3) مثال عن إعادة الترتيب (Reordering)

```
DADDU   R1, R2, R3
BEQZ    R12, skip
DSUBU   R4, R5, R6
DADDU   R5, R4, R9
skip: OR R7, R8, R9
```

لنحدد فيما إذا كان مخالفة الـ control dependence يؤثر على الاستثناءات وتدفق المعطيات أم لا (كما رأينا في المثالين السابقين) :

نلاحظ أن السجل R4 لم يتم استخدامه في تعليمة التفرع أو في الـ skip

لذلك ماذا لو تحقق الشرط ولم يتم تنفيذ هاتين التعليمتين.

لذلك نستطيع نقلهما إلى قبل التفرع، وذلك حتى لو تحقق شرط التفرع

سيتم تنفيذ التعليمات المرتبطة بـ R4

لكن ليس له فائدة في التفرع و لا يتأثر الـ data control dependences بهذا التغير .

ملاحظة: إن أي قيمة يتم استخدامها في تعليمات تالية تسمى Liveness

أما القيم التي لا تستخدم تسمى dead.

- فرضاً تعليمة التفرع تحقق شروطها، فإنه بعد إعادة ترتيب البرنامج فستكون تعليمة DSUBU قد نفذت ولم تؤثر على أداء البرنامج ولم يتم استخدامها
- إن هذا النوع من جدولة البرنامج (إعادة ترتيبه) هي نوع من Speculation وتسمى Software speculation بينما الـ Compiler يكون مسؤول عن خرج عملية التفرع وفي هذه الحالة غالباً يتوقع عدم تحقق شرط التفرع.

<THE END>