Pipelining

درسنا في المحاضرة السابقة:

- تصميم وحدة التحكم CU
- والآلات منتهية الحالات FSM
- ورأينا تصميم معالجات MIPS

في هذه المحاضرة سنتحدث عن أهم التقنيات المستخدمة في تحسين أداء المعالج والتي تدعى الـ Pipelining

المحاضرة مليئة بالمخططات التي ستسهل عليك الفهم (Don't worry)

مقدمة : قلنا سابقا أنه :

- إذا كان المعالج Multi Cycle فإن التعليمة الواحدة تمر بخمس مراحل .

  وكانت أطول تعليمة هي تعليمة ال load (والتي تأخذ 5 نبضات CPI=5)

- لنفكر، لو كنا نعمل بـ Single Cycle ونريد تنفيذ تعليمتين متتاليتين فإننا نحتاج لكل تعليمة نبضة (لأن CPI = 1)

  أي أننا نحتاج نبضتين لتنفيذ تعليمتين، ولكن إذا انتقلنا إلى Multi Cycle فإنه لتنفيذ هاتين التعليمتين المتتاليتين نحتاج 10 نبضات وهو زمن كبير نسبياً

- فكان الحل هو أن يكون في تنفيذ التعليمات نوع من التوازي، وهو التراكب أي أنه عندما يتم تنفيذ مرحلة من تعليمة ما فإنه في نفس النبضة يتم تنفيذ مرحلة مختلفة من تعليمة أخرى

ILP: Instruction Level Parallelism

- نعلم أنه في حالتي Single cycle و Multi cycle فإنه في المسار الذي تمر به التعليمة لا يمكن أن تمر ومعها تعليمة أخرى في نفس الزمن أي أنه يتم تنفيذ تعليمة واحدة في وحدة الزمن وهذا يخفض من الأداء

- لتحسين الأداء يجب تنفيذ تعليمات متعددة في واحدة الزمن، ويتم ذلك عن طريق:

- Parallelism (التوازي): جلب عدة تعليمات معاً خلال كل دورة (نبضة).
- Pipelining (التوارد): وهو التراكب لتحسين الـMulti cycle يتم فيها جلب تعليمة واحدة في كل نبضة.

- ويسمى هذا التراكب (التداخل) بين التعليمات بـ (التوازي على مستوى التعليمة ILP)

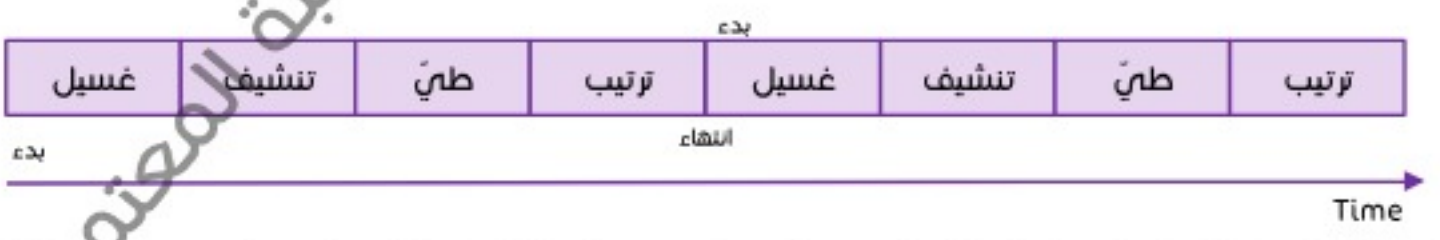
- ليس الهدف تقصير الزمن لتنفيذ المهمة الواحدة وإنما زيادة عدد التعليمات المنتهية في واحدة الزمن عن طريق هذا التداخل والذي يعطي إنتاجية (Throughput) أفضل.

مثال 1: Team assembly

مثال من الحياة العملية يوضح عملية التراكب:

- كما نعلم أن غسيل الملابس كعملية كاملة تتم على عدة مراحل:

- فرضاً أريد الغسل مرتين متتاليتين فسيكون الأداء كالتالي:



- كما نرى أننا احتجنا إلى ثماني فترات للإنهاء مرتيّ غسيل، حيث أن الدفعة الثانية لم تبدأ حتى تنتهي الأولى.

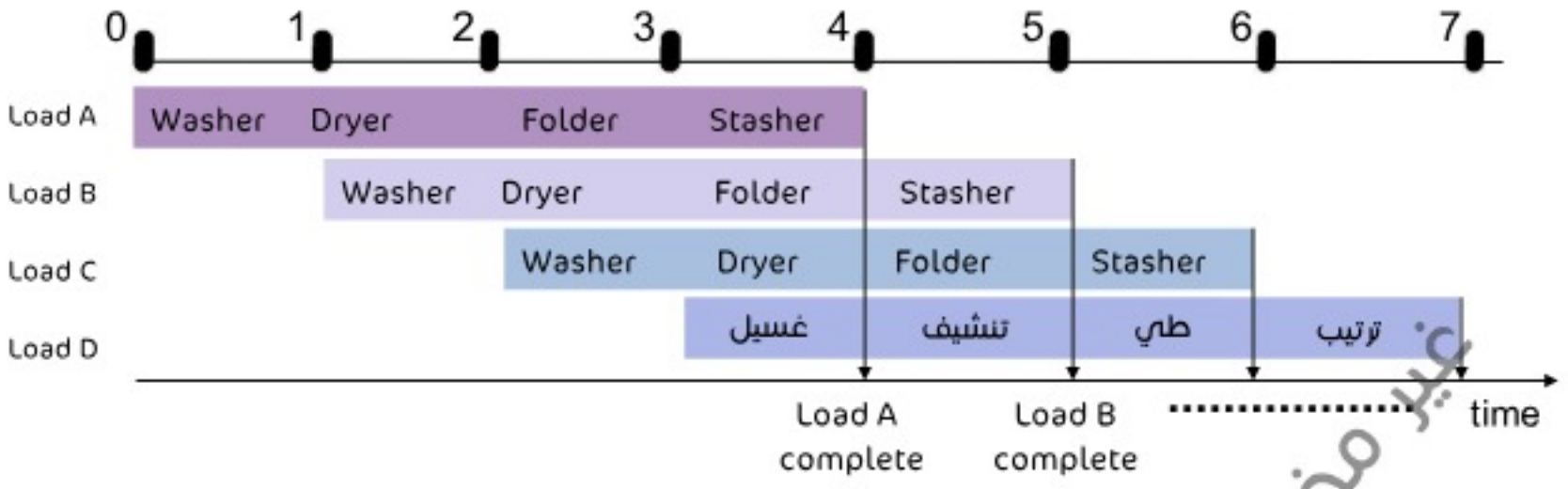
- الوقت المطلوب لإتمام عملية ما {مثلاً A} = (n) (عدد المراحل الفرعية المقسمة للعملية)

حيث n : وحدة الزمن التي تنجز فيها مرحلة واحدة من العملية

- Throughput (الإنتاجية) = (1/n) "عملية في واحدة الزمن"

- حيث أن الإنتاجية تعبر عن كم العمليات المنجزة خلال واحدة الزمن

مثال 2 : Assembly line



- أما هنا فكما نرى في المخطط أعلاه، عندما تنتهي مرحلة من العملية A تبدأ المرحلة المشابهة لها من العملية B، فعندما بدأت العملية A في اللحظة 1 بدأت بعدها مباشرة B في اللحظة 2 وC في اللحظة 3 وهكذا
- نتيجة لهذا التراكب ولأن العملية الكاملة تستغرق أربع مراحل زمنية، انتهت العملية الأولى A في اللحظة الزمنية 4 وانتهت B في اللحظة 5 لأنها بدأت بعد A بمرحلة واحدة

- (عدد العمليات الكاملة التي تمت في زمن T) = T − n + 1

حيث n الزمن المستغرق لإتمام العملية الأولى وهو ما يعرف ب (latency)

- Throughput (الإنتاجية) = $(1 - ((n-1))/T)$

تذكرة: إن مقدار التحسين يساوي:

$$Speedup = \frac{Original\ Execution}{New\ Execution}$$

وعلى مستوى الإنتاجية يكون:

$$Speedup = \frac{Throughput\ \text{بعد التراكب}\ (assembly\ line)}{Throughput\ \text{التسلسلي}\ (team\ assembly)}$$

$$= \frac{1 - \frac{(n-1)}{T}}{\frac{1}{n}} = n - \frac{(n-1)}{T} \rightarrow n$$

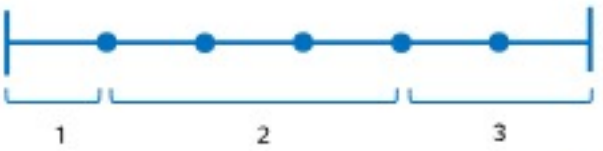
نرى أن مقدار التحسين هو n وهذا يعني أنه عند كل لحظة زمنية تكون قد تمت عملية كاملة.

Features of Pipelined Processor

- تعمل جميع الوحدات الوظيفية بشكل مستقل، حيث أن هناك مهام تعمل بنفس الوقت مستخدمة مصادر مختلفة فإذا كان هناك ترابط سيحدث خلل
- لا علاقة للتوارد Pipelining بتقليل زمن تنفيذ المهمة الواحدة وإنما لزيادة الإنتاجية الكلية
- التحسين المحتمل = عدد مراحل التوارد (Potential speedup = Number of pipe stages)
- يحدد معدل التـــوارد Pipelining بـــ:

1. مـرحلة التوارد الأبطأ (الأكثر زمناً):

مثلاً في المخطط المجاور هناك عملية ما تتم على ثلاث مراحل، فيتم تحديد معدل التوارد حسب المرحلة 2 التي تستغرق ثلاث فترات زمنية
وذلك لأنه لو حددناه بمرحلة أخرى أقل منها تستغرق فترة واحدة
فإن هذه المرحلة لن تكتمل
(لأن المراحل الأقصر زمناً ستكون مجبرة على انتظار المرحلة الأطول)



2. الـوقت اللازم لتعبئة pipeline خط التوارد لعملية من عدة مراحل:

الوقت المستغرق والوقت الضائع في النبضات الأولى يقللان من تحسين الإنتاجية

3. أطـوال مراحل التوارد غير المتوازنة:

إن اختلاف الوقت المستغرق لكل عملية (على مستوى الحاسوب عدد نبضات كل مرحلة من تعليمة ما) يقلل من السرعة
مثلاً إذا استغرقت الـwasher وقتاً أطول من الـdryer فيجب على المجفف الانتظار

4. التوقف بسبب الترابط (التضـارب):

لأن جميع الواحدات الوظيفية يجب أن تعمل بشكل مستقل.

↙ لتطبيق التوارد Pipelining يتطلب ذلك:

1. تقسيم ممر البيانات (Divide datapath)

- تقسيم ممر البيانات **datapath** إلى مهام متساوية الطول تقريباً، وتتطلب مصادر غير متداخلة ليتم تنفيذها بشكل تسلسلي

2. إدراج السجلات عند حدود المهمة (Insert registers at task boundaries)

- إضافة سجلات (ذواكر مؤقتة) إلى حواف نهاية كل مهمة في ممر المعطيات، حيث أن السجلات تحتفظ بخرج مهمة ما لتعيده كدخل للمهمة التي تليها

تذكرة : في التعليمات الرياضية التي تعتمد على نواتج بعضها, فرضاً استخدمنا ضمن الدارة بوابة and أعطتنا خرج عملية ما و نريد عمل عملية على الناتج فهنا يأتي دور السجلات التي تحتفظ بناتج الخرج وتعطيه إلى المرحلة المرتبطة بها التي تليها وهكذا..

3. مزامنة المهام مع الزمن (Synchronize tasks with a clock)

- **مزامنة كل المهام على طول نبضة واحدة، فتصميم التوارد Pipelining تبعاً للمرحلة الأطول**

مثلاً : في المخطط التالي نصمم التوارد تبعاً للمرحلة 2
بحيث يكون طول كل نبضة هو ثلاث ثواني

4. تقسيم كل تعليمة (Break each instruction down)

قسم التعليمة لعدة مهام بحيث يتم تنفيذها بنسق متناوب.

- ما المعالجات الأمثل للتوارد!!

بالتأكيد معالجات MIPS

MIPS pipelining

1- لمعالجات MIPS أنماط تعليمات قليلة، حيث أن حقل السجل المصدر (rs)يقع في نفس المكان لكل التعليمات
(J-Type , I-Type , R-Type)

- هذا التناظر يعني أنه في المرحلة الثانية يمكن البدء بقراءة ملف السجل بنفس الوقت الذي يتم فيه تحديد نوع التعليمة التي تم جلبها

- لو لم تكن متناظرة لكان علينا قسم هذه المرحلة إلى اثنتين، فيصبح لدينا 6 مراحل.

2- كل تعليمات MIPS لها نفس الطول، وذلك يسهل جلب التعليمة في المرحلة الأولى وفك تشفيرها في المرحلة الثانية.

3- نستخدم معاملات الذاكرة فقط في تعليمات التحميل و التخزين في MIPS، وهذا يعني أنه يمكننا أن نستخدم مرحلة التنفيذ (execute) لحساب عنوان الذاكرة و الوصول إلى الذاكرة في المرحلة التالية.

4- المعاملات تكون مرتبة في الذاكرة، فلا يوجد داعي لأكثر من عملية نقل للبيانات التي تتطلب وصولين في الذاكرة، أي يمكن نقل البيانات المطلوبة بين المعالج والذاكرة في مرحلة واحدة

Pipelining a Single-Cycle Datapath

الجدول التالي مثال يوضح العديد من التعليمات ومراحل تنفيذها في الـ Single cycle

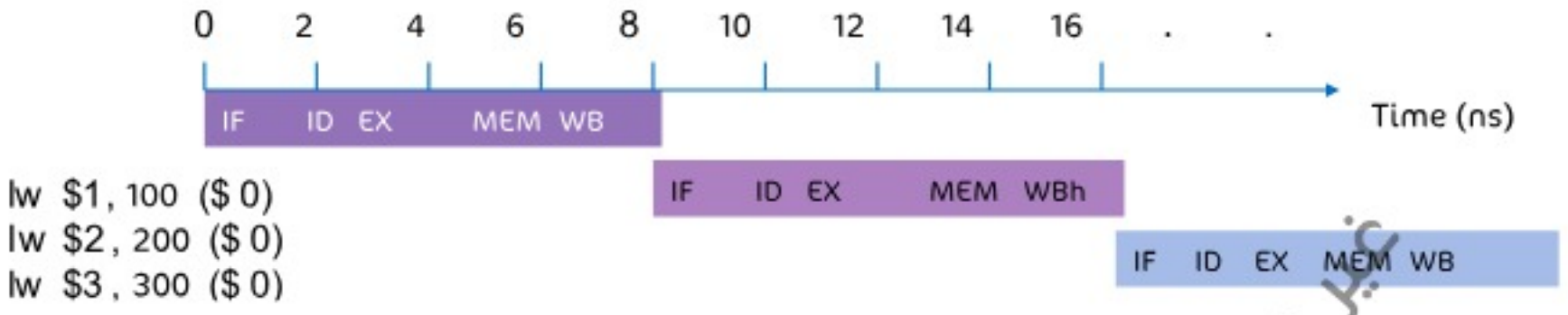
Instruction class	Instr. fetch (IF)	Instr. Decode (also reg. file read) (ID)	Execution (ALU Operation) (EX)	Data access (MEM)	Write Back (Reg. file write) (WB)	Total time
lw	2ns	1ns	2ns	2ns	1ns	8ns
sw	2ns	1ns	2ns	2ns		8ns
R-format add, sub, and, or, slt	2ns	1ns	2ns		1ns	8ns
B-format, beq	2ns	1ns	2ns			8ns

ماذا تلاحظ ! :

- نلاحظ منه أن أطول تعليمة هي تعليمة الـ Load لأنها تمر بكل الأقسام
- تعليمة Store نلاحظ أن CPI = 4 , حيث أنها تخزن في الذاكرة فقط ولا تعيد التحميل لذلك آخر مرحلة لا تمر بها
- نلاحظ أن تعليمات R-type لا تمر بالذاكرة
- تعليمات التفرع والقفز هي أقصر التعليمات وCPI = 3
- على الرغم من أن كل التعليمات عدا Load تنتهي بأقل من 8 نانو ثانية وإن أطول تعليمة تستغرق 8 , إلا أننا نتعامل بـ Single cycle لذلك فإن كل التعليمات الباقية سيكون زمن نبضتها 8 نانو ثانية.

(دراسة تأثير الـ Pipelining على التنفيذ قبل - بعد)

Execution Time: Single-Cycle without Pipelining



لنفرض كما نرى في المخطط أنه لدينا ثلاث تعليمات (lw) Load متتالية.

- زمن كل منها هو Clock cycle time = 8ns
- اذاً زمن التعليمات الثلاث إذا تم تنفيذها بشكل متسلسل هو 24ns

هذا الزمن سيكون كبير نسبياً اذاً ما كان عدد التعليمات أكثر

Single cycle بعد التوارد Pipelining:

Pipelined Datapath "Single Cycle"

Instruction class	Instr. fetch (IF)	Instr. Decode (also reg. file read) (ID)	Execu-tion (ALU Opera-tion) (EX)	Data access (MEM)	Write Back (Reg. file write) (WB)	Total time
lw	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10 ns
sw	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10 ns
R-format: add, sub, and, or, slt	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10 ns
B-format: beq	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10 ns

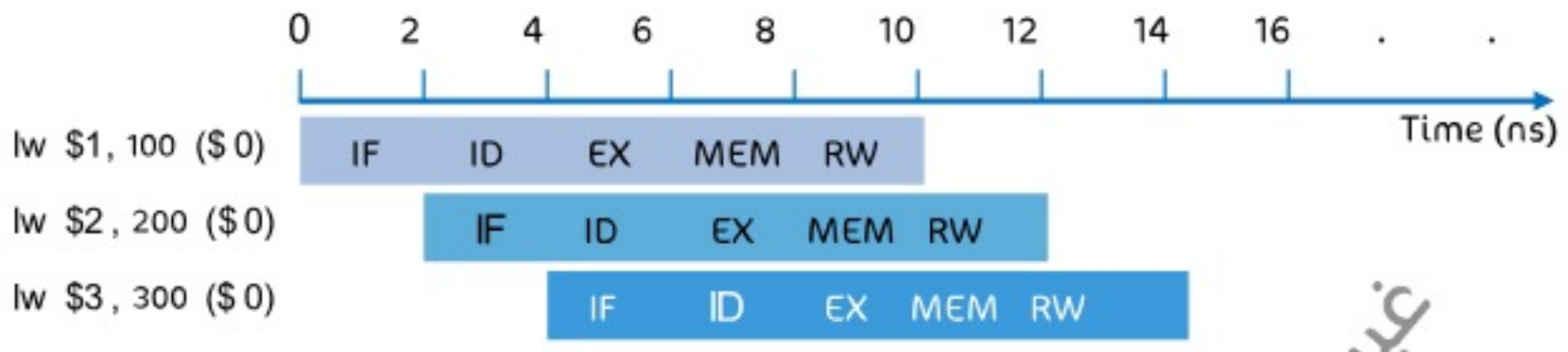
ماذا تلاحظ! :

- درسنا أننا في التوارد نعتمد الزمن الأطول، حيث نلاحظ في الجدول أن كل 1ns أصبحت 2ns
- الأماكن الفارغة أصبحت ممتلئة وذلك بسبب التوارد فإذا لم تكن التعليمة تعمل فهناك تعليمة أخرى تعمل
- نلاحظ أيضاً أن الزمن الكلي للتعليمة ازداد 10 ns = Total Time

- صحيح أن زمن تنفيذ التعليمة ازداد من 8 إلى 10 ولكن التحسين يتمثل في الإنتاجية الكلية

يجب مقارنة التنفيذ ليتضح الأمر.

Execution Time: Single-Cycle with Pipelining



- كما نلاحظ اذاً، صحيح أن زمن تنفيذ التعليمة ازداد بمقدار 2 ns،
- لكن الوقت اللازم لتنفيذ تعليمات الـ Load الثلاث هو 14 ns أي قللناه بمقدار 10 ns

نسبة التحسين :

$$\text{Performance ratio} = \frac{\text{Single cyble time}}{\text{Pipleine time}} = \frac{24}{14} = 1.7$$

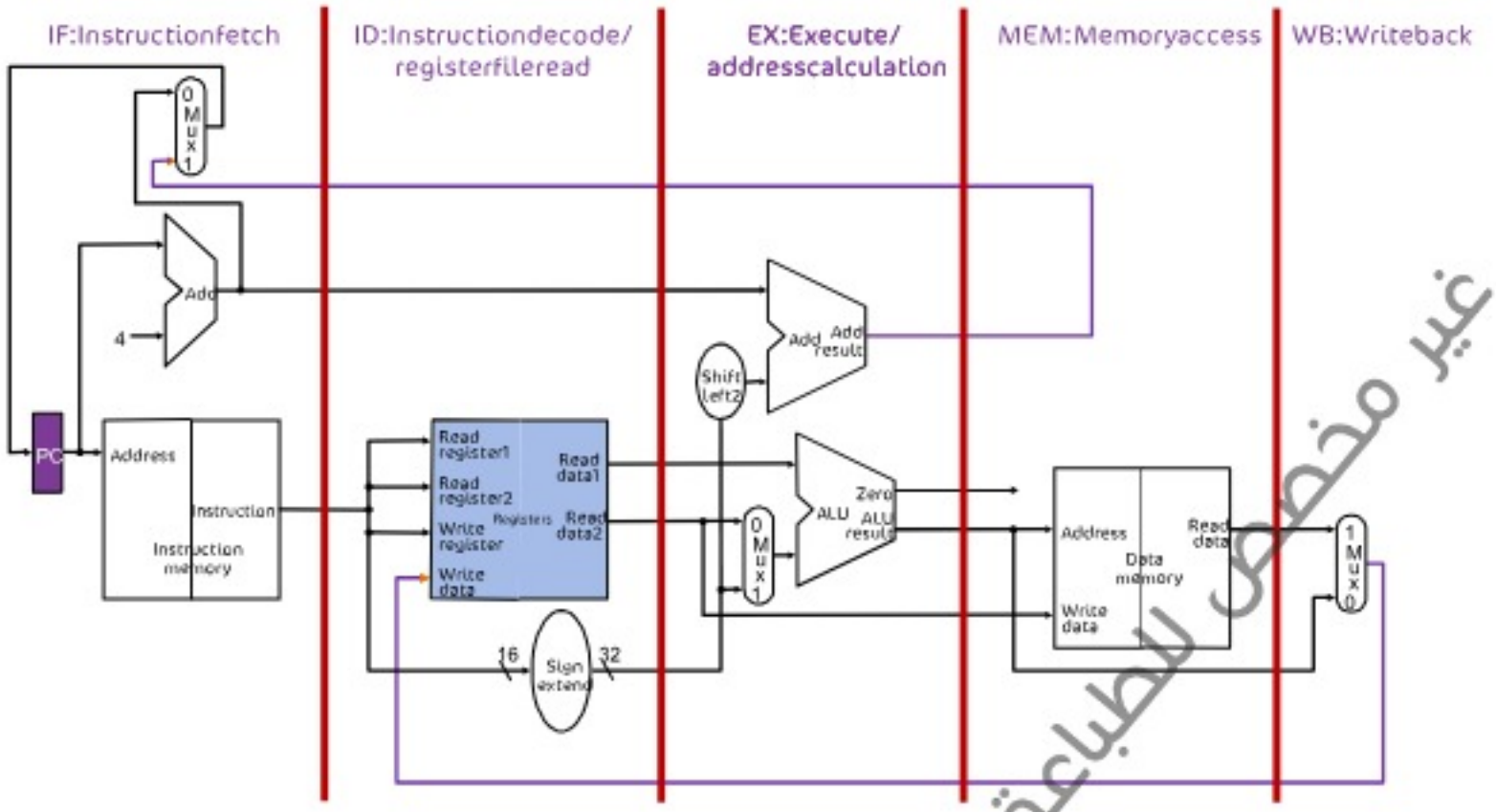
- النسبة المئوية للتحسين هي 70%

يعطى قانون الـ Speed up في الـ Pipeline كالتالي :

- $$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

لم نبدأ بعد..

Pipelined Single Cycle Processor Design



تم تقسيم الـ Hard ware لخمسة أقسام كما في المخطط أعلاه إلى:

(IF, ID , EX , MEM , WB) ونضع بين كل قسمين سجلات

1. مرحلة IF :

تحتاج الذاكرة الخاصة بالتعليمة وتزيد على الـ pc

2. مرحلة ID:

مرحلة ID إن معالج MIPS ينفذ الـDecode في الـHard ware

أي أن الخطوط موصولة دائماً للسجلات rs وrtg ويقرأ السجلات بنفس المرحلة، حتى لو كان هناك تعليمات لا تحتاج لقراءة السجلين لأن احداهما رقم فهو يقرأهما كليهما لكي يحافظ على التسلسل لكل التعليمات.

3. مرحلة EX:

تتم فيها الـ ALU.

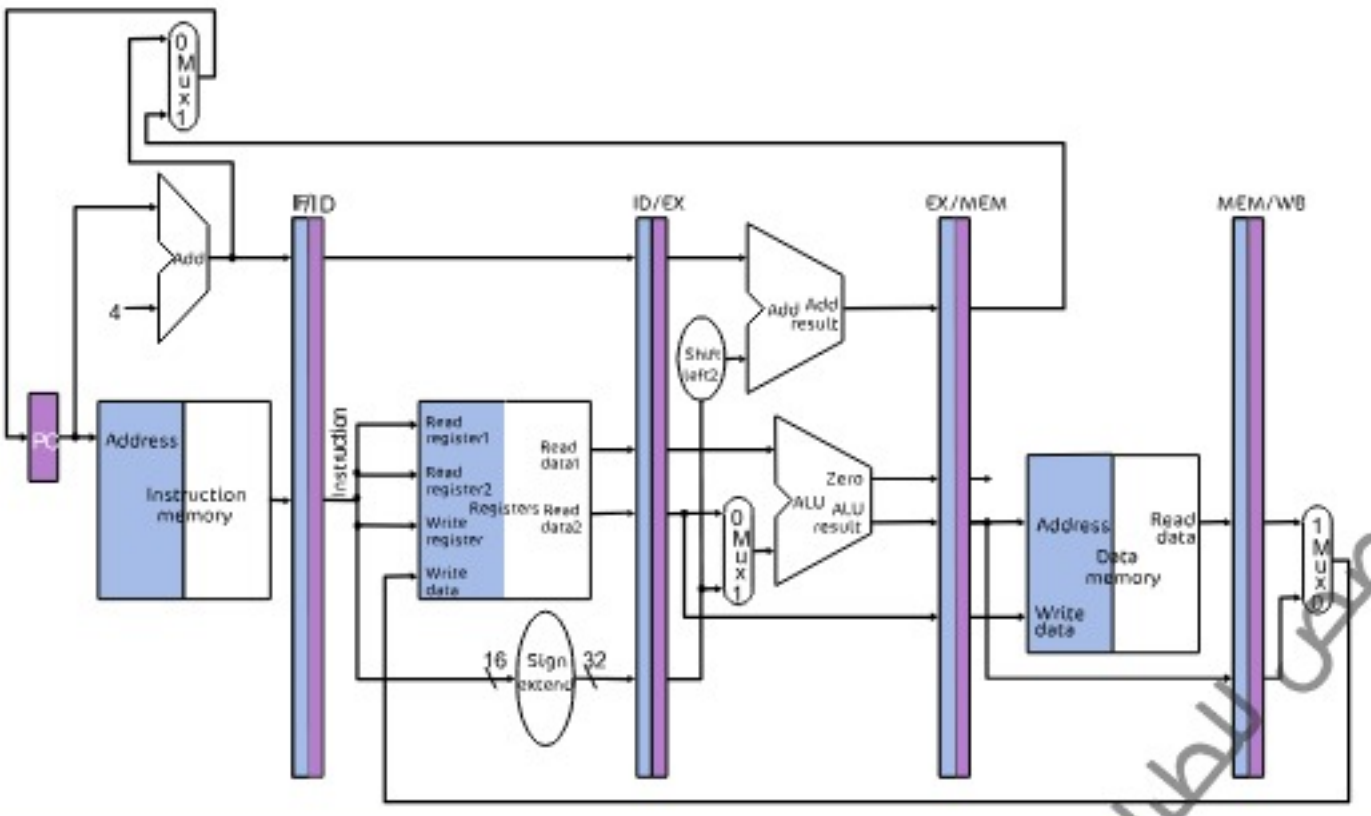
4. مرحلة MEM:

إما نقوم بـ save أو read.

5. مرحلة WB:

نكتب في سجل rd الناتج.

Pipelined Single Cycle Registers Adding

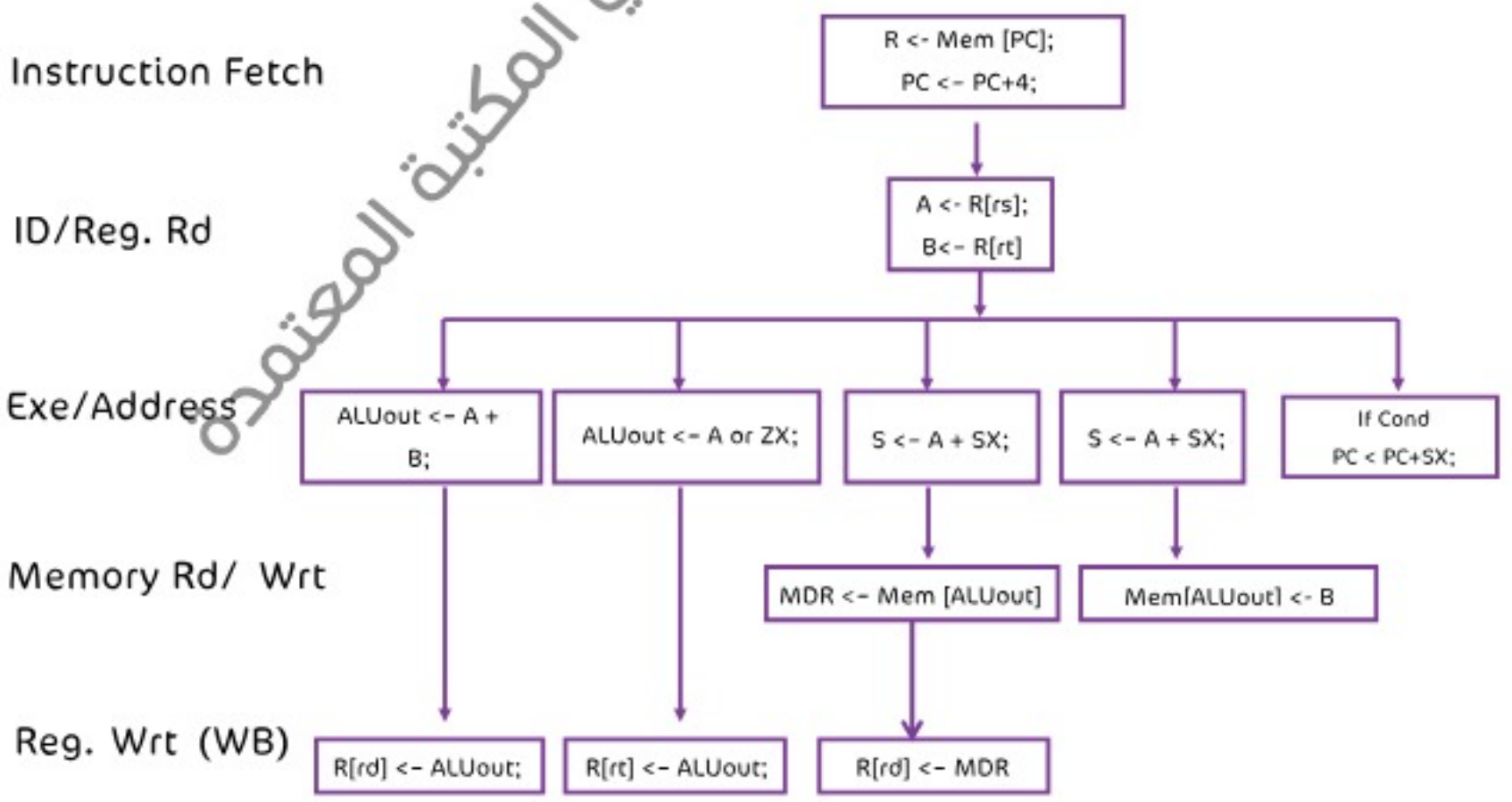


كما نرى المسجلات جزئين، جزء يخزن من المرحلة الأولى وجزء يعطي input للمرحلة التالية

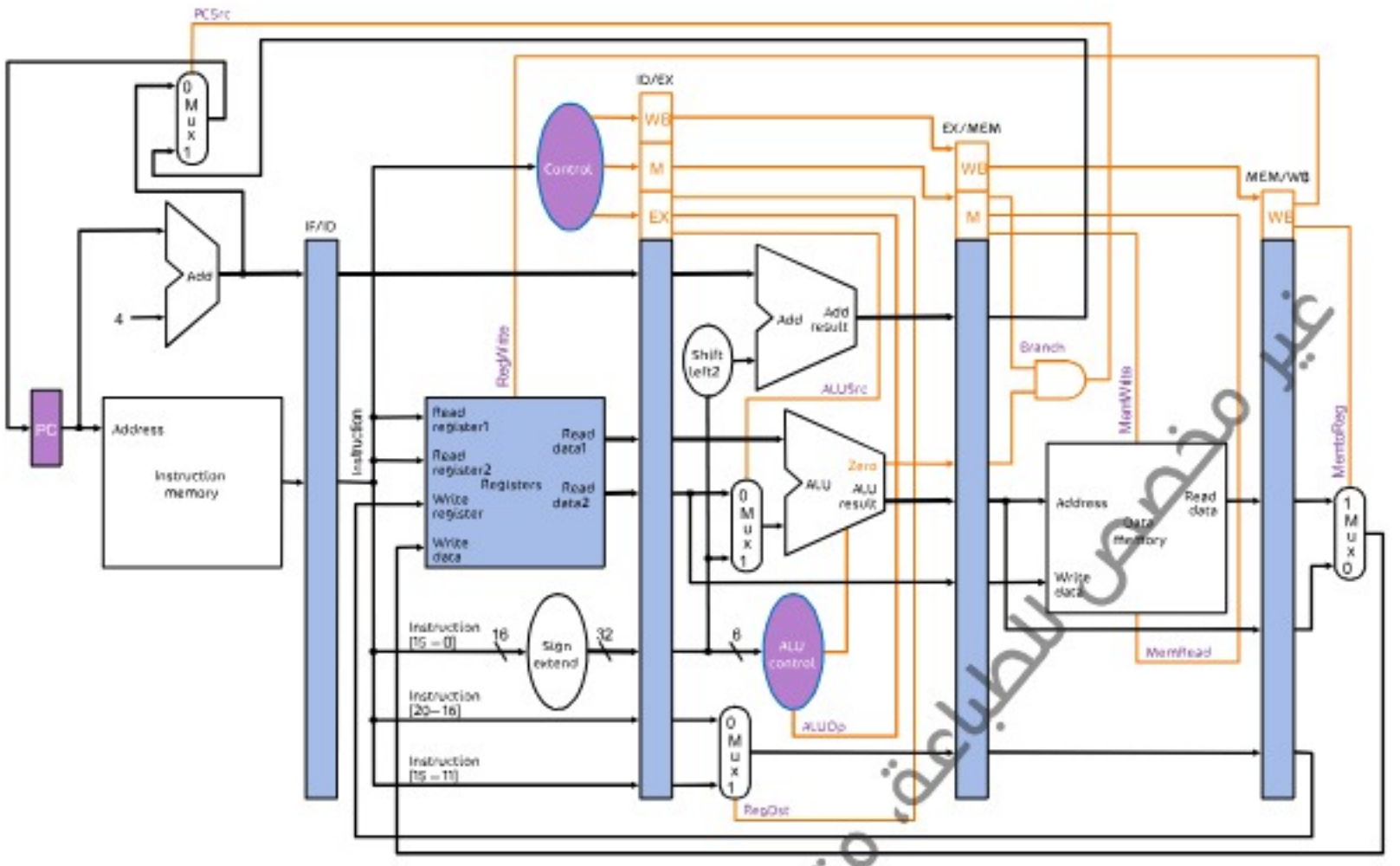
تمت تسميتهم بحسب المراحل فمثلا IF/ID أي هو المسجل الموجود بين مرحلة الجلب ومرحلة فك الترميز

كما نلاحظ File register ملون بلونين وذلك للتمييز أنه يمكن أن نقرأ منه ونكتب عليه ولكن ليس بنفس المرحلة أما إذا كان بنبضتين مختلفتين فيمكن ذلك

تصميم وحدة التحكم بتقنية التوارد، مخطط الحالات FSM:



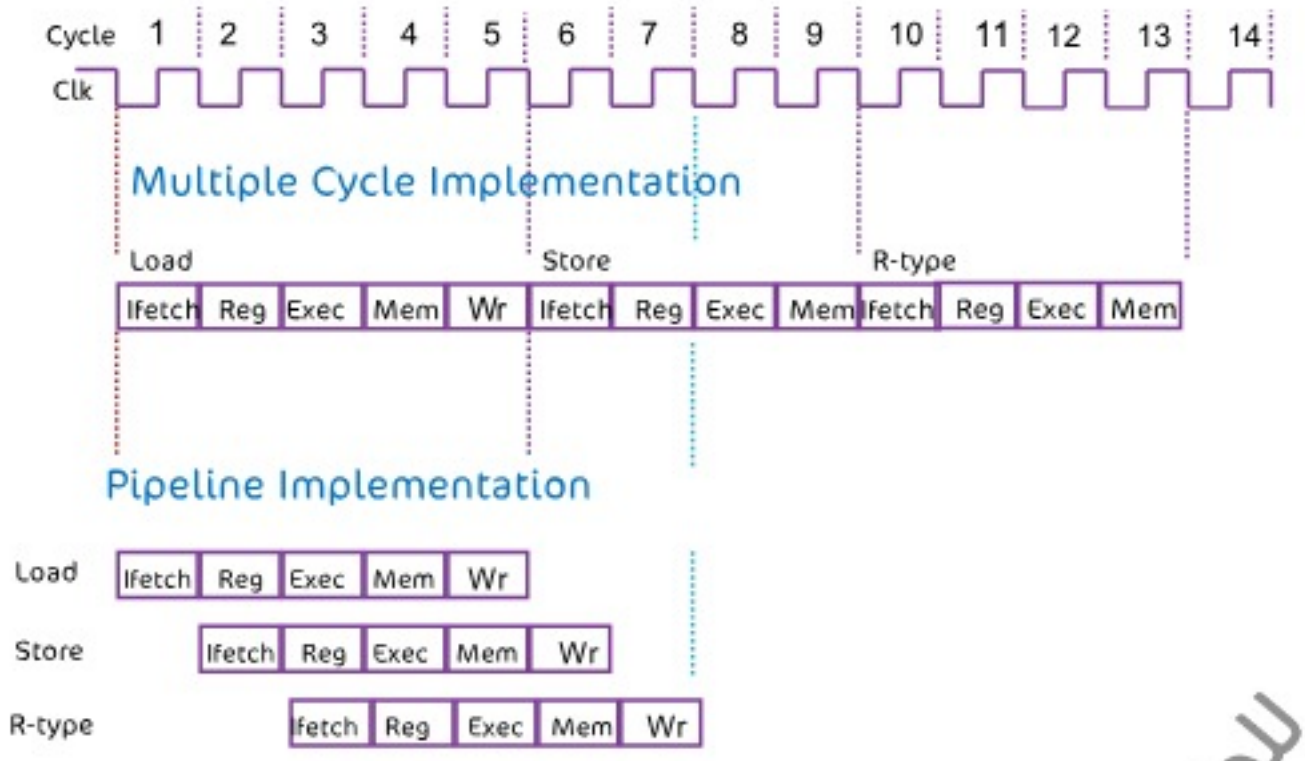
وبتنفيذ مخطط الحالات ينتج لدينا الشكل النهائي لوحدة التحكم بتقنية التوارد :



الجدول التالي يوضح ما تخزنه السجلات في كل مرحلة:

Register Name	Data Held
IF/ID	PC+4, Instruction Word (IW)
ID/EX	PC+4, R1 , R2, IW(0-15) sign ext. , IW(11-15), IW(16-20)
EX/MEM	PC+4, zero , ALUResult , R2,IW(11-15) OR IW(16-20)
MEM/WB	M[ALUResult] , ALUResult , IW(11-15)OR IW(16-20)

Multiple Cycle VS. Pipeline :



- في الـ Multi cycle فقط قسمنا النبضة لخمسة نبضات ونلاحظ أنه بقي الزمن طويل
- أما بعد تطبيق التوارد

مثال

زمن النبضة في آلة Single cycle هو 45 ns , وفي آلة Multi cycle و Pipelined هو 10 ns
ومتوسط CPI تبعاً لمزيج تعليمات داخل البرنامج في حالة Multi cycle هو 4.6
- ما هو زمن التنفيذ لكل نوع من الآلات ل 100 تعليمة:

الحل:

-Single Cycle Machine:

$$CPU_{Time} = IC \times CPI \times clock\ cycle\ time = 100 \times 1 \times 45(ns/cycle) = 4500\,ns$$

-Multi Cycle Machine:

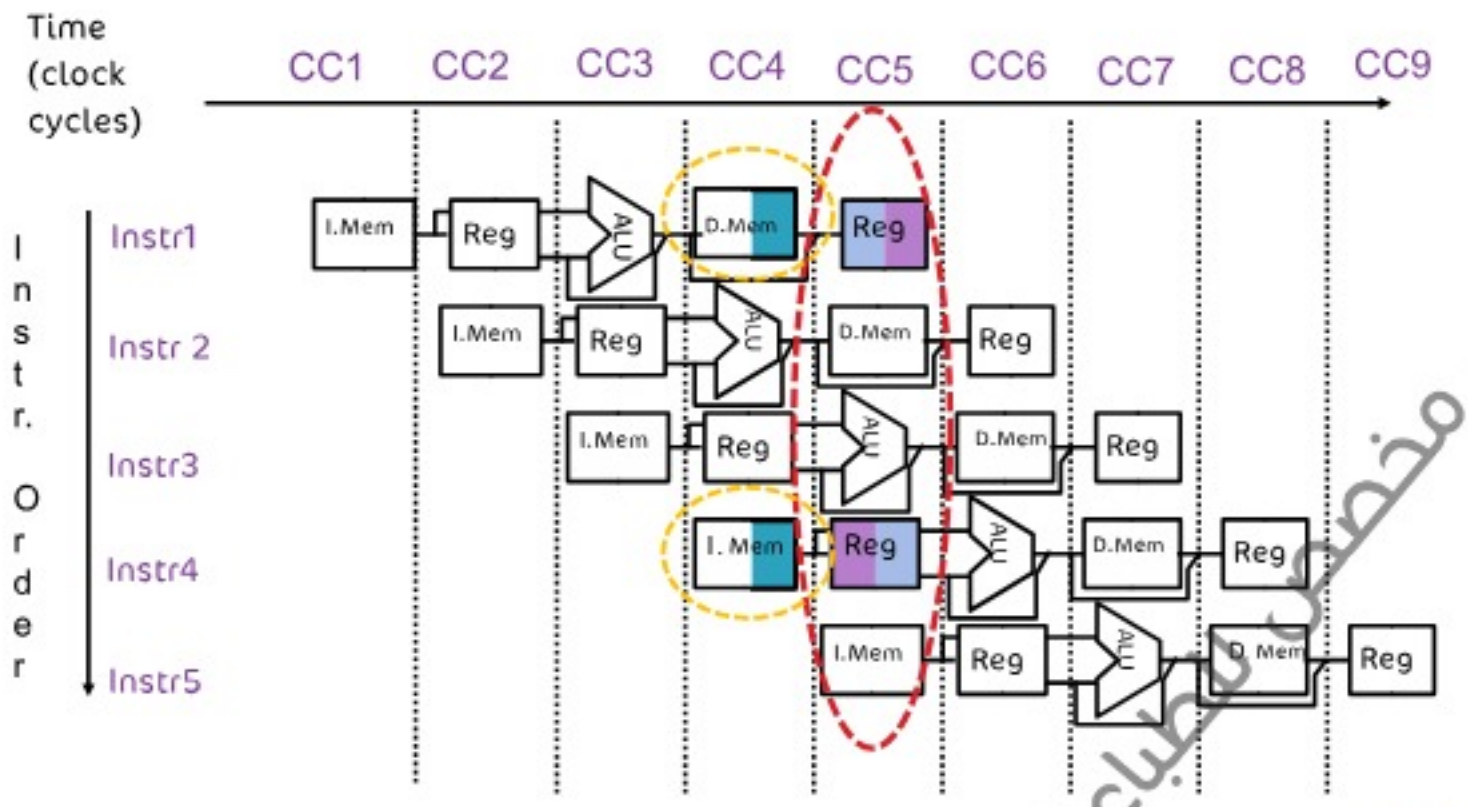
$$CPU_{Time} = 100 \times 4.6 \times 10(ns/cycle) = 4600\,ns$$

-Pipelined Machine:

$$CPU_{Time} = ((100 \times 1) + 4[cycle\ drain]) \times 10(ns/cycle) = 1040\,ns$$

دورات التعبئة (cycle drain): وهي تساوي (n-1) حيث n عدد المراحل المقسمة لها العملية
وبعد دورات التعبئة مع كل نبضة تنفذ تعليمة.
-هنا CPI = 1 لأنه بعد الدورة الرابعة أي بعد مرور دورات التعبئة مع كل نبضة ساعة تنفذ تعليمة

Graphical Representation - Single cycle



ماذا نلاحظ في المخطط ! :

- نلاحظ أن هناك عمليتان تتم على الذاكرة في النبضة الرابعة بنفس الوقت :
  ولكن لم يحصل تضارب لأنه عملية قراءة وعملية كتابة .
  وكذلك نفس الشيء على ملف السجلات في النبضة الخامسة .

لقد درسنا الـ Single cycle ولكن ماذا بشأن الـ Mutli cycle هل سيكون هناك تضارب بالذاكرة !

Pipeline Hazards

Hazard :

الخطر (أو الصعوبة) الذي سيواجه التوارد وهو حالة عندما لا تستطيع التعليمة التالية أن تنفذ في النبضة التالية للتعليمة السابقة

هناك ثلاثة أنواع للـ Hazard وهي : (1 .Structural Hazard 2 .Data Hazard 3 .Control Hazard)

مثال على التضارب، ليكن لدينا تسلسل التعليمات التالي:

A = B+C هنا تتم الكتابة على A بعد إيجاد الناتج في النبضة الخامسة

D = A+X هنا نحتاج A في النبضة الثانية قبل إيجاد قيمتها فهنا حصل تضارب

1. Structural Hazard (الخطر الهيكلي)

تعليمتين لا يمكن تنفيذهما بسبب تعارض المصادر.

مثلا:

لدينا عدة تعليمات تحتاج الوصول لذاكرة التعليمات مثلا الدورة الرابعة من تعليمة lw تتطلب وصول الذاكرة (memory read) .

وبنفس الوقت هناك التعليمة الرابعة فيها النبضة الأولى وهي إحضار التعليمة وتحتاج أيضا الوصول للذاكرة (memory read).

وهذا يسبب تعارض

الحلول :

1. تزويد الـ Hardware بمصادر مزدوجة في ممر المعطيات
2. يمكن أن تضيف وحدة التحكم أو الـ Compiler (نبضات NOP) للتأخير بين التعليمات

وهذه العملية تعرف بـ pipeline stall أو bubble

- هنا نلاحظ أن التوارد لم يعد مثالي , فيحدث تغيير بمعادلة الأداء (رأيناها بمثال سابق) :
- The pipelined CPI with stalls = Ideal CPI + Stall clock cycles per instruction

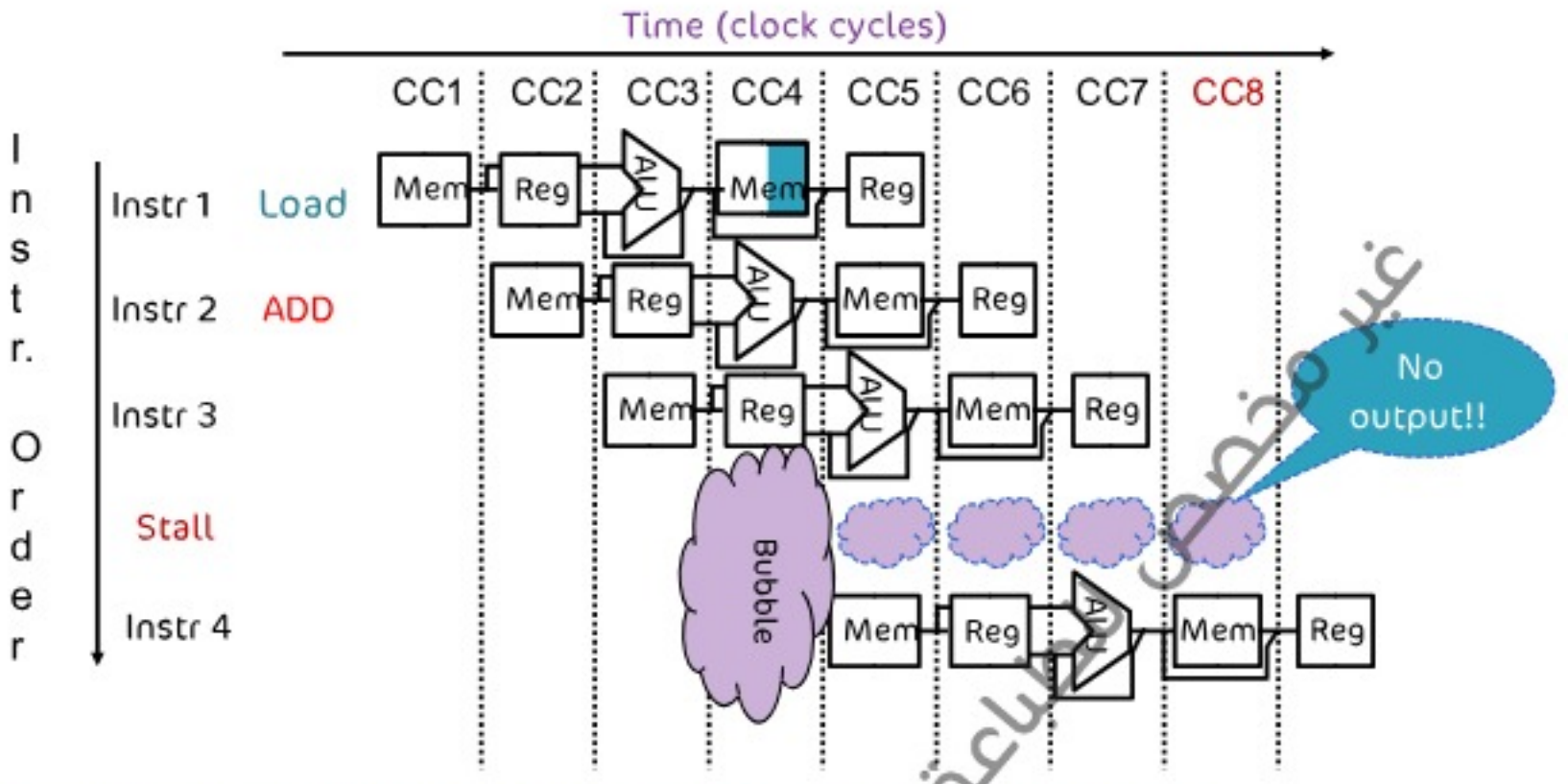
Structural Hazard Solutions

يمكن الحد أو التقليل من هذه التضاربات عن طريق استخدام:

1. Stall Operation: عملية إيقاف
   (ستؤثر على الأداء وتضعفه)
2. Additional Multiple Functional Units: وحدات وظيفية متعددة إضافية
   (مكلفة أكثر لأنها تحتاج قطع عتادية أكثر)

Still easy ...

1. Stalls Solution



One memory for Instruction and Data => hazard is solved by inserting a Stall (bubble)

- كما نلاحظ أنها Multi cycle في التعليمة الأولى (Instr 1) في النبضة الرابعة (CC4) يتم الكتابة على الذاكرة بنفس النبضة تريد التعليمة الرابعة الجلب من الذاكرة قبل الكتابة عليها
لذلك يحصل Stall لحين الكتابة على الذاكرة وتبدأ التعليمة الرابعة في النبضة الخامسة بدل الرابعة

مثال لتأثير هذا الحل على الأداء

لنفرض أن التعليمات المرجعية للبيانات تشكل 40% من عدة تعليمات أخرى، والمعالج فيه خطر بنيوي (structural hazard) وفيه معدل النبضة يساوي 1.05 من معدل النبضة للمعالج دون structural hazard ما هو معدل وقت تنفيذ كل تعليمة :

$$The\ Average\ Instruction\ time = CPI \times Clock\ Cycle\ Time$$
$$= (1 + 0.4 \times 1)) \times Clock\ Cycle\ Time_{ideal}) / (1.05)$$
$$= (1.4 / 1.05) \times Clock\ Cycle\ Time_{ideal}$$
$$= 1.3\ Clock\ Cycle\ Time_{ideal}$$

- نلاحظ: المعالج الذي لا يحتوي structural hazard أسرع بـ (1.3) مرة من المعالج الذي يحوي structural hazard

2. Adding Functional units Solution

مثال لفهم تأثير الحل الثاني على الأداء

- لدينا آلتين A وB:

- A (Dual ported memory)
- B (Single ported memory and pipelined) "1.05 times faster clock rate"
- Ideal CPI = 1 for both

بفرض أن هناك تعليمات Stores/Loads تشكل40% من التعليمات المنفذة يكون:

$$Speedup\ (A) = \frac{Pipeline\ Depth}{(1+0) \times \frac{clockunpipe}{clockpipe}} = Pipeline\ Depth\ .$$

وهنا لا يوجد توارد

- $Pipeline\ Depth$: عدد المراحل

$$Speedup\ (B) = \frac{Pipeline\ Depth}{(1+0.4\times 1) \times \frac{clockunpipe}{(clockunpipe\ /\ 1.05)}}$$

$$= (Pipeline\ Depth\ )/1.4 \times 1.05 = 0.75 \times Pipeline\ Depth\ .$$

$$SpeedUp(A)\ /\ SpeedUp(B) = \frac{Pipeline\ Depth}{0.75 \times Pipeline\ Depth} = 1.33$$

- A أسرع من B بـ 1.33 مرة

- معالجة الخطر بطريقة Adding Functional:

1. Memory structural Hazard

- تتم ازالتها باستخدام وحدتي ذاكرة مؤقتة (cache memory):
  - Instruction memory
  - Data memory

2. Register File structural hazard

- مدخلي كتابة في ملف السجلات يسمح للمرحلة 4 و5 بأن يعملا بتوارد.
- الوصول لملف السجلات وأخذ معلومة منه أسرع من تنفيذ الـALU.
- حيث تكون الكتابة على ملف السجل في النصف الأول من النبضة وتتم القراءة في النصف الثاني.
- يصبح بإمكاننا القراءة والكتابة في ملف السجلات (كما في تعليمات MIPS).

2. Data Hazard (خطر البيانات)

و هو يعني أن تعليمة ما لا يمكن أن يتم تنفيذها بسبب ارتباطها بتعليمة قبلها ستأخذ منها معلومات لتنفذ، وبسبب التوارد و هي حاليا غير متاحة بعد.

مثال بسيط

```
add $s0, $t0, $t1
sub $t2, $s0, $t3 # needs $s0
```

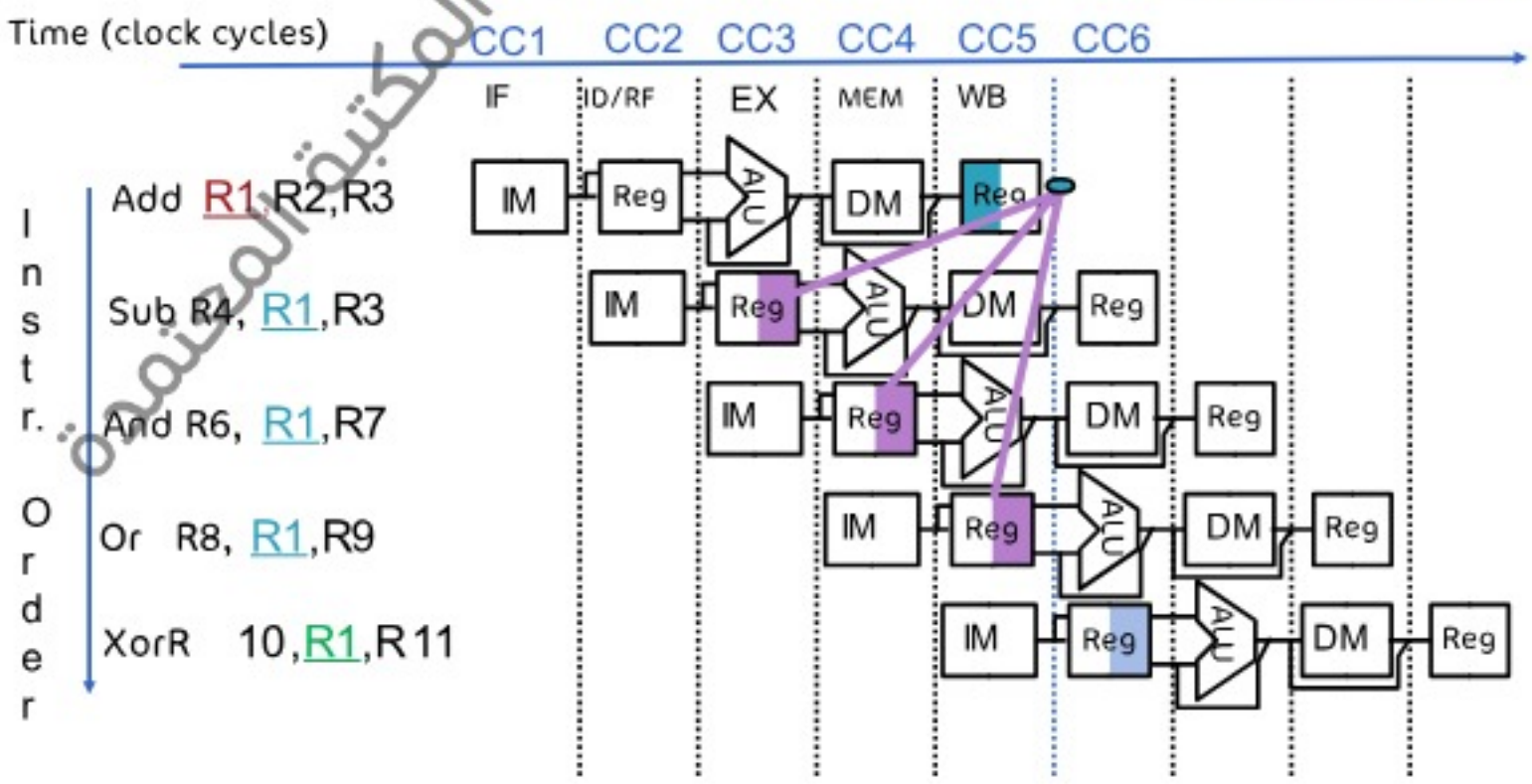
- كما ترى تعليمة الطرح تحتاج السجل $s0

مثال ثان للتوضيح

```
1: Add R1, R2, R3
2: Sub R4, R1, R3
3: And R6, R1, R7
4: Or R8, R1, R9
5: Xor R10, R1, R11
```

One of the solutions

توضيح المثال الثاني :

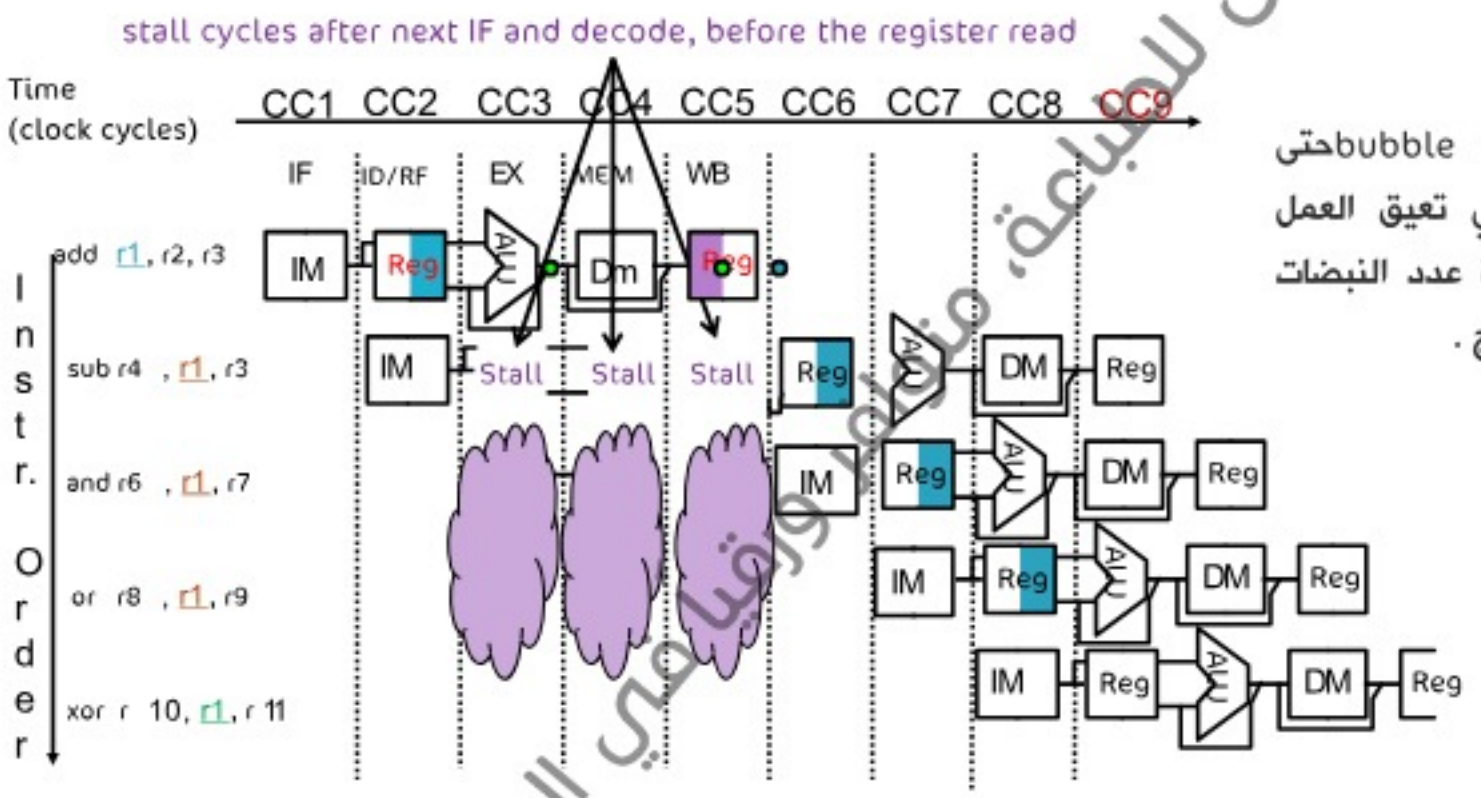


Data Hazard due to Dependencies backwards in time are HAZARDS

- كما لاحظنا أعلاه أن المعلومات التي نحتاجها للقيام بالتعليمات 4,3,2 لا يمكن الحصول عليها
  حتى نهاية النبضة الخامسة
  ونحتاجها في النبضات التي تسبقها

- لحل هذه المشكلة في Data Hazard هناك أحد الحلين:
- Stall solution
- Forwarding solution

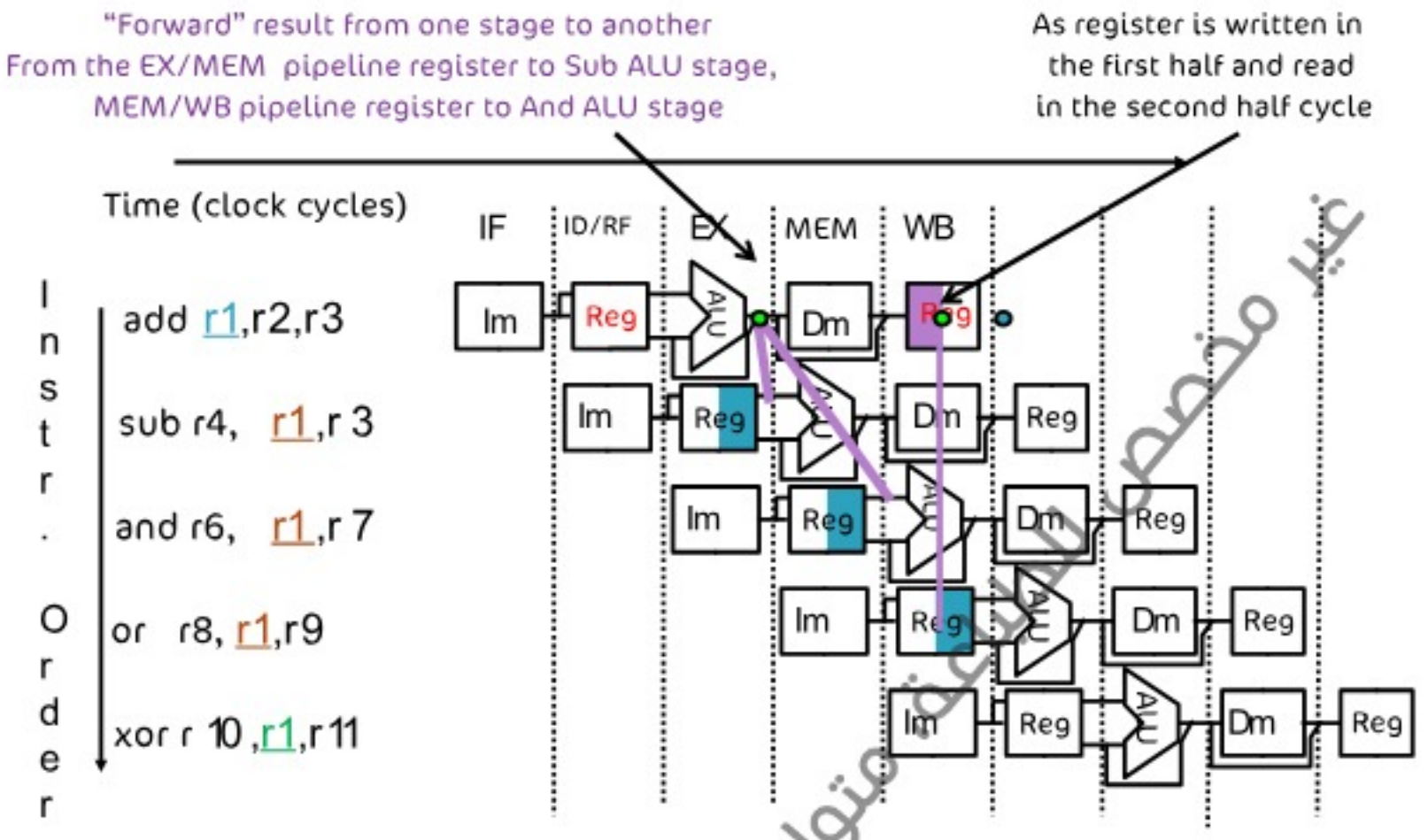
1. Stalls Solution



يتم عمل stall أو bubble حتى نهاية النبضة التي تعيق العمل وذلك يؤدي لزيادة عدد النبضات لحين انتهاء البرنامج .

Keep going ...

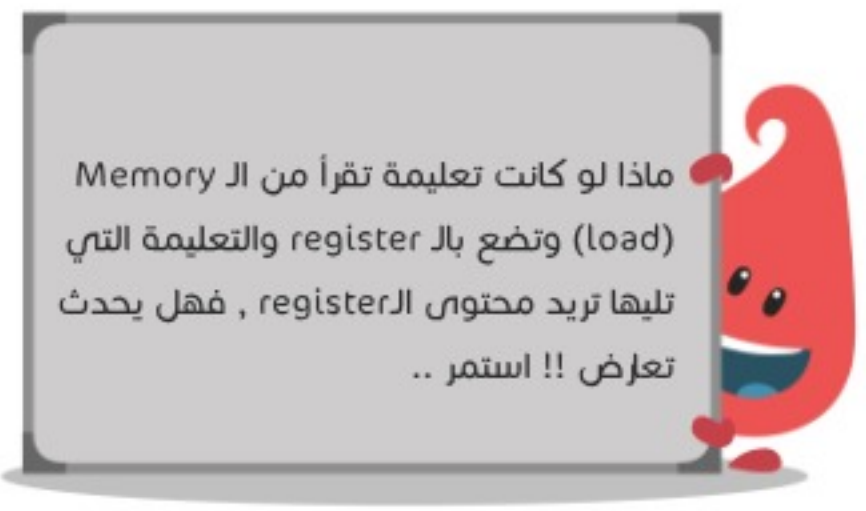
2. Forwarding Solution or "By passing"



- نلاحظ من خرج الـ ALU يمكن أن نأخذ الناتج حتى قبل أن يخزن في السجلات
- لذلك نجد وصلة من خرج الـALU الى مدخلها في كل من التعليمات التي تليها وتحتاجها

وهذا ما يعرف بالـ Forwarding

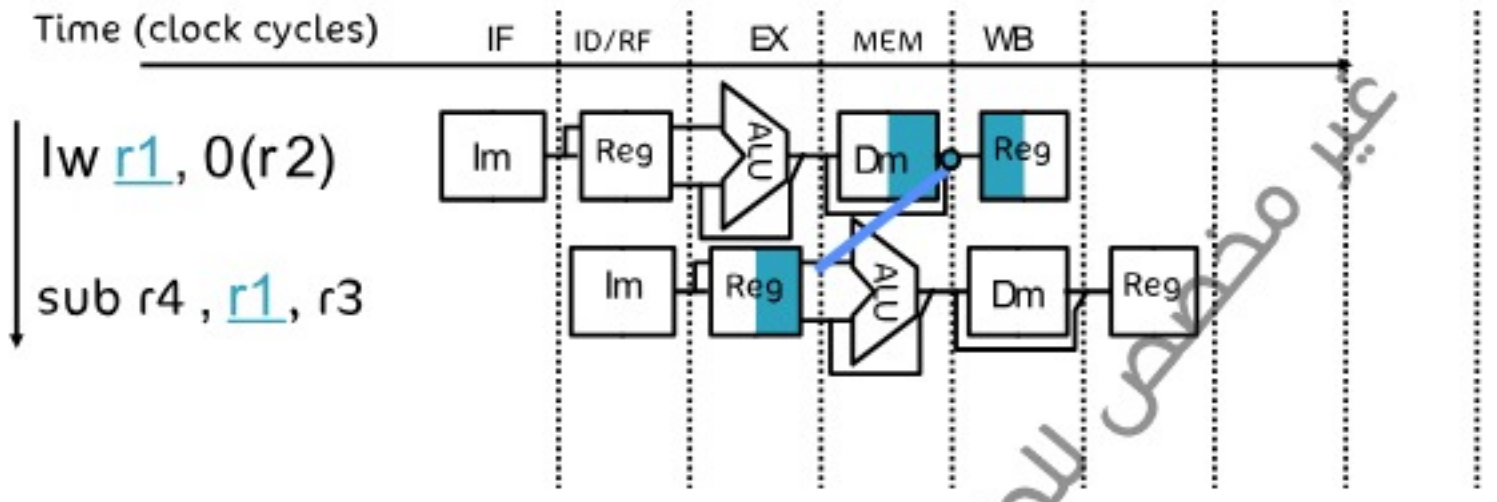
- ملخص:
- للـ Data Hazard حلان: Stalls و Forwarding
- خرج بعض المصادر يتم توجيهه لدخل مصادر أخرى.
- التوجيه يمكن أن يحد من بعض Data Hazard و ليس كلها .



Data Hazard: Load-Use

شكل خاص من أشـكال الـ Data Hazard :

حيث يتم طلب معلومة بواسطة تعليمة Load وهذه المعلومة ليـست متاحة عندما طُلبت.

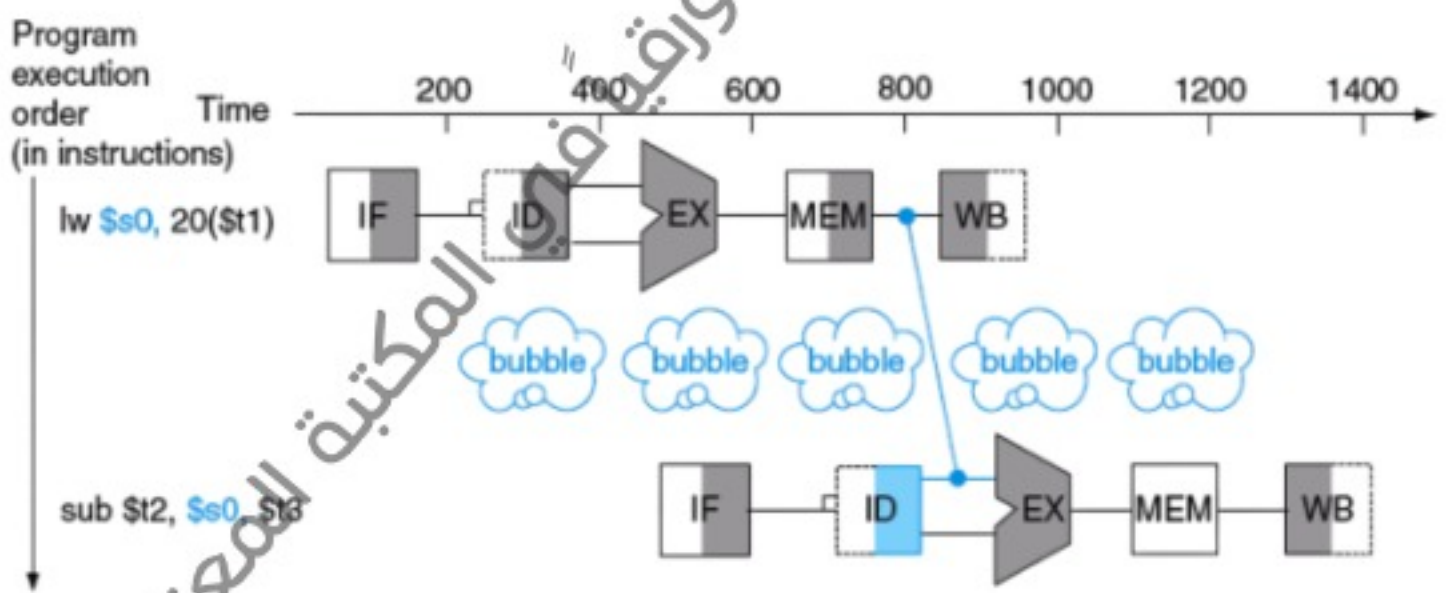


- نلاحظ ترابط الزمن وهذا معناه يوجد Hazard:

  في هذه الحالة لا نستطيع بالتوجيه (Forwarding) فقط.

  بل يجب إيقاف أو تأخير التعليمات المعتمدة على تعليمة Load.

- لتصبح كالتالي :

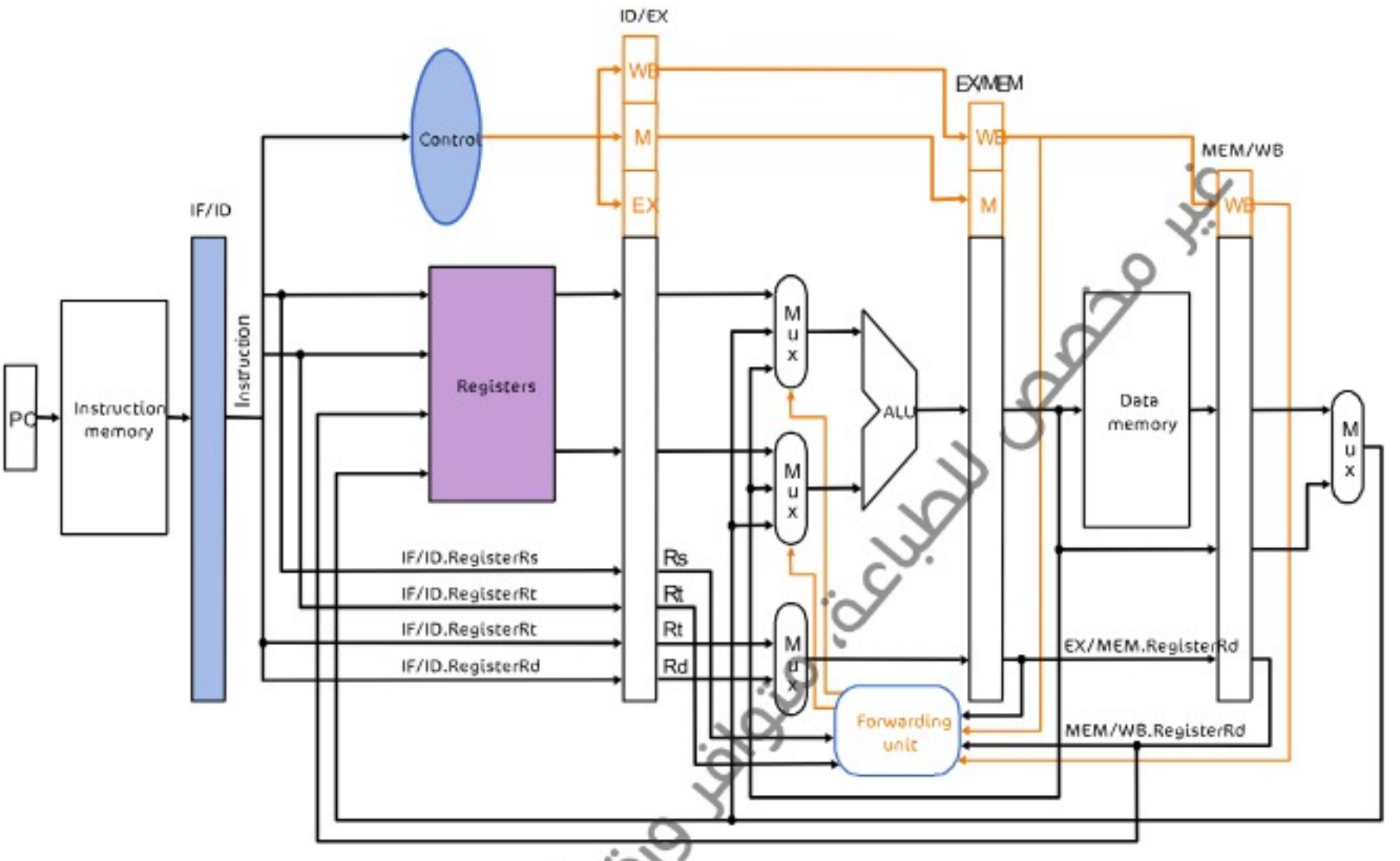


Lw $so, 20($t1)
Sub $t2, $so, $t3

Lw $so, 20($t1)
NOP
Sub $t2, $so, $t3

- حيث أن خرج النبضة الرابعة نريده في التعليمة الثانية لمدخل ALU فنؤخر نبضة ونعمل Forwarding من خرج الMemory إلى دخل الـ ALU

Forwarding Unit Hardware



- نرى أن وحدة التحكم للـ Forwarding تستقبل مداخل من كل أماكن التخزين الممكنة وذلك لتحكم بتوجيه البيانات التي يتم جلبها وفيها مخارج لـ دخل الـ ALU
- Forwarding unit يجب عليها التعامل مع كل أنواع الـ Hazard وحلها

مثال آخر عن Load-use :

```
lw R1, 0(R2)
Sub R3, R4, R5
Add R6, R1, R3
```

- هنا نريد موجه بين الأولى والثالثة وموجه بين الثانية والثالثة أو Stall

See You soon ...