ORGANISASI DAN ARSITEKTUR KOMPUTER

Pipeline processing

PIPELINE DASAR

- What is pipelining
- Clock & Latches
- Contoh 5 Stage Pipeline
- Load/Store & RISC/CISC
- Hazard
- Examples of Hazards

GENERAL CONCEPTS

- Pipelining merupakan teknik yang membagi task kedalam sejumlah subtask yang perlu dilakukan dalam sebuah *sequence.*
- Setiap subtask dikerjakan oleh sebuah fungsional unit. Unit-unit terhubung secara serial dan semuanya beropreasi secara simultan.
- Penggunaan pipelining untuk memperbaiki performa.

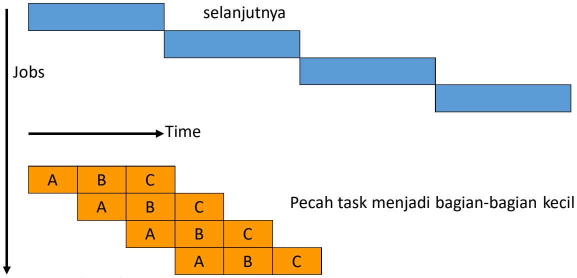
GENERAL CONCEPTS

- Teknik pipeline ini dapat diterapkan pada berbagai tingkatan dalam sistem komputer.
- Bisa pada level yang tinggi, misalnya program aplikasi, sampai pada tingkat yang rendah, seperti pada instruksi yang dijalankan oleh microprocessor.

The Assembly Line

Unpipelined

Start dan finish sebuah task sebelum pindah pada task selanjutnya



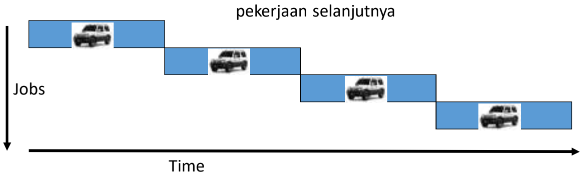
Pecah task menjadi bagian-bagian kecil

Pipelined

ILUSTRASI

Unpipelined

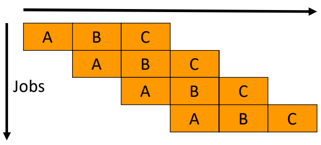
Start dan finish sebuah task sebelum pindah pada pekerjaan selanjutnya



- Misal untuk 1 mobil butuh 24 jam
- Tidak ada paralelisme
- Troughput → 1mobil/24jam

ILUSTRASI

Pipelined



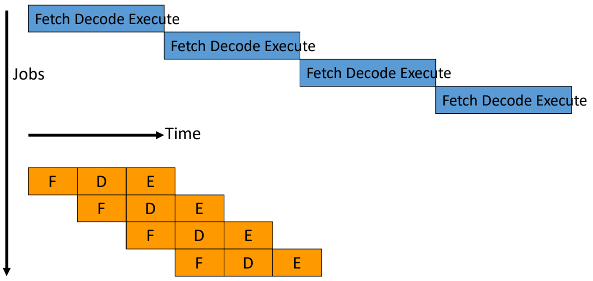
Pecah task menjadi bagian-bagian kecil

Misal:
A → Engine
B → Body
C → Paint

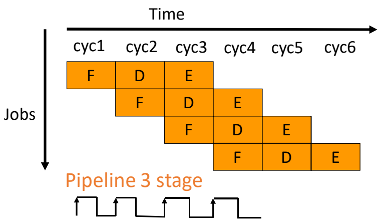
- Dengan paralelisme meningkatkan hasil
- Troughput → 1mobil/8jam

Waktu per stage = 8 jam

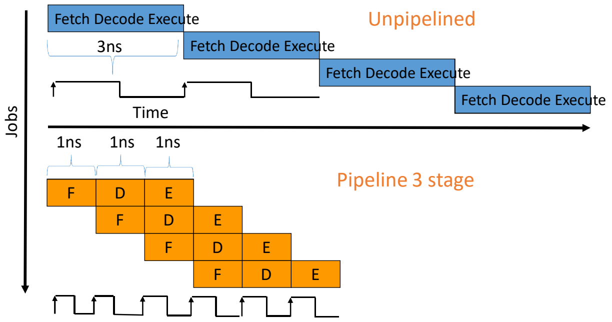
Unpipelined



Pipeline 3 stage



- Setiap clock naik ⎍⎍ → sebuah instruksi selesai



Unpipelined clock speed= 1/3ns =333MHz

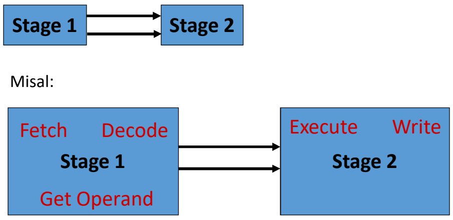
Pipelined clock speed= 1/1ns =1GHz

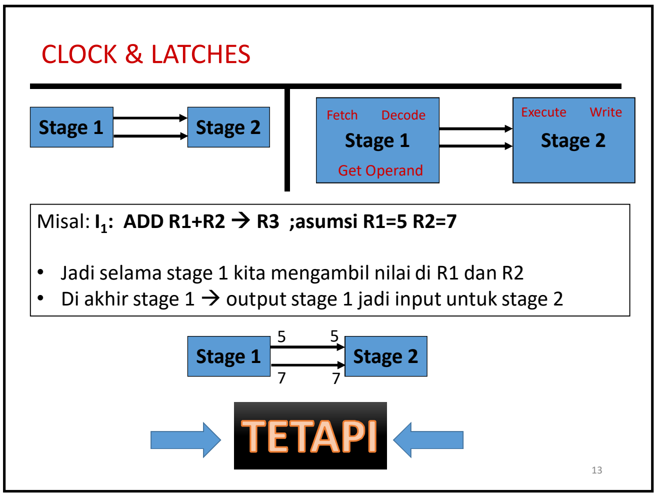
EFEK

Maka dengan pipelining:

- Waktu per instruksi naik
- Jumlah siklus per instruksi naik (perhatikan peningkatan kecepatan clock)
- Total waktu eksekusi turun
- Jumlah pipeline stage = peningkatan kecepatan clock

CLOCK & LATCHES

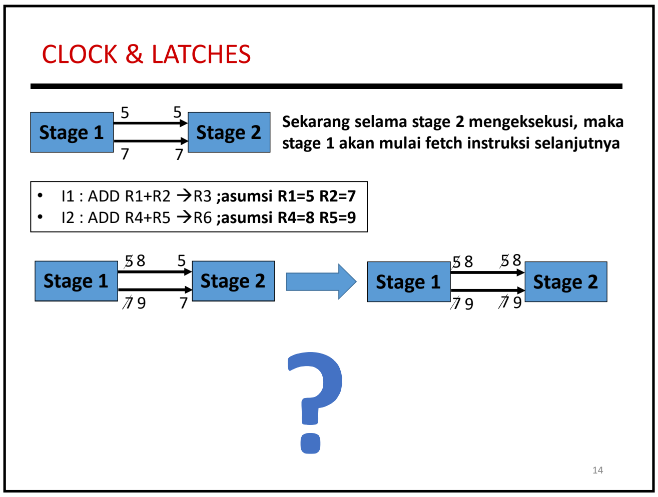




CLOCK & LATCHES

Misal: I_1: **ADD R1+R2 → R3 ;asumsi R1=5 R2=7**

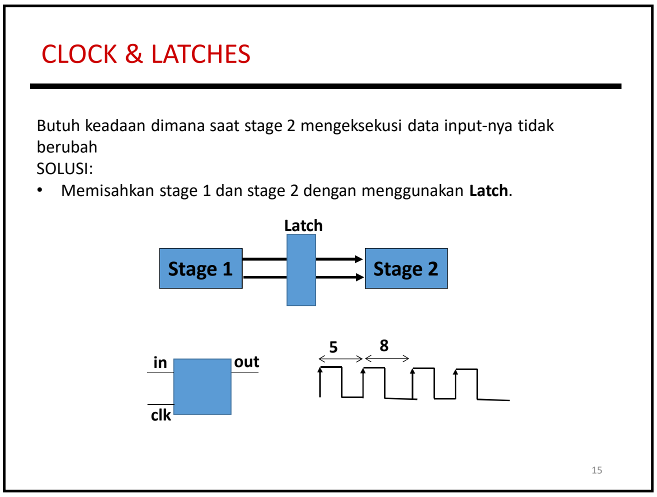
- Jadi selama stage 1 kita mengambil nilai di R1 dan R2
- Di akhir stage 1 → output stage 1 jadi input untuk stage 2



CLOCK & LATCHES

Sekarang selama stage 2 mengeksekusi, maka stage 1 akan mulai fetch instruksi selanjutnya

- I1 : ADD R1+R2 →R3 **;asumsi R1=5 R2=7**
- I2 : ADD R4+R5 →R6 **;asumsi R4=8 R5=9**

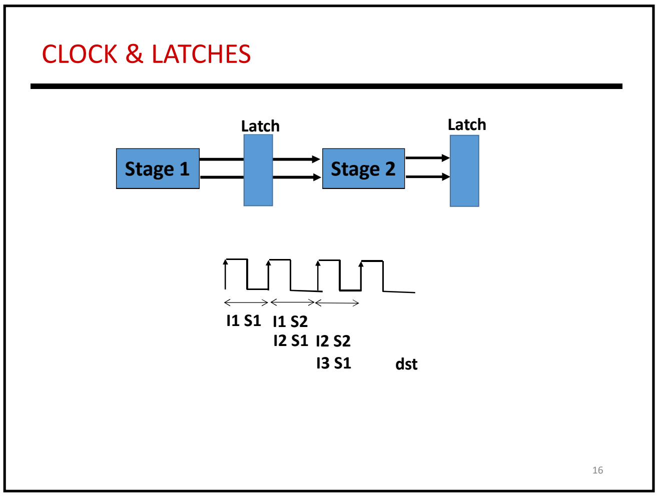


CLOCK & LATCHES

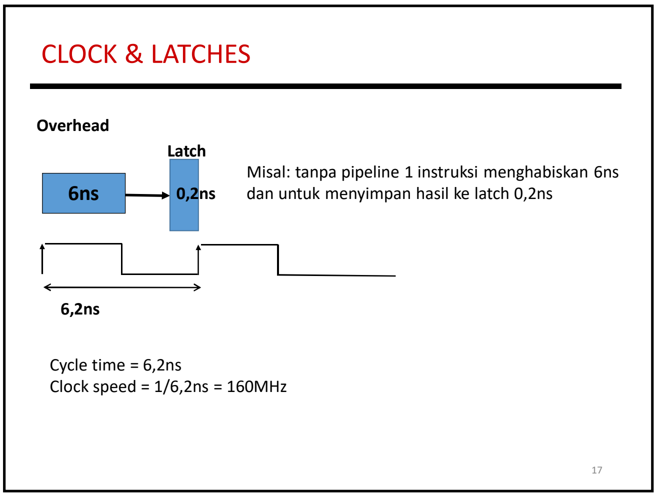
Butuh keadaan dimana saat stage 2 mengeksekusi data input-nya tidak berubah

SOLUSI:

- Memisahkan stage 1 dan stage 2 dengan menggunakan **Latch**.



CLOCK & LATCHES

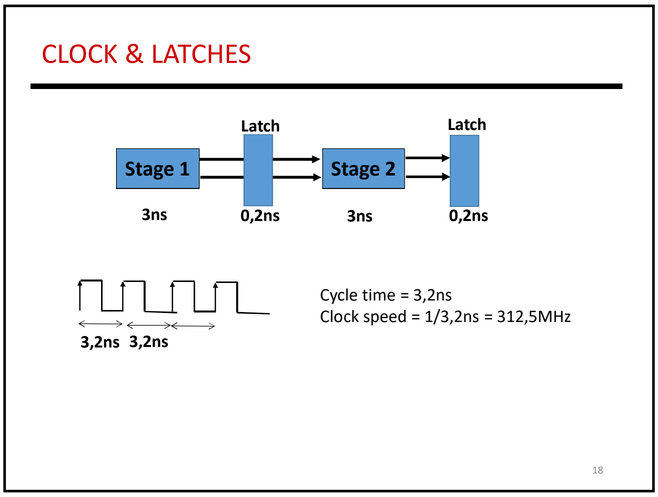


CLOCK & LATCHES

Overhead

Misal: tanpa pipeline 1 instruksi menghabiskan 6ns dan untuk menyimpan hasil ke latch 0,2ns

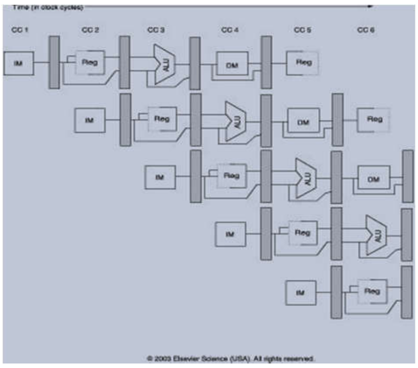
Cycle time = 6,2ns
Clock speed = 1/6,2ns = 160MHz



CLOCK & LATCHES

Cycle time = 3,2ns
Clock speed = 1/3,2ns = 312,5MHz

A 5-Stage Pipeline

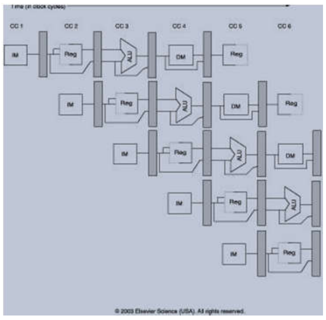


© 2003 Elsevier Science (USA). All rights reserved.

DETAIL SETIAP STAGE

A 5-Stage Pipeline

Use the PC to access the I-cache and increment PC by 4



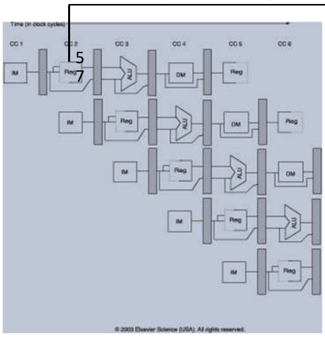
© 2003 Elsevier Science (USA). All rights reserved.

- PC
- Nilai PC bertambah
- Latch

PC

A 5-Stage Pipeline

Read registers, compare registers, compute branch target; for now, assume branches take 2 cyc (there is enough work that branches can easily take more)



© 2003 Elsevier Science (USA). All rights reserved.

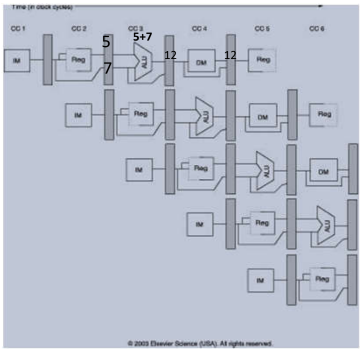
Dec RR

Misal:
ADD R1+R2

Misal jika ada instruksi lompat:
BRZ R3,[24]

A 5-Stage Pipeline

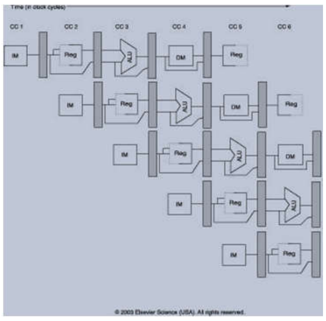
ALU computation, effective address computation for load/store



© 2003 Elsevier Science (USA). All rights reserved.

A 5-Stage Pipeline

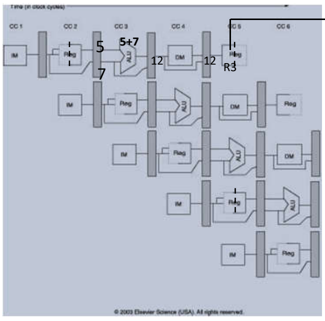
Memory access to/from data cache, stores finish in 4 cycles



© 2003 Elsevier Science (USA). All rights reserved.

A 5-Stage Pipeline

Write result of ALU computation or load into register file

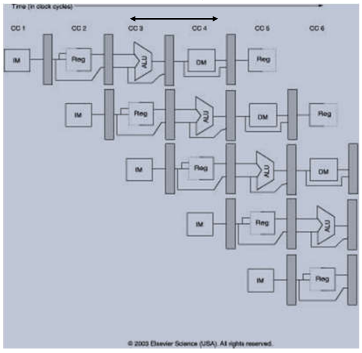


© 2003 Elsevier Science (USA). All rights reserved.

WR

A 5-Stage Pipeline

ALU computation, effective address computation for load/store

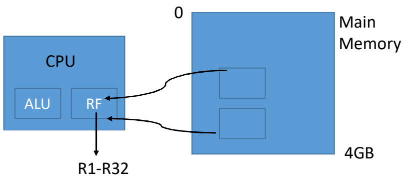


© 2003 Elsevier Science (USA). All rights reserved.

RISC/ CISC Load/Stores

- RISC
  - Reduce instruction set computer
  - Setiap instruksi sederhana

  Misal:
  ADD R1,R2 → R3
  OR R1,R2 → R3
  LD [ ]→R4
  ST [ ]←R5



Misal:

Kita ingin menyalin nilai dari Main memory ke RF (register File) R1 dan R2 dan menjumlahkannya dan simpan ke R3 lalu disimpan ke alamat memori yang ditunjuk.

Jadi:

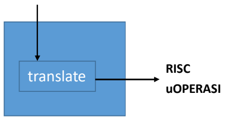
LD [ ]→R1
LD [ ]→R2
Add R1,R2 →R3
ST [ ]←R3

- CISC (Complex Instruction Set Computer)
- 1 sequence instruksi sama dengan beberapa instruksi sederhana

- MUL R1,R2→R3
- ADD R3,R4 →R4

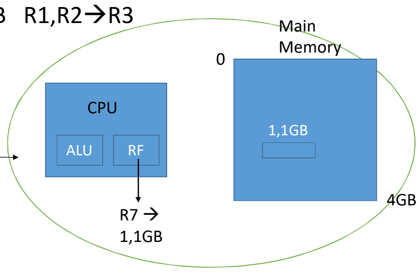
→ • MAC R1,R2,R4

Intel x86 → CISC



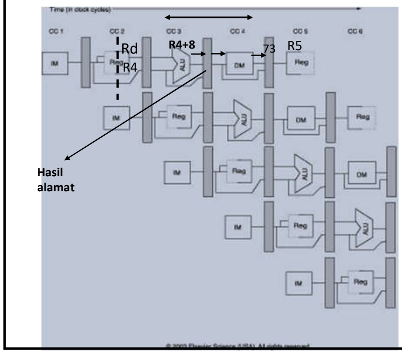
Misal:

- ALU ADD/OR/SUB R1,R2→R3
- LD [ ]→R4
- ST [ ]←R5
- LD [R7]→R4
- LD 8[R7]→R4
- ST 24[R7]←R5



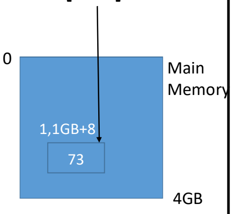
A 5-Stage Pipeline

ALU computation, effective address computation for load/store



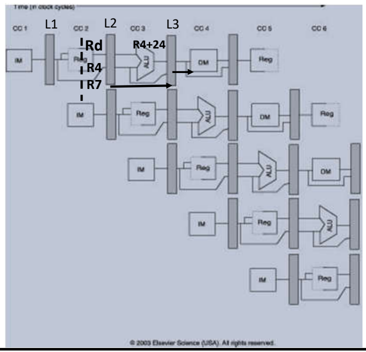
© 2003 Elsevier Science (USA). All rights reserved.

LD 8[R4]→R5



A 5-Stage Pipeline

ALU computation, effective address computation for load/store



© 2003 Elsevier Science (USA). All rights reserved.

ST 24[R4]←R7

TERIMA KASIH

Hazards

- Hazard adalah keadaan yang dapat menimbulkan tunda (delay, stall) pada pipeline.
- Pada keadaan stall, pipeline tidak menghasilkan output sehingga peningkatan keluaran ideal tidak dapat dicapai.

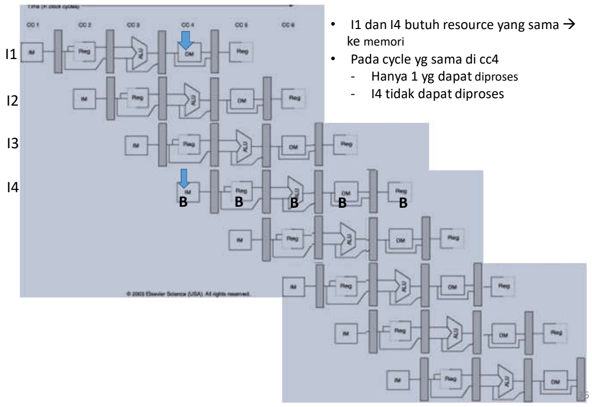
Hazards

- Operasi pipeline akan stall jika salah satu unit atau stage membutuhkan lebih banyak waktu untuk melakukan fungsinya dan memaksa stage lainnya untuk idle.
- Situasi seperti itu disebut pipeline bubble (pipeline hazards)

Hazards

- Structural hazards: different instructions in different stages (or the same stage) conflicting for the same resource
- Data hazards: an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction
- Control hazard: fetch cannot continue because it does not know the outcome of an earlier branch – special case of a data hazard – separate category because they are treated in different ways

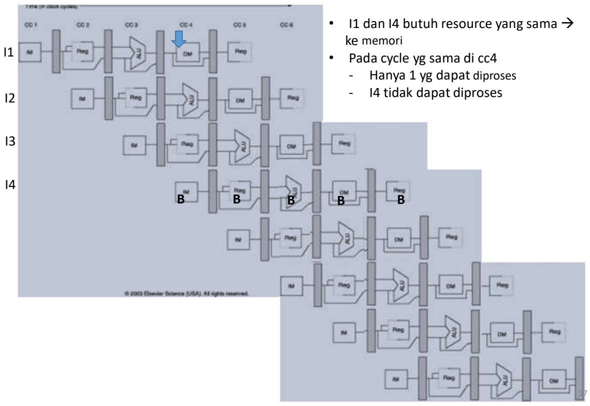
A 5-Stage Pipeline



© 2003 Elsevier Science (USA). All rights reserved.

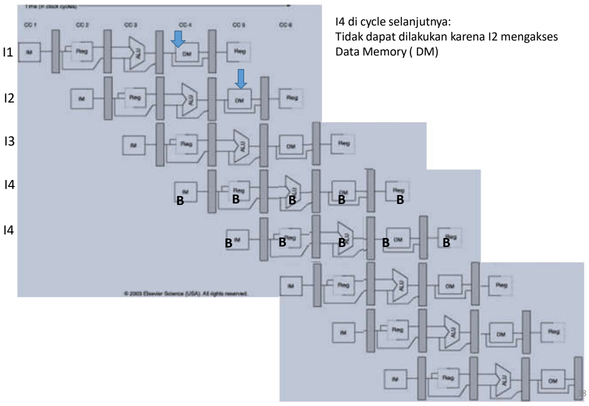
- I1 dan I4 butuh resource yang sama → ke memori
- Pada cycle yg sama di cc4
  - Hanya 1 yg dapat diproses
  - I4 tidak dapat diproses

A 5-Stage Pipeline



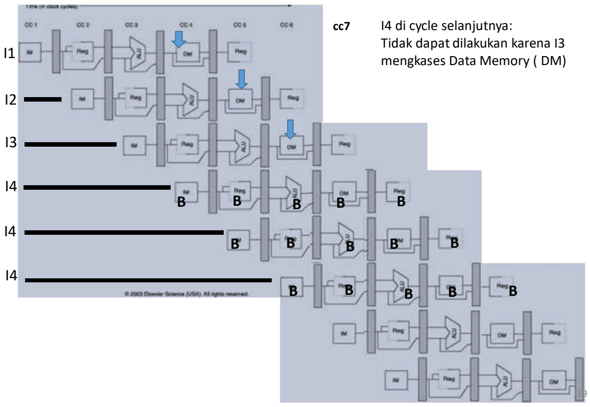
- I1 dan I4 butuh resource yang sama → ke memori
- Pada cycle yg sama di cc4
  - Hanya 1 yg dapat diproses
  - I4 tidak dapat diproses

A 5-Stage Pipeline



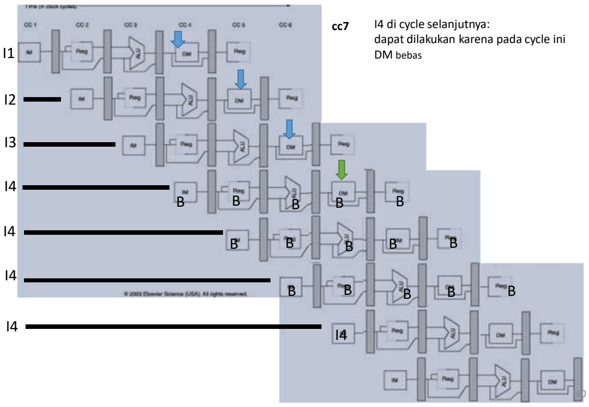
I4 di cycle selanjutnya:
Tidak dapat dilakukan karena I2 mengakses Data Memory (DM)

A 5-Stage Pipeline



cc7 I4 di cycle selanjutnya:
Tidak dapat dilakukan karena I3 mengkases Data Memory (DM)

A 5-Stage Pipeline

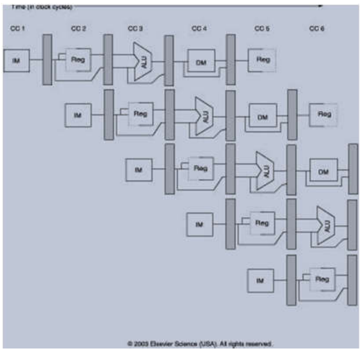


cc7 I4 di cycle selanjutnya:
dapat dilakukan karena pada cycle ini DM bebas

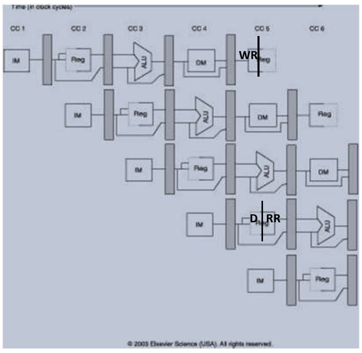
Solusi:
Membagi memori menjadi dua yaitu:

- **memori instruksi dan**
- **memori data**

Masalah di cc5 bagian register file



Masalah di cc5 bagian register file

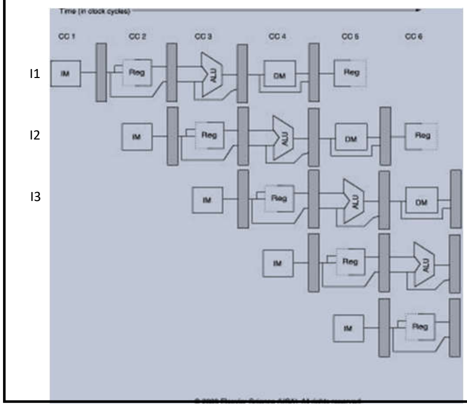


© 2003 Elsevier Science (USA). All rights reserved.

Structural Hazards

- Example: a unified instruction and data cache → stage 4 (MEM) and stage 1 (IF) can never coincide
- The later instruction and all its successors are delayed until a cycle is found when the resource is free → these are pipeline bubbles
- Structural hazards are easy to eliminate – increase the number of resources (for example, implement a separate instruction and data cache)

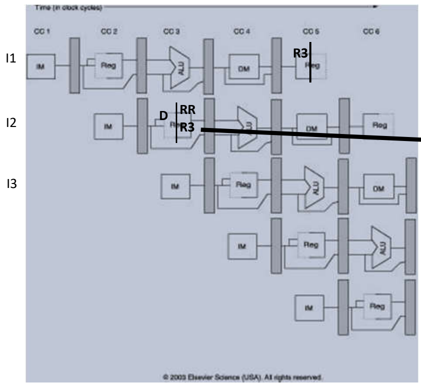
Data Hazards



I1 : R1+R2 →R3
I2 : R3+R4 → R5

© 2003 Elsevier Science (USA). All rights reserved.

Data Hazards



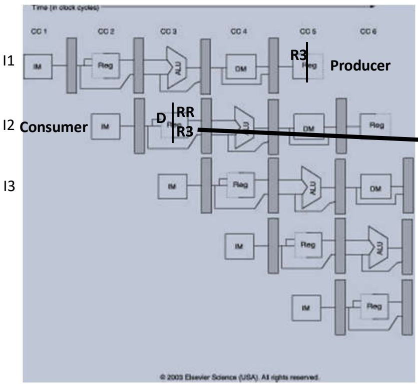
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
I2 : R3+R4 → R5

?

early

Data Hazards



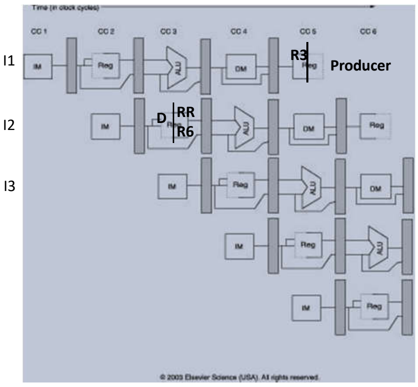
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
I2 : R3+R4 → R5

?

Wrong result

Data Hazards

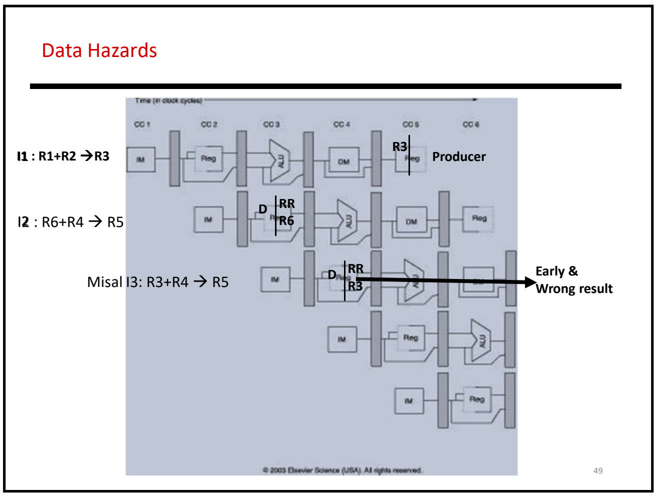


© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3

I2 : ~~R3~~+R4 → R5

I2 : R6+R4 → R5

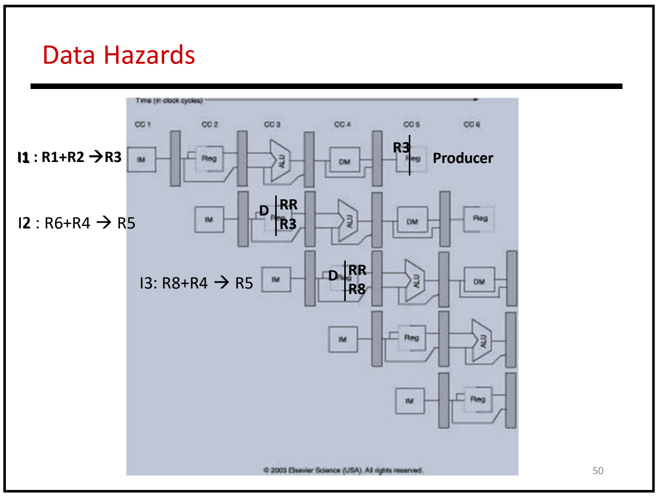


Data Hazards

I1 : R1+R2 →R3

I2 : R6+R4 → R5

Misal I3: R3+R4 → R5

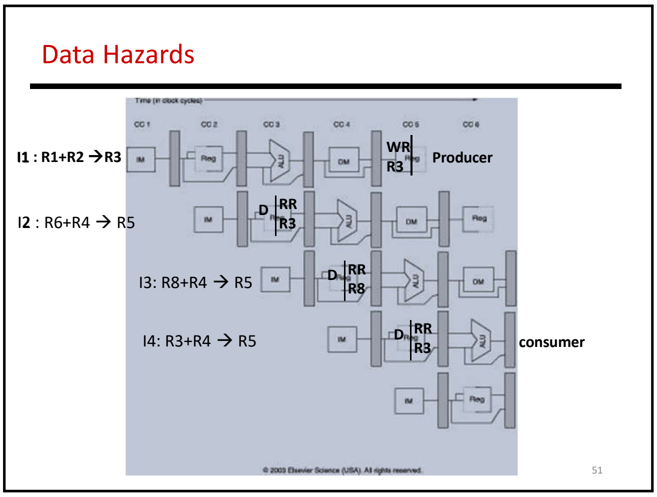


Data Hazards

I1 : R1+R2 →R3

I2 : R6+R4 → R5

I3: R8+R4 → R5



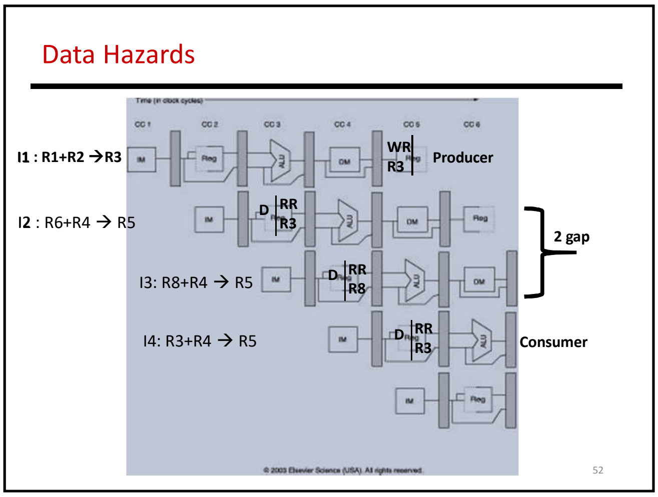
Data Hazards

I1 : R1+R2 →R3

I2 : R6+R4 → R5

I3: R8+R4 → R5

I4: R3+R4 → R5



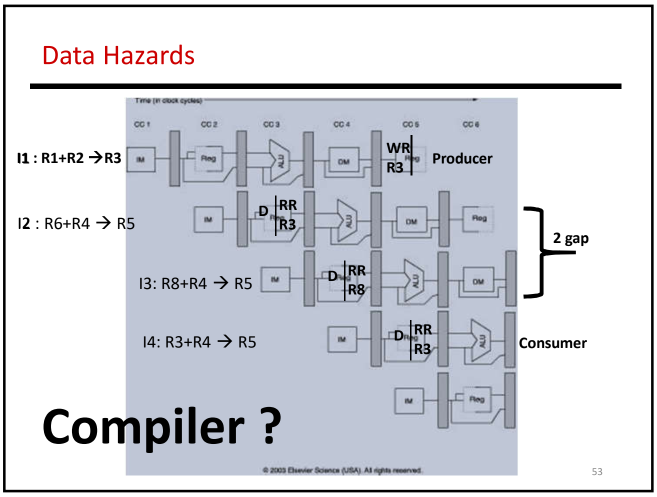
Data Hazards

I1 : R1+R2 →R3

I2 : R6+R4 → R5

I3: R8+R4 → R5

I4: R3+R4 → R5



Data Hazards

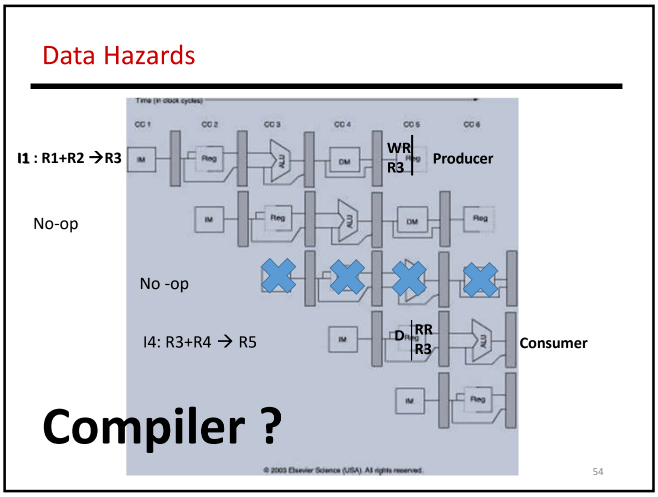
I1 : R1+R2 →R3

I2 : R6+R4 → R5

I3: R8+R4 → R5

I4: R3+R4 → R5

Compiler ?



Data Hazards

I1 : R1+R2 →R3

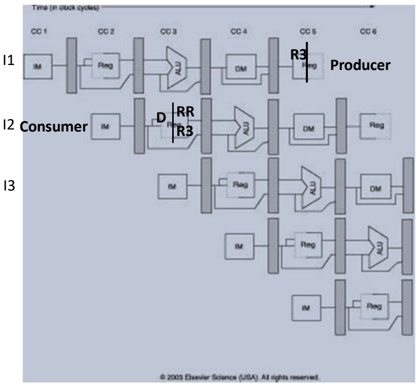
No-op

No -op

I4: R3+R4 → R5

Compiler ?

Data Hazards



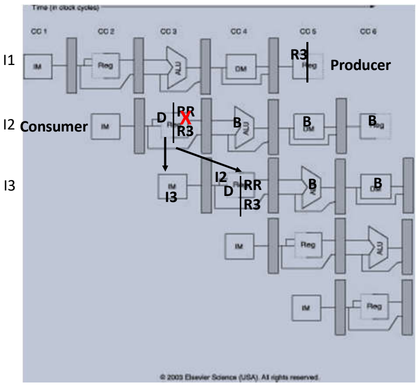
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 → R3

I2 : R3+R4 → R5

- Deteksi
- Hardware
- Decode
- Sinyal
- Kirim buble ke pipeline dan mengulang D|RR

Data Hazards



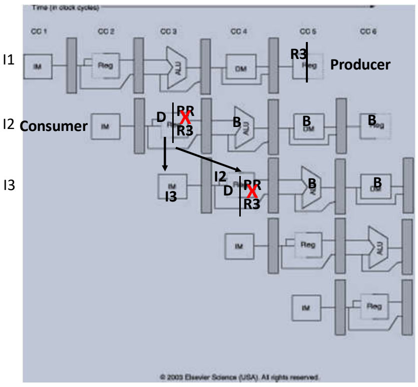
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 → R3

I2 : R3+R4 → R5

- Deteksi
- Hardware
- Decode
- Sinyal
- Kirim buble ke pipeline dan mengulang D|RR

Data Hazards



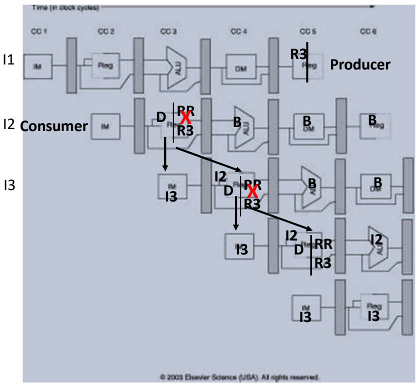
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 → R3

I2 : R3+R4 → R5

- Deteksi
- Hardware
- Decode
- Sinyal
- Kirim buble ke pipeline dan mengulang D|RR

Data Hazards



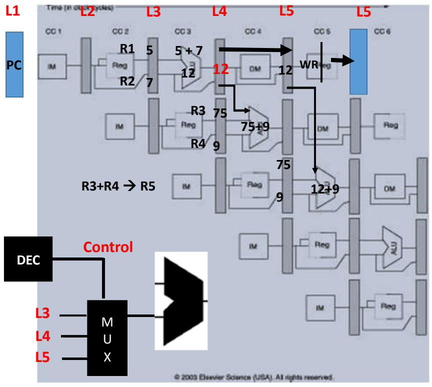
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 → R3

I2 : R3+R4 → R5

- Deteksi
- Hardware
- Decode
- Sinyal
- Kirim buble ke pipeline dan mengulang D|RR

A 5-Stage Pipeline

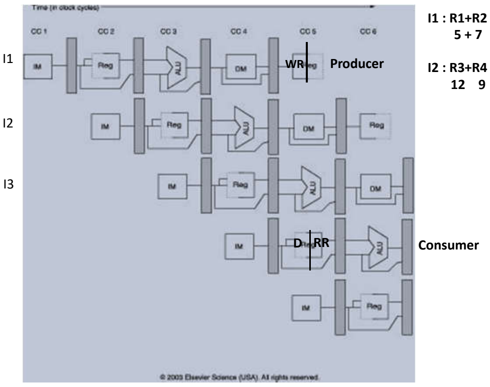


© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 → R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

A 5-Stage Pipeline

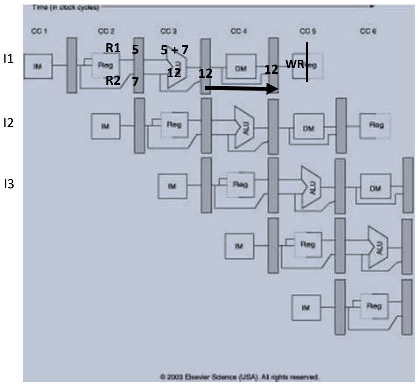


© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

A 5-Stage Pipeline

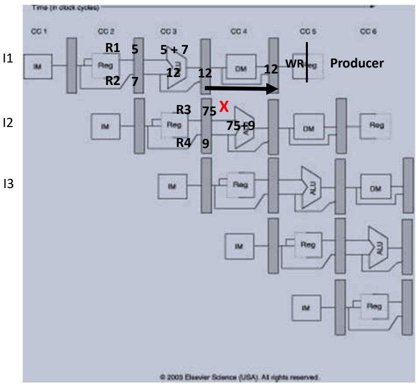


© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

A 5-Stage Pipeline

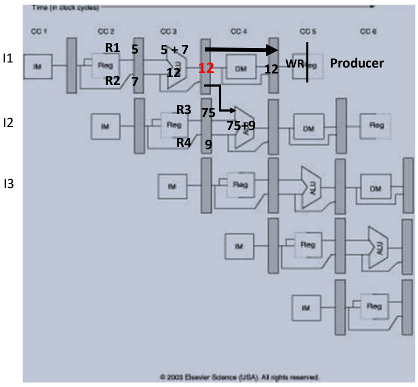


© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

A 5-Stage Pipeline



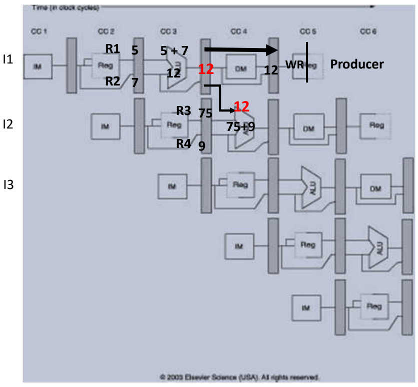
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

- Angka 12 hasil ada di pipeline I1

A 5-Stage Pipeline



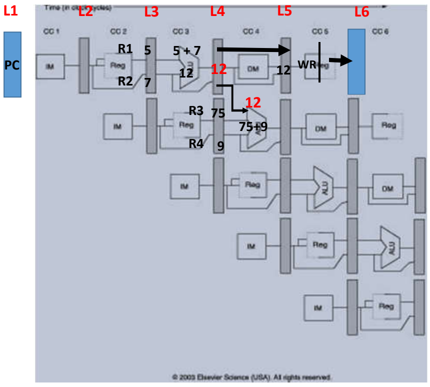
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

- Angka 12 hasil ada di pipeline I1

A 5-Stage Pipeline



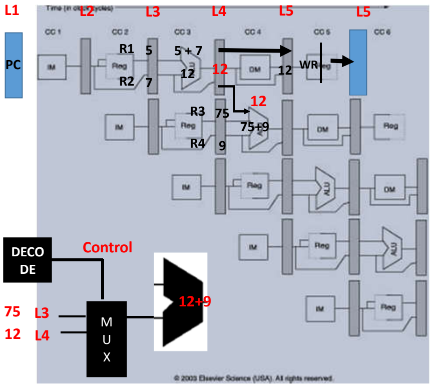
© 2003 Elsevier Science (USA). All rights reserved.

I1 : R1+R2 →R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

- Angka 12 hasil ada di pipeline I1

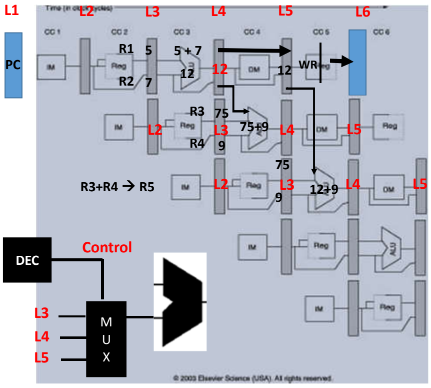
A 5-Stage Pipeline



I1 : R1+R2 → R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

A 5-Stage Pipeline

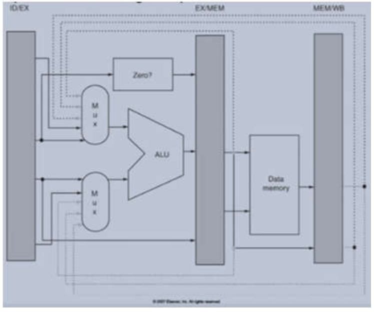


I1 : R1+R2 → R3
5 + 7 12

I2 : R3+R4 → R5
12 9 21

Pipeline Implementation

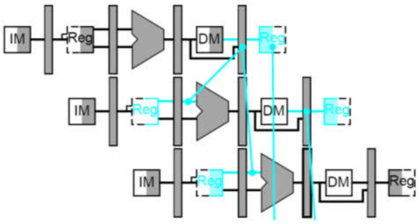
- Signals for the muxes have to be generated – some of this can happen during ID
- Need look-up tables to identify situations that merit bypassing/stalling – the number of inputs to the muxes goes up



Example

add R1, R2, R3
;(R2+R3 → R1)

lw R4, 8(R)1)

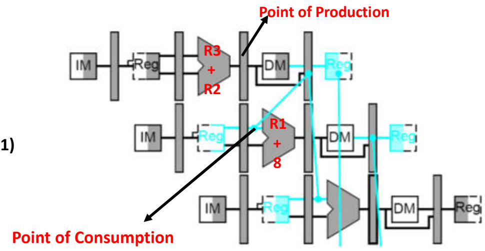


- **Point of Production**
- **Point of Consumption**

Example

add R1, R2, R3
;(R2+R3 → R1)

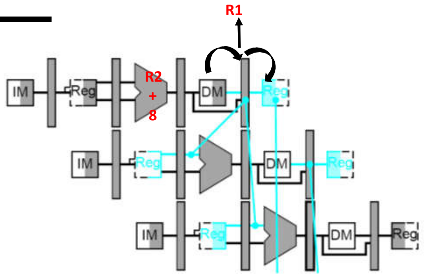
lw R4, 8(R1)



Example 2

lw R1, 8(R2)

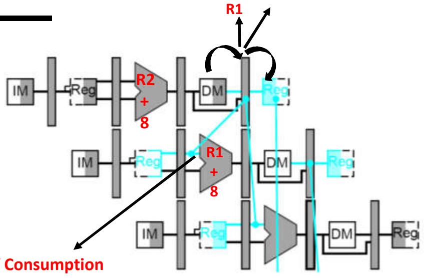
lw R4, 8(R1)



Example 2

lw R1, 8(R2)

lw R4, 8(R1)



Point of Production

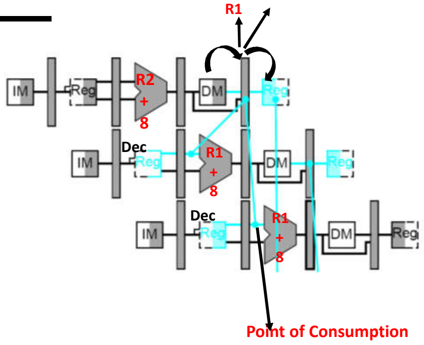
Point of Consumption

Will not work !

Example 2

lw R1, 8(R2)

lw R4, 8(R1)



Point of Production

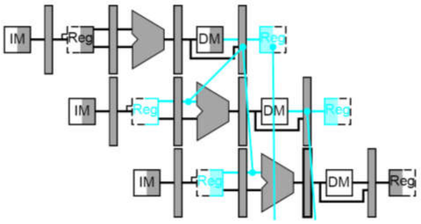
Point of Consumption

- **Decode**
- **Stall**

Example 3

lw R1, 8(R2)

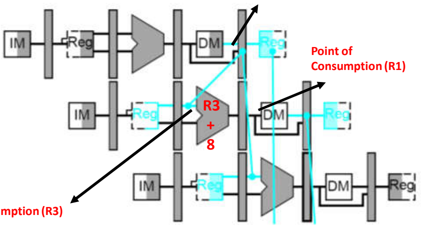
sw R1, 8(R3)



Example 3

lw R1, 8(R2)

sw R1, 8(R3)



Point of Production R1

Point of Consumption (R1)

Point of Consumption (R3)

Summary

- For the 5-stage pipeline, bypassing can eliminate delays between the following example pairs of instructions:

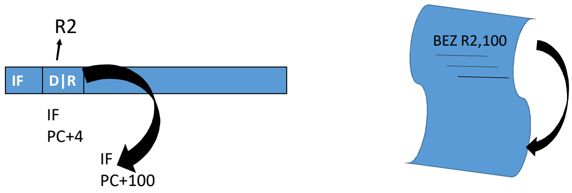
```
add/sub         R1, R2, R3
add/sub/lw/sw   R4, R1, R5

lw      R1, 8(R2)
sw      R1, 4(R3)
```

- The following pairs of instructions will have intermediate stalls:

```
lw              R1, 8(R2)
add/sub/lw      R3, R1, R4     or   sw   R3, 8(R1)
```

Control Hazards

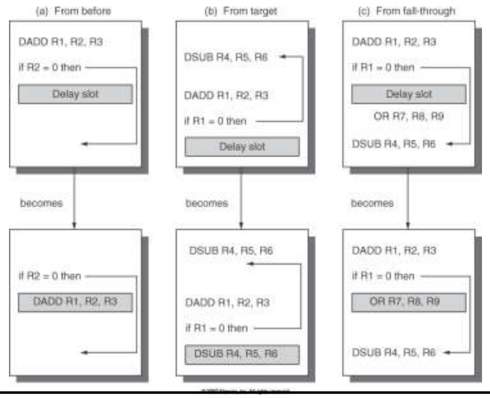


✓ IF
✗ IF PC+4
✓ IF PC+100

Control Hazards

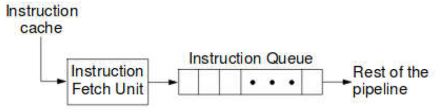
- Simple techniques to handle control hazard stalls:
  - for every branch, introduce a stall cycle (note: every 6th instruction is a branch!)
  - assume the branch is not taken and start fetching the next instruction – if the branch is taken, need **hardware** to cancel the effect of the wrong-path instruction
  - Predict the next PC and fetch that instruction
  - fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost

Branch Delay Slots



Instruction Fetch Units and Instruction Queues

- Most processors employ sophisticated fetch units that fetch instructions before they are needed and store them in a queue.



- The fetch unit also has the ability to recognize branch instructions and to generate the target address.

Instruction Fetch Units and Instruction Queues

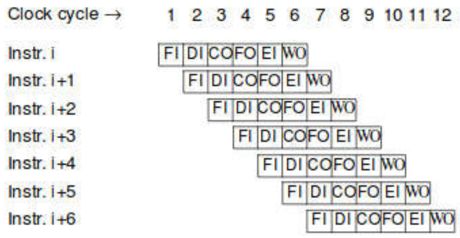
- Penalty produced by unconditional branches can be drastically reduced: the fetch unit computes the target address and continues to fetch instructions from that address, which are sent to the queue.
- The rest of the pipeline gets a continuous stream of instructions, without stalling.

Instruction Fetch Units and Instruction Queues

- The rate at which instructions can be read (from the instruction cache) must be sufficiently high to avoid an empty queue.
- With conditional branches penalties can not be avoided.
- The branch condition, which usually depends on the result of the preceding instruction, has to be known in order to determine the following instruction.

CONTROL HAZARD (6 stage pipeline example)

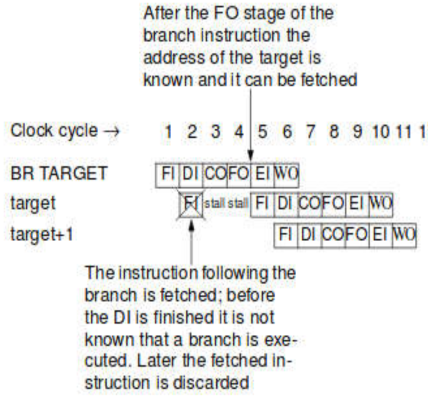
- FI: fetch instruction — FO: fetch operand
- DI: decode instruction — EI: execute instruction
- CO: calculate operand address — WO: write operand



Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
Instr. i	FI	DI	CO	FO	EI	WO						
Instr. i+1		FI	DI	CO	FO	EI	WO					
Instr. i+2			FI	DI	CO	FO	EI	WO				
Instr. i+3				FI	DI	CO	FO	EI	WO			
Instr. i+4					FI	DI	CO	FO	EI	WO		
Instr. i+5						FI	DI	CO	FO	EI	WO	
Instr. i+6							FI	DI	CO	FO	EI	WO

Unconditional branch

```
                - - - - - - - - - - - - - -
                BR      TARGET
                - - - - - - - - - - - - - -
  TARGET        - - - - - - - - - - - - - -
```



After the FO stage of the branch instruction the address of the target is known and it can be fetched

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
BR TARGET	FI	DI	CO	FO	EI	WO						
target		~~FI~~	stall	stall	FI	DI	CO	FO	EI	WO		
target+1						FI	DI	CO	FO	EI	WO	

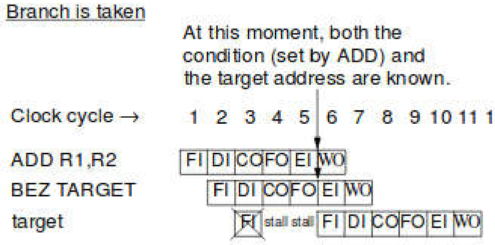
The instruction following the branch is fetched; before the DI is finished it is not known that a branch is executed. Later the fetched instruction is discarded

Penalty: 3 cycles

Conditional branch

```
              ADD      R1,R2          R1 ← R1 + R2
              BEZ      TARGET         branch if zero
              instruction i+1
              - - - - - - - - - - - - -
  TARGET      - - - - - - - - - - - - -
```

Branch is taken



At this moment, both the condition (set by ADD) and the target address are known.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TARGET		FI	DI	CO	FO	EI	WO					
target			~~FI~~	stall	stall	FI	DI	CO	FO	EI	WO	

Penalty: 3 cycles

Branch **not** taken

At this moment the condition is known and instr+1 can go on.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TARGET		FI	DI	CO	FO	EI	WO					
instr i+1			FI	stall	stall	DI	CO	FO	EI	WO		

Penalty: 2 cycles

With conditional branch we have a penalty even if the branch has *not* been taken. This is because we have to wait until the branch condition is available.

DELAYED BRANCHING

- The idea with delayed branching is to let the CPU do some useful work during some of the cycles which are shown above to be stalled.
- With delayed branching the CPU **always** executes the instruction that immediately follows after the branch and only then alters (if necessary) the sequence of execution. The instruction after the branch is said to be in the *branch delay slot*.

This is what the programmer has written

This instruction does not influence any of the instructions which follow until the branch; it also doesn't influence the outcome of the branch. →

```
      MUL  R3,R4     R3 ← R3*R4
      SUB  #1,R2     R2 ← R2-1
      ADD  R1,R2     R1 ← R1+R2
      BEZ  TAR       branch if zero
      MOVE #10,R1    R1 ← 10
      -------------
TAR   -------------
```

This instruction should be executed only if the branch is not taken. → MOVE #10,R1

- **The compiler** (assembler) has to find an instruction which can be moved from its original place into *the* branch delay slot after the branch and which will be executed regardless of the outcome of the branch.

This is what the compiler (assembler) has produced and what actually will be executed:

```
      SUB   #1,R2
      ADD   R1,R2
      BEZ   TAR
      MUL   R3,R4
      MOVE  #10,R1
      -------------
TAR   -------------
```

MUL R3,R4 ← This instruction will be executed regardless of the condition.

MOVE #10,R1 ← This will be executed only if the branch has not been taken

This happens in the pipeline:

Branch is taken

At this moment, both the condition (set by ADD) and the target address are known.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TAR		FI	DI	CO	FO	EI	WO					
MUL R3,R4			FI	DI	CO	FO	EI	WO				
the target				~~FI~~	stall	FI	DI	CO	FO	EI	WO	

Penalty: 2 cycles

Branch is **not** take

At this moment the condition is known and the MOVE can go on.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TAR		FI	DI	CO	FO	EI	WO					
MUL R3,R4			FI	DI	CO	FO	EI	WO				
MOVE				FI	stall	DI	CO	FO	EI	WO		

Penalty: 1 cycle

What happens if the compiler is not able to find an instruction to be moved after the branch, into the branch delay slot?

In this case a NOP instruction (an instruction that does nothing) has to be placed after the branch. In this case the penalty will be the same as without delayed branching.

```
        MUL   R2,R4  ← Now, with R2, this instruction in-
        SUB   #1,R2    fluences the following ones and
        ADD   R1,R2    cannot be moved from its place.
        BEZ   TAR
        NOP
        MOVE  #10,R1
        - - - - - - - - - - - - -
TAR     - - - - - - - - - - - - -
```

Branch Prediction

- In the last example we have considered that the *branch will not be taken* and we fetched the instruction following the branch; in the case the branch was taken the fetched instruction was discarded. As result, we had

branch penalty of
- 1 if the branch is **not** taken (prediction fulfilled)
- 2 if the branch is taken (prediction not fulfilled)

- Correct branch prediction is very important and can produce substantial performance improvements.
- Based on the predicted outcome, the respective instruction can be fetched, as well as the instructions following it, and they can be placed into the instruction queue.
- If, after the branch condition is computed, it turns out that the prediction was correct, execution continues.
- On the other hand, if the prediction is not fulfilled, the fetched instruction(s) must be discarded and the correct instruction must be fetched.

- To take full advantage of branch prediction, we can have the instructions not only fetched but also begin execution. **This is known as speculative execution**.
- **Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution path.** If it turns out that the prediction was correct, execution goes on without introducing any branch penalty.
- If, however, the prediction is not fulfilled, the instruction(s) started in advance and all their associated data must be purged and the state previous to their execution restored.

Branch prediction strategies:

1. Static prediction
2. Dynamic prediction

Static Branch Prediction

Static prediction techniques do not take into consideration execution history.

Static approaches:

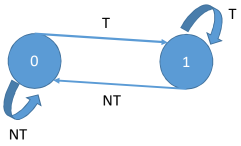
1. Predict never taken (Motorola 68020): assumes that the branch is not taken.
2. Predict always taken: assumes that the branch is taken.
3. Predict depending on the branch direction (PowerPC 601):
   - predict branch taken for backward branches;
   - predict branch not taken for forward branches.

Dynamic Branch Prediction

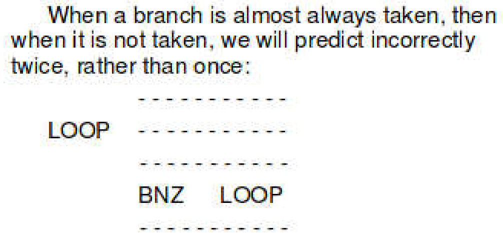
- Dynamic prediction techniques improve the accuracy of the prediction by recording the history of conditional branches.
- **One-Bit Prediction Scheme**
- **Two-Bit Prediction Scheme**

One-Bit Prediction Scheme

- One-bit is used in order to record if the last execution resulted in a branch taken or not. The system predicts the same behavior as for the last time.



Dynamic Branch Prediction



When a branch is almost always taken, then when it is not taken, we will predict incorrectly twice, rather than once:

```
        - - - - - - - - - - -
LOOP    - - - - - - - - - - -
        - - - - - - - - - - -
        BNZ    LOOP
        - - - - - - - - - - -
```

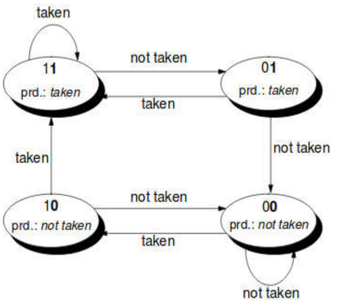
Dynamic Branch Prediction

- After the loop has been executed for the first time and left, it will be remembered that BNZ has not been taken. Now, when the loop is executed again, after the first iteration there will be a false prediction; following predictions are OK until the last iteration, when there will be a second false prediction.
- In this case the result is even worse than with static prediction considering that backward loops are always taken (PowerPC 601 approach).

Two-Bit Prediction Scheme

- With a two-bit scheme predictions can be made depending on the last two instances of execution.
- A typical scheme is to change the prediction only if there have been two incorrect predictions in a row.

Two-Bit Prediction Scheme



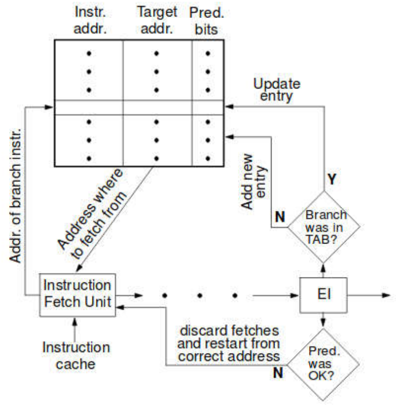
```
LOOP   - - - - - - - - - - -
       - - - - - - - - - - -
       - - - - - - - - - - -
       BNZ     LOOP
       - - - - - - - - - - -
```

After the first execution of the loop the bits attached to BNZ will be 01; now, there will be always one false prediction for the loop, at its exit.

Branch History Table

- History information can be used not only to predict the outcome of a conditional branch but also to avoid recalculation of the target address.
- Together with the bits used for prediction, the target address can be stored for later use in a branch history table

Branch History Table



Branch History Table

- Address where to fetch from : If the branch instruction is not in the table the next instruction (address PC+1) is to be fetched. If the branch instruction is in the table first of all a prediction based on theprediction bits is made. Depending on the prediction outcome the next instruction (address PC+1) or the instruction at the target address is to be fetched.
- Update entry : If the branch instruction has been in the table, the respective entry has to be updated to reflect the correct or incorrect prediction.
- Add new entry: If the branch instruction has not been in the table, it is added to the table with the corresponding information concerning branch outcome and target address. If needed one of the existing table entries is discarded. Replacement algorithms similar to those for cache memories are used.

Branch History Table

- Using dynamic branch prediction with history tables up to 90% of predictions can be correct.
- Both Pentium and PowerPC 620 use speculative execution with dynamic branch prediction based on a branch history table

TERIMA KASIH