Pipeline Processing

PIPELINING

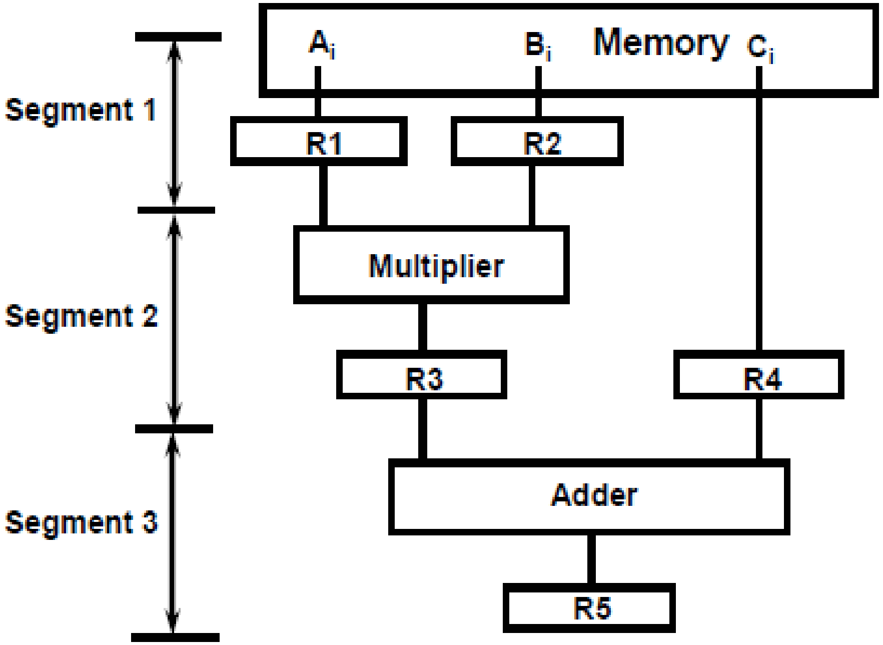
A technique of decomposing a sequential process into sub operations, with each sub process being executed in a partial dedicated segment that operates concurrently with all other segments.

A pipeline can be visualized as a collection of processing segments through which binary information flows.

The name “pipeline” implies a flow of information analogous to an industrial assembly line.

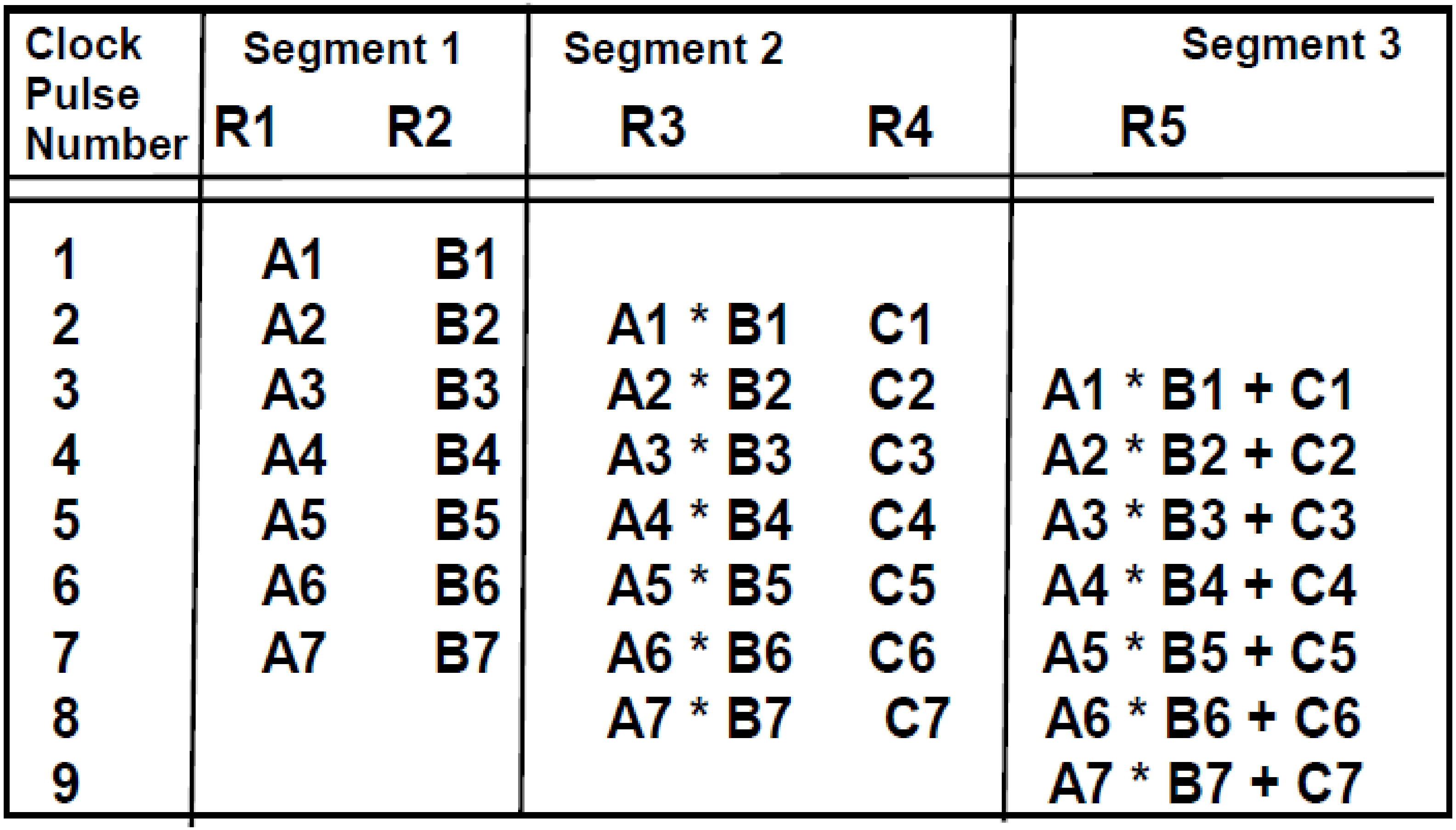
Example of the Pipeline Organization

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7



R1 ← A_i, R2 ← B_i	Load A_i and B_i
R3 ← R1 * R2, R4 ← C_i	Multiply and load C_i
R5 ← R3 + R4	Add

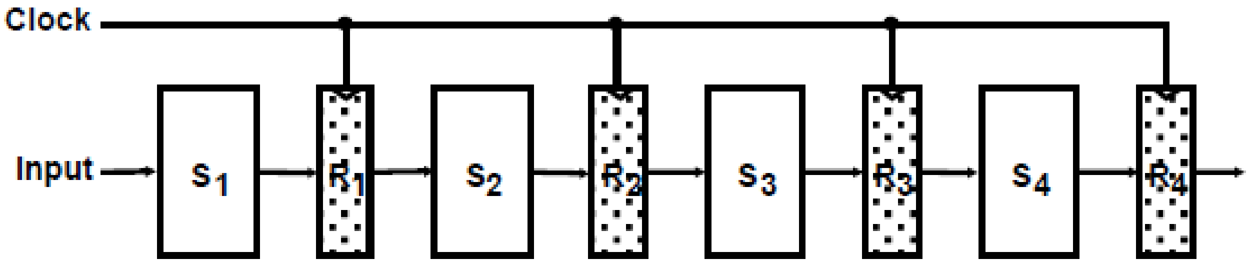
OPERATIONS IN EACH PIPELINE STAGE



Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

Segment	1	2	3	4	5	6	7	8	9
1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

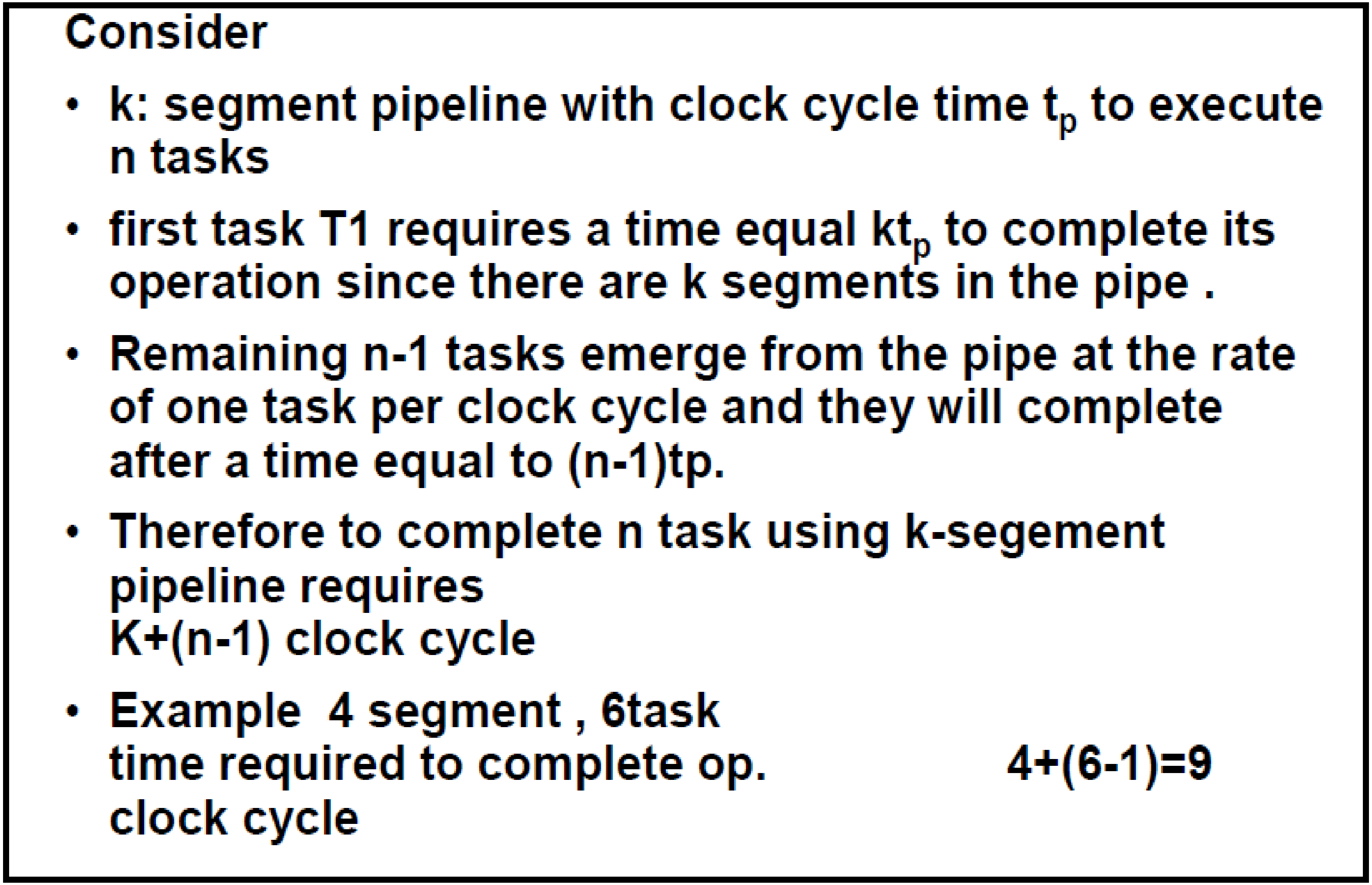
Clock cycles

Behavior of the pipeline is illustrated with a space time diagram.

Space time diagram:

This shows the segment utilization as a function of time.

Speedup ratio of pipeline



Consider

- k: segment pipeline with clock cycle time t_p to execute n tasks
- first task T1 requires a time equal kt_p to complete its operation since there are k segments in the pipe .
- Remaining n-1 tasks emerge from the pipe at the rate of one task per clock cycle and they will complete after a time equal to (n-1)tp.
- Therefore to complete n task using k-segement pipeline requires
  K+(n-1) clock cycle
- Example 4 segment , 6task
  time required to complete op. 4+(6-1)=9
  clock cycle

Cont.

- For nonpipeline unit that perform the same operation and takes a time equal to t_n to complete each [redacted] task.
- The total time required for n tasks $=nt_n$
- Speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio
- $S=nt_n / (K+n-1)t_p$
- As the number of tasks increases , n beomes larger the k-1, and k+n-1 approaches the value of n under this condition ,the speedup becomes $S=t_n /t_p$
- If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuit, tn=ktp
- Including the assumption speedup reduces to $S=Kt_p/t_p=K$
- This shows that the theoretical max. speedup that a pipeline can provide is k, where k is the no. of segment in the pipeline

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline
- subopertion in each stage; $t_p = 20nS$
- 100 tasks to be executed
- 1 task in non-pipelined system; $20*4 = 80nS$

Pipelined System

$$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$$

Non-Pipelined System

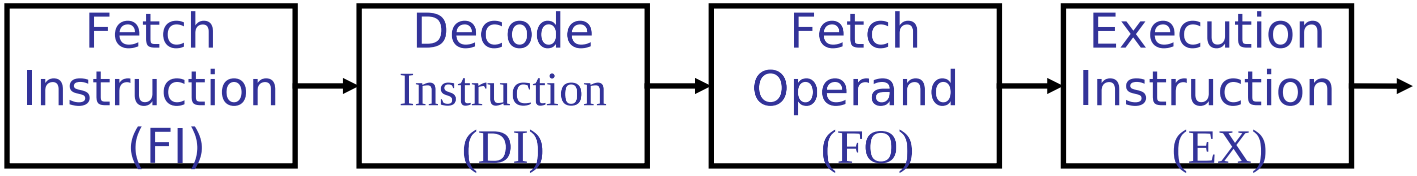
$$tn = n*k*t_p = 100 * 80 = 8000nS$$

Speedup

$$S_k = 8000 / 2060 = 3.88$$

4-Stage Pipeline is basically identical to the system with 4 identical function units

4-Stage instruction Pipelining



- Fetch instruction (FI)
- Decode instruction (DI)
- Fetch operands (FO)
- Execute instructions and store result (EX)

Clock Cycle

Instruction

	1	2	3	4	5	6	7
1	FI	DI	FO	EX			
2		FI	DI	FO	EX		
3			FI	DI	FO	EX	
4				FI	DI	FO	EX

Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated cycle

- There are three classes of hazards
  - Structural hazard
  - Data hazard
  - Branch hazard

Pipeline Hazards

- Structural hazard
  - Resource conflicts when the hardware cannot support all possible combination of instructions simultaneously
- Data hazard
  - An instruction depends on the results of a previous instruction
- Branch hazard
  - Instructions that change the value of PC

Structural hazard

- It is caused by access to memory by two segment of pipeline at the same time.
- Some pipeline processors have shared a single-memory pipeline for data and instructions.

Structural hazard

- If the EX segment needs to store the result of operation in the data memory, while at the same time FI segment needs to fetch the instruction from memory.
- This can be resolved by using separate instruction and data memories.

Structural hazard

- To solve this hazard, we “stall” the pipeline until the resource is freed
- A stall is commonly called pipeline bubble, since it passes through the pipeline taking space but carry no useful work

Data Dependencies

- Three types of data dependencies defined in terms of how succeeding instruction depends on preceding instruction
  - RAW: Read after Write or Flow dependency
  - WAR: Write after Read or anti dependency
  - WAW: Write after Write or output dependency

Three Generic Data Hazards

- Read After Write (RAW)

  Instr$_J$ tries to read operand before Instr$_I$ writes it.

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

RAW Dependency

- Example program (a) with two instructions
  - i1: load r1, a;
  - i2: add r2, r1,r3;
- Program (b) with two instructions
  - i1: mul r1, r4, r5;
  - i2: add r2, r1, r3;
- Both cases we cannot read in i2 until i1 has completed writing the result

Three Generic Data Hazards

Write After Read (WAR)

Instr$_J$ writes operand ***before*** Instr$_I$ reads it

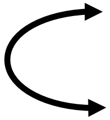
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called as “anti-dependence”

Three Generic Data Hazards

- Write After Write (WAW)

Instr$_J$ writes operand ***before*** Instr$_I$ writes it.



```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called as “output dependence”

WAR and WAW Dependency

- Example program (a):
  - i1: mul r1, r2, r3;
  - i2: add r2, r4, r5;
- Example program (b):
  - i1: mul r1, r2, r3;
  - i2: add r1, r4, r5;
- both cases we have dependence between i1 and i2
  - in (a), r2 must be read before it is written into
  - in (b), r1 must be written by i2 after it has been written by i1

Data hazard

Example:

ADD R1←R2+4

SUB R4←R1-R5

ADD	FI	DI	FO R2=10	EX R1=14	
SUB		FI	DI	FO R1=?	EX

Delayed load

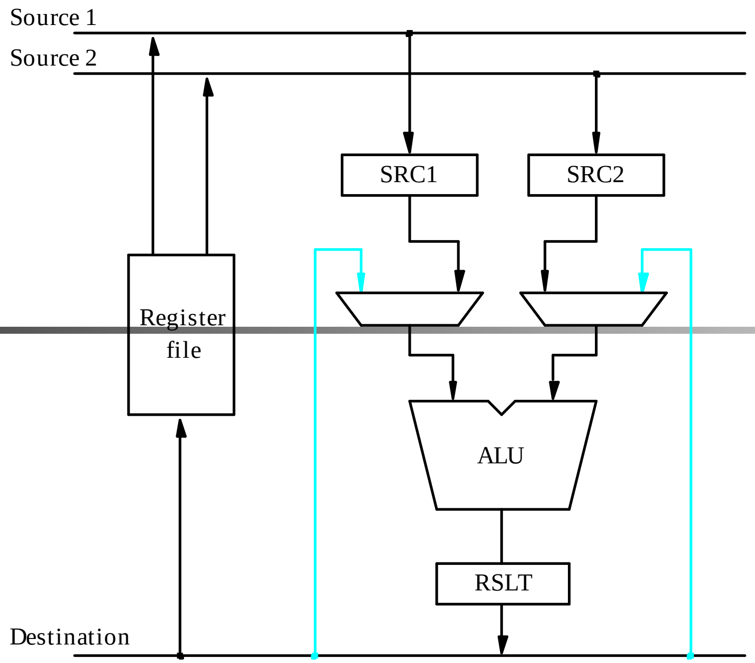
- Delayed load approach inserts a no-operation instruction to avoid the data conflict

ADD R1←R2+R3
NOP
SUB R4←R1-R5

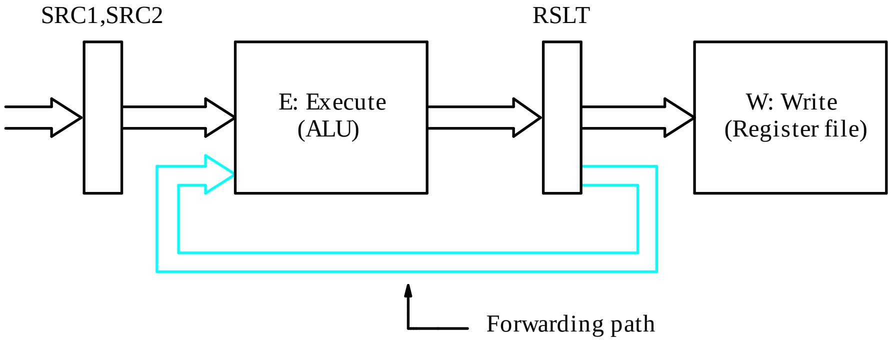
ADD	FI	DI	FO R2=10	EX R1=14		
NOP		FI	DI	FO	EX	
SUB			FI	DI	FO R1=14	EX

Operand Forwarding

- It can be further solved by a simple hardware technique called *forwarding* (also called *bypassing* or *short-circuiting*)
- Control hardware checks the destination operand.
- If it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file.



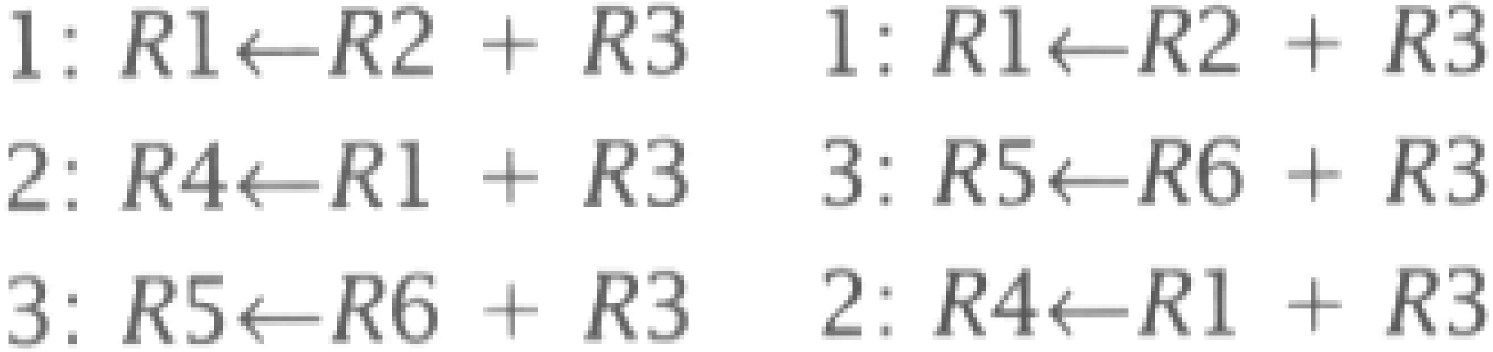
(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Compiler Optimization

- Compiler has to be optimized to reorder instructions in order to avoid problems with data dependencies.



$$
\begin{array}{ll}
1: R1 \leftarrow R2 + R3 & 1: R1 \leftarrow R2 + R3 \\
2: R4 \leftarrow R1 + R3 & 3: R5 \leftarrow R6 + R3 \\
3: R5 \leftarrow R6 + R3 & 2: R4 \leftarrow R1 + R3
\end{array}
$$

Control Hazards

This conflicts arise from branch and other instructions that change the value of PC.

Two types of control hazards.

- **Unconditional branch:-** Always alter the sequential program flow by loading PC with the target address.
    - JMP 2000
- **Conditional branch:-**
    - Control selects the target instruction if condition is satisfied.
    - Control selects the next sequential instruction if condition is not satisfied.
        - JNC 2050

Control Hazards

- 1000: R1<- Load[memory]
- 1001: R3<-R3+ R4
- 1002: JMP 2050
- 1003: R2<- R5+ R6

		1	2	3	4	5	6	7	8	9	10
	1	FI	DI	FO	EX						
	2		FI	DI	FO	EX					
Branch	3			FI	DI	FO	EX				
	4				FI	-	-	FI	DI	FO	EX

Inst from 2050

Inst from 1003

As soon as Instruction 3 is decoded in the DI, transfer from FI to DI of the instruction 4 will be halted.

Branch instruction delays the pipeline until PC is loaded with the branch target address.

Control Hazards

Solutions:

- Delayed Branch
- Branch target buffer (BTB)
- Prefetch target instruction
- Branch Prediction

Delayed Branch

Insert 3 NOP after branch instruction to maintain continuous flow of pipeline. This technique is called delayed branch.

R1<- Load[memory]
R3<-R3+ R4
JMP 2050
NOP
NOP
NOP
INSTRUCTION FROM 2050

	1	2	3	4	5	6	7	8	9	10
1	FI	DI	FO	EX						
2		FI	DI	FO	EX					
Branch			FI	DI	FO	EX				
NOP				FI	DI	FO	EX			
NOP					FI	DI	FO	EX		
NOP						FI	DI	FO	EX	
Inst from 2050							FI	DI	FO	EX

Compiler optimization

(Rearrange the Instruction)

- In this procedure, the compiler detects the branch instruction and rearrange the instruction sequence by inserting useful instructions in the delayed slot to maintain continuous flow of pipeline.

Branch target buffer (BTB)

- BTB is an associative memory
- Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for the branch.
- When pipeline decodes a branch instruction, it searches BTB for target instruction.
  - If found, instruction will be fetched directly from BTB.
- 1000: R1<- Load[memory]
- 1001: R3<-R3+ R4
- 1002: JMP 2050
- 1003: R2<- R5+ R6

Address of Branch	Target Address
1002	2050

Prefetch target instruction

- Prefetch the target instruction in addition to the instruction following the branch.
- Fetch instruction in both path, branch taken and branch not taken.
- Both are saved until the branch is executed.
- Then select instruction from right path and discard the wrong path.

Branch Prediction

- A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed
- The idea is to assign a prediction bit P to the branch instruction when it is first executed.
- Then pipeline begins prefetching the instruction from the predicted path.
- A correct prediction eliminates the wasteful time caused by branch penalties.

Branch Prediction

- I: JNC 2000 <P>
- If P=1 -> prediction is to go 2000
- P predicts whether branch will occur or not.
- When loop iteration is controlled by I, once the loop execution path is entered P predicts that the same loop path will be followed each time I will encountered.
- Misprediction eventually results when loop is exited, but it can be expected to be right most of the time.