CHAPTER:9

INTRODUCTION TO RISC

BY: ER. PRALHAD CHAPAGAIN

(Contact: pralhad.chapagain@gmail.com)

Syllabus:

- RISC Fundamentals, RISC Instruction Set
- Instruction pipeline, Register Windows and renaming
- Conflicts in Instruction Pipeline: Data Conflicts, Branch Conflicts
- RISC vs. CISC

Reduced Instruction Set Computing (1):

- An important aspect of computer architecture is the design of instruction set for the processor.
- The instruction set chosen for a particular computer determines the way, the machine language programs are constructed.
- Many computers have instruction set that include more than 100 and sometimes even more than 300 instructions.
- In 1980s, a number of computer designers recommended that computers use fewer instructions with simple construct so that they can be executed much faster within the CPU.
- This type of computer is classified as reduced instruction set computing(RISC).
  - The greater the number of instructions in an instruction set, the larger the propagation delay.
  - For e.g.; if the CPU had 32 instruction, a 5 x 32 decoder would have been needed. This decoder would require more time to generate output than the smaller 4 x 16 decoder, which would reduce the maximum clock rate of CPU.
- Relatively few instructions.
- Fixed-length instructions:
  - instructions of same size; format can be different.

Reduced Instruction Set Computing (2):

- Limited loading and storing instructions Access memory.
- Fewer Addressing modes.
- Instruction pipelining.
- Large number of registers so as to store many operands internally. When the operands are needed, the CPU fetches them from registers, rather than from memory, thereby reducing the access time.
- Hardwired control unit rather than micro-program control so as to have a lower propagation delay.

RISC Instruction Set (1):

- The instruction set of RISC processor is reduced whereas CISC(Complex Instruction Set Computing) processor might have over 300 instructions in its instruction set, RISC CPU typically have less than 100.
- These instructions perform a wide variety of function, each of which is being executed in a single clock cycle.
- When developing a RISC instruction set, it is important not to reduce the set too much.
- Consider, for example, the instruction set for a processor includes AND, OR, NOT and XOR instruction. Using De-Morgan's law, an OR can be implemented using only AND and NOT. A OR B = NOT((NOT A) AND (NOT B))
- Similarly, an XOR can be realized using the same operation
  A XOR B = NOT ((NOT (A AND (NOT B))) AND (NOT ((NOT A) AND B)))
- Therefore, we can exclude OR and XOR instruction from the instruction set and still allow the CPU to perform the same function.

RISC Instruction Set (2):

- Instruction breakdown for the MIPS 4000 CPU

Instruction Type	Number of instructions
Data Move	15
ALU	16
Multiply/Divide	8
Branch	25
Coprocessor	11
Exception	12
Special	2

Instruction Pipelining (1):

- A pipelining is a series of stages, where some work is done at each stage in parallel.
- The stages are connected one to the next to form a pipe- instruction enter at one end, progress through the stages, and exit at the other end.
- Pipelining is an speed up technique where multiple instructions are overlapped in execution on a processors.
- In RISC processors, one instruction is executed while the next is being decoded and its operands are being loaded, while the following instruction is being fetched.
- By overlapping these operations, the CPU executes one instruction per clock cycle, even though each instruction requires three cycles to be fetched, decoded and executed.

Instruction Pipelining (2):

Different stages on pipelining:

Fetch Instruction:

- Responsible for obtaining the requested instruction from memory. The instruction and the program counter are stored in the register as temporary storage

Decode Instruction:

- Responsible for decoding the instruction and sending out the various control line to the other parts of the processors.

Select Register:

- Responsible for selecting the register where the operand is stored.

Execute Instruction:

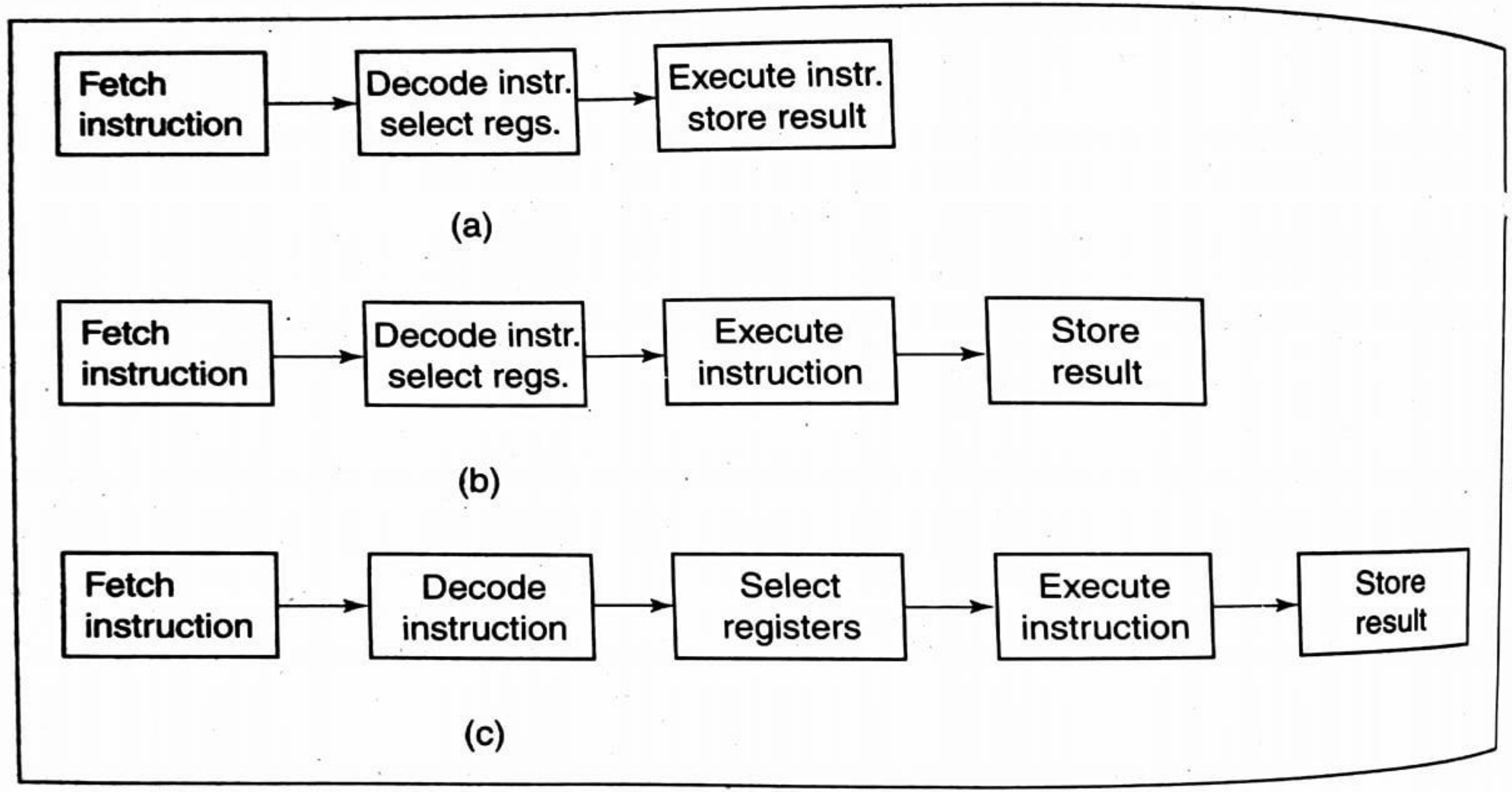
- Responsible for executing the instruction.

Store Result:

- Responsible for writing the result of a calculation, memory access, or input into the register file.

Instruction Pipelining (3):

a) Three , b) four, and c) five stage RISC pipelines



Instruction Pipelining (4):

Data flow through a) three, b) four, and c) five stage RISC pipelines

Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5

(a)

Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5
4	–	–	–	I1	I2	I3	I4

(b)

Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5
4	–	–	–	I1	I2	I3	I4
5	–	–	–	–	I1	I2	I3

(c)

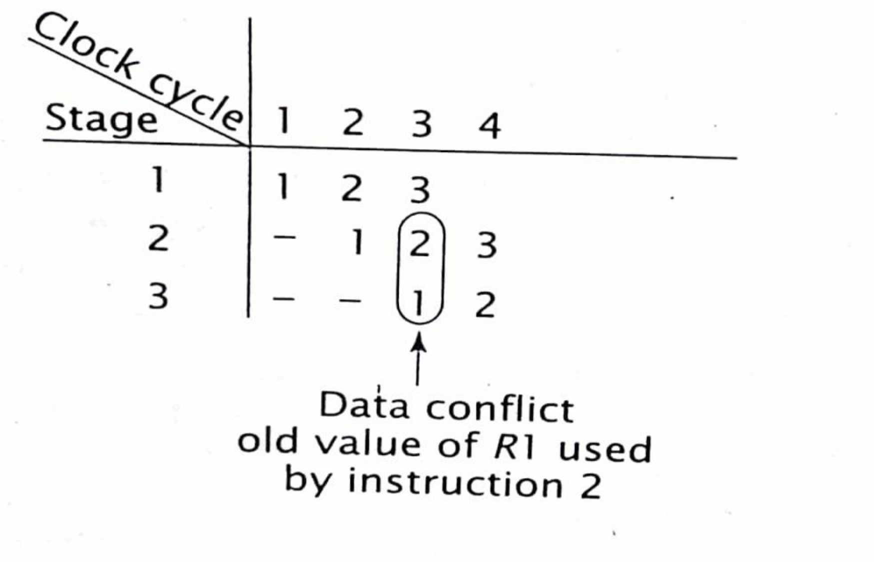
Conflicts in Instruction Pipeline (1):

- A pipeline conflicts occurs when the instruction pipeline deviates at some phase, some operational conditions that do not permit the continued execution.
- The conflicts caused by pipelines fall into two categories:
  - Data conflicts
  - Branch conflicts

Data Conflicts:

- Data Conflicts occur when the pipeline causes an incorrect data value to be used.
- Consider the following consecutive program statements:

  1: R1←R2 + R3
  2: R4←R1 + R3
  3: R5←R6 + R3



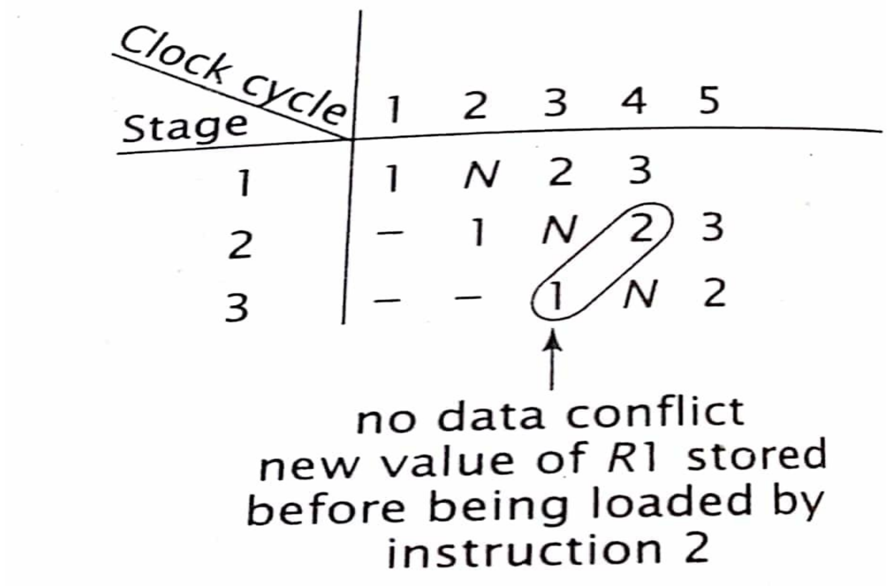
Solution to Data Conflicts Problem (1):

No-op insertion:

- The simplest is to have the compiler detect data conflicts and insert no-ops to avoid the conflict.

  1: R1←R2 + R3
  N: no-op
  2: R4←R1 + R3
  3: R5←R6 + R3

- The no-op statement in the second block delays the fetching of operands for the second instruction by one clock cycle.
- This delay allows the last stage of the pipeline to store the new value of R1 before it is loaded for use in executing the second instruction, thus avoiding data conflict.
- It reduces the overall system performance.
- The no-op instruction does not perform useful work and require an extra clock cycle



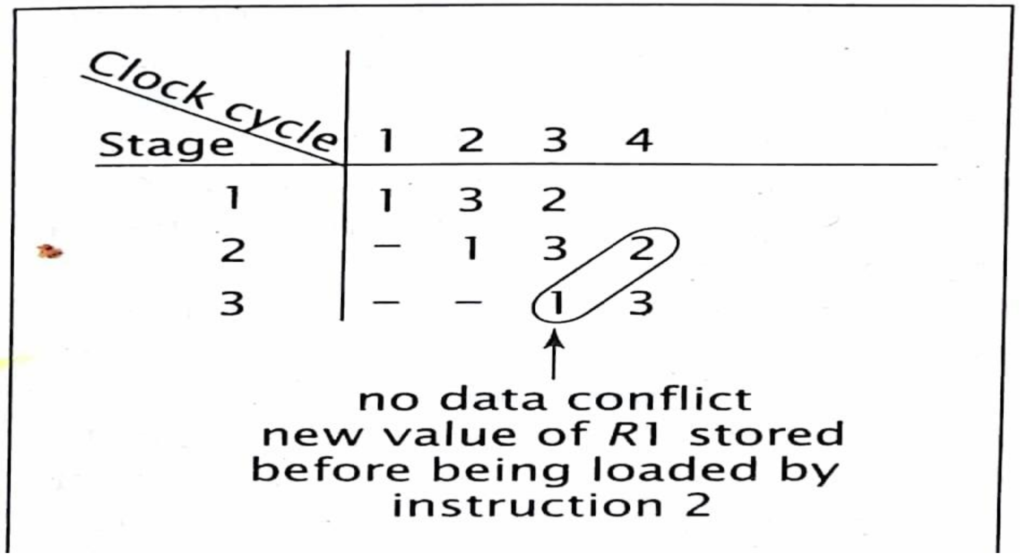
Solution to Data Conflicts Problem (2):

Instruction Re-Ordering:

- For some programs, the compiler can reorder some of the instructions to remove the data conflict.

  1: R1←R2 + R3
  3: R5←R6 + R3
  2: R4←R1 + R3

- Instruction reordering resolves the data conflict by introducing a delay between the conflicting instructions.



- Unlike no-op insertion, instruction reordering performs useful work during this time.
- It is not always possible to reorder the instructions of a program to avoid data conflicts.
- For example, the following code segment cannot be reordered successfully:

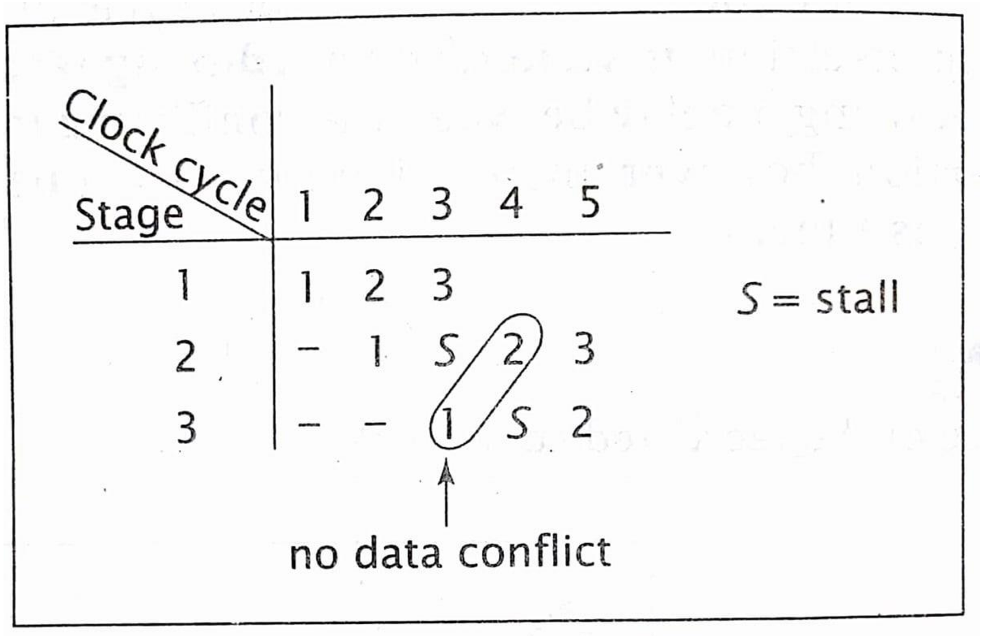
  1: R1←R1 + R2
  2: R1←R1 + R3
  3: R1←R1 + R4

- No-op insertion and instruction reordering resolve data conflicts using only the compiler.

Solution to Data Conflicts Problem (3):

Stall insertion:

- Use additional hardware in the RISC instruction pipeline.
- The additional hardware detects data conflicts between instructions in the pipeline and inserts stalls, or introduces delays, to resolve the data conflicts.
- This is similar to no-op insertion, except it is handled by hardware, while the program is executing, rather than by the compiler while the program is being compiled.

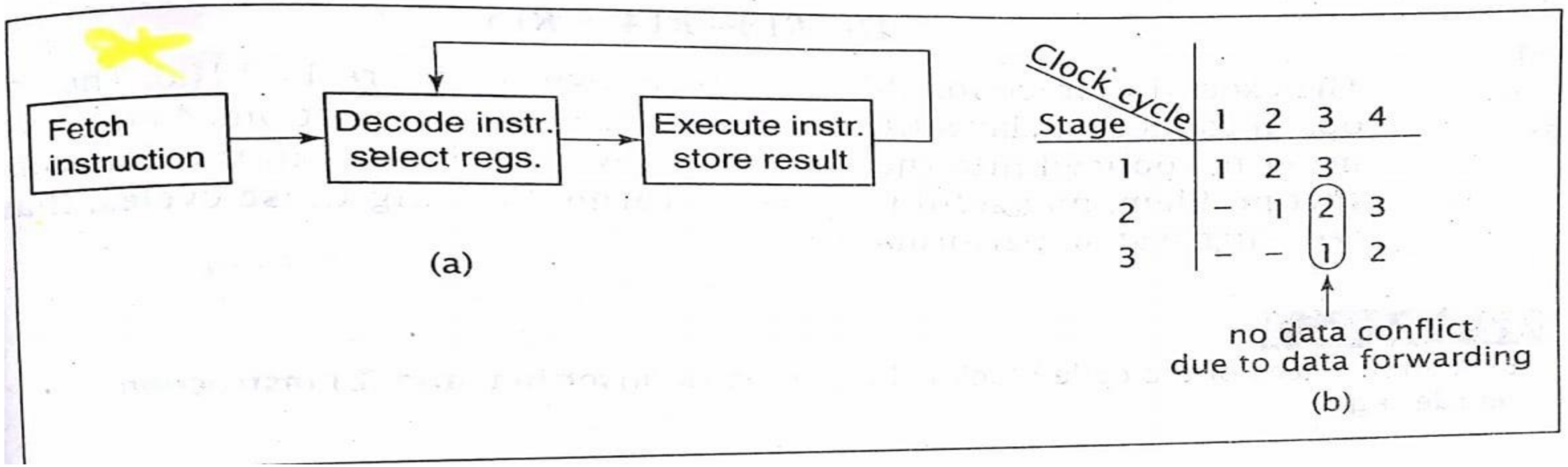


Solution to Data Conflicts Problem (4):

Data forwarding:

- Hardware solution to the data conflict problem.
- After the instruction is executed, its result is stored just as before, but the result is also forwarded directly to the stage that selects registers(retrieves operands).
- That stage gets the new value of the operand directly from the execute instruction stage of the pipeline before(or at the same time as) it is stored in the appropriate register.

Data forwarding: (a) modified three-stage pipeline and (b) execution trace of the code block



Conflicts in Instruction Pipeline (2):

Branch conflicts:

- The following code segment illustrates the branch conflict problem:

  1: R1←R2 + R3
  2: R4←R5 + R6
  3: JUMP 10
  4: R7←R8 + R9
  5: R10←R11 + R12
  : : :
  10: R13←R14 + R15

- After instruction 3 is executed, the CPU should branch to instruction 10; however, instructions 4 and 5 are already in the pipeline before instruction 3 executed.

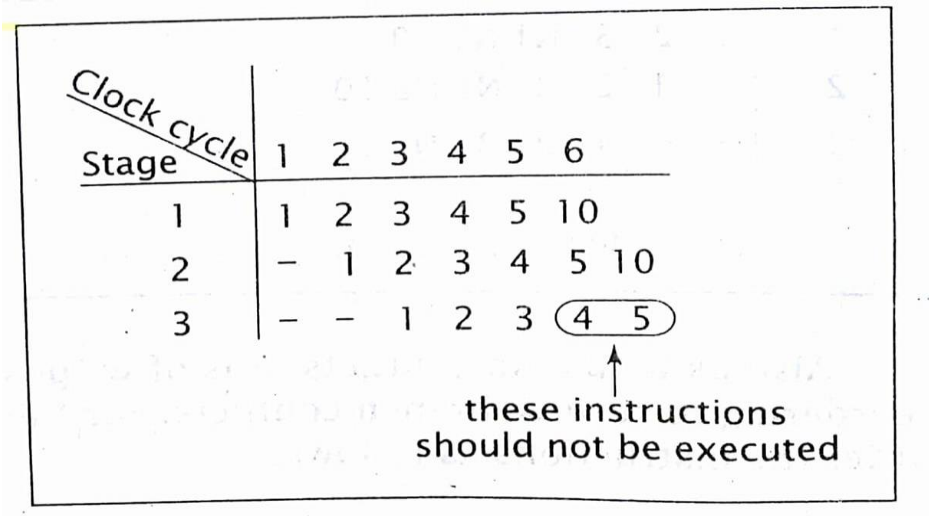


Fig: execution trace of the code block illustrating a branch conflict

Solution to Branch Conflicts Problem (1):

((consider the simpler case of unconditional branch instructions))

No-op insertion:

1:	R1←R2 + R3
2:	R4←R5 + R6
3:	JUMP 10
N1:	no-op
N2:	no-op
4:	R7←R8 + R9
5:	R10←R11 + R12
	:
10:	R13←R14 + R15

- The no-ops introduce a delay sufficient to ensure that instructions 4 and 5 are never introduced into the pipeline
- **Instruction reordering:**

3:	JUMP 10
1:	R1←R2 + R3
2:	R4←R5 + R6
4:	R7←R8 + R9
5:	R10←R11 + R12
	:
10:	R13←R14 + R15

Stage \ Clock cycle	1	2	3	4	5	6	
1	1	2	3	N1	N2	10	
2	—	1	2	3	N1	N2	10
3	—	—	1	2	3	N1	

(a)

Stage \ Clock cycle	1	2	3	4	5
1	3	1	2	10	
2	—	3	1		
3	—	—	3		

(b)

Fig: execution traces of the code block using a) no-op insertion b) Instruction reordering

Solution to Branch Conflicts Problem (2):

Stall insertion can also be used to handle branch conflicts.

- When the pipeline recognized a branch instruction, it inserts stalls into the pipeline to delay the fetching of next instruction until after the branch instruction had been completed.

Annulling (1):

- In annulling, the instructions proceed through the pipeline as they normally would.
- If an instruction should not have been executed, because a previous instruction branched away from it, its result are not stored.
- Even though it might have been executed, as long as no results are stored, it is as if the instruction was never processed.

1:	$R10 \leftarrow 10$
2:	$R1 \leftarrow R1 + R3$
3:	$R2 \leftarrow R2 + R3$
4:	$R10 \leftarrow R10 - 1$
5:	IF($R10 \neq 0$) THEN GOTO 2
6:	$R4 \leftarrow R5 + R6$
7:	$R7 \leftarrow R8 + R9$

Solution to Branch Conflicts Problem (2):

Annulling (2):

- The statements 5, 6, and 7 are all in the pipeline during clock cycle 7, even though statement 5 should be followed by statement 2.
- The execution of statements 6 and 7 are annulled by the pipeline hardware, which knows that the branch in statement 5 is taken.
- During clock cycle 61, the branch is not taken and the loop terminates.
- This time, the execution of statements 6 and 7 during the following two clock cycles are not annulled and their results are stored.

Stage \ Clock cycle	1	2	3	4	5	6	7	8	9	10	• • •	59	60	61	62	63
1	1	2	3	4	5	6	7	2	3	4	• • •	5	6	7	8	9
2	–	1	2	3	4	5	6	7	2	3	• • •	4	5	6	7	8
3	–	–	1	2	3	4	5	6	7	2	• • •	3	4	5	6	7

branch taken annulled branch not taken not annulled

Solution to Branch Conflicts Problem (3):

Branch Prediction:

- Allows the compiler or pipeline hardware to make an assumption as to whether or not the conditional branch will be taken.
- If its guess is right, the correct next instruction occurs immediately after the conditional branch instruction and is executed during the next clock cycle; no delay is introduced.
- If the guess is wrong, the results are annulled before.

Stage \ Clock cycle	1	2	3	4	5	6	7	8	• • •	41	42	43	44	45	46	47
1	1	2	3	4	5	2	3	4		5	2	3	6	7	8	9
2	–	1	2	3	4	5	2	3		4	5	2	3	6	7	8
3	–	–	1	2	3	4	5	2		3	4	5	2	3	6	7

assumption correct

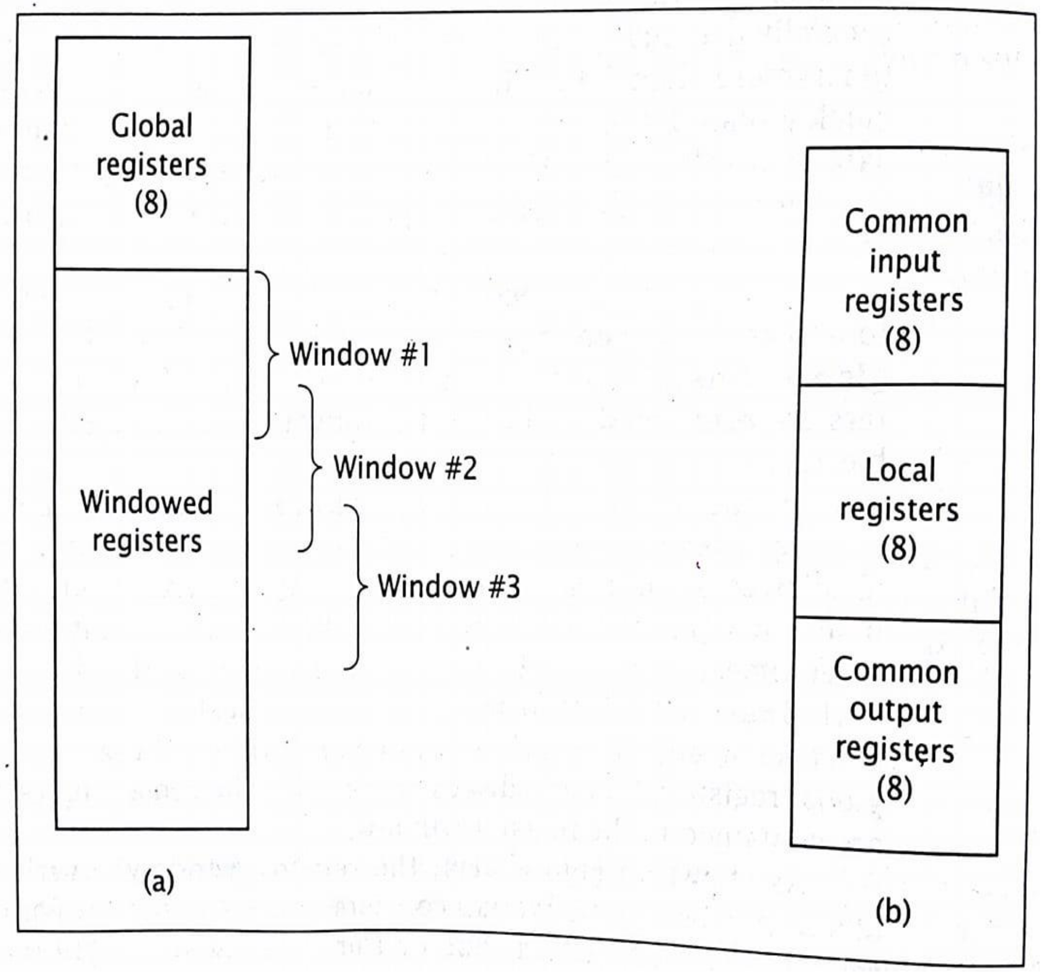
assumption incorrect

annulled

Register Windows and Renaming (1):

- The reduced hardware requirement of RISC processor leave additional space available on the chip for the system designer.
- RISC CPU generally use this space to include a large number of registers, sometimes more than 100.
- The CPU can access data in register more quickly than data in memory.
- Although, a RISC processor has many register, it may not be able to access all of them at any time.
- Most RISC CPU has some global register which are always accessible.
- The remaining register are windowed such that only a subset of the registers are accessible at any specific time.

Register windowing in the SPARC processor

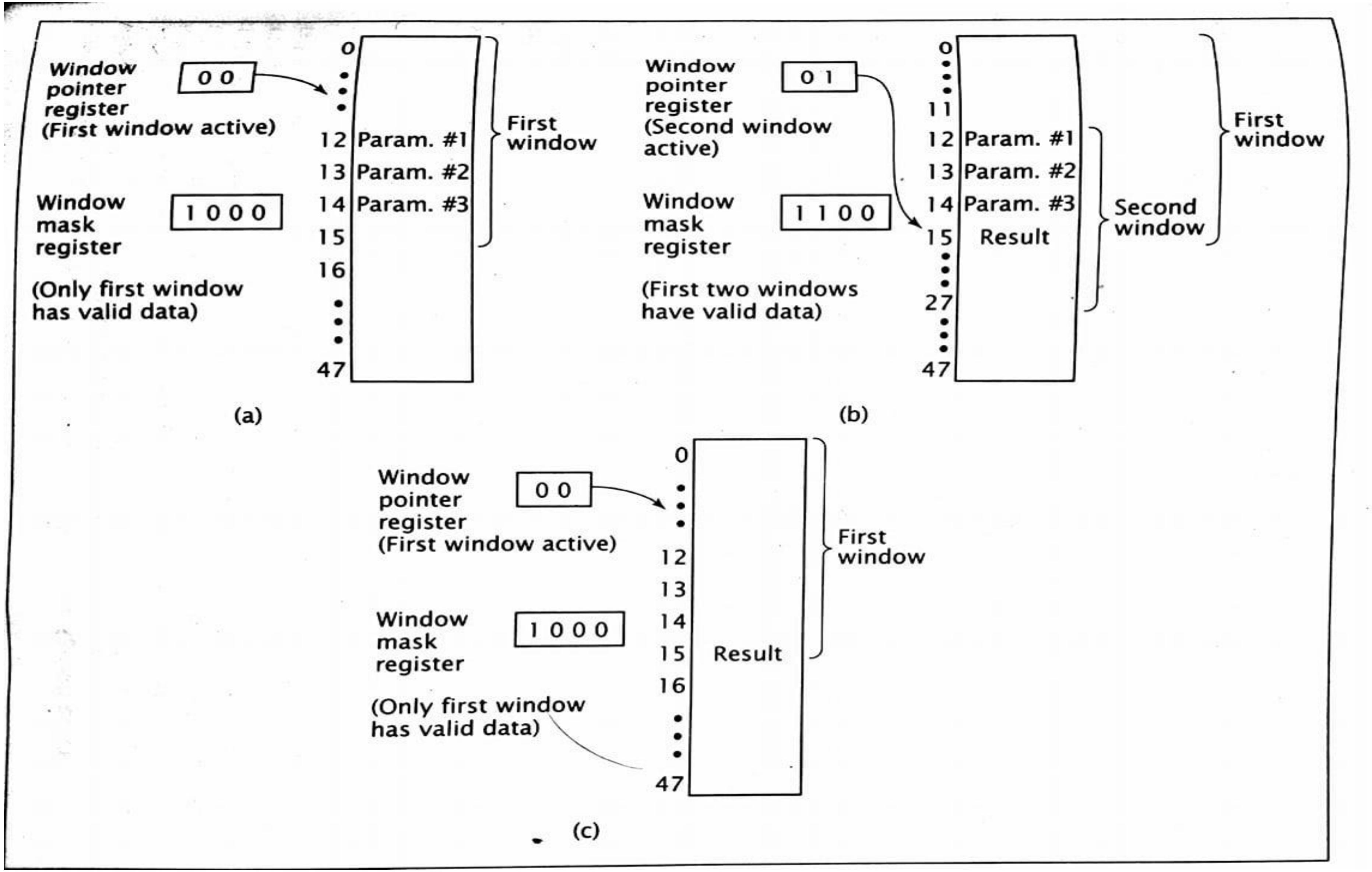


Register Windows and Renaming (3):

Illustration:

- Consider a CPU with 48 registers.
- The CPU has four windows with 16 registers each, and an overlap of four registers between windows.
- Initially, CPU is running a program using its first register window.
- It must call a subroutine and pass three parameters to it.
- The CPU stores these parameters in three of the four overlapping registers and calls subroutine.
- The subroutine can directly access these parameters.
- Subroutine calculates a result, stores the value in one of the overlapping registers and returns to main program.
- This deactivates the second window; the CPU now works with the first window and can directly access the result.
- A **window pointer register** contains the value of the window that is currently active.
- A **window mask register** contains 1 bit per window and denotes which windows contain valid data.

Register Windows and Renaming (4):



Register windowing in a CPU: (a) during execution of the main routine, (b) executing a subroutine, and (c) after returning from the subroutine

<u>Register Windows and Renaming (5):</u>

<u>Register Renaming</u>:

- More recent processors may use register renaming to add flexibility to the idea of register windowing.
- A processor that uses register renaming can select any register to comprise its working register window.
- The CPU uses pointers to keep track of which registers are active and which physical register correspond to each logical register.
- Unlike register windowing, in which only specific group of physical registers are active at any given time, register renaming allows any group of physical registers to be active.

(Q).**Calculate the window size and total number of registers. Given: No. of Global registers = 10, No. of local registers = 10, No. of common registers = 6, No. of windows = 4.**

- Total number of registers = **G + W(L +C)** = 10 + 4(10 + 6) = 74
- Windows size **= L + 2C + G** = 10 + 2 x 6 + 10 = 32
- Where, G = no. of global registers
  W= no of windows
  L=no of local registers
  C=no of common registers

RISC vs. CISC (1)

RISC	CISC
Multiple register sets, often consisting of more than 256 registers	Single register set, typically 6 to 16 registers total
Three register operands allowed per instruction (e.g., `add R1, R2, R3`)	One or two register operands allowed per instruction (e.g., `add R1, R2`)
Parameter passing through efficient on-chip register windows	Parameter passing through inefficient off-chip memory
Single-cycle instructions (except for `load` and `store`)	Multiple-cycle instructions
Hardwired control	Microprogrammed control

RISC vs. CISC (2)

RISC	CISC
Highly pipelined	Less pipelined
Simple instructions that are few in number	Many complex instructions
Fixed length instructions	Variable length instructions
Complexity in compiler	Complexity in microcode
Only load and store instructions can access memory	Many instructions can access memory
Few addressing modes	Many addressing modes

ASSIGNMENT QUESTIONS: (from old question set):

1. *Differentiate between RISC and CISC processors. Mention major conflicts occurring due to instruction pipelining in RISC, also explain remedies to those conflicts.*
2. *Illustrate with example how branch and data conflicts occurs? List out the solutions to the data conflicts.*
3. *Show how instruction pipelining can improve the performance of system. Explain the different instruction pipelining conflicts.*
4. *Describe register windows. In a system, there are 8 windows of registers, each register share 8 input registers, and 8 output registers. Each of the windows has 4 local registers. The system has 10 global registers. Calculate the total no of registers in the system. Show the pictorial representation as well.*
5. *Describe register windows. In a system, there are 5 windows of registers, each register share 4 input registers, and 4 output registers. Each of the windows has 10 local registers. The system has 20 global registers. Calculate the total no of registers in the system. Show the pictorial representation as well.*
6. *Explain major sets of design principles related to RISC architecture. Calculate the window size and total number of registers. Given: No. of global register=10, no of local register=10, no of common register =6, no of windows = 4*
7. *Write short Notes:*
    1. *Register windows*

CHAPTER COMPLETED