Chapter 9

Reduced Instruction Set Code (RISC)

Assistant Professor

Er. Shiva Ram Dam

Contents:

1. RISC Fundamentals
2. RISC Instruction set
3. Instruction Pipeline
4. Register Windows and Renaming
5. Conflicts in Instruction Pipeline: Data Conflicts, Branch Conflicts
6. RISC Vs CISC

9.1 RISC Fundamentals

- Reduced Instruction set computers
- This architecture contains less no of instructions than that of CISC.
- Reduces the cycles per instruction at the cost of the number of instructions per program.
- The no of registers in RISC is usually 32 or more.
- The First RISC CPU (MIPS 2000) had 32 general-purpose registers.
- They are processors that use a small instruction set and simple addressing mode so that their instructions can be executed much faster within the CPU with less referring to the memory. This type of processor is classified as a reduced instruction set computer(RISC).

Features of RISC

1. **Fixed Length instructions.**
   - Instructions are of the same size (say 8 bit), unlike CISC, where instructions are of variable length.
   - This reduces decoding overhead and saves time.
2. **Limited interaction with memory.**
   - Since memory is slower, RISC CPUs are designed not to interact extensively with memory.
   - Thus, RISC processors have limited interaction with memory to load and store data.
3. **Fewer addressing modes.**
   - Fewer no of instruction = fewer no of addressing modes. And vice versa.
4. **Instruction pipeline**
   - Pipelining can be achieved.
   - Pipelining is to fetch another instruction side by side while current instruction is being decoded.
5. **Large no of registers.**
   - RISC architecture has a larger number of registers to reduce the use of interaction with memory.
   - Large no of registers also means less access time and greater speed.
6. **Hardwired control unit**
   - RISC CPUs use hardwired control units.
   - The fact that the hardwired control unit can run at a greater clock frequency than that of the micro-coded control unit(used by CISC CPUs).
7. **Fewer Addressing modes**

9.2 RISC Instruction Set

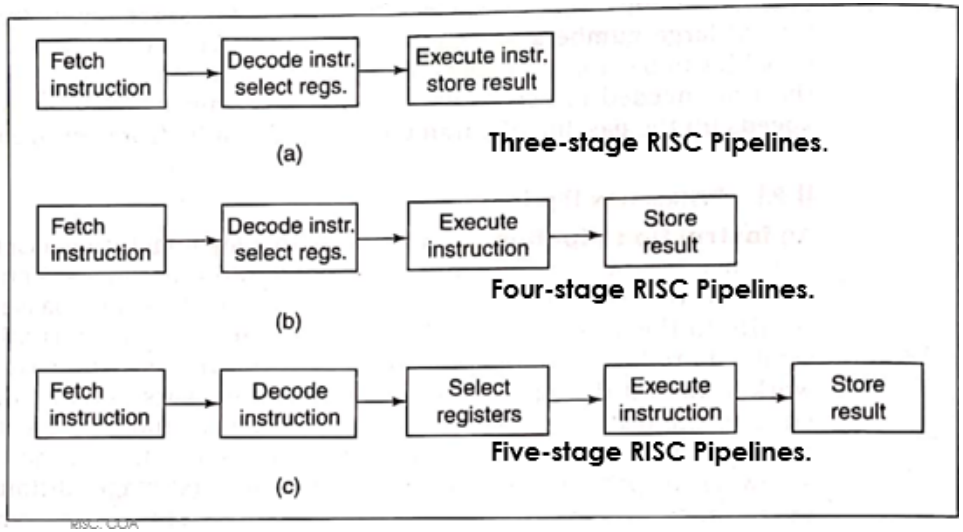
- All the basic types of instructions must be represented in the RISC instruction set, to perform different functions.
- The basic instructions are
  - Data movement(load, store, register move)
  - Arithmetic
  - Logical
  - Branch
  - And Shift
- When developing a RISC instruction set, it is important not to reduce the set too much.
- CISC CPUs might have over 300 instructions in their instruction set, RISC CPUs has typically less than 100 instructions.

9.2 RISC Instruction Set

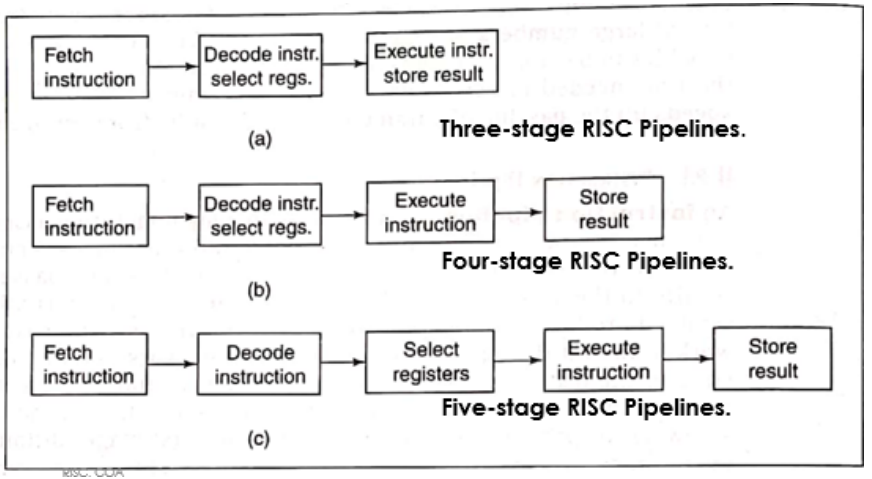
- All the basic types of instructions must be represented in the RISC instruction set, to perform different functions.
- The basic instructions are
  - Data movement(load, store, register move)
  - Arithmetic
  - Logical
  - Branch
  - And Shift
- When developing a RISC instruction set, it is important not to reduce the set too much.
- CISC CPUs might have over 300 instructions in their instruction set, RISC CPUs has typically less than 100 instructions.

9.3 Instruction Pipeline

- A pipeline is like an assembly line in which many products are being worked on simultaneously, each at a different station.
- 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.



9.3 Instruction Pipeline



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)

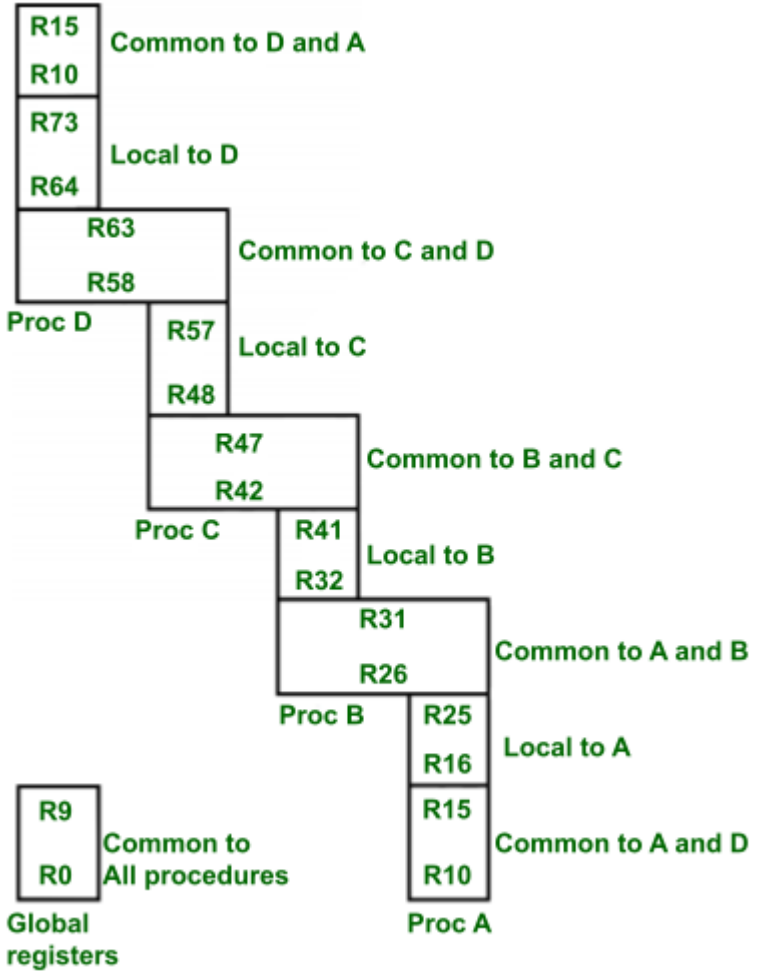
Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5
4	–	–	–	I1	I2	I3	I4
5	–	–	–	–	I1	I2	I3

(c)

Figure: Data flow through Three, Four, and Five stage RISC pipelines

9.4 Register Organization in RISC CPU

- The characteristics of some RISC CPUs is to use an overlapped register window that provides passing of parameters to called procedure and stores the result to the calling procedure.
- For each procedure call, the new register window is assigned from register file used by the new procedure.
- Each procedure call activates the new register window by increment a pointer and the return statement decrements the pointer which causes the activation of the previous window.
- Windows of adjacent procedures have overlapping registers that are shared to provide passing of parameters and storage of results.



9.4 Register Organization in RISC CPU (Contd.)

- In this organization, the RISC CPU contains 74 registers. Registers R0 to R9 are global registers that contain parameters which are common to all procedures.
- The remaining registers(R10 to R73) are divided into four windows to contain procedures A, B, C and D.
- Each window contains 10 local registers and two sets of 6 registers common to consecutive windows. Local registers contain local variables and common overlapped registers help to pass parameters between adjacent procedures without actual movement of data.
- One of the register window is activated at a time. The high registers of the calling procedure overlap the low registers of the called procedures, therefore parameters are passed from calling to called procedure easily.

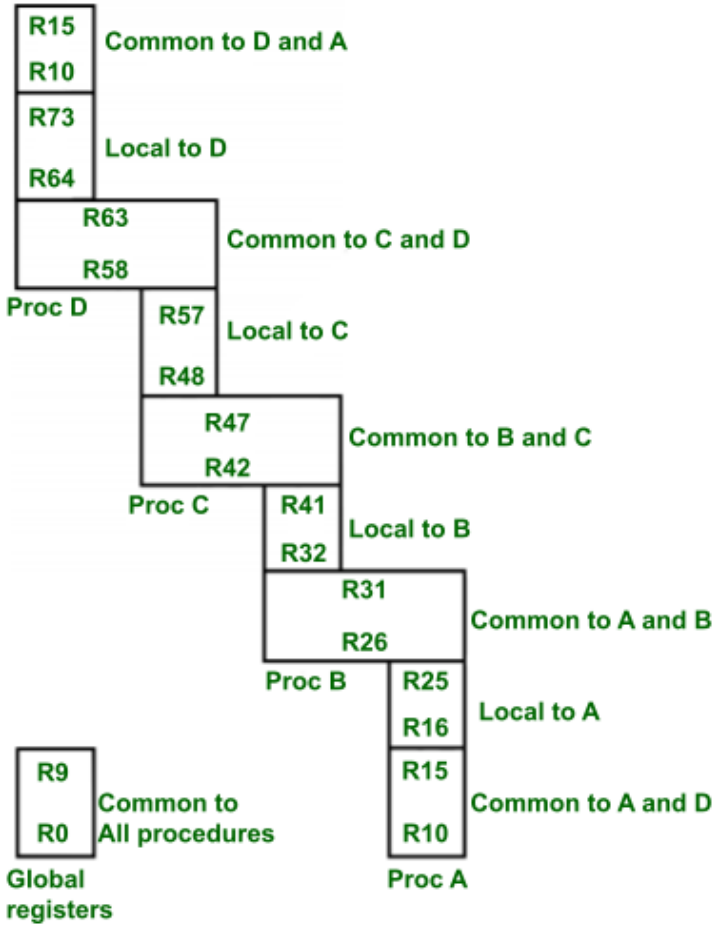
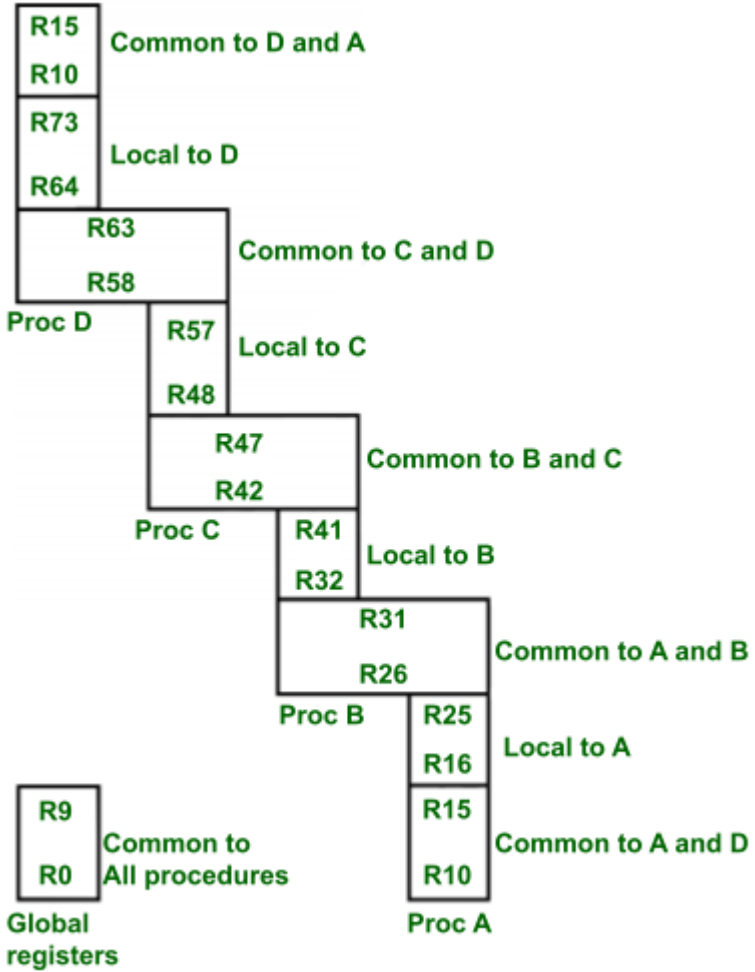


Figure: Overlapped register window of RISC CPU

9.4 Register Organization in RISC CPU (Contd.)

- The organization of register windows has the relationship as
  - Number of global registers = G
  - Number of local registers present in each window = L
  - Number of registers common to two adjacent windows = C
  - Number of windows = W
- Then the number of registers available for each window is calculated by :
  - Window size (S)= L+2C+G
  - The total number of registers required in the processor is
  - Register File = (L+C)W + G

- **Here:**
  - In the above figure we have g= 10, l= 10, c= 6, w= 4, then
  - the window size = 10+2×6+4 = 36
  - register file size = (10+6)×4 + 10 = 74



Numerical:

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 number of registers in the system. Show the pictorial representation as well. [PU 2018 Spring]

- Here:
  - Number of global registers (G) =10
  - Number of local registers present in each window (L)= 4
  - Number of registers common to two adjacent windows (C) = 8+8 = 16
  - Number of windows (W)= 8
- We have:
  - Window size (S)= L+2C+G =4+2*16+10 =46
  - The total number of registers required in the processor is
  - Register File = (L+C)W + G =(4+16) *8+10= 170

Draw figure here as in the previous slide 11

Reference: https://www.youtube.com/watch?v=GPIgi-OIe_g

9.5 Register Windowing

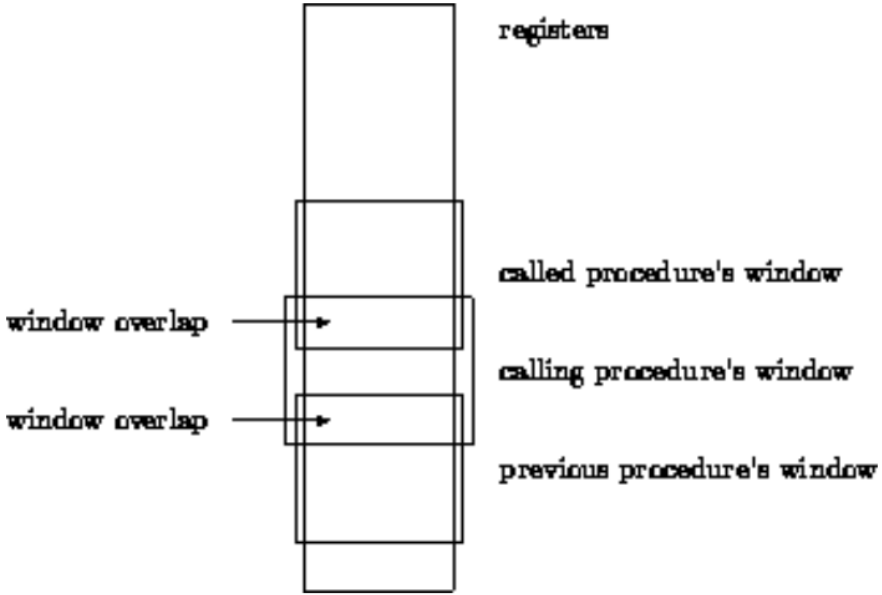
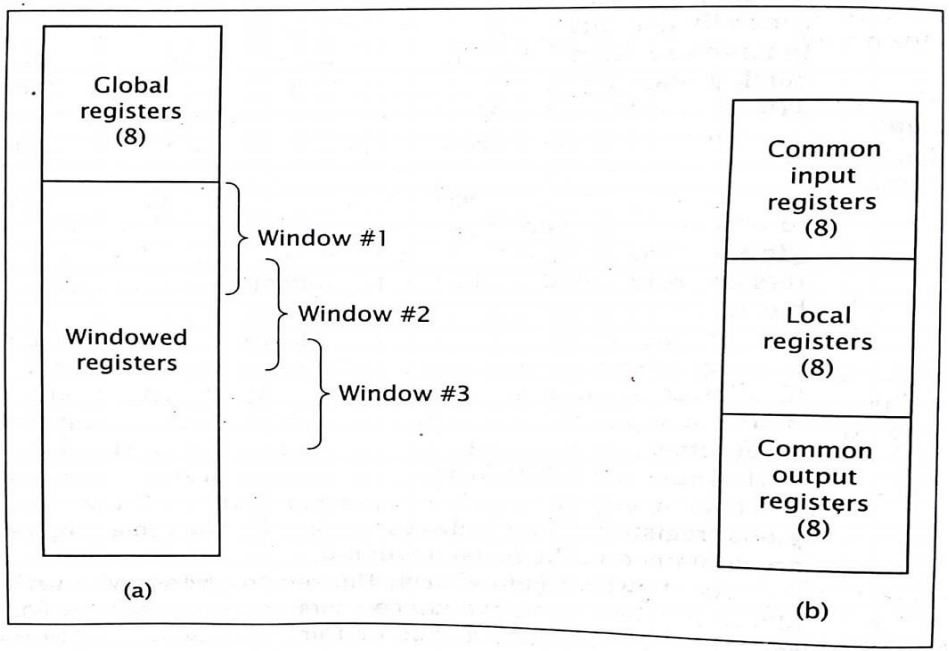


Figure: Overlapping register windows

9.5 Register Windowing (Contd.)

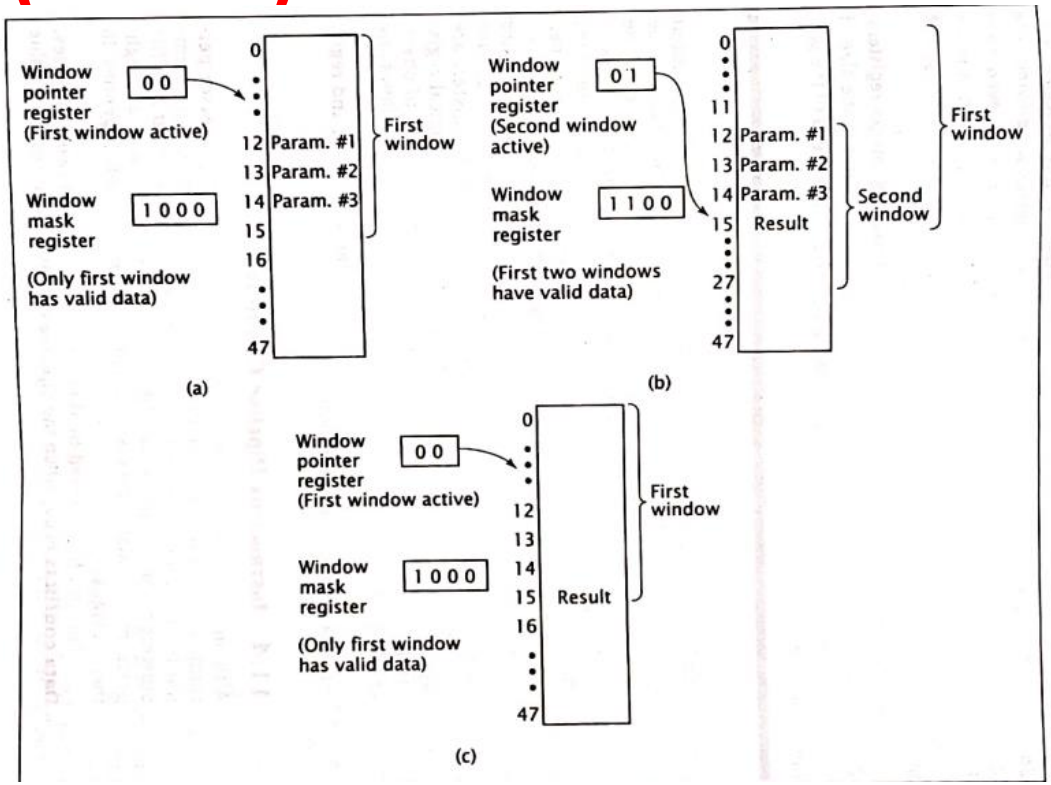
- The reduced hardware requirement of RISC processor leave additional space available on the chip for the system designer.
- RISC CPU generally use this space to include a large number of registers, sometimes more than 100.
- The CPU can access data in register more quickly than data in memory.
- Although, 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 Windowing (Contd.)

- Consider a CPU with 48 registers.
- The CPU has four windows with 16 registers each, and overlap of four registers between windows.
- Initially, the 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 the main program.
- This deactivates the second window; the CPU now works with the first window and can directly access the result.



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

Register Renaming

- More recent processors may use register renaming **to add flexibility** to the idea of register renaming.
- 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 corresponds to each logical register.
- Unlike register windowing, in which only specific groups of physical registers are active at any given time, register renaming allows any groups of physical registers to be active.

9.6 Conflicts in Instruction Pipeline

- Instructing pipelining of course improves the overall system performance but it doesn't come without hazards.
- A hazard is a potential problem that occurs while CPU tries to simultaneously execute multiple instructions which exhibit data dependence.
- Types of Conflict/ Hazard
  - Data Conflict
  - Structural Conflict
  - Branch Conflict

1. Data Conflict

- A data conflict occurs within a RISC pipeline when one instruction stores a result in a register and another instruction uses that value as an operand.
- Under certain condition, the second instruction will read the old value before the first instruction has stored the new value.
- Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline.
- Hazard cause delays in the pipeline.
- There are mainly three types of data hazards:
    1. RAW (Read after Write) [Flow/True data dependency]
    2. WAR (Write after Read) [Anti-Data dependency]
    3. WAW (Write after Write) [Output data dependency]
- WAR and WAW hazards occur during the out-of-order execution of the instructions.

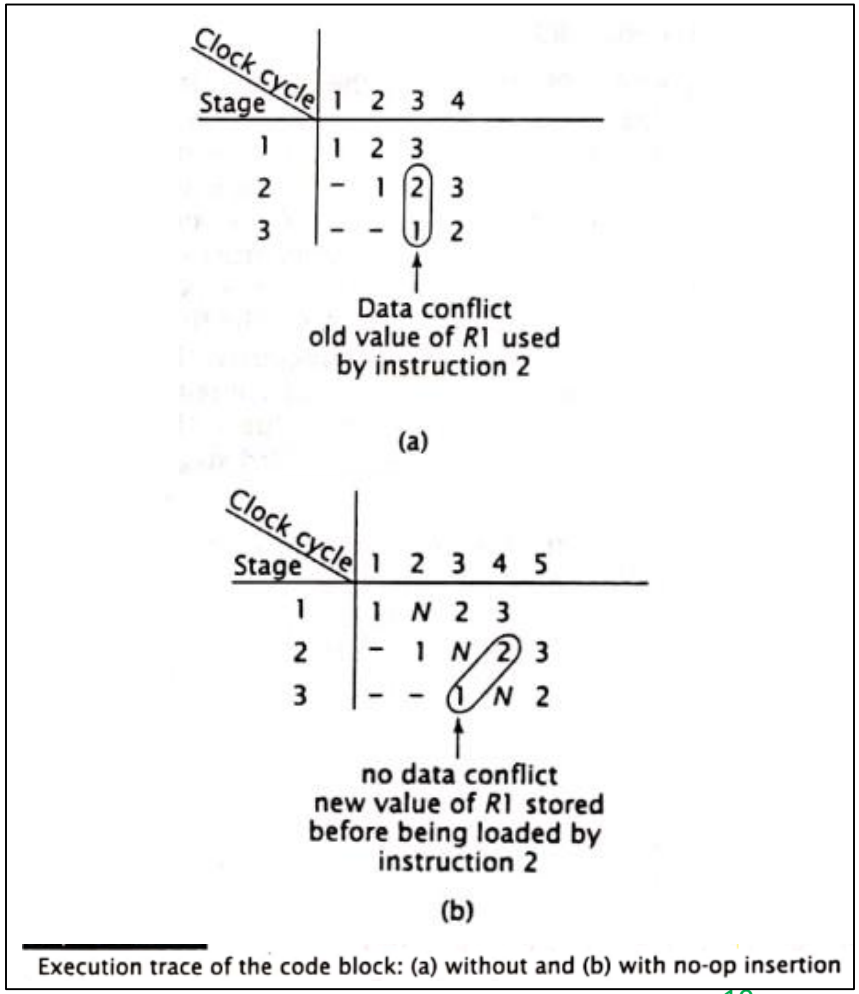
Solution to Data Conflict

- Consider the following consecutive program statements. Let registers R1, R2, R3, R4, R5 and R6 contain values 1, 2, 3, 4, 5 and 6 respectively.

1: $R1 \leftarrow R2 + R3$
2: $R4 \leftarrow R1 + R3$
3: $R5 \leftarrow R6 + R3$

- One of the simplest solution is to have the compiler detect data conflict and **insert no-ops** to avoid the conflict. The no-op statement in the second block delays the fetching of operands for the second instruction by one clock cycle.

1: $R1 \leftarrow R2 + R3$
N: *no-op*
2: $R4 \leftarrow R1 + R3$
3: $R5 \leftarrow R6 + R3$



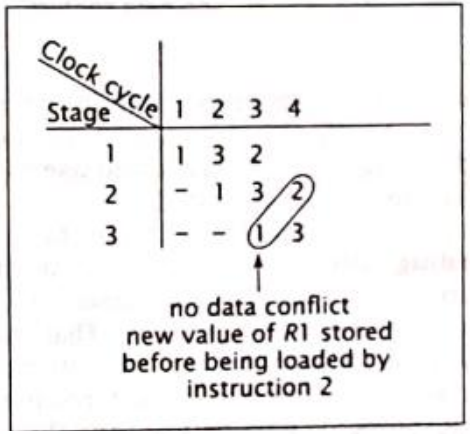
Execution trace of the code block: (a) without and (b) with no-op insertion

Solution to Data Conflict

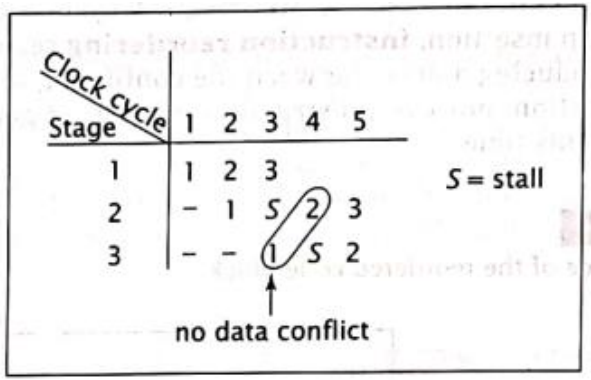
- Although no-op insertion solves the data conflict problem, it reduces overall system performance.
- It spoils an extra cycle. Thus, not optimal solution.
- Next solution can be **reordering of the program** instructions as:

1: $R1 \leftarrow R2 + R3$
3: $R5 \leftarrow R6 + R3$
2: $R4 \leftarrow R1 + R3$

- This is not always possible and feasible.
- Another solution is to insert Stall. In this method, the additional hardware detects data conflicts between instructions in the pipeline and inserts stalls and introduces delays.
- The optimal solution to data conflict is **Data forwarding.** 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.



Execution trace of the reordered code block



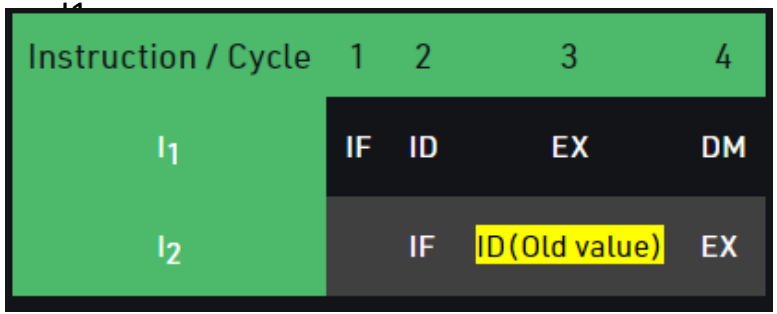
Execution trace of the code block with instruction stalls

Data Conflict: Optimal Solution

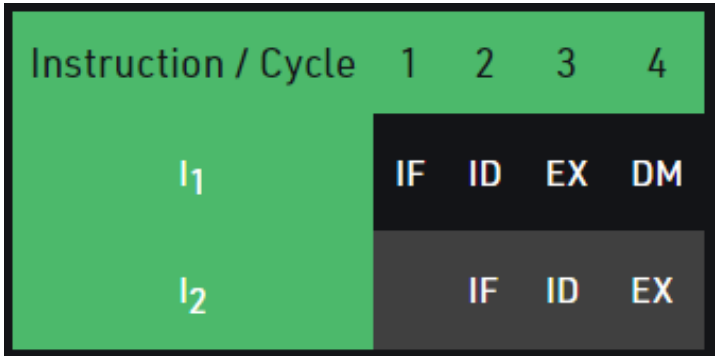
- Example: Let there be two instructions I1 and I2 such that:

  I1 : ADD R1, R2, R3
  I2 : SUB R4, R1, R2

- When the above instructions are executed in a pipelined processor, then data dependency condition will occur,
  - which means that I2 tries to read the data before I1 writes it,
  - therefore, I2 incorrectly gets the old value from

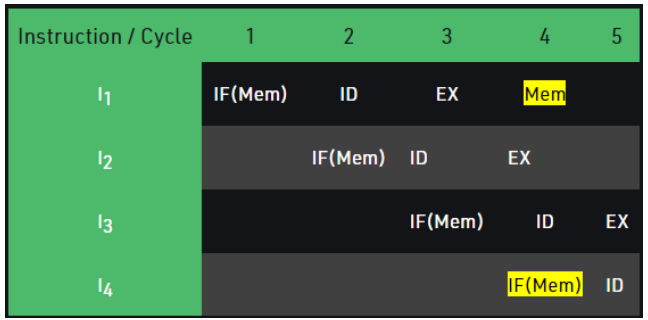


- **Solution to Data conflict:** Using Operand Forwarding
- In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly.
- Considering the same example:
  I1 : ADD R1, R2, R3
  I2 : SUB R4, R1, R2



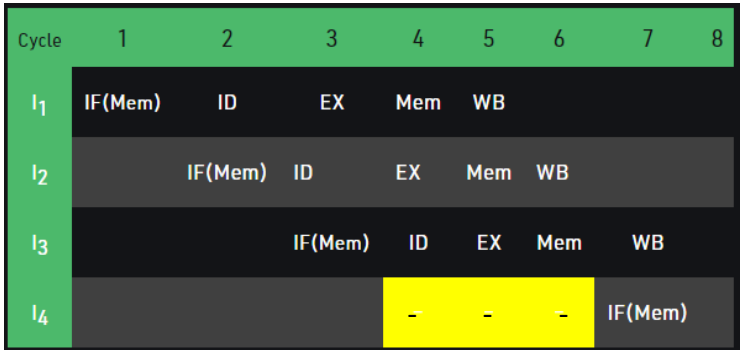
2. Structural Conflict

- This dependency arises due to the resource conflict in the pipeline.
- A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle.
- A resource can be a register, memory, or ALU.



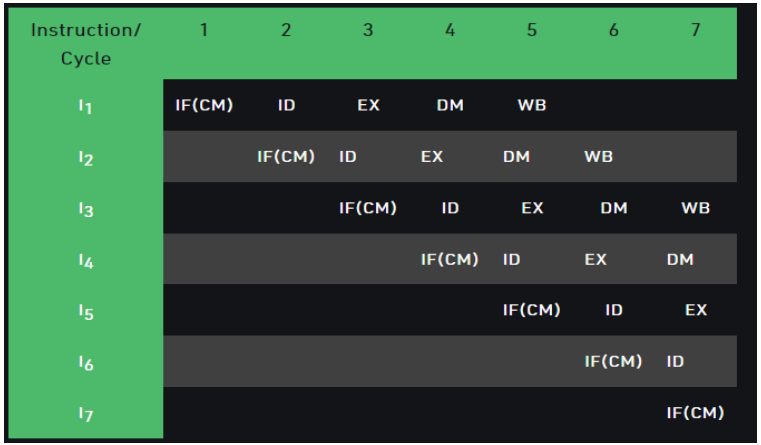
- In the above scenario, in cycle 4, instructions I_1 and I_4 are trying to access same resource (Memory) which introduces a resource conflict.

- To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:



Solution to Structural conflict:

- **Solution to Structural conflict: Using Renaming**
- To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called **Renaming.**
- According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively.
  - CM will contain all the instructions and
  - DM will contain all the operands that are required for the instructions.



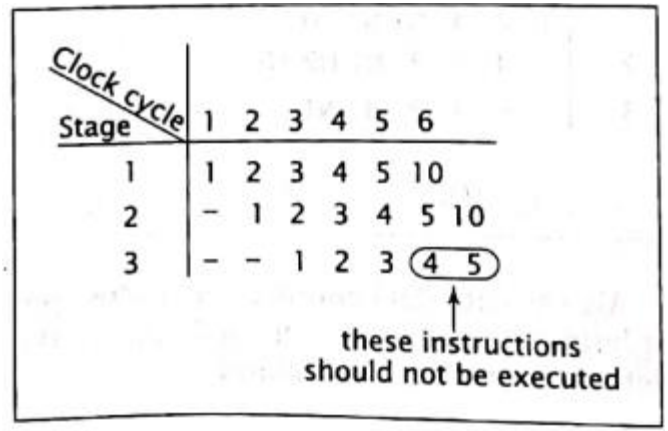
WB: Write Bytes to register

3. Branch Conflict

- Branch or Jump statements can also cause conflicts within a RISC instruction pipeline.
- Unlike data conflicts, branch conflicts do not cause incorrect data values to be used. Instead, branch conflicts cause the CPU to execute instructions at times when they should not be executed.
- 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
```

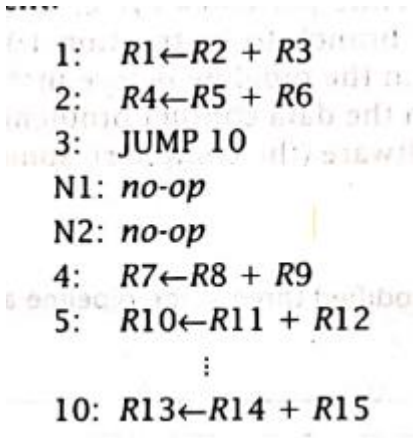
- After instruction 3 is executed, the CPU should branch to instruction 10.
- However, instruction 4 and 5 are already in the pipeline before instruction 3 is executed.

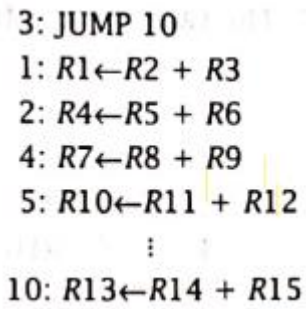


Execution trace of the code block illustrating a branch conflict

3. Branch Conflict (Contd.)

- This problem is more complex than data conflict, though, because of conditional branch instructions, which may or may not branch during a given execution.
- One of the solution might be: **insertion of no-op** and another might be : **reordering of instructions.**
- However, both the techniques are not optimal.





```
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
```

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

Clock cycle / Stage	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)

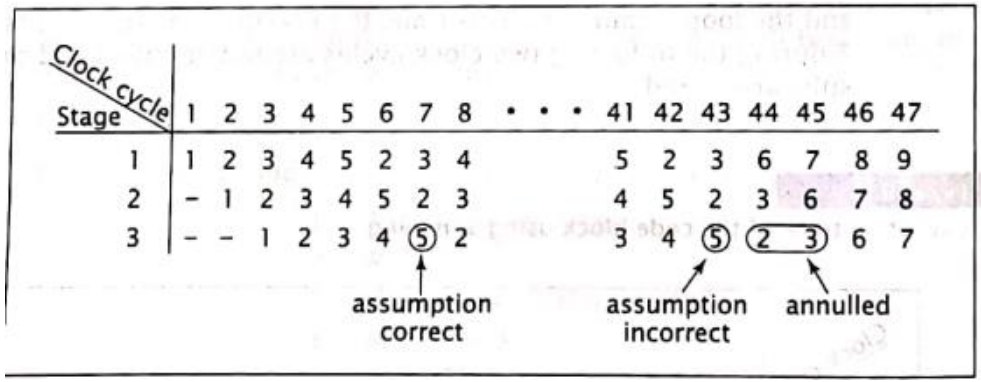
Clock cycle / Stage	1	2	3	4	5
1	3	1	2	10	
2	–	3	1		
3	–	–	3		

(b)

Execution traces of the code block using: (a) no-op insertion, and (b) instruction reordering

3. Branch Conflict (Contd.)

- The performance of the pipeline could be improved by using **branch prediction.**
- Branch prediction allows the compiler or pipeline hardware to make an assumption as to whether or not the conditional branch will be taken.
- The CPU enters the most likely next instruction into the pipeline immediately after the conditional branch instruction.
- 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 introduce.
- If the guess is wrong, the results are annulled as before.



Execution trace of the code block using branch prediction

9.7 RISC Vs CISC

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and microprogrammed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word	Instructions are larger than the size of one word

Exam questions:

1. Differentiate between RISC and CISC processors. Also describe how branch conflicts can be resolved? [PU 2017 Fall]
2. Illustrate with an example how branch of data conflicts occurs? List out the solutions to the data conflicts. [PU 2017 Spring]
3. What is 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 number of registers in the system. Show the pictorial representation as well. [PU 2018 Fall]
4. What is 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 number of registers in the system. Show the pictorial representation as well. [PU 2018 Spring]
5. What is Register Windows? In a system, there are 8 windows of registers, each register shares 8 input registers, and 8 output registers. Each of the windows has 4 local registers. The system has 10 global registers. Calculate the total number of registers in the system. Show the pictorial representation as well. [PU 2019 Spring]
6. What is instruction pipelining? Explain the data conflicts and branch conflicts along with is remedies. [PU 2020 Spring]

End of chapter 9