Computer Architecture

V semester

Chapter 8: Reduced Instruction Set Computer (RISC)

5 hours

Compiled by :

Ankit Bhattarai, Assistant Professor
Email: ankitbhattarai@cosmoscollege.edu.np

Topics to be discussed

- Reduced Instruction Set Computing (RISC)
- RISC vs CISC
- Instruction Pipelines
- Conflicts & Solutions in Instruction Pipelines
- Register Window and Renaming

Reduced Instruction Set Computing (RISC)

- 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.

Reduced Instruction Set Computing (RISC)

- 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.

RISC & its Characteristics:

- Relatively few instructions.
- Fixed-length instructions: instructions of same size; format can be different.
- Limited loading and storing instructions Access memory.
- Fewer Addressing modes.
- Instruction pipelining.

RISC & its Characteristics:

- Large number of registers so as to 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 microprogram control so as to have a lower propagation delay.

RISC Instruction Set

- 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))

Contd…

- 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 & CISC : Comparison

RISC	CISC
It mainly emphasis on software.	It mainly emphasis on hardware.
Single-clock, reduced instruction only.	Includes multi-clock complex instructions
Low cycles per second, large code sizes	Small code sizes, high cycles per second
Spends more transistors on memory registers	Transistors used for storing complex instructions
Fixed sized instructions	Variable sized instructions.
Code size is large.	Code size is small.

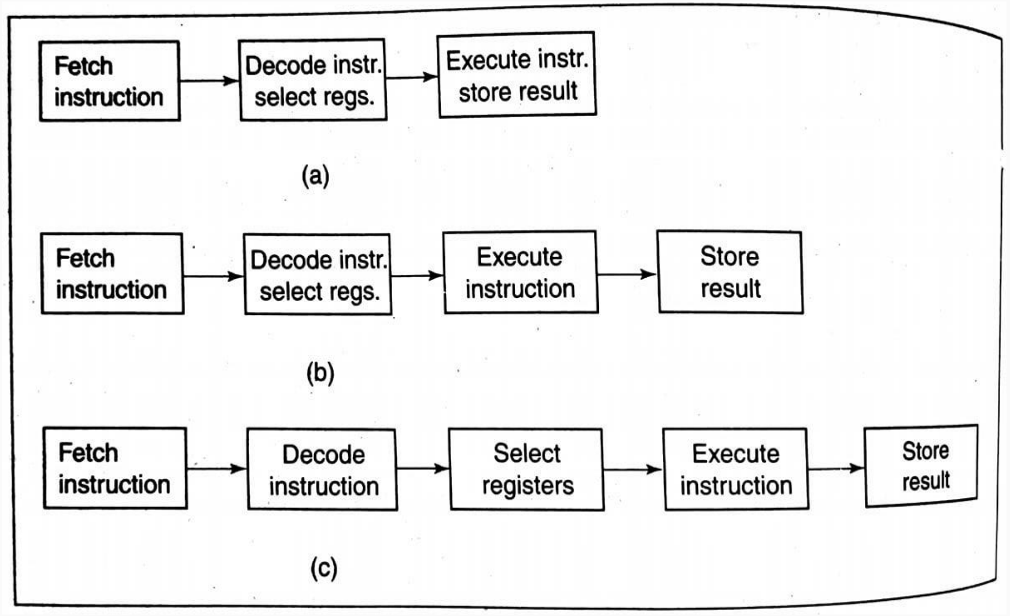
Pipelining: Instruction Pipelining

- Pipelining is the process of accumulating instruction from the processor through a pipeline.
- It allows storing and executing instructions in an orderly process. It is also known as pipeline processing.
- 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.

Pipelining : Instruction Pipelining

- In this a stream of instructions can be executed by overlapping *fetch*, *decode* and *execute* phases of an instruction cycle.
- This type of technique is used to increase the throughput of the computer system.
- An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline.
- Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Stages RISC pipelines

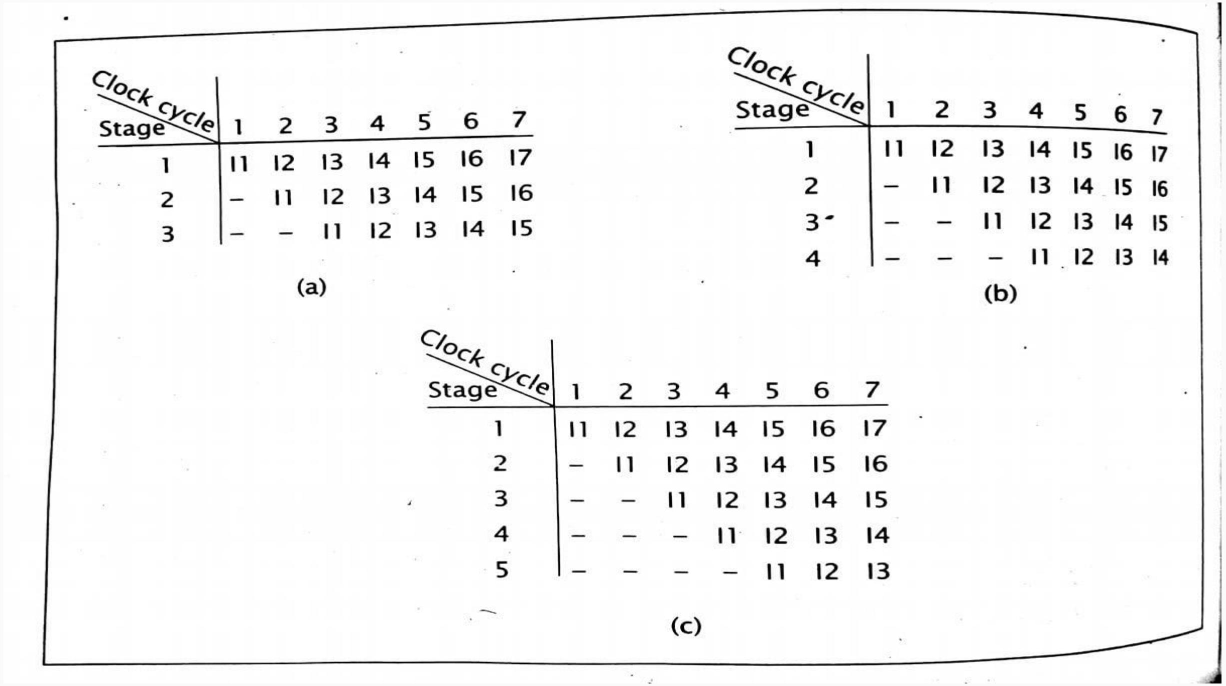


Three stages RISC pipelines

Four stages RISC pipelines

Five stages RISC pipelines

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



Clock cycle / Stage	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)

Clock cycle / Stage	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)

Clock cycle / Stage	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:

- **Timing Variations**

All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

- **Data Conflict:**

Data Conflicts occur when the pipeline causes an incorrect data value to be used.

- **Branch Conflict:**

Branch Conflicts occur when a branch statement results in incorrect instructions being executed.

Conflicts in Instruction Pipeline:

- **Interrupts**

Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

- **Data Dependency**

It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Pros and cons of Pipelining:

Advantages of Pipelining

1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

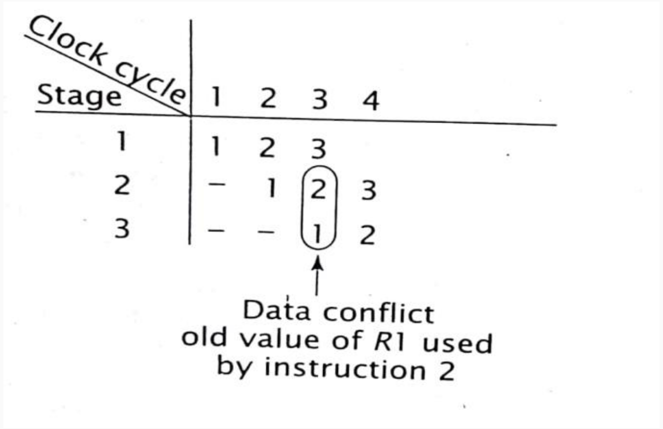
Conflicts in Instruction Pipeline: **Data Conflicts**

- Consider the following consecutive program statements:

  1: R1←R2 + R3

  2: R4←R1 + R3

  3: R5←R6 + R3



Stage \ Clock cycle	1	2	3	4
1	1	2	3	
2	–	1	2	3
3	–	–	1	2

Data conflict
old value of *R*1 used
by instruction 2

Solutions to data conflict problem: **No-op insertion**

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

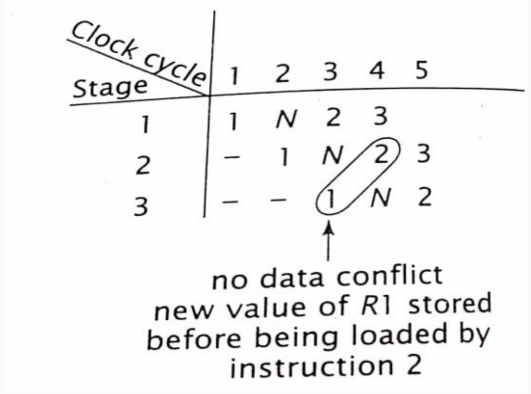
  1: R1←R2 + R3

  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.

Contd…



Stage \ Clock cycle	1	2	3	4	5
1	1	*N*	2	3	
2	–	1	*N*	2	3
3	–	–	1	*N*	2

no data conflict new value of *R*1 stored before being loaded by instruction 2

- It reduces the overall system performance.
- The no-op instruction does not perform useful work and require an extra clock cycle.

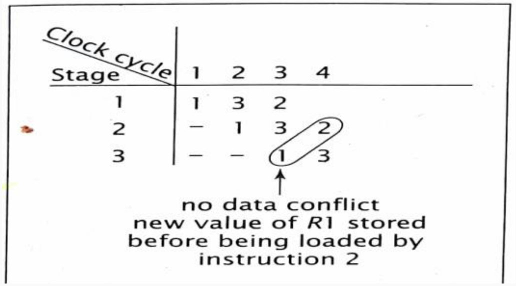
Solutions to data conflict problem: **Instruction reordering:**

- 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.



Stage \ Clock cycle	1	2	3	4
1	1	3	2	
2	–	1	3	2
3	–	–	1	3

no data conflict
new value of *R1* stored
before being loaded by
instruction 2

Contd…

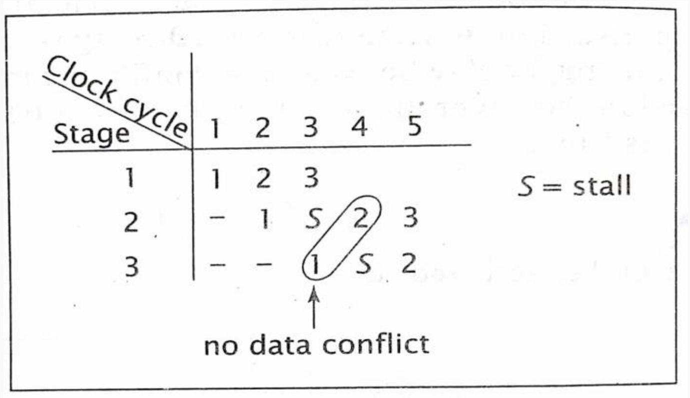
- It is not always possible to reorder the instructions of a program to avoid data conflicts. For example, the following code segment cannot be reordered successfully:

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

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

Solutions to data conflict problem: **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.



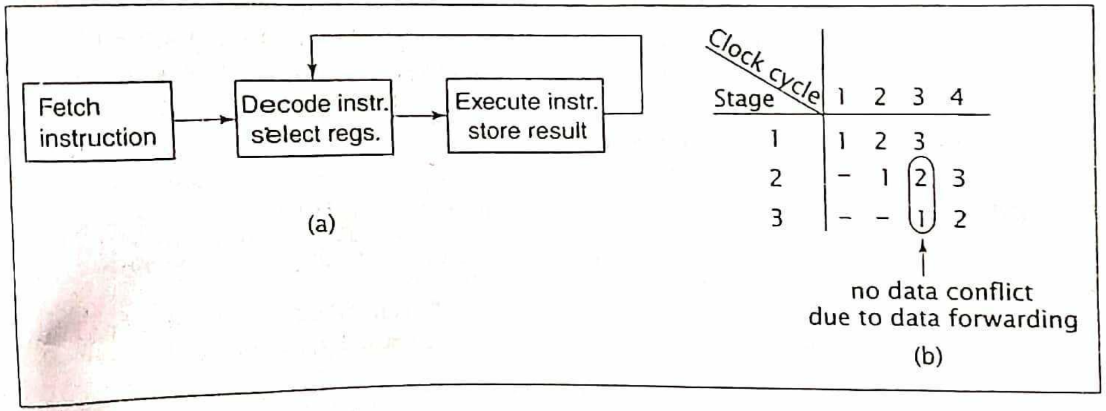
Stage \ Clock cycle	1	2	3	4	5
1	1	2	3		
2	–	1	S	2	3
3	–	–	1	S	2

S = stall

no data conflict

Solutions to data conflict problem: **Data forwarding**

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


Fetch instruction
Decode instr. select regs.
Execute instr. store result
(a)
Clock cycle
Stage
1 2 3 4
1 | 1 2 3
2 | – 1 2 3
3 | – – 1 2
no data conflict due to data forwarding
(b)

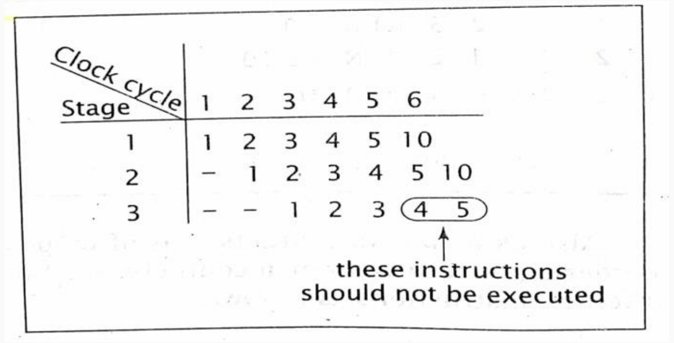
- Hardware solution to the data conflict problem.
- After the instruction is executed, its result is stored just as before, but the result is also forwarded directly to the stage that selects registers(retrieves operands).
- That stage gets the new value of the operand directly from the execute instruction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

Conflicts in Instruction Pipeline: **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.



Stage \ Clock cycle	1	2	3	4	5	6	
1	1	2	3	4	5	10	
2	–	1	2	3	4	5	10
3	–	–	1	2	3	$\boxed{4}$	$\boxed{5}$

↑ these instructions should not be executed

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

Solutions to Branch Conflict: No-op insertion, reordering & stall insertion

- **No-op insertion**

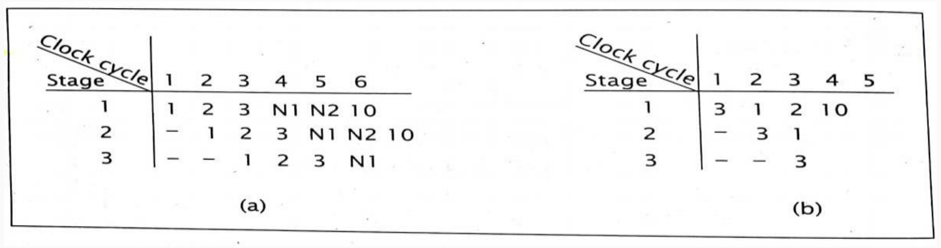
```
1:      R1←R2 + R3
2:      R4←R5 + R6
3:      JUMP 10
N1:     no-op
N2:     no-op
4:      R7←R8 + R9
5:      R10←R11 + R12
                :
10:     R13←R14 + R15
```

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

Contd…

- **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

- **Stall insertion** can also be used to handle branch conflicts.

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

Solutions to Branch Conflict: **Annulling**

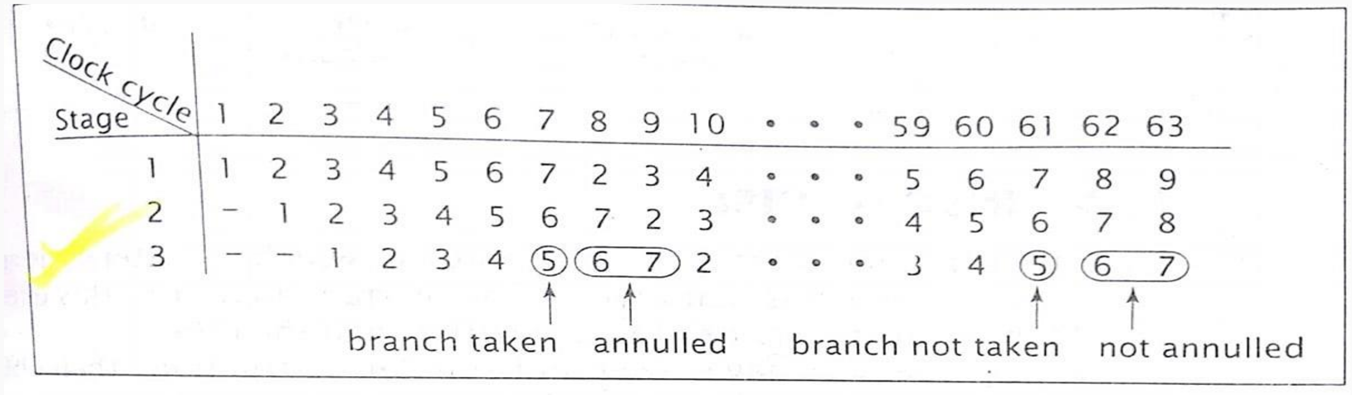
- In annulling, the instructions proceed through the pipeline as they normally would.
- If an instruction should not have been executed, because a previous instruction branched away from it, its result are not stored.
- Even though it might have been executed, as long as no results are stored, it is 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:	$R7 \leftarrow R8 + R9$

- 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.

Contd…

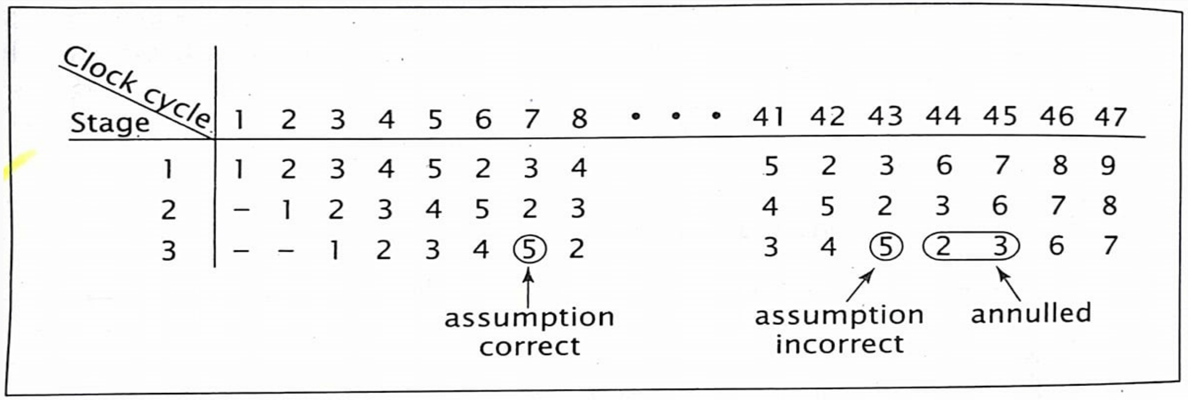
- During clock cycle 61, the branch is not taken and the loop terminates.
- This time, the execution of statements 6 and 7 during the following two clock cycles are not annulled and their results are stored.



Clock cycle / Stage	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	$\boxed{5}$	$\boxed{6}$	$\boxed{7}$	2	•	•	•	3	4	$\boxed{5}$	$\boxed{6}$	$\boxed{7}$
							branch taken	annulled								branch not taken	not annulled	

Solutions to Branch Conflict: **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

- 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, so having more registers makes data available faster.
- Having more registers also helps to reduce the number of memory references, particularly calling and returning from subroutines.

Register Windows and Renaming

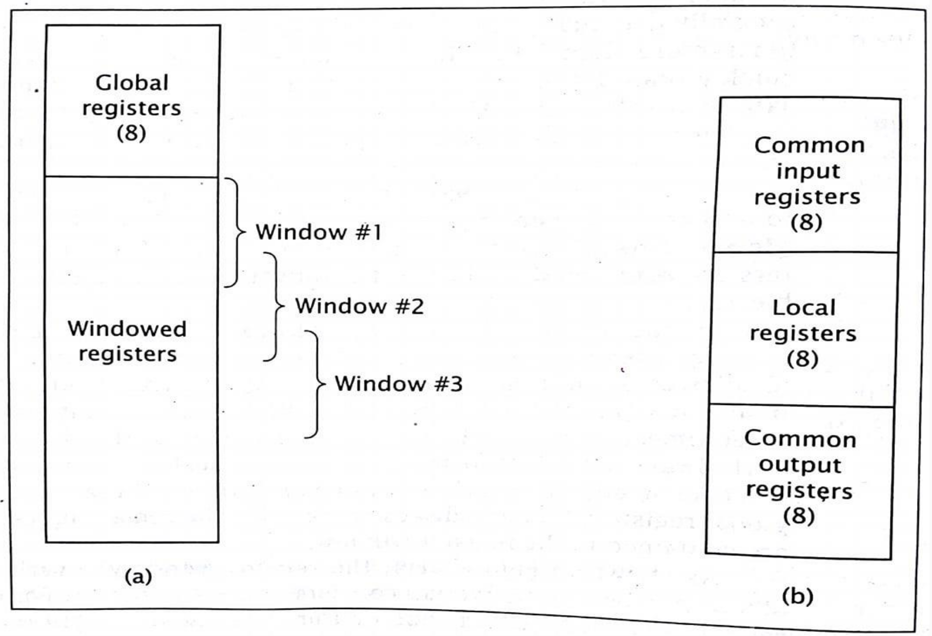
- Although, a RISC processor has many register, it may not be able to access all of them at any time.
- Most RISC CPU has some global register which are always accessible.
- The remaining register are windowed such that only a subset of the registers are accessible at any specific time.

- To visualize we take an example of SPARC processor as shown in figure on next slide; which can access any 32 different registers at a given time.

Register Windows and Renaming

- Of these 32 registers, 8 are global registers that are always accessible.
- The remaining 24 registers are contained in the register window.

Register windowing in the SPARC processor



Register Windows and Renaming

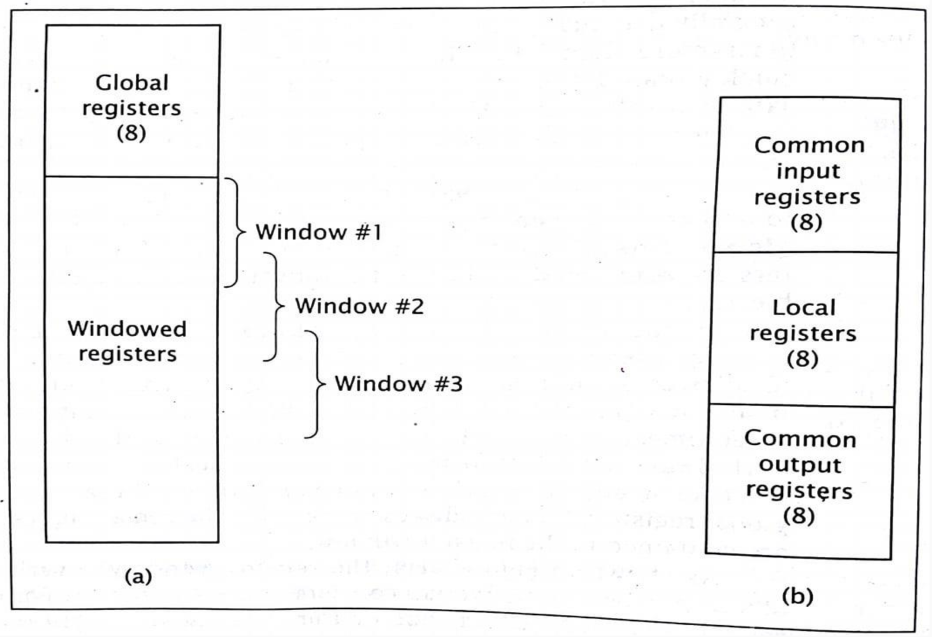
- Register windows gets overlapped.
- ***Example:***

✓ The last eight register of windows 1 are also the first eight registers of windows 2.

✓ Similarly, the last eight registers of windows 2 are also the first eight registers of windows 3.

✓ The middle eight registers of window 2 are local; they are not shared with any other window.

Register windowing in the SPARC processor



Register Windows and Renaming

- The RISC CPU must keep track of which windows is active and which windows contains valid data.
- A windows 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 is advantageous when the CPU calls its subroutine.
- ***Example:***

✓ In SPARC example, if window 1 is active and the CPU calls the subroutine, the processor activates window 2 by updating the window pointer register and window mask register.

✓ The CPU can pass parameters to the subroutine via the registers that overlap both windows, instead of through memory; this saves a significant amount of time in accessing data.

✓ The CPU can use the same registers to return results to the calling routine.

Register Windows and Renaming

- The RISC CPU must keep track of which windows is active and which windows contains valid data.
- A windows 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 is advantageous when the CPU calls its subroutine.
- ***Example:***

✓ In SPARC example, if window 1 is active and the CPU calls the subroutine, the processor activates window 2 by updating the window pointer register and window mask register.

✓ The CPU can pass parameters to the subroutine via the registers that overlap both windows, instead of through memory; this saves a significant amount of time in accessing data.

✓ The CPU can use the same registers to return results to the calling routine.

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.

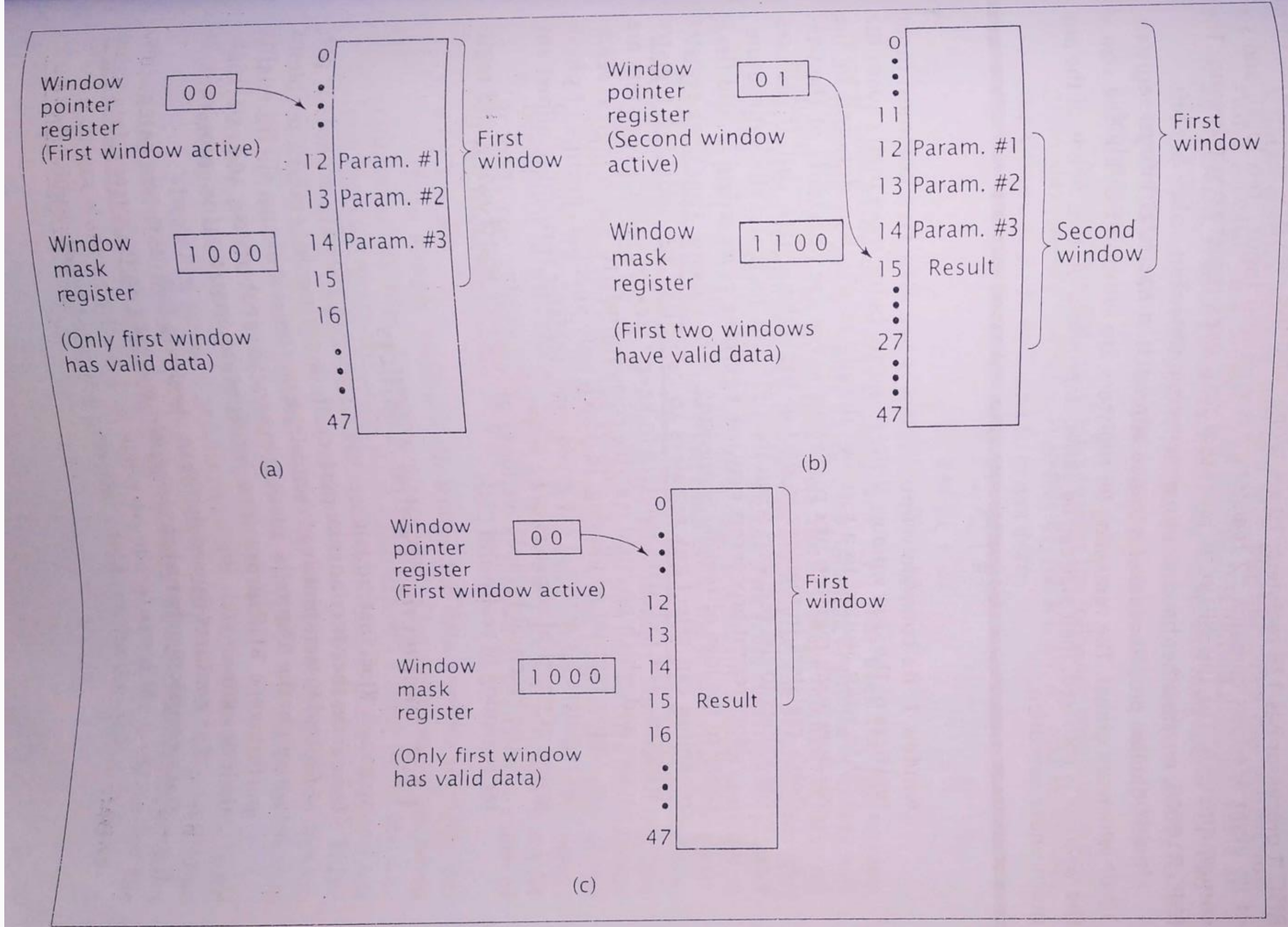


FIGURE 11.5 Register windowing in a CPU: (a) during execution of the main routine, (b) executing a subroutine, and (c) after returning from the subroutine

Register renaming

- More recent processors may use register renaming to add flexibility to the idea of 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 correspond to each logical register.
- In computer architecture, register renaming is a technique that abstracts logical registers from physical registers.
- Every logical register has a set of physical registers associated with it.

Register renaming

- When a machine language instruction refers to a particular logical register, the processor transposes this name to one specific physical register on the fly.
- The physical registers are opaque and cannot be referenced directly but only via the canonical names.
- This technique is used to eliminate false data dependencies arising from the reuse of registers by successive instructions that do not have any real data dependencies between them.
- The elimination of these false data dependencies reveals more instruction-level parallelism in an instruction stream, which can be exploited by various and complementary techniques such as superscalar and out-of-order execution for better performance.

THANK YOU

Any Queries ?

ankitbhattarai@cosmoscollege.edu.np