Chapter-8

RISC (Reduced Instruction Set Computer)

RISC (Reduced Instruction Set Computer)

A computer with fewer instructions with simple constructs for faster execution within CPU without having use of memory is classified as RISC.

RISC Characteristics

The concept of RISC architecture includes an attempt to reduce execution time by simplifying the instruction set of computer. Characteristics of RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes.
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed length, easily decoded instruction format
6. Single cycle instruction execution
7. Hardwired rather than micro programmed control unit
8. For LOAD and STORE it uses register to register movement.

Instructions of RISC processor consists of register-to-register operations with load and store operations for memory access. In load instruction operand are brought into processor register. The RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. As all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible. RISC processors are used for high level programming. Due to less instruction, it works with higher clock frequencies and uses small space on the chip. Thus remaining space can be used to add registers and other component.

CISC (Complex Instruction Set Computer)

A computer with large number of instructions is classified as complex instruction set computer. Although it has complex instruction set, it does not increase development cost. CISC processors provide background compatibility with other processors in their families. As technology allows more devices to be incorporated in a single microprocessor chip, CISC CPUs are adding more registers to their designs to achieve the performance improvement as that of RISC processor

1. Large number of instructions – typically 100 to 250 instructions;
2. Large variety of addressing modes, 5 to 20 different modes.

3. Variable-length instruction format.
4. These instructions manipulateoperand in memory.
5. For LOAD and STORE it uses memory to memory movement.

Parallel processing

Parallel processing is the processing of data concurrently to achieve the faster execution time or simultaneous data processing for the purpose of increasing computational speed of a computer system. For e.g. when an instruction is being executed in ALU, the next instruction can be read from memory. The purpose of parallel processing is to speed up the computer processing capability and increase its through put i.e. the amount of processing that can be accomplished in given interval of time. Parallel processing increase the hardware and thus cost of the system increases.

Levels of complexity in parallel processing

1. Lowest level- Distinguish between serial and parallel operations. For parallel operation registers with parallel load operate all the bits of the word simultaneously.
2. Highest level- At highest level parallel processing can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Data is distributed among multiple functional units.

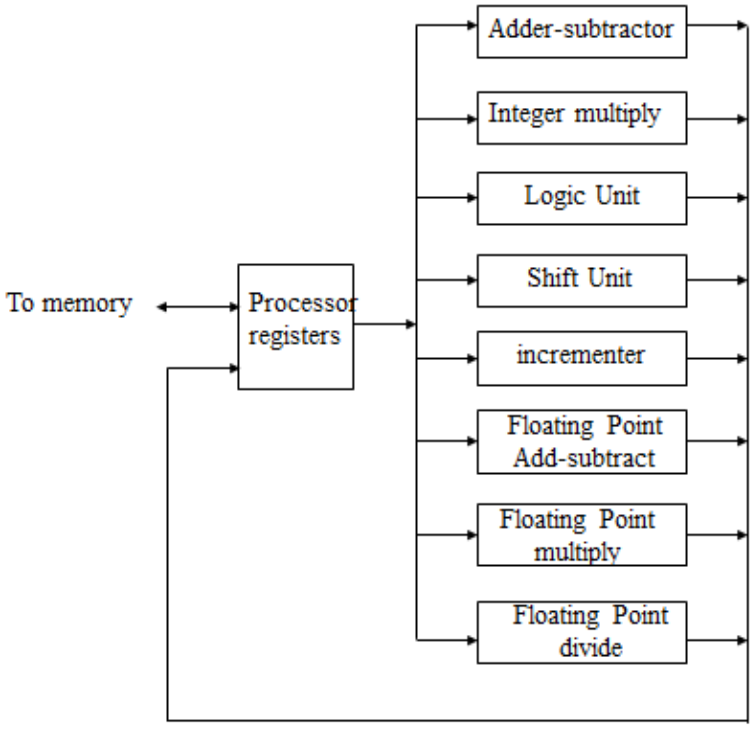


Fig. Processor with multiple functional units.

Above figure gives one possible way of separating execution unit into eight functional units operating in parallel.

Classification of Parallel Processing

It is preferred according to how data and instruction can be executed simultaneously. Instruction stream is the sequence of instruction read from memory. Data stream is the operation performed on data in the processor. Parallel processing occurs in instruction stream or data stream or in both. Flynn's classified computer in four groups:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data stream (SIMD)
3. Multiple instruction stream, single data stream (MISD)
4. Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit a processor and a memory unit. Instructions are executed simultaneously and the system may or may not have internal parallel processing. Parallel processing is through multiple functional unit.

SIMD has organization including many processing units under the supervision of a common control unit. Recieves same instructions but operates on different items of data.

MISD is only theoretical implementation; no practical system has been constructed using organization.

MIMD refers to computer capable of processing several programs at the same time. Most multiprocessor and multiple computer systems is classified in this category.

Pipelining

Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. It is a collection of processing segments through which binary information flows. Each segments process partially the partitioned task. The result from each segment is transferred to next segment in pipeline. The final result is obtained after the data have passed through all segments.

E.g. $A_i*B_i+C_i$for i=1,2,3…7

Here this example says to perform the combined multiply and add operations with a stream of numbers. Each sub operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit. Sub operations in each segment are

$R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$ Input A_i and B_i

$R_3 \leftarrow R_1*R_2$, $R_4 \leftarrow C_i$ Multiply and input C_i

$R_5 \leftarrow R_3+R_4$ Add C_i to product

R_1, R_2, R_3, R_4, R_5 receives new data every clock pulse. The first clock pulse transfers A_1& B_1 into R_1 and R_2. The second clock pulse transfers product of R_1 and R_2 into $R_3$and C_1 into R_4. The same clock pulse transfers A_2 and B_2r seginto R_1 and R_2. The third cock pulse operates on all the three segments simultaneously. It takes three clock pulses to fill up the pipe and retrieve the first output form R_5. Now, each clock produces a new output and moves data step down the pipeline. This continues till the input flow in system.

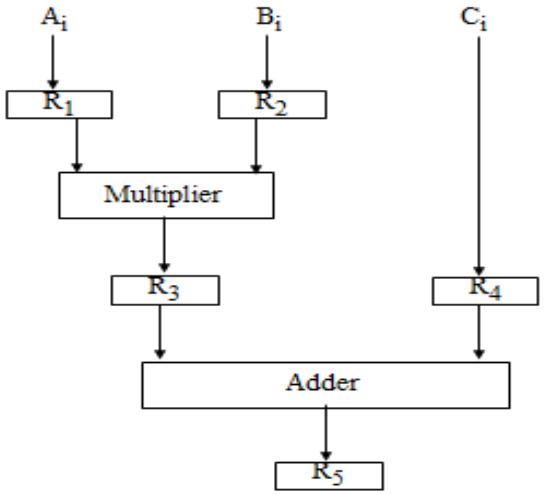


Fig. Example of pipeline processing

Clock Pulse no	Segment 1		Segment 2		Segment 3
	R_1	R_2	R_3	R_4	R_5
1	A_1	B_1	-	-	-
2	A_2	B_2	$A_1 * B_1$	C_1	-
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	-	-	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	-	-	-	-	$A_7 * B_7 + C_7$

General Consideration

EgThe general structure of a four segment pipeline is illustrated in figure below. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a sub operation over the data stream following through pipe. The segments are separated by registers R_i that holds intermediate results between the stages. Task is the total operation performed going through all segments in pipeline.

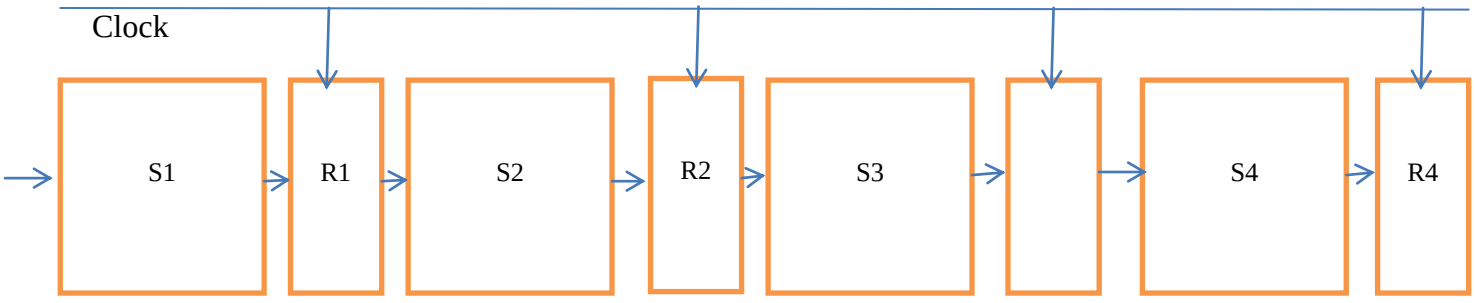


Fig: Four- segment Pipeline

Behavior of pipeline can be illustrated with a space-time diagram. It gives segment utilization as a function of time. Space-time diagram for above four-segment pipeline.

Segment/Clock cycles	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Here horizontal axis gives the clock cycle and the vertical axis gives the segment number. Six task T_1 to T_6 in four segment. Task T_1 is handled by segment1. After first clock cycle segment 1 is busy with T_2 and segment2 with T_1. continuing in this manner T_1 is completed in fourth clock cycle. from then pipeline completes task in each clock cycle. When pipeline is full, it takes only one clock period to obtain an output.

Now, k-segment pipeline with time period T_p execute n tasks. First task T_1 requires time equals to kT_p. The remaining n-1 tasks emerge from the pipe at one task per cycle. Therefore total time is $(n-1)t_p$.

total clock cycle=k+(n-1)clock cycles

Therfore from above example=4+(6-1)=9 clock cycles

For non pipeline unit performing the same operation takes time equal to t_n to complete each task. Total time required is nT_n for n tasks.

The speed up of a pipeline processing over an equivalent non pipeline is given by ratio

$$s = \frac{n\text{tn}}{k + (n-1)\text{tp}}$$

As number of task increases n becomes much larger than k-1 and k+n-1 approaches to the value of n. Then

$$s = \frac{\text{tn}}{\text{tp}}$$

For time taken being same in pipeline and non pipeline we will have $t_n=kt_p$

$$s = \frac{k\text{tp}}{\text{tp}} = k$$

This gives maximum speed up that a pipeline can provide is k, where k is the number of segment in pipeline.

egt_p=20 ns

k=4

n=100 tasks

Pipeline system will take (k+n-1)t_p

=(4+99)*20

=2060ns

Assuming $t_n=kt_p$

=4*20=80ns for nonpipeline

nkt_p=100*80=8000ns to complete 100 tasks

s=8000/2060 =3.88

As n increases the speed up will approach 4 which is equal to k.

Arithmetic Pipeline

Arithmetic pipeline units are used in high speed computers. They are used to implement floating point operations, multiplication of fixed-point numbers.

Eg Floating-point addition and subtraction

The inputs of floating-point adder pipeline are two normalized floating-point binary numbers.

$X=A*2^a$

$Y=B*2^b$

A and B are exponents. Here addition and subtraction can be performed in four segments.

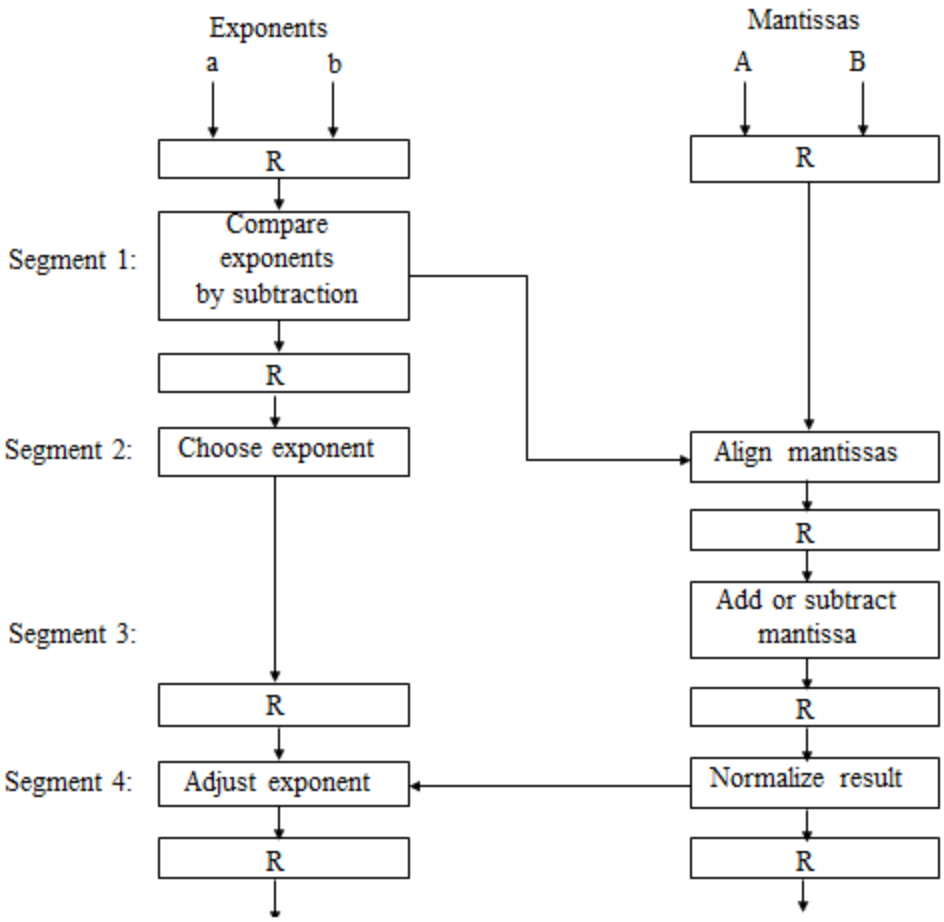


Fig: Pipeline for floating-point addition and subtraction

The registers R are placed between the segments to store intermediate results. The sub-operations are:

1. Compare the components
2. Align mantissas
3. Add or subtract mantissas
4. Normalize the result

The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of result. The exponent determines how many times the mantissa associated with smaller exponent must be shifted to right. This gives alignment of mantissas. Shift must be designed as a combinational circuit to reduce the shift time. Two mantissas are added or subtracted in segment 3. The result is normalized in segment4.For overflow the sum or difference of mantissas are shifted right and exponent is increased by one. For underflow, the number of leading zeros in mantissas determines the number of left shift and the number is subtracted from exponent.

eg $X=0.9504*10^3$

$Y=0.8200*10^2$

In first segment exponents are subtracted ie 3-2=1. The larger exponent is 3 thus the result is chosen. In next segment Y is align that is shift to right to obtain

$X=0.9504*10^3$

$Y=0.0820*10^3$

The addition of two mantissas

$Z=X+Y$

$=(0.9504+0.0820)*10^3$

$=1.0324*10^3$

Now for normalizing mantissa is shifted right and exponent is increased by 1.

$Z=0.10324*10^4$

The comparator, shifter, adder-sub tractor, incrementer and decrementer in the floating point pipeline are implemented with combinational circuits.

Suppose time delays for four segment are t_1=60 ns, t_2=70 ns, t_3=100 ns, t_4=80 ns interface register have t_r=10 ns

clock cycle $t_p= t_3+t_r =(100+10) = 110$ ns

For nonpipeline$t_n=t_1+t_2+t_3+t_4+t_r= 320$ ns

s=320/110=2.9 over non pipelined.

Instruction Pipeline

Pipeline can also occur in instruction stream. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segment. Here, instruction fetch and execute phases overlap and perform simultaneous operation.

E.g. An instruction fetch unit and an instruction execute unit which provides two-segment pipeline. The instruction fetch segment can be implemented by first-in-first-out (FIFO) buffer. Here when execute unit is not using memory; the control increases the program counter and uses its address value to read consecutive instruction from memory. The instructions are stored if FIFO buffer for execution in FIFO basis. Thus it records the average access time of memoryfor reading instructions. FIFO buffer acts as queue for execution unit to extract instructions.

For complex instructions, computer requires additional phases other than fetch and execute. Sequence of steps to process an instruction:

1. Fetch the instruction from memory

2. Decode the instruction
3. Calculate the effective address
4. Fetch the operands from memory
5. Execute the instruction
6. Store the result in proper sequence

These are difficulties thatwill prevent instruction pipeline to operate at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations.

To design an instruction pipeline in efficient way the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its functional depends on the instruction and the way it is executed.

E.g. four segment Instruction Pipeline. Here decoding of the instruction and calculation of effective address is combined in one segment. The instruction execution and storing of the result can be combined into one segment. This reduces instruction pipeline into four segments.

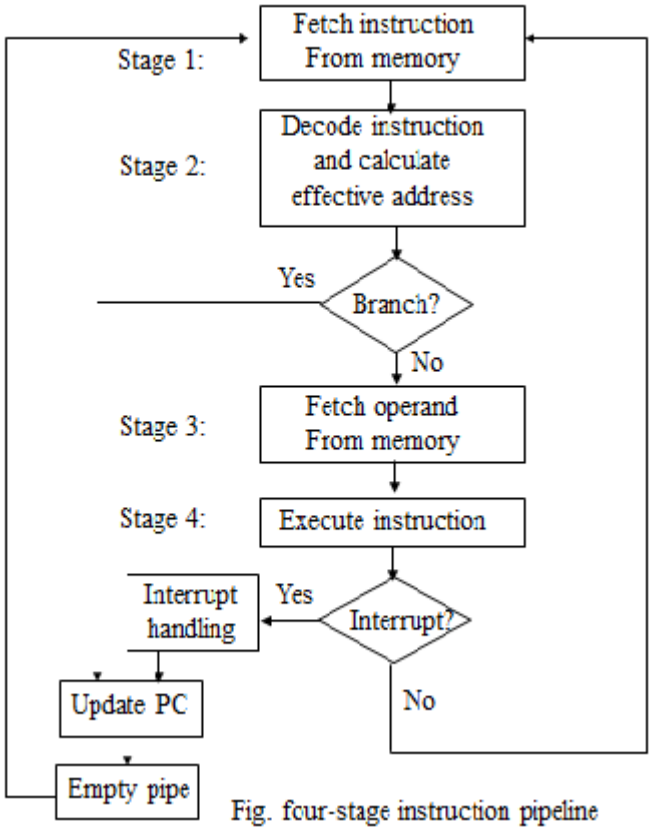


Fig. four-stage instruction pipeline

An instruction is being executed in segment4. In the same time next instruction is fetching operand in segment3. Effective address is being calculated in a separate arithmetic circuit for

third instruction and when memory is available instruction is being fetched and placed in FIFO. Thus four operations can be at same time.

FI-segment that fetches instruction

DA-segment that decode instruction and calculate effective address.

FO-segment that fetch operand

EX-segment that executes instruction

Step Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3(branch)			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5								FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Assume that processor has separate instruction and data memories. Thus FI & FO proceed at same time. In absence of branch each segment operates on different instructions. Thus in step 4 instruction 1 is being executed in segment EX, the operand for instruction2 is being fetched in segment FO, instruction3 is being decoded in segment DA and instruction4 is being fetched from memory in segment F1.

For instruction3 being a branch instruction. Here as soon as this instruction is decoded in segment DA in step4, the transfer from FI to DA is halted until the branch instruction is executed in step 6. If the branch is taken a new instruction is fetched in step 7. If the branch is not taken instruction fetched previously in step4 can be used.

Another delay may occur in the pipeline if the EX segment needs to store result of the operation in the data memory while FO needs to fetch an operand. In that case FO must wait till EX finishes its operation.

Difficulties in Instruction Pipeline

1. Resource conflicts- It is caused when two segments access memory at same time. These conflicts can be removed by using separate memories for data and instructions.
2. Data Dependency- It arises when an instruction depends on the result of previous one.
3. Branch difficulties- It arises from branch and other instructions that change the value of PC.

Data Dependency

It occurs due to collision of data or address. A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations. A data dependency occurs when an instruction need data that are not yet available. E.g Instruction FO may need the operand that is generated by EX of previous instruction and that may be still executing. Here, the second instruction must wait till the execution of first instruction. An address dependency occurs when an operand address cannot be calculated due to the information not being available needed by addressing mode. E,g. An instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into registers. Thus operandaccess to memory must be delayed till the required address is available.

Dealing with difficulties

1. Hardware interlocks- Hardware interlocks can be inserted. Interlock is a circuit that detects the instruction whose source operands are the destinations for other instructions in pipeline. After detecting the instruction it is delayed by enough clock cycles to remove the conflicts.
2. Operand Forwarding- It needs a special hardware to detect the conflicts and then avoids it by routing the data through special paths through pipeline. E.g. Instead of transferring ALU result to destination register the hardware checks the operand and if it is required as source in next instruction, it passes the result directly into ALU input. This method requires additional hardware to detect the conflicts.
3. Delayed Load- Here compiler detects the conflicts and inserts no-operations to avoid the conflicts. The no-op instructions can be done by reordering the instructions that is necessary to delay.

   E.g operations of four instructions

   1. LOAD:R1←M[address]
   2. LOAD: R2←M[address]
   3. ADD:R3←R1+R2
   4. STORE:M[address]←R3

   Let us assume that this proceeds in three segment pipeline without interruptions

   I: Instruction fetch

   A: ALU Operation(Instruction decoding, address decoding, data manipulation and ALU operation)

   E:Execute instruction

   Here data conflicts occur in instruction3 because the operand R2 is not available in A segment.

   Timing diagram with data conflicts

Clock Cycle	1	2	3	4	5	6
1. LOAD:R1	I	A	E			
2. LOAD:R2		I	A	E		
3. ADD:R1+R2			I	A	E	
4. STORE:R3				I	A	E

   Here E segment in clock cycle 4 is in a process of placing memory data into R2. A is using the data from R2 in same clock cycle, but value in R2 is not correct value since it has not yet

been transferred from memory. Now, compiler must make sure that instruction following load instruction uses the data fetch from memory. If the compiler cannot find useful instruction to put after load, it inserts no-operation instruction. It has no-operation and wastes clock cycle. This is known as delayed load.

Timing diagram without data conflicts:

Clock Cycle	1	2	3	4	5	6	7
1. LOAD:R1	I	A	E				
2. LOAD:R2		I	A	E			
3. NO-operation			I	A	E		
4. ADD:R1+R2				I	A	E	
5. STORE:R3					I	A	E

Advantage: No hardware is required i.e. compiler removes the conflicts.

Branch Conflicts

A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential flow by loading the PC with target address. In a conditional branch the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. The branch instruction breaks the normal sequence.

Handling of conditional branch

1. Pre fetch target instruction: Here the target instruction is pre fetched along with the instruction following the branch. Both are saved till the branch is executed. If the branch condition is successful the pipeline continues from the branch target instruction.
2. Branch Target Buffer(BTB): It is an associative memory included in the fetch segment of the pipeline. Each entry in BTB consists of address of a previously executed branch instruction and the target instruction for the branch. It also stores the next few instructions after the branch target instruction. When pipeline decodes the branch instruction, it searches in BTB for the address of the instruction. If it is in BTBthe instruction is available and is pre fetch directly. If not present in BTB the pipeline shift towards new instruction and stores the target instruction in BTB.
3. Loop Buffer: Loop buffer is a small high speed register maintain by fetch segment in pipeline. When a program loop is detected, it is stored in loop buffer with all branches. Now, it can be executed directly without having to access memory till loop mode is removed by final branching out.
4. Branch Prediction: A pipeline with branch prediction uses some logic to guess outcome of a conditional branch instruction before it is executed. The pipeline then pre fetches the instruction stream from predicted path.
5. Delayed Branch: Here compiler detects the branch instructions and rearrange instructions for operation of pipeline without interrupt. No-operation instruction is inserted after a branch instruction. This causes the computer to fetch the target instruction during no-operation.

E.g. Load from memory to R1
Increment R2
ADD R3 to R4
Subtract R5 form R6
Branch to address X

Clock Cycles	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. NO-operation						I	A	E		
7. NO-operation							I	A	E	
8. Instruction in X								I	A	E

Using NO-operation

Clock Cycles	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

Rearranging the instruction

RISC Pipeline

RISC is the ability to use an efficient instruction pipeline. It uses instruction pipeline using a small no. of sub operations with each being executed in one clock cycle. Because of fixed length instruction format the decoding of the operation can occur at the same time as the register selection. All operands are present in registers thus there is no need to calculate effective address or fetching operands from memory. Thus instruction pipeline can be implemented with three segments.

1: fetch the instruction from program memory

2: executes instruction in ALU

3: Stores the result

RISC machines use two memories for storing instructions and data. This reduces conflicts between a memory access to fetch an instruction and to load or store and operand.

RISC executes instruction at the rate of one per clock cycle. It is not possible to expect that every instructionbe fetched from memory and executed in one clock cycle. What is done is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single clock cycle instruction execution. Advantage of RISC over CISC is it uses pipeline segments that requires only one clock cycle.

RISC compiler translates high level program into machine language program. And instead of having additional hardware to handle branch and data conflicts, it relies on minimizing these effects by compiler by inserting no-operation instruction or rearranging instructions.

Register Windows and register renaming

The reduced hardware requirements of RISC processors leave additional space available on chip. RISC CPUs use this space to include a large number of registers, sometimes more than 100.CPU accesses data more quickly in registers than in memory. Thus having more registers makes data available fast and also reduces the memory references.

Although a RISC processor has large number of registers it may not be able to access all of them at any given time. Global registers are always available. Other registers are windowed. That is only a subset of registers is accessible at any given time.

Figure below explains windowing of registers. Here this processor can access any of 32 different registers at a given time. Of these 32 registers, eight are available global registers that are always accessible. The remaining registers are contained in the register window.

Register windows overlap each other. E.g. in SPARC processor last eight register of first window are also the first eight register of window2 and last eight register of window2 is first eight register of window3. The middle eight register of window2 are local.

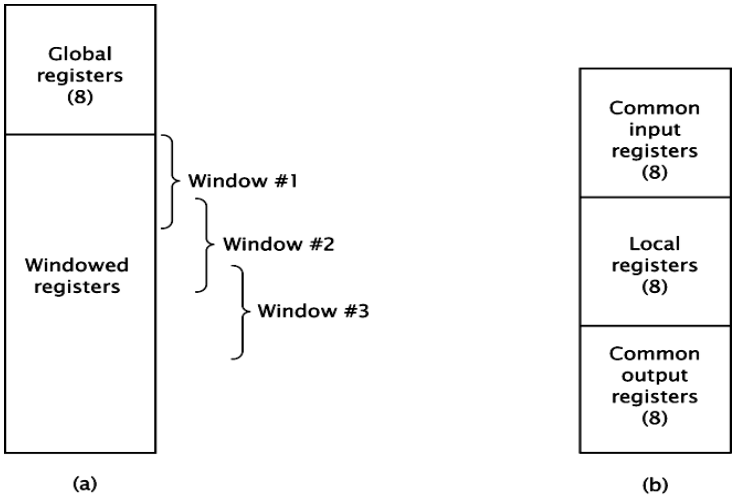
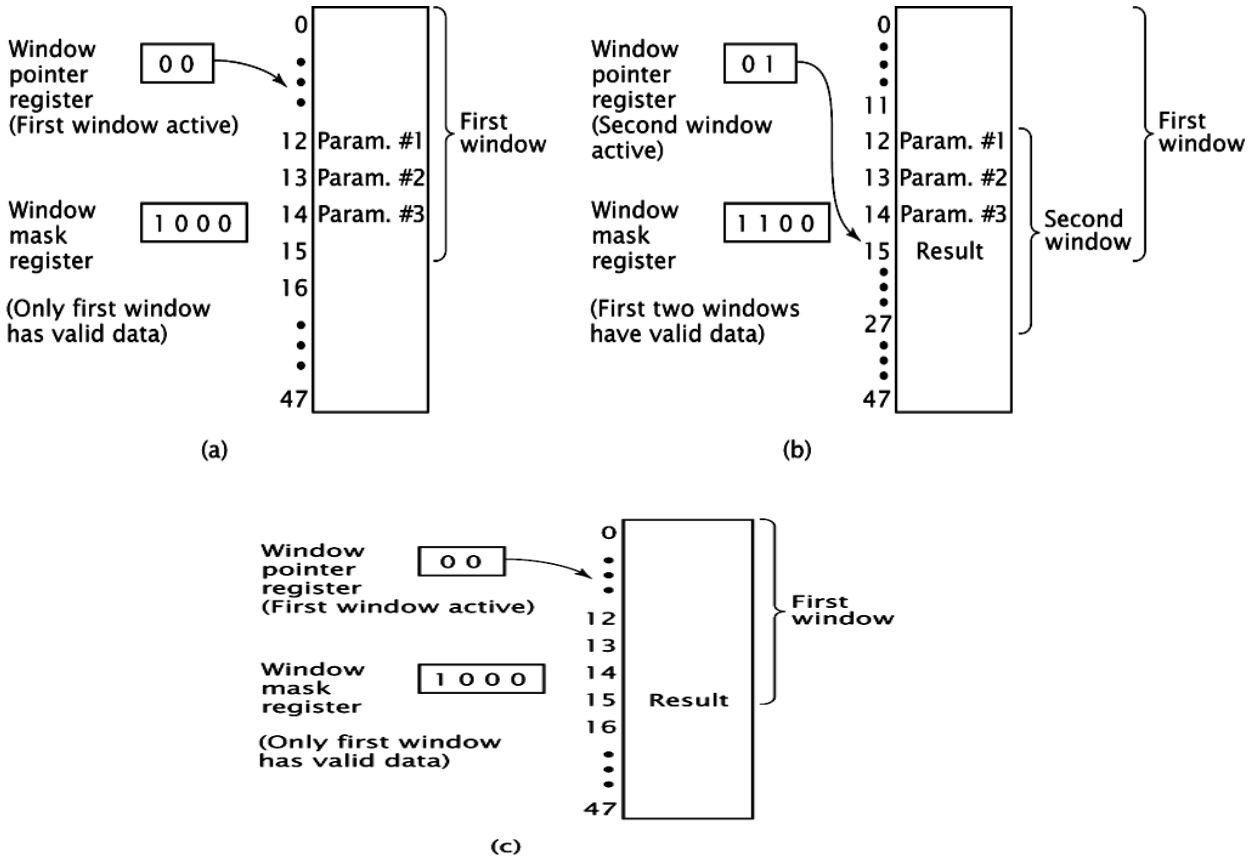


Fig: Register windowing in the SPARC processor

RISC CPU must keep the track of which window is active and which contains valid data. For this window pointer register is used to point the active window. A 1 bit window mask register is used that denotes valid windows.

Register window is beneficial when CPU calls a subroutine. When calling process takes place, processor activates register window is moved down one position. E.g. let us consider a CPU with 48 registers with three windows with 16 registers each and a overlap of four registers between windows.



Initially CPU is running on the first window as in figure (a). Here a subroutine is called and three parameters are to be passed. CPU stores these parameters in three of overlapping registers and calls the subroutine. Subroutine accesses the parameter, calculates and result to be returned to the main program is again stored in the overlapping register. Now second window is deactivated and CPU works with first window as shown in figure (c). Register windowing is not linear but circular. The last window overlaps window1.

Register renaming is used to add flexibility to the idea of register windowing. A processor that uses register renaming can use any register to compromise its working register "window". The CPU uses pointer to keep track of active register and which physical register corresponds to which logical register. Register renaming allows any group of physical registers to be active.