3.0 Objective

The main objective of this lesson is to known the basic properties of pipelining, classification of pipeline processors and the required memory support. The main aim this lesson is to learn the how pipelining is implemented in various computer architecture like RISC and CISC etc. How the issues related to limitations of pipelining and are overcame by using superscalar pipeline architecture.

3.1 Introduction

Pipeline is similar to the assembly line in industrial plant. To achieve pipelining one must divide the input process into a sequence of sub tasks and each of which can be executed concurrently with other stages. The various classification or pipeline line processor are arithmetic pipelining, instruction pipelining, processor pipelining have also been briefly discussed. Limitations of pipelining are discussed and shift to Pipeline architecture to

Superscalar architecture is also discussed. Superscalar pipeline organization and design are discussed.

3.2 Linear pipelining

Pipelining is a technique of that decompose any sequential process into small subprocesses, which are independent of each other so that each subprocess can be executed in a special dedicated segment and all these segments operates concurrently. Thus whole task is partitioned to independent tasks and these subtask are executed by a segment. The result obtained as an output of a segment (after performing all computation in it) is transferred to next segment in pipeline and the final result is obtained after the data have been through all segments. Thus it could understand if take each segment consists of an input register followed by a combinational circuit. This combinational circuit performs the required sub operation and register holds the intermediate result. The output of one combinational circuit is given as input to the next segment.

The concept of pipelining in computer organization is analogous to an industrial assembly line. As in industry there different division like manufacturing, packing and delivery division, a product is manufactured by manufacturing division, while it is packed by packing division a new product is manufactured by manufacturing unit. While this product is delivered by delivery unit a third product is manufactured by manufacturing unit and second product has been packed. Thus pipeline results in speeding the overall process. Pipelining can be effectively implemented for systems having following characteristics:

- A system is repeatedly executes a *basic function*.
- A basic function must be divisible into independent *stages* such that each stage have minimal overlap.
- The complexity of the stages should be roughly similar.

The pipelining in computer organization is basically flow of information. To understand how it works for the computer system lets consider an process which involves four steps / segment and the process is to be repeated six times. If single steps take t nsec time then time required to complete one process is 4 t nsec and to repeat it 6 times we require 24t nsec.

Now let’s see how problem works behaves with pipelining concept. This can be illustrated with a space time diagram given below figure 3.1, which shows the segment utilization as function of time. Lets us take there are 6 processes to be handled (represented in figure as P1, P2, P3, P4, P5 and P6) and each process is divided into 4 segments (S1, S2, S3, S4). For sake of simplicity we take each segment takes equal time to complete the assigned job i.e., equal to one clock cycle. The horizontal axis displays the time in clock cycles and vertical axis gives the segment number. Initially, process1 is handled by the segment 1. After the first clock segment 2 handles process 1 and segment 1 handles new process P2. Thus first process will take 4 clock cycles and remaining processes will be completed one process each clock cycle. Thus for above example total time required to complete whole job will be 9 clock cycles (with pipeline organization) instead of 24 clock cycles required for non pipeline configuration.

	1	2	3	4	5	6	7	8	9
P1	S1	S2	S3	S4					
P2		S1	S2	S3	S4				
P3			S1	S2	S3	S4			
P4				S1	S2	S3	S4		
P5					S1	S2	S3	S4	
P6						S1	S2	S3	S4

Figure 3.1 Space –time diagram for pipeline

Speedup ratio : The speed up ratio is ratio between maximum time taken by non pipeline process over process using pipelining. Thus in general if there are n processes and each process is divided into k segments (subprocesses). The first process will take k segments to complete the processes, but once the pipeline is full that is first process is complete, it will take only one clock period to obtain an output for each process. Thus first process will take k clock cycles and remaining n-1 processes will emerge from the pipe at the one process per clock cycle thus total time taken by remaining process will be (n-1) clock cycle time.

Let t_p be the one clock cycle time.

The time taken for n processes having k segments in pipeline configuration will be

$= k*t_p + (n-1)*t_p= (k+n-1)*t_p$

the time taken for one process is t_n thus the time taken to complete n process in non pipeline configuration will be

$= n*t_n$

Thus speed up ratio for one process in non pipeline and pipeline configuration is

$= n*t_n / (n+k-1)*t_p$

if n is very large compared to k than

$=t_n / t_p$

if a process takes same time in both case with pipeline and non pipeline configuration than $t_n = k*t_p$

Thus speed up ratio will $S_k =k*t_p/t_p =k$

Theoretically maximum speedup ratio will be k where k are the total number of segments in which process is divided. The following are various limitations due to which any pipeline system cannot operate at its maximum theoretical rate i.e., k (speed up ratio).

a. Different segments take different time to complete there suboperations, and in pipelining clock cycle must be chosen equal to time delay of the segment with maximum propagation time. Thus all other segments have to waste time waiting for next clock cycle. The possible solution for improvement here can if possible subdivide the segment into different stages i.e., increase the number of stages and if segment is not subdivisible than use multiple of resource for segment causing maximum delay so that more than one instruction can be executed in to different resources and overall performance will improve.
b. Additional time delay may be introduced because of extra circuitry or additional software requirement is needed to overcome various hazards, and store the result in the intermediate registers. Such delays are not found in non pipeline circuit.
c. Further pipelining can be of maximum benefit if whole process can be divided into suboperations which are independent to each other. But if there is some resource conflict or data dependency i.e., a instruction depends on the result of pervious instruction which is not yet available than instruction has to wait till result become available or conditional or non conditional branching i.e., the bubbles or time delay is introduced.

Efficiency : The efficiency of linear pipeline is measured by the percentage of time when processor are busy over total time taken i.e., sum of busy time plus idle time. Thus if n is number of task , k is stage of pipeline and t is clock period then efficiency is given by

$\eta = n/ [k + n -1]$

Thus larger number of task in pipeline more will be pipeline busy hence better will be efficiency. It can be easily seen from expression as $n \rightarrow \infty$, $\eta \rightarrow 1$.

$\eta = S_k/k$

Thus efficiency η of the pipeline is the speedup divided by the number of stages, or one can say actual speed ratio over ideal speed up ratio. In steady stage where n>>k, η approaches 1.

Throughput: The number of task completed by a pipeline per unit time is called throughput, this represents computing power of pipeline. We define throughput as

$W= n/[k*t + (n-1) *t] = \eta/t$

In ideal case as η -> 1 the throughout is equal to 1/t that is equal to frequency. Thus maximum throughput is obtained is there is one output per clock pulse.

Que 3.1. A non-pipeline system takes 60 ns to process a task. The same task can be processed in six segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?

Soln. Total time taken by for non pipeline to complete 100 task is = 100 * 60 = 6000 ns

Total time taken by pipeline configuration to complete 100 task is

= (100 + 6 –1) *10 = 1050 ns

Thus speed up ratio will be = 6000 / 1050 = 4.76

The maximum speedup that can be achieved for this process is = 60 / 10 = 6

Thus, if total speed of non pipeline process is same as that of total time taken to complete a process with pipeline than maximum speed up ratio is equal to number of segments.

Que 3.2. A non-pipeline system takes 50 ns to process a task. The same task can be processed in a six segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?

Soln. Total time taken by for non pipeline to complete 100 task is = 100 * 50 = 5000 ns

Total time taken by pipeline configuration to complete 100 task is

= (100 + 6 –1) *10 = 1050 ns

Thus speed up ratio will be = 5000 / 1050 = 4.76

The maximum speedup that can be achieved for this process is = 50 / 10 = 5

The two areas where pipeline organization is most commonly used are arithmetic pipeline and instruction pipeline. An arithmetic pipeline where different stages of an arithmetic operation are handled along the stages of a pipeline i.e., divides the arithmetic operation into suboperations for execution of pipeline segments. An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle as different stages of pipeline. RISC architecture supports pipelining more than a CISC architecture does. There are three prime disadvantages of pipeline architecture.

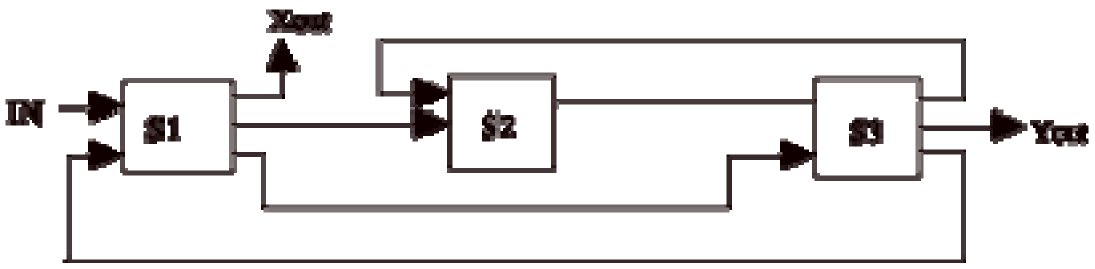
1. The first is complexity i.e., to divide the process into dependent subtask
2. Many intermediate registers are required to hold the intermediate information as output of one stage which will be input of next stage. These are not required for single unit circuit thus it is usually constructed entirely as combinational circuit
3. The third disadvantage is its inability to continuously run the pipeline at full speed, i.e. the pipeline stalls for some cycle. There are phenomena called pipeline hazards which disrupt the smooth execution of the pipeline if these hazards are not handled properly they may gave wrong result. Often it is required insert delays in the pipeline flow in order to manage these hazards such delays are called bubbles. Often it is managed by using special hardware techniques while sometime using software techniques such as compiler or code reordering, etc. Various types of pipeline hazards include:
   - structural hazards that happens due to hardware conflicts
   - data hazards that happen due to data dependencies
   - control hazards that happens when there is change in flow of statement like due to branch, jump, or any other control flow changes conditions
   - Exception hazard that happens due to some exception or interrupt occurred while execution in a pipeline system.

3.3 Non linear pipeline

A dynamic pipeline can be reconfigured to perform variable function at different times. The traditional linear pipelines are static pipeline because they used to perform fixed function. A dynamic pipeline allows feed forward and feedback connections in addition to streamline connection. A dynamic pipelining may initiate tasks from different reservation tables simultaneously to allow multiple numbers of initiations of different functions in the same pipeline.

3.3.1 Reservation Tables and latency analysis

Reservation tables are used how successive pipeline stages are utilized for a specific evaluation function. These reservation tables show the sequence in which each function utilizes each stage. The rows correspond to pipeline stages and the columns to clock time units. The total number of clock units in the table is called the evaluation time. A reservation table represents the flow of data through the pipeline for one complete evaluation of a given function. (For example, think of X as being a floating square root, and Y as being a floating cosine. A simple floating multiply might occupy just S1 and S2 in sequence.) We could also denote multiple stages being used in parallel, or a stage being drawn out for more than one cycle with these diagrams.



S1	X					X		X
S2		X		X				
S3			X		X		X	

S1	Y				Y	
S2			Y			
S3		Y		Y		Y

We determine the next start time for one or the other of the functions by lining up the diagrams and sliding one with respect to another to see where one can fit into the open slots. Once an X function has been scheduled, another X function can start after 1, 3 or 6 cycles. A Y function can start after 2 or 4 cycles. Once a Y function has been scheduled, another Y function can start after 1, 3 or 5 cycles. An X function can start after 2 or 4 cycles. After two functions have been scheduled, no more can be started until both are complete.

Job Sequencing and Collision Prevention

Initiation the start a single function evaluation collision may occur as two or more initiations attempt to use the same stage at the same time. Thus it is required to properly schedule queued tasks awaiting initiation in order to avoid collisions and to achieve high throughput. We can define collision as:

1. A collision occurs when two tasks are initiated with latency (initiation interval) equal to the column distance between two “X” on some row of the reservation table.

2. The set of column distances F ={l1,l2,…,lr} between all possible pairs of “X” on each row of the reservation table is called the forbidden set of latencies.

3. The collision vector is a binary vector C = (Cn…C2 C1), Where Ci=1 if i belongs to F (set of forbidden latencies) and Ci=0 otherwise.

Some fundamental concepts used in it are**:**

Latency - number of time units between two initiations (any positive integer 1, 2,…)

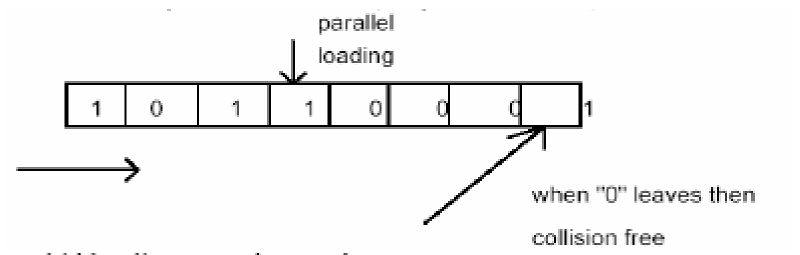
Latency sequence – sequence of latencies between successive initiations

Latency cycle – a latency sequence that repeats itself

Control strategy – the procedure to choose a latency sequence

Greedy strategy – a control strategy that always minimizes the latency between the current initiation and the very last initiation

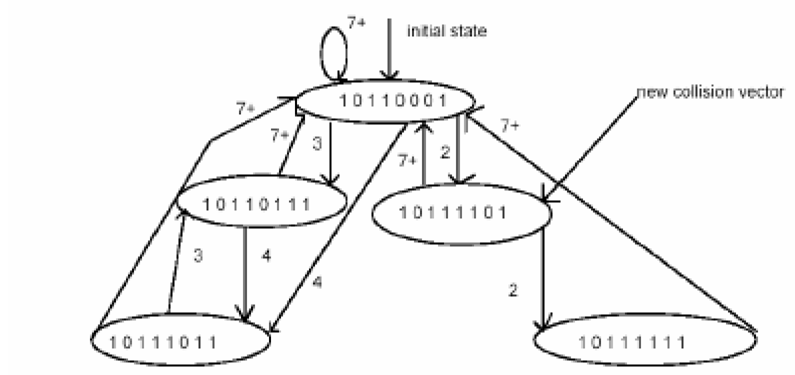
Example: Let us consider a Reservation Table with the following set of forbidden latencies F and permitted latencies P (complementation of F).



Forbidden list = F = {1,5,6,8}
Collision vector: C={1 0 1 1 0 0 0 1)
8 7 6 5 4 3 2 1

	0	1	2	3	4	5	6	7	8
1	X								X
2		X	X					X	
3				X					
4					X	X			
5							X	X	

```
1 0 1 1 0 0 0 1
1 0 1 1 0 1 1 1
OR     1 0 1 1
1 0 1 1 1 0 1 1
```



```
    1 0 1 1 0 0 0 1     initial
(+) 0 0 1 0 1 1 0 0     →2 (initial)
    1 0 1 1 1 1 0 1

    1 0 1 1 0 0 0 1     initial
    0 0 0 1 0 1 1 0     →2 (initial)
    1 0 1 1 0 1 1 1

    1 0 1 1 0 0 0 1     initial
    0 0 0 0 1 0 1 1     →4 (initial)

    1 0 1 1 0 0 0 1     initial
    0 0 0 0 0 0 0 1     →7 (initial)
    1 0 1 1 0 0 0 1
```

It has been observed that

1. The collision vector shows both permitted and forbidden latencies from the same reservation table.

2. One can use n-bit shift register to hold the collision vector for implementing a control strategy for successive task initiations in the pipeline. Upon initiation of the first task, the collision vector is parallel-loaded into the shift register as the initial state. The shift register is then shifted right one bit at a time, entering 0's from the left end. A collision free initiation is allowed at time instant t+k a bit 0 is being shifted at of the register after k shifts from time t.

A **state diagram** is used to characterize the successive initiations of tasks in the pipeline in order to find the shortest latency sequence to optimize the control strategy. A **state** on the diagram is represented by the contents of the shift register after the proper number of shifts is made, which is equal to the latency between the current and next task initiations.

3. The successive collision vectors are used to prevent future task collisions with previously initiated tasks, while the collision vector C is used to prevent possible collisions with the current task. If a collision vector has a "1" in the ith bit (from the right), at time t, then the task sequence should avoid the initiation of a task at time t+i.

4. Closed logs or cycles in the state diagram indicate the steady – state sustainable latency sequence of task initiations without collisions. The **average latency** of a cycle is the sum of its latencies (period) divided by the number of states in the cycle.

5. The throughput of a pipeline is inversely proportional to the reciprocal of the average latency. A latency sequence is called **permissible** if no collisions exist in the successive initiations governed by the given latency sequence.

6. The maximum throughput is achieved by an optimal scheduling strategy that achieves the (MAL) minimum average latency without collisions.

Simple cycles are those latency cycles in which each state appears only once per each iteration of the cycle. A single cycle is a **greedy cycle** if each latency contained in the cycle is the minimal latency (outgoing arc) from a state in the cycle. A good task-initiation sequence should include the greedy cycle.

Procedure to determine the greedy cycles

1. From each of the state diagram, one chooses the arc with the smallest latency label unit; a closed simple cycle can formed.

2. The average latency of any greedy cycle is no greater than the number of latencies in the forbidden set, which equals the number of 1's in the initial collision vector.

3. The average latency of any greedy cycle is always lower-bounded by the
MAL in the collision vector

Two methods for improving dynamic pipeline throughput have been proposed by Davidson and Patel these are

- The reservation of a pipeline can be modified with insertion of non complete delays
- Use of internal buffer at each stage.

Thus high throughput can be achieved by using the modified reservation table yielding a more desirable latency pattern such the each stage is maximum utilized. Any computation can be delayed by inserting a non compute stage.

Reconfigurable pipelines with different function types are more desirable. This requires an extensive resource sharing among different functions. To achieve this one need a more complicated structure of pipeline segments and their interconnection controls like bypass techniques to avoid unwanted stage.

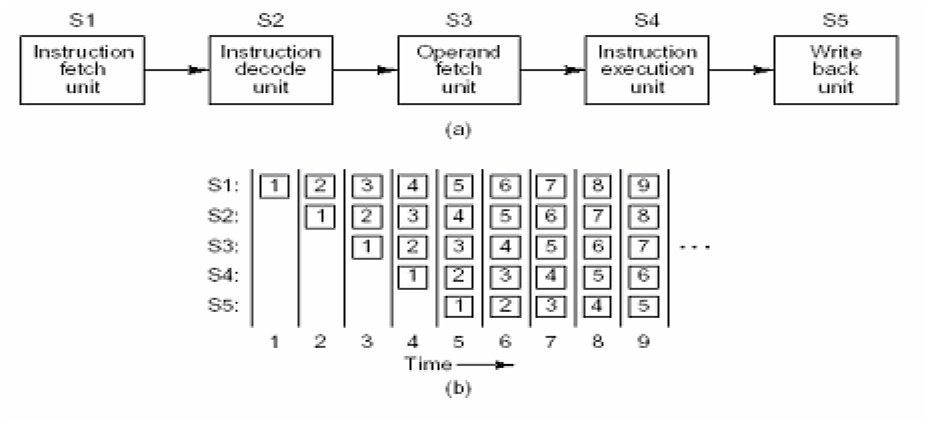
A dynamic pipeline would allow several configurations to be simultaneously present like arithmetic unit performing both addition as well as multiplication at same time. But to achieve this tremendous control overhead and increased interconnection complexity would be expected.

3.4 Design of Instruction pipeline

As we know that in general case, the each instruction to execute in computer undergo following steps:

- Fetch the instruction from the memory.
- Decode the instruction.
- Calculate the effective address.
- Fetch the operands from the memory.
- Execute the instruction (EX).
- Store the result back into memory (WB).

For sake of simplicity we take calculation of the effective address and fetch operand from memory as single segment as operand fetch unit. Thus below figure shows how the instruction cycle in CPU can be processed with five segment instruction pipeline.



While the instruction is decoded (ID) in segment 2 the new instruction is fetched (IF) from segment 1. Similarly in third time cycle when first instruction effective operand is fetch (OF), the 2nd instruction is decoded and the 3rd instruction is fetched. In same manner in fourth clock cycle, and subsequent cycles all subsequent instructions can be fetched and placed in instruction FIFO. Thus up to five different instructions can be processed at the same time. The figure show how the instruction pipeline works, where time is in the horizontal axis and divided into steps of equal duration. Although the major difficulty with instruction pipeline is that different segment may take different time to operate the forth coming information. For example if operand is in register mode require much less time as compared if operand has to be fetched from memory that to with indirect addressing modes. The design of an instruction pipeline will be most effective if the instruction cycle is divided into segments of equal duration. As there can be resource conflict, data dependency, branching, interrupts and other reasons due to pipelining can branch out of normal sequence.

Que 5.3 Consider a program of 15,000 instructions executed by a linear pipeline processor with a clock rate of 25MHz. The instruction pipeline has five stages and one instruction is issued per clock cycle. Calculate speed up ratio, efficiency and throughput of this pipelined processor?

Soln: Time taken to execute without pipeline is = 15000 * 5* (1/25) microsecs

Time taken with pipeline = (15000 + 5 -1)*(1/ 25) microsecs

Speed up ratio = (15000*5*25) / (15000+ 5 -1)*25 = 4.99

Efficiency = Speed up ratio/ number of segment in pipeline = 4.99/5= 0.99

Throughput = number of task completed in unit time = 0.99 * 25 = 24.9 MIPS

Principles of designing pipeline processor

Buffers are used to speed close up the speed gap between memory access for either instructions or operands. Buffering can avoid unnecessary idling of the processing stages caused by memory access conflicts or by unexpected branching or interrupts. The concepts of busing eliminates the time delay to store and to retrieve intermediate results or to from the registers.

The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine transitive or multiple fetch-store operations with faster register operations. This is carried by register tagging and forwarding.

Another method to smooth the traffic flow in a pipeline is to use buffers to close up the speed gap between the memory accesses for either instructions or operands and arithmetic and logic executions in the functional pipes. The instruction or operand buffers provide a continuous supply of instructions or operands to the appropriate pipeline units. Buffering can avoid unnecessary idling of the processing stages caused by memory access conflicts or by unexpected branching or interrupts. Sometimes the entire loop instructions can be stored in the buffer to avoid repeated fetch of the same instructions loop, if the buffer size is sufficiently large. It is very large in the usage of pipeline computers.

Three buffer types are used in various instructions and data types. Instructions are fetched to the instruction fetch buffer before sending them to the instruction unit. After decoding, fixed point and floating point instructions and data are sent to their dedicated buffers. The store address and data buffers are used for continuously storing results back to memory.

Busing Buffers

The sub function being executed by one stage should be independent of the other sub functions being executed by the remaining stages; otherwise some process in the pipeline must be halted until the dependency is removed. When one instruction waiting to be executed is first to be modified by a future instruction, the execution of this instruction must be suspended until the dependency is released.

Another example is the conflicting use of some registers or memory locations by different segments of a pipeline. These problems cause additional time delays. An efficient internal busing structure is desired to route the resulting stations with minimum time delays.

Internal Forwarding and Register Tagging

To enhance the performance of computers with multiple execution pipelines

1. **Internal Forwarding** refers to a short circuit technique for replacing unnecessary memory accesses by register -to-register transfers in a sequence of fetch-arithmetic-store operations
2. **Register Tagging** refers to the use of tagged registers, buffers and reservations stations for exploiting concurrent activities among multiple arithmetic units.

The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine transitive or multiple fetch-store operations with faster register operations. This concept of internal data forwarding can be explored in three directions. The symbols Mi and Rj to represent the ith word in the memory and jth fetch, store and register-to register transfer. The contents of Mi and Rj are represented by (Mi) and Rj

Store-Fetch Forwarding The store the n fetch can be replaced by 2 parallel operations, one store and one register transfer.

2 memory accesses

Mi -> (R1) (store)

R2 -> (Mi) (Fetch)

Cab be replaced by only one memory access

Mi -> (R1) (store)

R2 -> (R1) (register Transfer)

Fetch-Fetch Forwarding The following fetch operations can be replaced by one fetch and one register transfer. One memory access has been eliminated.

2 memory accesses

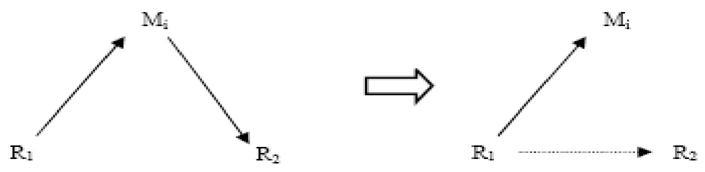
R1 -> (Mi) (fetch)

R2 -> (Mi) (Fetch)

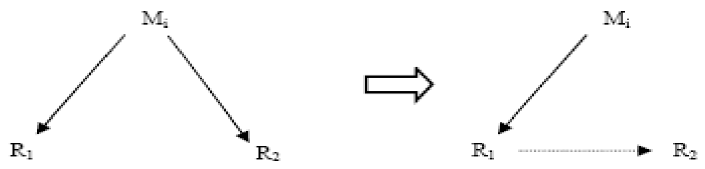
Is being replaced by Only by one memory access

R1 -> (Mi) (Fetch)

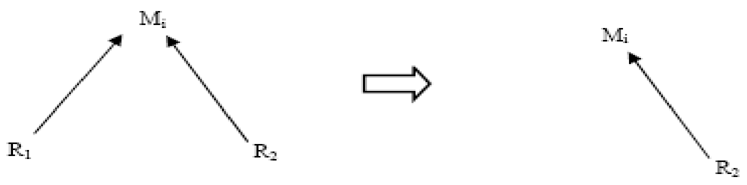
R2 -> (R1) (register Transfer)



Store – Fetch Forwarding



Fetch – Fetch Forwarding



Store-Store overwriting

Store-Store Overwriting

The following two memory updates of the same word can be combined into one; since the second store overwrites the first. 2 memory accesses

Mi -> (R1) (store)

Mi -> (R2) (store)

Is being replaced by only by one memory access

Mi -> (R2) (store)

The above steps shows how to apply internal forwarding to simplify a sequence of arithmetic and memory access operations in figure thick arrows for memory accesses and dotted arrows for register transfers

Forwarding and Data Hazards

Sometimes it is possible to avoid data hazards by noting that a value that results from one instruction is not needed until a late stage in a following instruction, and sending the data directly from the output of the first functional unit back to the input of the second one (which is sometimes the same unit). In the general case, this would require the output of every functional unit to be connected through switching logic to the input of every functional unit.

Data hazards can take three forms:

Read after write (RAW): Attempting to read a value that hasn't been written yet. This is the most common type, and can be overcome by forwarding.

Write after write (WAW): Writing a value before a preceding write has completed. This can only happen in complex pipes that allow instructions to proceed out of order, or that have multiple write-back stages (mostly CISC), or when we have multiple pipes that can write (superscalar).

Write after read (WAR): Writing a value before a preceding read has completed. These also require a complex pipeline that can sometimes write in an early stage, and read in a later stage. It is also possible when multiple pipelines (superscalar) or out-of-order issue are employed.

The fourth situation, read after read (RAR) does not produce a hazard.

Forwarding does not solve every RAW hazard situation. For example, if a functional unit is merely slow and fails to produce a result that can be forwarded in time, then the pipeline must stall. A simple example is the case of a load, which has a high latency. This is the sort of situation where compiler scheduling of instructions can help, by rearranging independent instructions to fill the delay slots. The processor can also rearrange the instructions at run time, if it has access to a window of prefetched instructions (called a prefetch buffer). It must perform much the same analysis as the compiler to determine which instructions are dependent on each other, but because the window is usually small, the analysis is more limited in scope. The small size of the window is due to the cost of providing a wide enough datapath to predecode multiple instructions at once, and the complexity of the dependence testing logic.

Out of order execution introduces another level of complexity in the control of the pipeline, because it is desirable to preserve the abstraction of in-order issue, even in the presence of exceptions that could flush the pipe at any stage. But we'll defer this to later.

Branch Penalty Hiding

The control hazards due to branches can cause a large part of the pipeline to be flushed, greatly reducing its performance. One way of hiding the branch penalty is to fill the pipe behind the branch with instructions that would be executed whether or not the branch is taken. If we can find the right number of instructions that precede the branch and are independent of the test, then the compiler can move them immediately following the branch and tag them as branch delay filling instructions. The processor can then execute the branch, and when it determines the appropriate target, the instruction is fetched into the pipeline with no penalty.

The filling of branch delays can be done dynamically in hardware by reordering instructions out of the prefetch buffer. But this leads to other problems. Another way to hide branch penalties is to avoid certain kinds of branches. For example, if we have

```
IF A < 0
  THEN A = -A
```

we would normally implement this with a nearby branch. However, we could instead use an instruction that performs the arithmetic conditionally (skips the write back if the condition fails). The advantage of this scheme is that, although one pipeline cycle is wasted, we do not have to flush the rest of the pipe (also, for a dynamic branch prediction scheme, we need not put an extra branch into the prediction unit). These are called predicated instructions, and the concept can be extended to other sorts of operations, such as conditional loading of a value from memory.

Branch Prediction

Branches are the bane of any pipeline, causing a potentially large decrease in performance as we saw earlier. There are several ways to reduce this loss by predicting the action of the branch ahead of time.

Simple static prediction assumes that all branches will be taken or not. The designer decides which way is predicted from instruction trace statistics. Once the choice is made, the compiler can help by properly ordering local jumps. A slightly more complex static branch prediction heuristic is that backward branches are usually taken and forward branches are not (backwards taken, forwards not or BTFN). This assumes that most backward branches are loop returns and that most forward branches are the less likely cases of a conditional branch.

Compiler static prediction involves the use of special branches that indicate the most likely choice (taken or not, or more typically taken or other, since the most predictable branches are those at the ends of loops that are mostly taken). If the prediction fails in this case, then the usual cancellation of the instructions in the delay slots occurs and a branch penalty results.

Dynamic instruction scheduling

As discussed above the static instruction scheduling can be optimized by compiler the dynamic scheduling is achieved either by using scoreboard or with Tomasulo's register tagging algorithm and discussed in superscalar processors

3.5 Arithmetic pipeline

Pipeline arithmetic is used in very high speed computers specially involved in scientific computations a basic principle behind vector processor and array processor. They are used to implement floating – point operations, multiplication of fixed – point numbers and similar computations encountered in computation problems. These computation problems can easily decomposed in suboperations. Arithmetic pipelining is well implemented in the systems involved with repeated calculations such as calculations involved with matrices and vectors. Let us consider a simple vector calculation like

A[i] + b[i] * c[i] for I = 1,2,3,……,8

The above operation can be subdivided into three segment pipeline such each segment has some registers and combinational circuits. Segment 1 load contents of b[i] and c[i] in register R1 and R2 , segment 2 load a[i] content to R3 and multiply content of R1, R2 and store them R4 finally segment 3 add content of R3 and R4 and store in R5 as shown in figure below.

Clock pulse number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	B1	C1	-	-	-
2	B2	C2	B1*C1	A1	
3	B3	C3	B2*C2	A2	A1+ B1*C1
4	B4	C4	B3*C3	A3	A2+ B2*C2
5	B5	C5	B4*C4	A4	A3+ B3*C3

6	B6	C6	B5*C5	A5	A4+ B4*C4
7	B7	C7	B6*C6	A6	A5+ B5*C5
8	B8	C8	B7*C7	A7	A6+ B6*C6
9			B8*C8	A8	A7+ B7*C7
10					A8+ B8*C8

To illustrate the operation principles of a pipeline computation, the design of a pipeline floating point adder is given. It is constructed in four stages. The inputs are

A = a x 2p

B = b x 2q

Where a and b are 2 fractions and p and q are their exponents and here base 2 is assumed.

To compute the sum

C = A+ B = c x 2r = d x 2s

Operations performed in the four pipeline stages are specified.

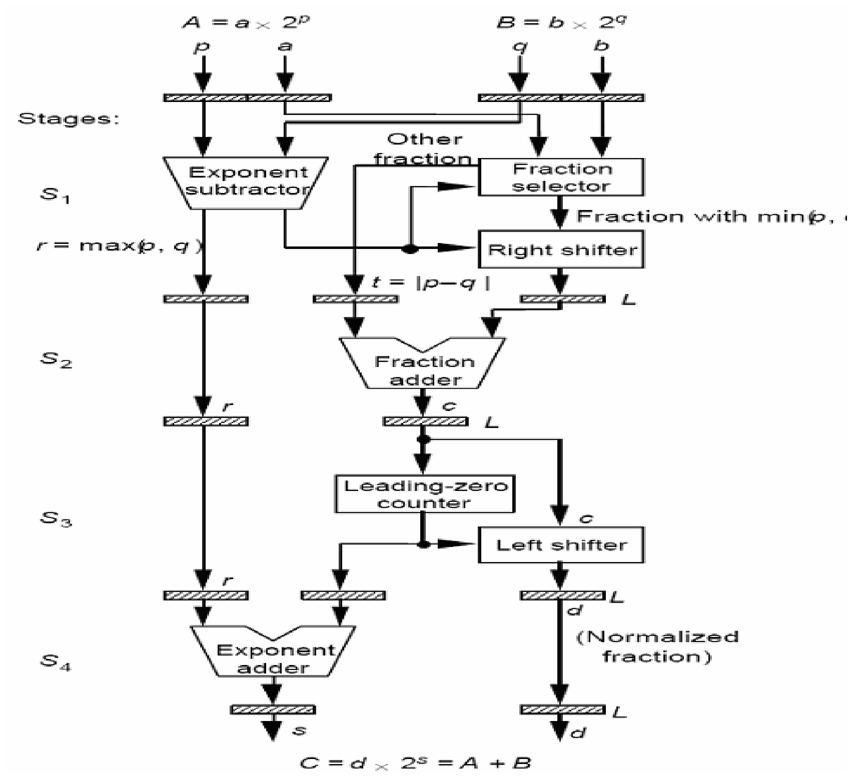
1. Compare the 2 exponents p and q to reveal the larger exponent r =max(p,q) and to determine their difference t =p-q

2. Shift right the fraction associated with the smaller exponent by t bits to equalize the two components before fraction addition.

3. Add the preshifted fraction with the other fraction to produce the intermediate sum fraction c where 0 <= c <1.

4. Count the number of leading zeroes, say u, in fraction c and shift left c by u bits to produce the normalized fraction sum d = c x 2u, with a leading bit 1. Update the large exponent s by subtracting s= r – u to produce the output exponent.

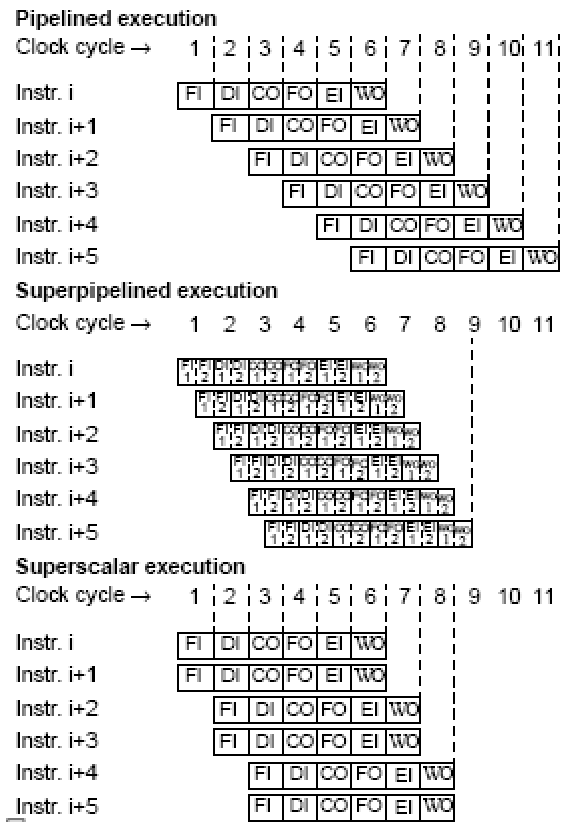
The given below is figure show how pipeline can be implemented in floating point addition and subtraction. Segment 1 compare the two exponents this is done using subtraction. Segment2 we chose the larger exponents the one larger exponent as exponent of result also it align the other mantissa by viewing the difference between two and smaller number mantissa should be shifted to right by difference amount. Segment 3 performs addition or subtraction of mantissa while segment 4 normalize the result for that it adjust exponent care must be taken in case of overflow, where we had to shift the mantissa right and increment exponent by one and for underflow the leading zeros of mantissa determines the left shift in mantissa and same number should be subtracted for exponent. Various registers R are used to hold intermediate results.



In order to implement pipelined adder we need extra circuitry but its cost is compensated if we have implement it for large number of floating point numbers. Operations at each stage can be done on different pairs of inputs, e.g. one stage can be comparing the exponents in one pair of operands at the same time another stage is adding the mantissas of a different pair of operands.

3.6 S**uperpipeline and Superscalar technique**

Instruction level parallelism is obtained primarily in two ways in uniprocessors: through pipelining and through keeping multiple functional units busy executing multiple instructions at the same time. When a pipeline is extended in length beyond the normal five or six stages (e.g., I-Fetch, Decode/Dispatch, Execute, D-fetch, Writeback), then it may be called Superpipelined. If a processor executes more than one instruction at a time, it may be called Superscalar. A superscalar architecture is one in which several instructions can be initiated simultaneously and executed independently. These two techniques can be combined into a Superscalar pipeline architecture.



3.6.1 Superpipeline

In order to make processors even faster, various methods of optimizing pipelines have been devised. Superpipelining refers to dividing the pipeline into more steps. The more pipe stages there are, the faster the pipeline is because each stage is then shorter. thus Superpipelining increases the number of instructions which are supported by the pipeline at a given moment. For example if we divide each stage into two, the clock cycle period t will be reduced to the half, t/2; hence, at the maximum capacity, the pipeline produces a result every t/2 s. For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance Ideally, a pipeline with five stages should be five times faster than a non-pipelined processor (or rather, a pipeline with one stage). The instructions are executed at the speed at which each stage is completed, and each stage takes one fifth of the amount of time that the non-pipelined instruction takes. Thus, a processor with an 8-step pipeline (the MIPS R4000) will be even faster than its 5-step counterpart. The MIPS R4000 chops its pipeline into more pieces by dividing some steps into two. Instruction fetching, for example, is now done in two stages rather than one. The stages are as shown:

Instruction Fetch (First Half)

Instruction Fetch (Second Half)

Register Fetch

Instruction Execute

Data Cache Access (First Half)

Data Cache Access (Second Half)

Tag Check

Write Back

Given a pipeline stage time T, it may be possible to execute at a higher rate by starting operations at intervals of T/n. This can be accomplished in two ways:

Further divide each of the pipeline stages into n substages.

Provide n pipelines that are overlapped.

The first approach requires faster logic and the ability to subdivide the stages into segments with uniform latency. It may also require more complex inter-stage interlocking and stall-restart logic.

The second approach could be viewed in a sense as staggered superscalar operation, and has associated with it all of the same requirements except that instructions and data can be fetched with a slight offset in time. In addition, inter-pipeline interlocking is more difficult to manage because of the sub-clock period differences in timing between the pipelines.

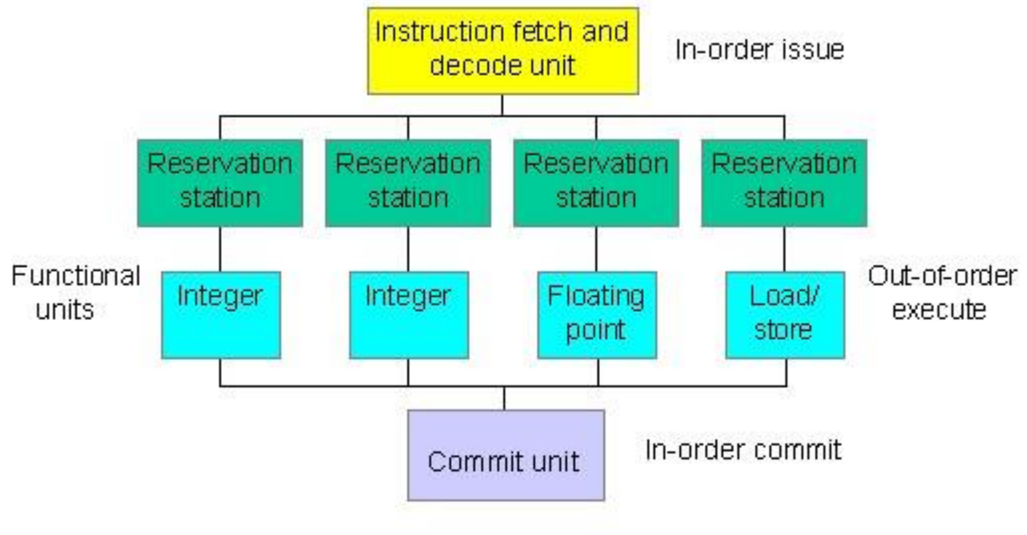
Inevitably, superpipelining is limited by the speed of logic, and the frequency of unpredictable branches. Stage time cannot productively grow shorter than the interstage latch time, and so this is a limit for the number of stages.

The MIPS R4000 is sometimes called a superpipelined machine, although its 8 stages really only split the I-fetch and D-fetch stages of the pipe and add a Tag Check stage. Nonetheless, the extra stages enable it to operate with higher throughput. The UltraSPARC's 9-stage pipe definitely qualifies it as a superpipelined machine, and in fact it is a Super-Super design because of its superscalar issue. The Pentium 4 splits the pipeline into 20 stages to enable increased clock rate. The benefit of such extensive pipelining is really only gained for very regular applications such as graphics. On more irregular applications, there is little performance advantage.

3.6.2 Superscalar

A solution to further improve speed is the superscalar architecture. Superscalar pipelining involves multiple pipelines in parallel. Internal components of the processor are replicated so it can launch multiple instructions in some or all of its pipeline stages. The RISC System/6000 has a forked pipeline with different paths for floating-point and integer instructions. If there is a mixture of both types in a program, the processor can keep both forks running simultaneously. Both types of instructions share two initial stages (Instruction Fetch and Instruction Dispatch) before they fork. Often, however, superscalar pipelining refers to multiple copies of all pipeline stages (In terms of laundry, this would mean four washers, four dryers, and four people who fold clothes). Many of today's machines attempt to find two to six instructions that it can execute in every pipeline stage. If some of the instructions are dependent, however, only the first instruction or instructions are issued.

Dynamic pipelines have the capability to schedule around stalls. A dynamic pipeline is divided into three units: the instruction fetch and decode unit, five to ten execute or functional units, and a commit unit. Each execute unit has reservation stations, which act as buffers and hold the operands and operations.



While the functional units have the freedom to execute out of order, the instruction fetch/decode and commit units must operate in-order to maintain simple pipeline behavior. When the instruction is executed and the result is calculated, the commit unit decides when it is safe to store the result. If a stall occurs, the processor can schedule other instructions to be executed until the stall is resolved. This, coupled with the efficiency of multiple units executing instructions simultaneously, makes a dynamic pipeline an attractive alternative

Superscalar processing has its origins in the Cray-designed CDC supercomputers, in which multiple functional units are kept busy by multiple instructions. The CDC machines could pack as many as 4 instructions in a word at once, and these were fetched together and dispatched via a pipeline. Given the technology of the time, this configuration was fast enough to keep the functional units busy without outpacing the instruction memory.

In some cases superscalar machines still employ a single fetch-decode-dispatch pipe that drives all of the units. For example, the UltraSPARC splits execution after the third stage of a unified pipeline. However, it is becoming more common to have multiple fetch-decode-dispatch pipes feeding the functional units.

The choice of approach depends on tradeoffs of the average execute time vs. the speed with which instructions can be issued. For example, if execution averages several cycles, and the number of functional units is small, then a single pipe may be able to keep the units utilized. When the number of functional units grows large and/or their execution time approaches the issue time, then multiple issue pipes may be necessary.

Having multiple issue pipes requires

- being able to fetch instructions for that many pipes at once
- inter-pipeline interlocking
- reordering of instructions for multiple interlocked pipelines
- multiple write-back stages
- multiport D-cache and/or register file, and/or functionally split register file

Reordering may be either static (compiler) or dynamic (using hardware lookahead). It can be difficult to combine the two approaches because the compiler may not be able to predict the actions of the hardware reordering mechanism.

Superscalar operation is limited by the number of independent operations that can be extracted from an instruction stream. It has been shown in early studies on simpler processor models, that this is limited, mostly by branches, to a small number (<10, typically about 4). More recent work has shown that, with speculative execution and aggressive branch prediction, higher levels may be achievable. On certain highly regular codes, the level of parallelism may be quite high (around 50). Of course, such highly regular codes are just as amenable to other forms of parallel processing that can be employed more directly, and are also the exception rather than the rule. Current thinking is that about 6-way instruction level parallelism for a typical program mix may be the natural limit, with 4-way being likely for integer codes. Potential ILP may be three times this, but it will be very difficult to exploit even a majority of this parallelism. Nonetheless, obtaining a factor of 4 to 6 boost in performance is quite significant, especially as processor speeds approach their limits.

Going beyond a single instruction stream and allowing multiple tasks (or threads) to operate at the same time can enable greater system throughput. Because these are naturally independent at the fine-grained level, we can select instructions from different streams to fill pipeline slots that would otherwise go vacant in the case of issuing from a single thread. In turn, this makes it useful to add more functional units. We shall further explore these multithreaded architectures later in the course.

Hardware Support for Superscalar Operation

There are two basic hardware techniques that are used to manage the simultaneous execution of multiple instructions on multiple functional units: Scoreboarding and reservation stations. Scoreboarding originated in the Cray-designed CDC-6600 in 1964, and reservation stations first appeared in the IBM 360/91 in 1967, as designed by Tomasulo.

Scoreboard

A scoreboard is a centralized table that keeps track of the instructions to be performed and the available resources and issues the instructions to the functional units when everything is ready for them to proceed. As the instructions execute, dependences are checked and execution is stalled as necessary to ensure that in-order semantics are preserved. Out of order execution is possible, but is limited by the size of the scoreboard and the execution rules. The scoreboard can be thought of as preceding dispatch, but it also controls execution after the issue. In a scoreboarded system, the results can be forwarded directly to their destination register (as long as there are no write after read hazards, in which case their execution is stalled), rather than having to proceed to a final write-back stage.

In the CDC scoreboard, each register has a matching Result Register Designator that indicates which functional unit will write a result into it. The fact that only one functional unit can be designated for writing to a register at a time ensures that WAW dependences cannot occur. Each functional unit also has a corresponding set of Entry-Operand Register Designators that indicate what register will hold each operand, whether the value is valid (or pending) and if it is pending, what functional unit will produce it (to facilitate forwarding). None of the operands is released to a functional unit until they are all valid, precluding RAW dependences. In addition , the scoreboard stalls any functional unit whose result would write a register that is still listed as an Entry-Operand to a functional unit that is waiting for an operand or is busy, thus avoiding WAR violations. An instruction is only allowed to issue if its specified functional unit is free and its result register is not reserved by another functional unit that has not yet completed. Four Stages of Scoreboard Control

1. Issue—decode instructions & check for structural hazards (ID1) If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. Read operands—wait until no data hazards, then read operands (ID2) A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

3. Execution—operate on operands (EX) The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. Write result—finish execution (WB) Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards.

If none, it writes results. If WAR, then it stalls the instruction. Example:

DIVD F0,F2,F4

ADDD F10,F0,**F8**

SUBD **F8**,F8,F14

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Three Parts of the Scoreboard

1. Instruction status—which of 4 steps the instruction is in

2. Functional unit status—Indicates the state of the functional unit (FU). 9 fields for each functional unit

Busy—Indicates whether the unit is busy or not

Op—Operation to perform in the unit (e.g., + or –)

Fi—Destination register

Fj, Fk—Source-register numbers

Qj, Qk—Functional units producing source registers Fj, Fk

Rj, Rk—Flags indicating when Fj, Fk are ready and not yet read. Set to

No after operands are read.

3. Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

Scoreboard Implications

• provide solution for WAR, WAW hazards

• Solution for WAR – Stall Write in WB to allow Reads to take place; Read registers only during Read Operands stage.

• For WAW, must detect hazard: stall in the Issue stage until other completes

• Need to have multiple instructions in execution phase

• Scoreboard keeps track of dependencies, state or operations

– Monitors every change in the hardware.

– Determines when to read ops, when can execute, when can wb.

– Hazard detection and resolution is centralized.

Reservation Stations The reservation station approach releases instructions directly to a pool of buffers associated with their intended functional units (if more than one unit of a particular type is present, then the units may share a single station). The reservation stations are a distributed resource, rather than being centralized, and can be thought of as following dispatch. A reservation is a record consisting of an instruction and its requirements to execute -- its operands as specified by their sources and destination and bits indicating when valid values are available for the sources. The instruction is released to the functional unit when its requirements are satisfied, but it is important to note that satisfaction doesn't require an operand to actually be in a register -- it can be forwarded to the reservation station for immediate release or to be buffered (see below) for later release. Thus, the reservation station's influence on execution can be thought of as more implicit and data dependent than the explicit control exercised by the scoreboard.

Tomasulo Algorithm

The hardware dependence resolution technique used **For IBM 360/91 about 3 years after CDC 6600.** Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

If reservation station free, then issue instruction & send operands (renames registers).

2. Execution—operate on operands (EX)

When both operands ready then execute; if not ready, watch CDB for result

3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting units; mark reservation station available.

Here the storage of operands resulting from instructions that completed out of order is done through renaming of the registers. There are two mechanisms commonly used for renaming. One is to assign physical registers from a free pool to the logical registers as they are identified in an instruction stream. A lookup table is then used to map the logical register references to their physical assignments. Usually the pool is larger than the logical register set to allow for temporary buffering of results that are computed but not yet ready to write back. Thus, the processor must keep track of a larger set of register names than the instruction set architecture specifies. When the pool is empty, instruction issue stalls.

The other mechanism is to keep the traditional association of logical and physical registers, but then provide additional buffers either associated with the reservation stations or kept in a central location. In either case, each of these "reorder buffers" is associated with a given instruction, and its contents (once computed) can be used in forwarding operations as long as the instruction has not completed. When an instruction reaches the point that it may complete in a manner that preserves sequential semantics, then its reservation station is freed and its result appears in the logical register that was originally specified. This is done either by renaming the temporary register to be one of the logical registers, or by transferring the contents of the reorder buffer to the appropriate physical register.

Out of Order Issue

To enable out-of-order dispatch of instructions to the pipelines, we must provide at least two reservation stations per pipe that are available for issue at once. An alternative would be to rearrange instructions in the prefetch buffer, but without knowing the status of the pipes, it would be difficult to make such a reordering effective. By providing multiple reservation stations, however, we can continue issuing instructions to pipes, even though an instruction may be stalled while awaiting resources. Then, whatever instruction is ready first can enter the pipe and execute. At the far end of the pipeline, the out-of-order instruction must wait to be retired in the proper order. This necessitates a mechanism for keeping track of the proper order of instructions (note that dependences alone cannot guarantee that instructions will be properly reordered when they complete).

3.7 RISC Pipelines

An efficient way to use instruction pipeline is one of characteristic feature of RISC architecture. A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. As discussed earlier, the length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in pre-RISC processors (now called CISC, or Complex Instruction Set Computer), they are more conducive to pipelining. While CISC instructions varied in length, RISC instructions are all the same length and can be fetched in a single operation.

Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI). Hence RISC can achieve pipeline segments, just requiring just one clock cycle, while CISC may use many segments in its pipeline, with longest segment requiring two or more clock cycles.

As most RISC data manipulation operations have register to register operations and an instruction cycle has following two phase.

1. I : Instruction fetch
2. E : Execute . Performs an ALU operation with register input and output.

The data transfer instructions in RISC are limited to Load and Store. These instructions use register indirect addressing and require three stages in pipeline

I : Instruction fetch

E: Calculate memory address

D: Memory. Register to memory or memory to register operation.

To prevent conflicts between memory access to fetch an instruction and to load or store operand, most RISC machine use two separate buses with two memories: one or storing the instruction and other for storing data.

Another feature of RISC over CSIC as far as pipelining is considered is compiler support. Instead of designing hardware to handle the data dependencies and branch penalties, RISC relies on efficiency of compiler to detect and minimize the delay encountered with these problems.

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of steps, lets us consider a three segment Instruction pipeline

I: Instruction fetch

A: ALU operation

E: Execute instruction

The I segment fetches the instruction from memory and decode it. The ALU is used for three different functions, it can be data manipulation , effective address calculation for LOAD and STORE operations, or calculation of the branch address for a program control instruction depending on type of instruction. The E segment directs the output of the

ALU to one of three destination i.e., a destination register or effective address to a data memory for loading or storing or the branch address to program counter, depending upon decode instruction.

Multiplying Two Numbers in Memory

Lets consider an example of matrix multiplication here the main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.

The CISC Approach

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

```
MULT 2:3, 5:2
```

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

CRICS : CISC and RISC Convergence

State of the art processor technology has changed significantly since RISC chips were first introduced in the early '80s. Because a number of advancements (including the ones described on this page) are used by both RISC *and* CISC processors, the lines between the two architectures have begun to blur. In fact, the two architectures almost seem to have adopted the strategies of the other. Because processor speeds have increased, CISC chips are now able to execute more than one instruction within a single clock. This also allows CISC chips to make use of pipelining. With other technological improvements, it is now possible to fit many more transistors on a single chip. This gives RISC processors enough space to incorporate more complicated, CISC-like commands. RISC chips also make use of more complicated hardware, making use of extra function units for superscalar execution. All of these factors have led some groups to argue that we are now in a "post-RISC" era, in which the two styles have become so similar that distinguishing between them is no longer relevant. This era is often called complex reduced instruction set CRISC.

3.8 VLIW Machines

Very Long Instruction Word machines typically have many more functional units that superscalars (and thus the need for longer – 256 to 1024 bits – instructions to provide control for them) usually hundreds of bits long. These machines mostly use microprogrammed control units with relatively slow clock rates because of the need to use ROM to hold the microcode. Each instruction word essentially carries multiple “short instructions.” Each of the “short instructions” are effectively issued at the same time. (This is related to the long words frequently used in microcode.) Compilers for VLIW architectures should optimally try to predict branch outcomes to properly group instructions.

Pipelining in VLIW Processors

Decoding of instructions is easier in VLIW than in superscalars, because each “region” of an instruction word is usually limited as to the type of instruction it can contain. Code density in VLIW is less than in superscalars, because if a “region” of a VLIW word isn’t needed in a particular instruction, it must still exist (to be filled with a “no op”). Superscalars can be compatible with scalar processors; this is difficult with VLIW parallel and non-parallel architectures. “Random” parallelism among scalar operations is exploited in VLIW, instead of regular parallelism in a vector or SIMD machine.

The efficiency of the machine is entirely dictated by the success, or “goodness,” of the compiler in planning the operations to be placed in the same instruction words.

Different implementations of the same VLIW architecture may not be binary-compatible with each other, resulting in different latencies.

5.7 Summary

1. The job-sequencing problem is equivalent to finding a permissible latency cycle with the MAL in the state diagram.

2. The minimum number of X’s in array single row of the reservation table is a lower bound of the MAL.

Pipelining allows several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment. Superscalar architectures include all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage. They have the ability to initiate multiple instructions during the same clock cycle. There are two typical approaches today, in order to improve performance:

1. Superpipelining
2. Superscalar

VLIW reduces the effort required to detect parallelism using hardware or software techniques.

The main advantage of VLIW architecture is its simplicity in hardware structure and instruction set. Unfortunately, VLIW does require careful analysis of code in order to "compact" the most appropriate "short" instructions into a VLIW word.

3.9 Keywords

pipelining Overlapping the execution of two or more operations. Pipelining is used within processors by *prefetching* instructions on the assumption that no branches are going to preempt their execution; in *vector processors*, in which application of a single operation to the elements of a vector or vectors may be pipelined to decrease the time needed to complete the aggregate operation; and in *multiprocessors* and *multicomputers*, in which a process may send a request for values before it reaches the computation that requires them..

scoreboard A hardware device that maintains the state of machine resources to enable instructions to execute without conflict at the earliest opportunity.

instruction pipelining strategy of allowing more than one instruction to be in some stage of execution at the same time.

3.10 Self assessment questions

1. Explain an asynchronous pipeline model, a synchronous pipeline model and reservation table of a four-stage linear pipeline with appropriate diagrams.
2. Define the following terms with regard to clocking and timing control.

a) Clock cycle and throughput b) Clock skewing c) Speedup factor

3. Describe the speedup factors and the optimal number of pipeline stages for a linear pipeline unit.
4. Explain the features of non-linear pipeline processors with feedforward and feedbackward connections.
5. Explain the pipelined execution of the following instructions with the following instructions:

a) X = Y + Z b) A = B X C

6. What are the possible hazards that can occur between read and write operations in an instruction pipeline?

3.11 References/Suggested readings

Advance Computer architecture: Kai Hwang