8

Pipelined Systems: Low Level Parallelism

Regardless of the application of a computer system, there is always some motivation to enhance the speed of execution of the system. One way to achieve this is to use a technology that operates at a higher speed. and many machines have used this method to produce a system that performs more work in a given amount of time. However, there are practical limits to this method since there are practical limits to how fast signals will travel. If speed increases are to be achieved without a faster technology, then some degree of concurrency is necessary. If the circuits can't function faster, then to increase the apparent speed of the machine, the basic modules of the machine should perform functions simultaneously. This can be done if the functions to be performed are independent. That is, if the results of one instruction are not needed for the execution of another. then the two functions can be done at the same time. If this technique is successful, then instead of one operation per unit time, N operations per unit time are performed (for N independent units).

A variety of ways have been utilized to increase the amount of concurrent execution in computers. One method is to organize a number of identical functional units in such a way that they can all perform the same operation on different sets of data simultaneously. This is called single instruction stream, multiple data stream processing (SIMD), since many processors are executing simultaneously, but the action is controlled by a single unit. For certain classes of problems, this can be a very beneficial organization. Machines that have used this method of organization include the Illiac N [BaBr68, Thur76], the Massively Parallel Processor [Batc80, HwBr84], and the Connection Machine [Hill85].

Another method of performing simultaneous tasks is to divide the work to be done into portions that can operate in the same period. For example, von Neumann suggested that the I/O operations could occur simultaneous with processing [BuGo46], which is a common practice in computer systems today. Overlap can also be achieved by divi ng instruction execution into its constituent pans: fetch.

decode, and execute. That is, while one instruction is being decoded, it may be possible to fetch the next instruction. And at the same time perform the work prescribed by the previous instruction. In this type of mechanism, the results of one step are used by the following step, and the process resembles a pipeline, which is why the method is called pipelining.

We are familiar with the use if pipelines to transport fluids across long distances. The fluid is placed into the pipe at the origin, and after some delay, the fluid becomes available at the destination. The material is kept flowing through the pipe by forcing more material into the beginning or front of the pipe. Because of the length of the pipe, a great deal of material may be sent into the pipe before anything is available at the end. But once the flow has begun, then there is a direct correlation between what enters the pipe and what leaves the pipe.

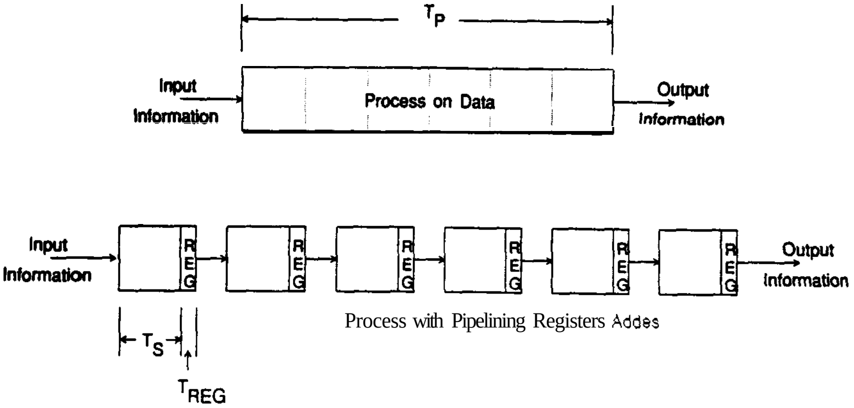
Other relatively common processes fit this description for pipelining. Manufacturing uses pipelines. called "assembly lines." to produce goods in a timely fashion. When the "pipeline" is flowing in an automobile factory, a new car exits the pipe every few seconds. This is an interesting example since the pipe is not homogeneous. That is. since not all automobiles are exactly alike, as the basic unit (in this case a car) moves through the pipe (the assembly line). features are added at the respective stations in the pipe according to a specification accompanying the unit through the pipe.

Other processes that form pipelines are plentiful. One example is food preparation in restaurants. where individuals perform specific functions on the food as it is processed. Another example is an automatic car wash system, where cars to be cleaned follow one another thouph a system where the various cleaning steps are applied successively to each unit. Even school systems can be considered a pipeline process, since one class follows another through the educational process, each learning concepts in a predetermined order.

In this chapter we will examine more closely this concept of pipelining, and apply the principles to the design of pipelined computer systems. Two basic kinds of pipes are used in computers: data pipes and control pipes. Both result in higher execution rates, since more answers are available per unit time. And both achieve effective results by overlapping independent functions. First we will identify the limits to the process, and then examine practical implementations of the mechanism. The information in this section is intended to be an introduction to the concept of pipelining, and an examination of some of its characteristics. Additional details can be determined by examining pipelines of real computer systems. and by looking at the implementations of some of the classical machines. Of particular interest are the scoreboard technique for reservation of time slots [Thor64], and the Tomasulo algorithm for utilization of multiple functional units with a pipeline [Toma67]. In addition, some texts present a more complete discussion of many of the aspects of pipelining than presented here [Kogg81, Ston80, HwBr84].

8.1 Pipelined Systems: Overlap of Independent Processes

The use of pipelining in computer systems is used to allow processes that can proceed independently to do so. For an example, consider the process of doing a floating point addition. A block diagram of one method of doing this is shown in Figure 3.25. which is repeated with a few modifications in Figure 8.1. It is possible to perform the addition function as shown in Figure 8.1 but since the



Process with Pipelining Registers Addes

Figure 8.3. Processing with Nonpipelined and Pipelined Mechanisms.

unrealistic assumption that T_S is the same for all sections. The results from the section are saved in a register. and the register time is represented as T_{REG}. The total time for a section. then is $T_S + T_{REG}$. Using this model of processing, let us follow the execution of the process in a nonpipelined and a pipelined system.

To pertorm M different operations on a nonpipelined implementation would require the application of the processing hardware M different times. as shown in Figure 8.4. Thus, the total processing time for the M instructions would be $M \times T_P$.

On the other hand, the use of pipelining allows different sections of the unit to be used in a single clock cycle. One way to visualize this is to construct a space-time diagram for the hardware: that is, time is shown on the X axis, and the functional units (in this case. sections of the pipelined process) are shown on the Y axis. Then the progress of the M instructions can be followed through the system. Such a diagram is shown in Figure 8.5 for the six section pipe of Figure 8.3. The time for the first instruction will be $N \times (T_S + T_{REG})$, where there are N sections of a pipe. In this case, $N = 6$. If we assume that the cost of the register is not included ($T_{REG} = 0$), and the initial process is equally divisible, so that $T_S = T_P / N$, then the first instruction takes the same amount of time as the first instruction of a nonpipelined implementation. T_P. The benefit comes fmm the other instructions, since the second instruction will be finished T_S after the first instruction, and so on. So, the total time required for M instructions is just the time required for the first instruction $[N \times (T_S + T_{REG})]$, plus $M - 1$ additional section times $[(M - 1) \times$

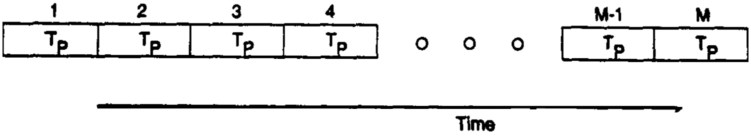
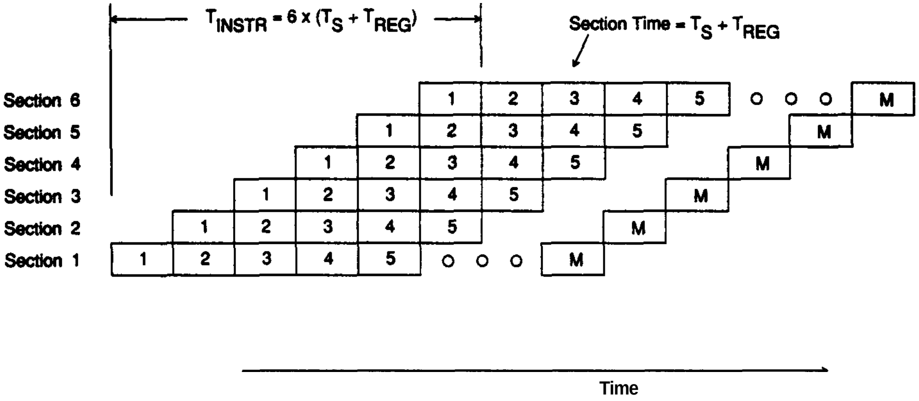


Figure 8.4. Timing Requirements for Nonpipelined Implementations.



Fire 8.5. **Overlapped Execution of Instructions.**

$(T_S + T_{REG})]$. So, the total time for M instructions is $(N + M - 1) \times (T_S + T_{REG})$. To see what the speedup is, we then calculate:

$$\text{Speedup} = \frac{\text{Time for } M \text{ instructions without pipelining}}{\text{Time for } M \text{ instructions with pipelining}}$$

From the above times, this becomes:

$$\text{Speedup} = \frac{M \times T_P}{(N + M - 1) \times (T_S + T_{REG})}$$

$$= \frac{T_P}{(1 + \frac{N-1}{M}) \times (T_S + T_{REG})}$$

For large M, the $(N-1)/M$ term becomes negligible, and the speedup is:

$$= \frac{T_P}{T_S + T_{REG}}$$

$$\approx \frac{T_P}{T_S}, \qquad \text{if } T_S \gg T_{REG}$$

$$= N$$

If we assume that T_{REG} is negligible, then the speedup is merely N, the number of sections in the pip. This agrees with our intuitive feeling of how much faster a pipelined implementation should be; indeed, we expect a six section pipe to produce results six times faster than the nonpipelined implementation. At the same time, we must remember that this observation is valid only if we disregard the time required for the storage function, and we also assume a steady state condition where M is large.

It is instructive to also examine some of the other information evident from the above discussion. First, pipelining implementations do not reduce the time for an instruction. That is, the time from the start to the finish of a single instruction will not decrease with the use of pipelining. If anything, the actual time to accomplish an instruction will be longer than the nonpipelined implementation, since the cycle time will be geared to the longest section time, and the registers will add real delays to the system. Thus, from the initiation of the operation to the completion of the first instruction will take a time dependent on the cycle time and the number of stages in the pipe. This time is sometimes called the "fill time," since it is the time required to fill all of the stages of the pipe. Obviously, for a larger number of stages in the pipe the fill time will be longer. This time may become important in the operation of the pipelined process, as we shall see.

Once the first operation has been completed, the remaining operations will follow at the rate of one result per clock cycle. This will continue as long as operations are available to perform, or until there is a conflict within the pipe for a resource. We will identify possible conflicts and some mechanisms for handling them in a later section. When the pipe is full and producing new results at the rate of one result per clock cycle, the system is operating at its highest efficiency. One of the tasks of the system architect is to create a unit capable of supporting the data movement needed to permit the pipeline to operate at maximum efficiency.

Another observation concerns the effective speedup over a nonpipelined system. The above equations can he plotted to identify the speedup achieved by using the pipeline technique. To give some physical "feel" for the observation. we will make some assumptions about the process we are pipelining. Assume that the nonpipelined system requires 100 nsec to complete. Thus, with no pipelining, the process will have a speedup of 1. Then. as the amount of pipelining increases, the speedup will increase. Fist, we will assume that the time required by the registers can be ignored, which is not a realistic assumption. That is. we will assume that T_{REG} is zero. With a T_S of 50 nsec, the speedup is 2; with a T_S of 25 nsec, the speedup is 4; and as the time per section, T_S, gets smaller, the effective speedup increases. It is evident from the equations that the effective speedup will asymptotically approach the Y axis. Thus, if the process were infinitely divisible, then the speedup could be infinitely large. A plot of effective speedup versus T_S is shown in Figure 8.6. The circled places on the curve are the effective speedups for N = 1 to 12.

The real gains available with pipelining are limited by two different mechanisms. The first is the fact that a real process is neither infinitely nor equally divisible. Thus, the time that needs to be considered is not T_P/N, but rather the maximum section time resulting from dividing the initial process into N sections. The second mechanism is the delay time added into the system by the use of registers or latches. If we assume a register delay of 10 nsec, then the curve of Figure 8.6 must be altered accordingly. This gives rise to the second curve of Figure 8.6, which indicates that the effective speedup is directly impacted by the register delays. An interesting observation can be made by following this second curve until the T_S is zero. At this point the effective speedup is only 10. Thus, even if the processing were free, the time penalty incurred by the use of real registers limits the speedup achievable with pipelining.

One of the assumptions implicit in the above observations is that there is no problem with keeping the pipe full. However, in real pipelines this is a challenging problem, and later in this chapter we will identify some techniques used to

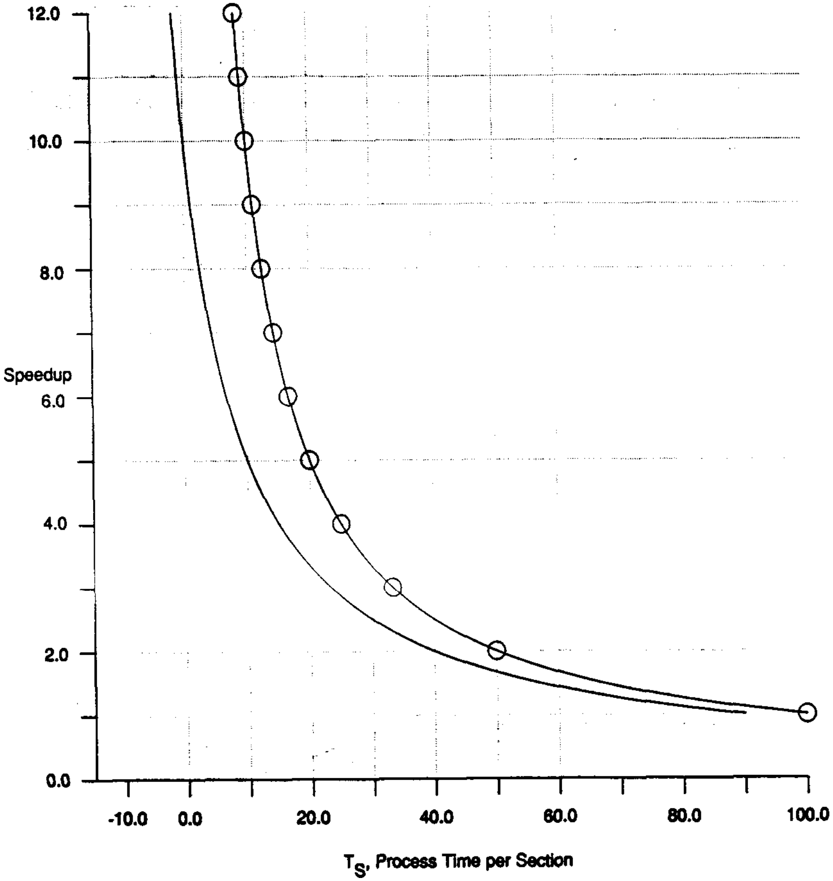


Figure 8.6. Speedup Achievable by Pipelining.

keep the pipe as full as possible. But suppose that the system is not able to keep the pipe full. but for some reason the sections of the pipe are not utilized for a fraction of clock cycles. How is the speedup affected? To answer this question. kt us return to the speedup calculation:

$$\text{Speedup} = \frac{\text{T i for } M \text{ instructions without pipelining}}{\text{Time for } M \text{ instructions with pipelining}}$$

Assume that a fraction of the cycles are unused, and let that fraction be represented by f. With this assumption, the amount of time required to execute M functions is not $N + M - 1$, but rather $(N + M - 1) \times (1 + f)$. And the equation developed above changes to become:

$$\text{Speedup} = \frac{M \times T_P}{(N + M - 1) \times (1 + f) \times (T_S + T_R)}$$

$$= \frac{T_P}{(1 + \frac{N - 1}{M}) \times (1 + f) \times (T_S + T_R)}$$

$$= \frac{T_P}{(1 + f) \times (T_S + T_R)}$$

$$= \frac{\text{Best speedup}}{1 + f}$$

Obviously, a **pipelined** processor is designed in such a way that *f* is kept *as* small **as** possible. However, it is instructive to note the effective degradation in performance that occurs with different values off. Consider the previous example of a process requiring 100 nsec to complete. and assume that the process is divided into 6 equal sections. Figure 8.7 gives a plot of the effective speedup versus the

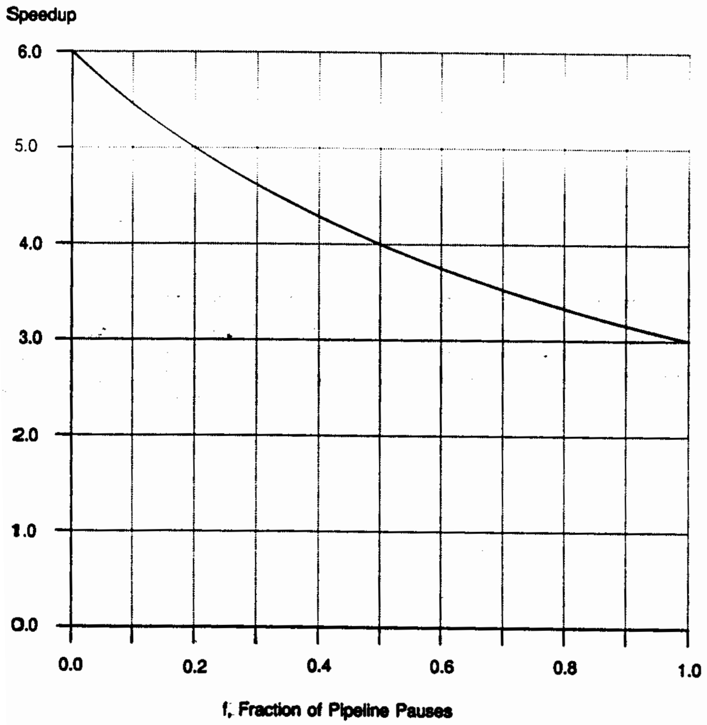


Figure 8.7. Effective Speedup as a Function of *f*, the Frequency of Pipeline Pauses.

frequency of the pauses of the pipe. If the frequency is 1, then the effective speedup is 3. That is, even if there are as many pause cycles as there are work cycles, then the effective speed is still 3 times faster than a nonpipelined implementation. Thus, the pipelining technique is effective for improving the speed of a process, even if the pipe cannot be kept full all of the time.

Pipelining is a technique that can be applied in any situation in which sequentially related events can proceed on independent operations in the same time frame. This will occur in the processing of data in arithmetic units and in the processing of instructions in control units. Let us examine these two mechanisms and identify some of the techniques that can be used.

8.2. Arithmetic Pipes: High Speed Calculations

In Chapter 3 we identified several mechanisms for doing high speed arithmetic. We will now examine some of these mechanisms with the intent of applying pipelining techniques to speed up the arithmetic process. Many metrics are considered during the process of dividing an arithmetic function into pipeline sections, and each designer will arrive at a compromise that meets the system design goals. As stated in the previous section, the objective of utilizing pipelining in arithmetic units is to achieve a speedup by performing operations concurrently for independent data sets. The questions to be asked by a designer in search of higher performance deal with timing issues and overall system issues:

- How can the initial process be subdivided to obtain the best results?
- What clock cycle time satisfies the various components of the process?
- What changes need to be made in the system to provide the overall data movement needed to sustain continuous operation by the pipeline?
- What metric is most meaningful to the overall system design goals?

The divisibility issue is one that can be dealt with in different ways to meet different design criteria Let us look at floating point multiplication for an example of a data operation which can be pipelined. The basic organization for this system is shown in Figure 8.8. The data movement within the system must

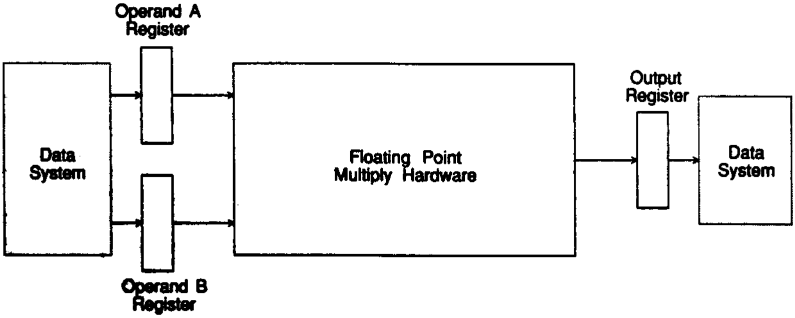


Figure 8.8. Diagram of Floating Point Multiplication Unit in System.

provide the operands for the unit in question, and in this case the multiply unit will perform the operation. The desired improvement is to speed up the operation as much as possible or feasible.

As can be seen from the diagram, one degree of pipelining is already available. That is, the floating point multiply unit can provide the action of multiplication, but the data system is responsible for supplying operands and handling the results. Thus, three operations can be overlapped in time, or pipelined: the fetching of the input operands to provide data to the multiplier, the multiplication function itself, and the storage of the result. Our concern here is with the actual floating point hardware, and the steps taken to pipeline the unit. We will examine the particular questions raised by this example, and also examine some of the other system questions.

As described in Chapter 3, the floating point multiplication can be accomplished in a number of ways. We will assume that the floating point number is in a 32-bit format. with a sign bit, an 8-bit exponent, and a 23-bit fractional mantissa. with hidden bit. Thus, to obtain

$$\text{Output} = A \times B$$

where the output and both inputs are in this format, then

$$\text{MANTISSA}_{\text{OUT}} \times 2^{\text{EXP}_{\text{OUT}}} = \text{MANTISSA}_A \times 2^{\text{EXP}_A} \times \text{MANTISSA}_B \times 2^{\text{EXP}_B}$$

$$= \text{MANTISSA}_A \times \text{MANTISSA}_B \times 2^{(\text{EXP}_A + \text{EXP}_B)}$$

The exponent is the sum of the two operand exponents, and the mantissa is the product of the mantissas of the two input operands. With a base two representation, the product can either be correct or require a I-bit normalization step. Hardware can be configured in many ways to perform the operations identified above, some more efficient than others. For the purposes of illustration of principles, we will select a mechanism and attempt to pipeline it.

The formation of the floating point product can be broken into different sections depending on the desired results. The initial hardware organization is shown in Figure 8.9. The process is performed in three steps: partial product formulation, partial product addition, postnormalization, and exponent formulation. The initial exponent formulation can be done in parallel with partial product formulation and addition, and then adjusted appropriately when the necessary postnormalization is performed.

The partial products are formed by Am27S558s, which are 8x8-bit multipliers. Thus, for 24-bit mantissas, nine individual multipliers are needed. This forms three rows of partial products, but the significance of the partial products formed in this process overlap one another. These partial products are identified as P1 through P9, with the significance of the bits identified in Figure 8.9. These partial products arc then summed in an adder tree made of 74AS881s, 74AS882s, and 74AS182s. The net result is a 48-bit number that may or may not have a "1" in the most significant bit position. Thus, a normalization step is required, and this is formed by a set of multiplexers. 74AS157s. The addition of the exponents is handled by 74AS881s. The element not shown is the sign bit. and the sign bit of the result will merely be the exclusive-OR of the sign bits of the two input operands.

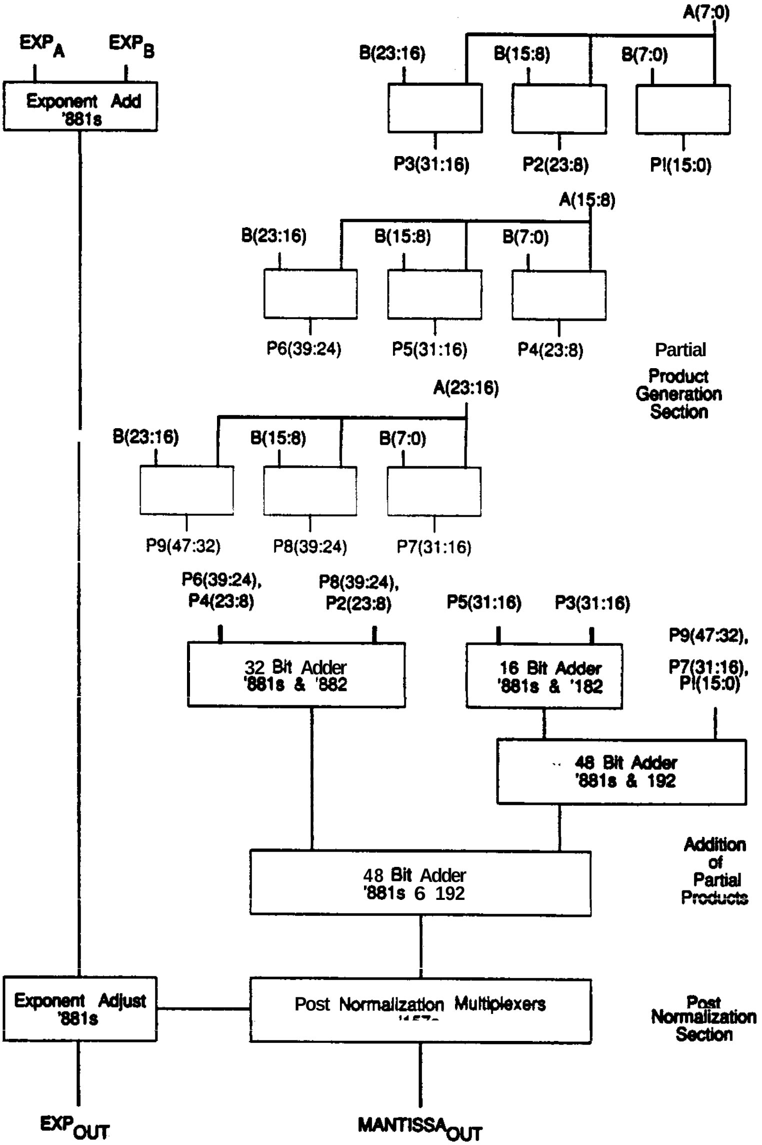


Figure 8.9. Block Diagram Level Representation of Hardware Floating Point Mechanism.

The time required for the hardware shown in Figure 8.9 is the sum of the time required for each of the three sections. The formation of the partial products requires 75 nsec. The addition of the partial products requires another 73 nsec. And the mantissa out will be available 11 nsec later. However, the adjustment of the exponent requires 18 nsec, so the mantissa is actually available about 9 nsec before the exponent. Thus, the whole process can be accomplished in 166 nsec. If this is to fit in a structure as shown in Figure 8.8, then an additional 14 nsec is required for the setup time, hold time, and the propagation delay time through a register, such as the 74AS574. Thus, the entire operation will require 180 nsec.

If we place registers in the process between the major sections, then the result could be represented as shown in Figure 8.10. With the initial process broken into three sections, we would like to see a speedup of three. But the clock cycle time of the system must be adjusted to accommodate the maximum time of the individual sections. Thus, for this example. $T_{\text{CLOCK CYCLE}} = T_P + T_{\text{REG}} = 75 + 14 = 89$ nsec. This results in a speedup of almost exactly two, which could be disappointing. However. in the process of doing this type of a design. locations that need attention if more speedup is required are identified. In this case, we are limited by the formation of the partial products. Slower speed parts can be used in the postnormalization section, and other changes can be made to the partial product addition section. But until a faster method of determining the partial products is obtained, the system will not run faster.

The multiplication example identifies several problems that need to be solved. The description above is for a very simple multiplier. and several things need to he done to the design to make it a real system. For example.

- How is overflow/underflow checked?
- How much hardware is required, and how does it affect the speed of the system?
- With respect to the allowable numbers, is unnormalized operation to be permitted? If so, how is the overall system to be changed?
- Is it necessary to compute the entire partial product array? Of the 48 bits which result from the multiplication, only 24 will actually be a part of the result. Therefore, may it be acceptable to create only the most significant portions of the partial product array under some conditions?
- When the appropriate bits are available, what kind of scheme is used for dealing with the extra bits? Truncation? Rounding? Round-to-zero? And is this to be done before or after post normalization?

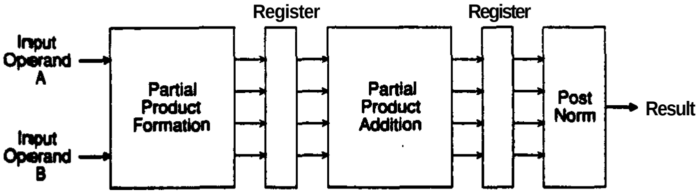


Figure 8.10. Block Diagram Level Representation of Hardware Floating Point Mechanism.

All of these questions must be addressed in a real implementation, and the answers will reflect the priorities of the designers.

Example 8.1: Costs of *pipelining:* Consider the floating point multiply example discussed above. With the use of pipelining, the speed of the system was doubted. What costs are associated with this speed increase?

Many different costs are associated with various designs. so we will identify only two: board space and power. The chip count is indicative of the amount of logic required by the system, and using board space is a more accurate measure of how "big" the system becomes. It is also indicative of how good the job of subdividing the system has been done. Another view of the system being used for this example is shown in Figure 8.11. The basic parts that could be used in a TTL implementation are identified with each major section. A summary of the parts needed is:

Parr Name	*Quantity*	*Unit Area (sq. in.)*	*Tor. Area (sq. in.)*	*Unit Power (W)*	*Toral Power (W)*
25S558	9	1.47	13.23	14	12.6
74AS881	36	52	18.72	.675	24.3
74AS882	4	.52	2.08	36	1.44
74AS182	1	.36	.36	.1	.1
74AS157	6	.36	2.16	.1	.6
			36.55		39.04

The total area needed by this system is *36.55* square inches, using DIP packages. A system that used leadless chip carriers would be smaller, but the

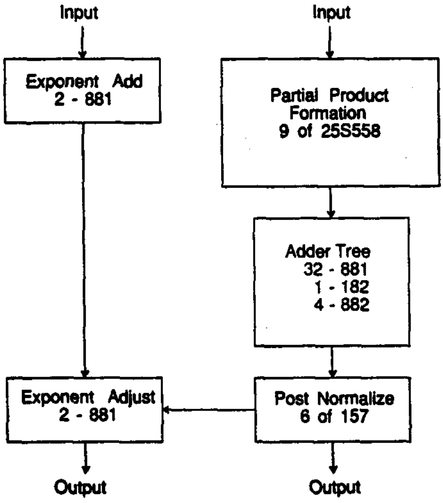


Figure 8.11. IC Requirements fa the Hardware Floating Point Unit.

same method of comparison would apply. We will assume that the registers between stages will be comprised of 74AS574s, which are 20 pin chips. Counting all of the lines that need to be saved from partial product formation to the adder tree (and the corresponding exponent), there are 153 bits of information to save. And between the adder tree and postnormalization there are an additional 33 bits. Thus, 25 register chips are needed, which will require an additional 11 square inches of space.

The power required by the devices is similar to the board space. The overall power for the unpipelined version is 39.04 watts. The register chips will require an additional 0.395 watts each. for a total of 9.875 watts. Thus. the speedup by a factor of two has caused an increase of board space of 30%. and an increase of power of 25%. For a resource investment of 25–30% the rate of operation of the floating point multiplier has been doubled.

The above example underscores some of the promises and pitfalls of pipelining. The original process was divided into three separate functions. bur the speedup was not three. Because of the real effect of adding registers, and the requirement that the clock cycle time be the maximum of the times for each of the individual functions. the resulting operational rate was twice the original rate. Thus, the actual maximum speedup is a function of all of the factors involved in the design of the system.

One of the basic tenets of pipelining is that to achieve the maximum available speedup $(T_P / (T_S + T_R))$ the pipeline must be kept full. To achieve this. the pieces of the "Data System" shown in Figure 8.8 must supply the appropriate operands in a timely fashion. and also handle the results as they become available. For the example system shown above, this means that every 180 nsec, two 4-byte operands must be made available to the floating point unit, and one Cbyte operand must be removed. This leads to a data rate of 12 bytes/180 nsec = 66.6 Mbytes/sec. If this data rate can be sustained, then the floating point unit is capable of achieving an operation rate of 5.55 MFLOPs. The rate of operation for the pipelined system is twice the rate of the unpipelined system. so to maintain the advantage of the speedup available with the pipelined implementation, the data system must be capable of handling information at a rate of 133.3 Mbytes/sec. This places a severe restriction on the types of information systems that can effectively be utilized by systems with data pipelines.

Example 8.2: Data rates for *pipelined systems:* The CRAY-2 computer system has a clock cycle time of 4.1 nsec. Assuming a single data pipeline system, what is the data rate necessary to keep a pipeline full?

A single pipeline will be full when two input operands and one result are handled in each data cycle. The CRAY-2 system, as well as other scientific systems, has a word width of 64 bits, or 8 bytes. Thus, for a full pipe. $3 \times 8 = 24$ bytes must be handled every 4.1 nsec. This is a data rate of 5854 Mbytes/sec. To achieve these data rates, multibank memories, wide data paths, and short transfer times are required.

As we have seen, the effectiveness of data pipelines is limited by several factors involved in real machines, such as the divisibility of the original process, the addition of registers to the system, and the problems associated with transferring information at the high data rates needed to keep a pipe full. In addition to the problem of physically supplying the information, there is a problem with the availability of the correct information. That is, even with a data system capable of extremely high data rates, there will be a problem when om operation cannot enter the pipe because of data conflicts. Data conflicts will occur when the results of one operation are needed by a following operation. Consider the following pair of operations:

$$A = B \times C$$
$$D = A X E$$

The value of A is needed before the second operation can proceed. This same behavior is also observed in some array operations:

```
for ( I = 1 ; I < 1000 ; I++ )

  X[ I ] = X[ I - 1 ] + Y[ I ];
```

In this operation, the calculation for X[9] cannot proceed until the value for X[8] has been obtained, and so on. Both of these operations exemplify the fact that a data calculation cannot proceed because a value is not available. The calculation that follows cannot proceed until the data from the preceding operation has been made available. Thus, the pipeline must halt until the data is ready, at which point it can proceed. This reduces the effective speedup, and hence it is a situation to be avoided as much as possible.

One observation about the interaction between the pipeline and the data processing concerns the length of the pipe. The longer the pipe, the longer it will take to get information from a previous operation. That is, if a process is subdivided into three sections. then the largest number of clock cycles needed to obtain a previous result is two. However, if the same process is divided into six sections. then up to five clock cycles can be needed to obtain the results of a previous calculation. Thus, two different arguments can be made for the optimal number of stages in a pipeline: for a large speedup, divide the initial process into many sections; to minimize the penalty of data conflicts, keep the number of sections small. The designer must then trade off the benefits and costs of processing with a data pipeline.

Another observation concerning the effective use of data pipelines deals with the operands used in the calculations. So long as the operands are independent, there is no possibility of penalties due to data conflicts. Thus, streams of operations constructed in such a fashion as to minimize the data conflicts will result in the highest performance. The guaranteed independence of vector operands is the mechanism used by vector machines to achieve very high data rates. For example, consider the problem of addiig two linear arrays of information together. The organization of the data into arrays corresponds to storing the information into vectors, where a vector is an organized set of data. The addition of the two arrays is then accomplished by streaming the information out of the storage locations to the arithmetic unit, and the results back again. Such an arrangement is shown in Figure 8.12 The two input operands actually consist of N pairs of numbers to be added. And the result consists of N numbers. each of which is the sum of the corresponding elements from the original vectors. Since all elements of the vector are available before the operation begins, the processing unit can process information without any conflicts.

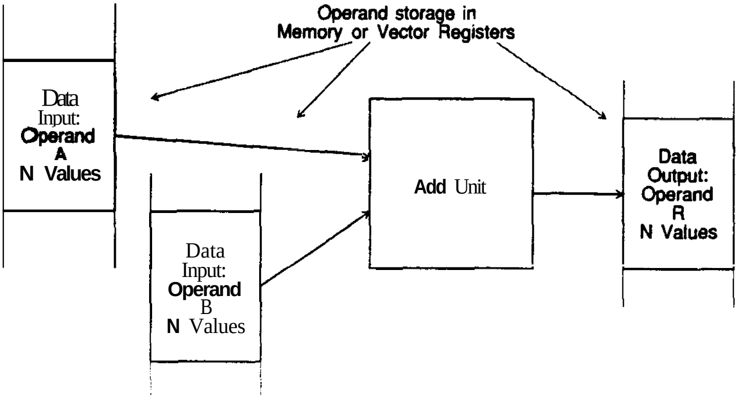

Operand storage in Memory or Vector Registers
Data Input: Operand A N Values
Data Input: Operand B N Values
Add Unit
Data Output: Operand R N Values

Figure 812. Processing Information with Vectors: Vector Addition.

The location of the vectors *A*, *B*, and R is dependent on the type of instruction level architecture used by the vector machine. One mechanism is to hold all of the vectors in memory. and stream the operands to the functional units directly from memory, and return the results to memory. This is a memory-to-memorv architecture. and was the design mechanism used. for example. in the Cyber 205 vector machine. The instruction must then identify the locations of the vectors in memory and the length of the vectors (how many numbers in each).

A more common mechanism is to use vector registers. a concept similar to the use of general purpose register sets in a "standard" general purpose machine. The operands for vector instructions are then supplied directly from high speed registers, and the results also stored in the registers. The vector instructions for this type of a machine need not identify memory locations. which require long addresses. but rather vector registers, which can be specified with a few bits. However, before the vectors can be combined from the registers the vectors must be moved there from memory. This type of architecture balances the probability that the information in the vectors can be used more than once before memory interaction is needed with the additional instructions required to transfer the data to and from memory.

Regardless of the mechanisms used for storage of the vector operands, one of the reasons that vector machines achieve high operational rates is the guaranteed independence of the operands being sent to the arithmetic units. The operand independence ensures that the pipeline will be kept full, and that there will be no data conflicts. This situation leads to the highest computational rates achievable by a pipelined machine,

Example 83: Pipelines in a vecror system: The CRAY-1 computer system was one of the first "popular" vector machines, and made extensive use of data pipelines to provide high computation rates. Other members of the CRAY family have added multiprocessing capabilities to the system, and extended some of the features available to the user. What are the data pipelines used in the CRAY-2 computer system? What is the peak floating point operation rate for the system?

The CRAY-2 computer is actually a multiprocessing system, with four processors available for use on programs. A block diagram of one computational section of a CRAY-2 is shown in Figure 8.13. As can be seen from the diagram. this system is not a memory-to-memory architecture. Information is transferred from the memory system to the vector registers (or scalar registers), and all arithmetic is done in the registers. Nine different data pipelines are available for use in the system, and they an:

Data Pipe	Pipe Sections
Address add	
Address multiply	
Scalar integer	
Scalar shift	
Scalar logical	
Vector integer	
Vector logical	
Floating point add	
Floating point multiply	

The complexity of the arithmetic to be done determines the number of sections required in the pipe for that arithmetic unit. The simpler operations listed in the table result in pipelines containing fewer sections than the more complex operations. The vector registers are each capable of storing 64 numbers, and so the vector instructions can operate on sets ot data containing up to 64 values (vector length ≤ 64). Longer vectors must be divided into sections of 64 elements or less.

When vector operands an being supplied to a pipelined functional unit, a new result is generated at the rate of one value each 4.1 nsec, This is a computational rate of 243.9 Mflops. When circumstances permit. two functional units can be utilized simultaneously, which gives a computational rate of 487.8 Mflops.

As we have seen, the time required for a pipeline section is dependent on several factors. We have partitioned the function performed in a section into two parts: the arithmetic or logical portion, and the storage or register function. In general, a designer will attempt to minimize the time required for both of these portions, so that the system will have a small clock cycle time. This situation is shown in Figure 8.14. Since any combinational function can be formed in two gate delays, if enough gates with a high enough fan in are used, a tradeoff is performed between the number of levels of logic and the total amount of gating required to accomplish the function. This may result in implementations that utilize many gate delays to accomplish their work, but that an beneficial because of a small gate count (or silicon area). The output of the function is directed to the storage element to be sent to the next section of the pipeline.

The ingenuity of the designer in using the available logic has a direct impact on the performance of the system. For example, consider the circuit shown in Figure 8.15(a). The logic portion is a two level gating circuit that implements the sum function, given proper logic levels for the two inputs and the carry. The output from this gating system is d i i to a gated latch. When the CLOCK-H line is asserted, the output will be set to agree with the level of the sum network. The total delay through this circuit. if the clock line is asserted, is six gate delays.

However, since this is a gated latch rather than an edge-triggered register, care must be taken to be sure that the value does not propagate too far while the clock is asserted.

The amount of time required for the sum and register functions of Figure 8.15(a) can be reduced by combining the sum logic with the latching logic. The function of Figure 8.15(a) is accomplished by the logic of Figure 8.15(b), with some obvious changes. The ORing function of the logic has been combined with the ORing function of the latching gates, and the ANDing function of the gated latch has been combined with the ANDing gates of the required logic function. The net result is a system that requires only three gate delays to complete, from clock and data stable to outputs stable. Note that both asserted high and asserted low outputs are available in the Figure 8.15(b). This will be useful for functions that follow this stage in the pipe.

Obviously, it would not be reasonable to combine all of the logic of a stage of a pipeline with the latching function, but the mechanism shown above of combining one level of the logic with the latch will reduce the timing impact of adding the latching function to the logic required by the function.

One of the disadvantages of the latches implemented in Figure 8.15 is that the time to output stable from the clock is not always equal. That is, the required time for the data to become stable is a function not only of logic input and the clock, but it is also a function of the level stored in the latch before the assertion of the clock. Consider the four possible combinations of the input data {LAT_IN-H in Figure 8.15(a)} and latch output:

LAT_IN-H	OUT-H	Delay from CLOCK-H
0	0	No change = zero delay
0	1	2 gate delays
1	0	3 gate delays
I	1	No change = zero delay

This difference in time required for the function results in an unwanted skew in the time for a section of logic. With latches designed as shown in Figure 8.15, the problem will always exist.

A number of different solutions to the problem have been suggested, one of which is the Earle Latch. which was used extensively in the IBM 360 pipelined machines. This latch is shown in Figure 8.16(a). One obvious difference is that the latch does not need (nor does it provide) both asserted high and asserted low inputs to function properly. If we repeat the above table to identify the speeds of the Earle Latch, we have:

LAT_IN-H	OUT-H	Delay from CLOCK-H
0	0	No change = zero delay
0	1	2 gate delays
1	0	2 gate delays
1	1	No change = zero delay

Thus the maximum time to data stable is always two gate delays, assuming that the propagation time through a gate is always the same. The only major difficulty with the system shown in Figure 8.16 is that both asserted high and asserted low clocks are required This requirement is not restrictive since the clock signals will be needed by all of the stages.

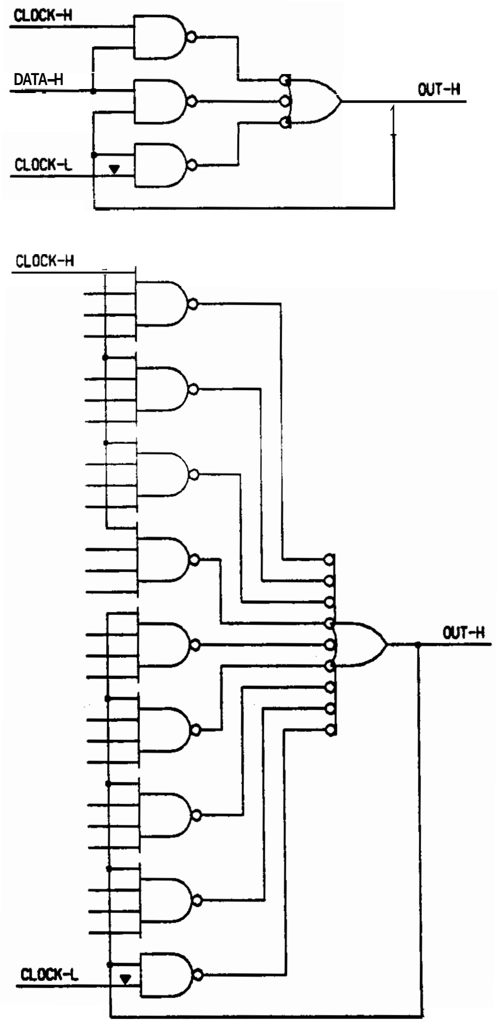


Figure 8.16. Earl Latch Designs: (a) Basic Latch; (b) Combined Latch and Logic.

The technique applied above of combining the logic of the function with the logic of the latch can also be utilized with the Earle Latch. The sum function of Figure 8.15 is combined with the Earl Latch as shown in Figure 8.16(b). Again. the designer must identify which combination of function and latch logic, with their associated costs (number of gates, or semiconductor area, or ...), matches the design goals of the system.

The use of pipelining techniques to speed up the processing of data manipulations results in enhanced throughput for data operations. The operations that must be performed are identified, and these are partitioned into appropriate sections. Storage elements are inserted between the sections to synchronize the actions of the system and to hold the data needed by the sections that follow. The performance achievable by the use of pipelining is a function of many factors, as we have seen. The divisibility of the original process, the register delays, and the amount of available logic all influence the basic data rate at which the pipe can operate. External influences that affect the operation of the pipe include the independence of the operands needed by the function, and the ability of the system to handle data at a sufficiently high rate to keep the pipeline full. A system that satisfies the internal and external requirements for correctness and data movement can achieve substantial speed improvements over nonpipelined implementations.

Example 8.4: Pipelining in data systems: The concept of pipelining for data operations can he used in manv applications where an increase of speed is needed. even if the operations do not lend themselves to division. Consider a hardware system constructed to calculate the fast fourier transform (FFT), as shown in Figure 8.17(a). Can pipelining be used to increase the speed of operation of the system?

In the system depicted in Figure 8.17(a), the data is stored in a memory and extracted as needed to perform the calculations. The arithmetic is performed in a set of special purpose hardware. One method for calculating the butterfly is shown in the data diagram: two values (D_U and D_L) form the inputs. and the outputs (D'_U and D'_L) are returned to the memory. The values are complex in nature, and as such consist of two parts, the real and the imaginary. The arithmetic involved consists of a complex multiply, a complex add. and a complex subtract. The weighting factor (W) is derived from a set of constants, and supplied by a memory not shown in the diagram. The complex arithmetic required by each set of butterfly calculations can be accomplished by four multiplications and six additions. With memory transactions, additions, and multiplications all requiring about the same amount of time, the system is fairly well matched at this point. That is, each butterfly will require a minimum of eight cycles, since that much time is required to extract a real and imaginary value for each of D_U and D_L, and place the calculated values back into memory. For six of the eight cycles the adder will be busy, and for four of the eight cycles the multiplier will be busy.

Pipelining can be applied to this system by recognizing that the FFT requires a number of passes through the data set The number of passes is $\log_2 N$, where N is the total number of data points, and also a factor of 2 The results of one pass form the information needed by the next pass. Thus, the basis for a pipeline exists, since the data is to be passed from one

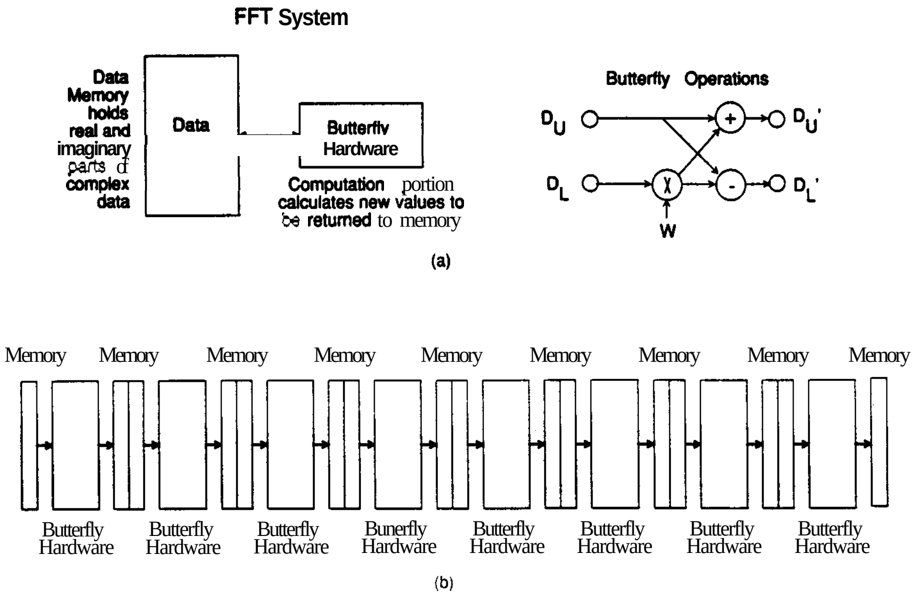


Figure 8.17. Fast Fourier Transform Systems: (a) Nonpipelined Implementation; (b) Pipelined Implementation.

module to another, where each module is responsible for one pass of the **butterfly** through the data This **arrangement** is depicted in Figure **8.17(b)** for a system which computes a 256 point transform. Eight stages are needed, and the memory output fnnn one stage feeds the memory input of the **next** stage. The memories are depicted as duals. since the **information** will be input by one **stage** and extracted fnnn the next. Thus, the **memories** must either "ping-pong" between two sections, or be interleaved in such a way that the desired information is available as **needed.**

Pipelining as shown in the figure will produce results eight times faster than a nonpipelined implementation, but requires eight times as much hardware. Nevertheless, if the speed is needed to maintain real time operation, then the hardware resources may be justified.

8.3. Control Pipelines: Overlap of Independent Control Operations

In the preceding section we looked at **improving** the **speed** of data operations by executing **different, independent** portions of the calculations at the same **time** in specifically designed portions of **hardware.** The principal **requirement** for **correct functionality** is that the **operations** be independent **one** stage of the pipe cannot produce correct **results** until all of the input information is correct. **Pipelining** is also applicable to other types of **processing,** so long as the independence requirement is satisfied, and the necessary **processing** can be **appropriately partitioned.**

In this section we will examine some of the mechanisms for pipelining control functions, and identify some of the limitations of the achievable performance.

As discussed in Chapter 4, a stored program computer basically operates on a **fetch-decode-execute** mechanism. An instruction is fetched **from** memory, decoded, and then the work specified by that instruction is executed. **These processes are** sequential in **nature,** and basically independent. so they satisfy the fundamental **requirements** for **pipelining.** The responsibility of the designer **and** system architect is to organize the data paths and registers in such a way that the various functions can be executed **concurrently.** If this can be accomplished, then the same type of speedup enjoyed by the data pipes of the preceding section can be realized.

One of the simplest pipelines of this **nature** is demonstrated by the **fetch**-execute mechanism of microcoded engines, one example of which is shown in Figure **5.31.** This is shown in block diagram form in Figure **8.18.** The address sequencer has the responsibility of identifying the next microinstruction to execute. and obtaining that instruction from microcode memory. This instruction is loaded into the microinstruction register. **During** the next clock cycle. the execute section will decode the **control** bits contained in the microinstruction register and perform the requested work. These two functions form a two stage pipeline, since the fetch section is always obtaining the instruction one clock period before the execute section performs the work. For this reason, the microinstruction register is sometimes referred to as the "pipeline register."

Each **of** the basic processes of instruction execution can he divided further than the microcode engine example. and many machines utilize this technique internal to the control unit. We will **partition** the activities of a control unit into the six units shown in Figure 8.19. These units have the following responsibilities:

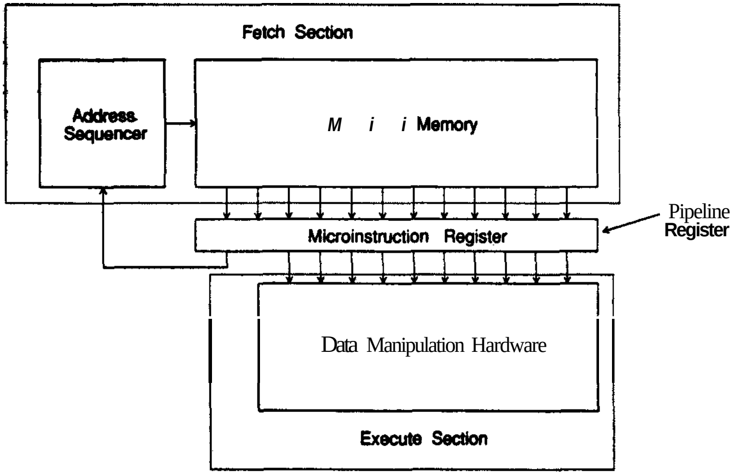


Figure 818. Fetch-Execute Mechanism of a **Pipelined Engine.**

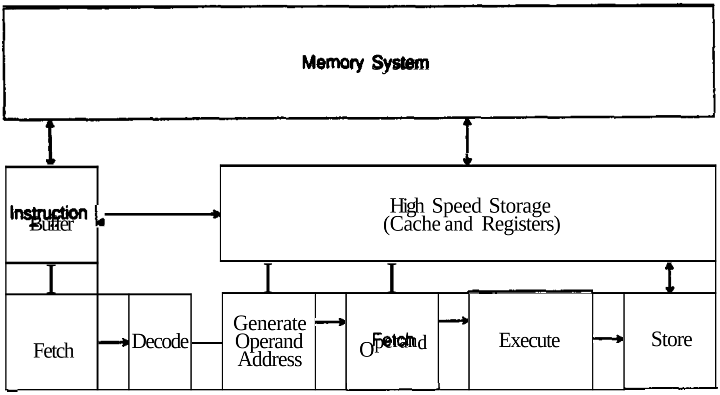


Figure 8.19. Block Diagram of Control Pipeline for Basic Machine.

- ***Memory System:*** The memory system contains all of the memory utilized by the computer system. This includes the mass storage devices as well as the main store. This portion of the system is responsible for providing the needed instructions and data as rapidly as possible.

- ***High Speed Storage:*** The high speed storage section contains both the cache memory and the registers utilized by the system. The active data is stored in such a way that it is available as needed by the program. In the discussions that follow, the high speed storage section will be considered perfect. This is not a reasonable assumption in real systems, but will facilitate understanding of the issues related to the pipeline. and move from consideration the problems resulting from the interaction of the pipeline with an imperfect memory system.

- ***Instruction Buffer:*** The instruction buffer is a small storage area that contains the instructions currently executing. This storage area is managed by the instruction fetch hardware, and contains the active portion of the currently executing program. Real sizes of instruction buffers vary with manufacturer and purpose from a few bytes to a few kilobytes. The information contained in the instruction buffer may come from directly from the memory system, or it may be provided by the cache memory. As with the high speed storage, we will assume that the instruction buffer is perfect, so that any information needed by the instruction fetch unit is immediately available.

- ***Instruction Fetch:*** Instructions needed by the program are obtained by the instruction fetch section. This unit identifies the next instruction to execute and presents it to the decode section.

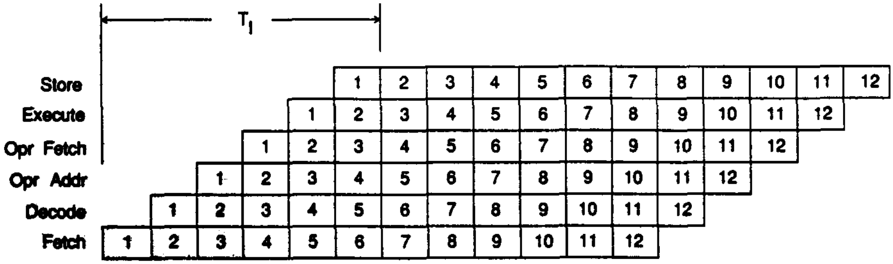
- Decode: The decode section obtains an instruction from the instruction fetch unit and identifies the work to be done. It then prepares the information that will be used by succeeding sections to identify operands and actions, and these bits will be forwarded to the sections as required by the instruction flow.

- ***Generate Operand Address:*** This unit is responsible for identifying addresses of operands. For example, in the two address move instruction:

  MOV *R1, R0

  data is moved **from** memory to **R0**. The generate operand address section would identify the fact that **R1** contains an address, and provide that address to the high speed storage section. More complicated addressing mechanisms are possible, and this section of the pipe must be able to provide the requested address. Any address generated by this section of the pipe will be utilized by the next pipe section to obtain the required data.
- ***Fetch Operand:*** The fetch operand unit identifies the location of the data needed by the instruction, and fetches that information fmm the high speed storage unit. The data is available to be utilized by the functional units in the data path during the cycle controlled by the execute section.
- ***Execute:*** The execute unit has the responsibility of doing the work called for by the instruction. The previous sections of the pipe will have prepared the data. and so both data and **instruction** information will be available. The result of **the** instruction will be provided to the store unit to be saved **as** needed.
- ***Store:*** The store unit takes the information resulting from the execution of the instruction and saves it **as** necessary in the high speed storage unit. Thus, any necessary modifications to registers or memory locations are performed by the store unit.

With the original process divided into six sections, it would appear that we should be able to get a speed up of six over a **nonpipelined implementation. As** we have seen with the data pipes, this will not be the case for various reasons: the process will not be equally divisible into six sections; the delays caused by registers adds a real increment of time to the process; and **increased** speed of data **transfers** may not be physically possible. However, if we assume that solutions to these problems have **been** provided, then we can envision the execution of the instructions as shown in **Figure** 8.20. **This** is essentially the same **as** Figure 8.5, but we have added a few more instructions. The reason for this will become evident in the following paragraphs. As with the data pipe, **there** is an inherent delay **caused** by the various **stages** of the pipe, and **instructions** will require a time (T_I) to complete. If **instructions** can be **inserted** into the pipe on each clock, then the effective instruction time will equal the clock cycle time.



Store						1	2	3	4	5	6	7	8	9	10	11	12
Execute					1	2	3	4	5	6	7	8	9	10	11	12	
Opr Fetch				1	2	3	4	5	6	7	8	9	10	11	12		
Opr Addr			1	2	3	4	5	6	7	8	9	10	11	12			
Decode		1	2	3	4	5	6	7	8	9	10	11	12				
Fetch	1	2	3	4	5	6	7	8	9	10	11	12					

Figure 8.20. Space T i **Representation of Instructions** in **Control** Pipe.

Aside from the physical problems that we have assumed can be handled in a reasonable way by a complex hardware system, the conflict problem limits the achievable performance. With a data pipe, we used the term "conflict" to describe the case when one operation could not proceed because it needed information from an operation that had not yet completed. Instructions in a pipe interact with one another in much the same way, which prevents the pipe from remaining full at all times. We will describe three different types of conflicts:

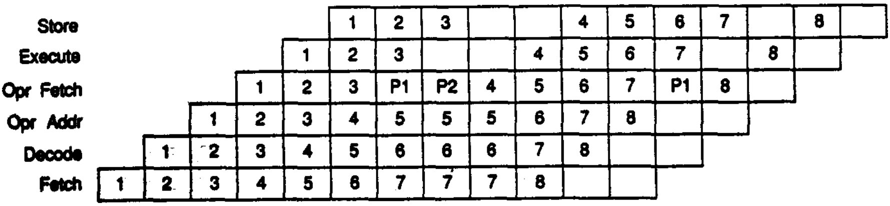
- A data conflict
- An address conflict

A branch conflict

1. As in a data pipe, a *data conflict* results when one instruction cannot proceed because an operand is needed that is the result of a previous instruction, and that instruction has not yet completed. For example. consider the following set of instructions:

```
1   ADD  R0, R3
2   MOV  R0. R7
3   ADD  R0, R5
4   ADD  RS. R4
5   SUB  R8, R9
6   MOV  R1. R2
7   ADD  R1. R3
8   ADD  R?. R6
```

The flow of these instructions through the pipeline is shown in Figure 8.21. The first three instructions have no difficulty executing, assuming that all of the information initially needed is available. However, there is a conflict between instruction 3 and instruction 4: the work specified by instruction 4 is to add the contents of R5 to the contents of R4; however, before this can occur, instruction 3 must first modify the contents of RS. Thus. the operand fetch section of the pipe will be unable to fetch the desired value until the store section of the pipe has placed the result of instruction 3 into the register. This results in the two penalty cycles shown in Figure 8.21. The instructions waiting in the pipeline pause until the request can be satisfied. and then proceed. Similarly. instruction 8 collides with instruction 6. When instruction 6 has modified R2, then instruction 8 can obtain the value and proceed. However, as shown in the figure, since there is an independent instruction between 6 and 8, the effective penalty incurred is only one cycle instead of two.



Store						1	2	3			4	5	6	7		8	
Execute					1	2	3			4	5	6	7		8		
Opr Fetch				1	2	3	P1	P2	4	5	6	7	P1	8			
Opr Addr			1	2	3	4	5	5	5	6	7	8					
Decode		1	2	3	4	5	6	6	6	7	8						
Fetch	1	2	3	4	5	6	7	7	7	8							

Figure 8.21. Penalties Associated with Data Conflicts.

The analysis of the effective execution rate is performed in the same manner as the analysis of the pauses which were discussed in the previous section. In fact, the resulting formula will have the same form:

$$\text{Effective speedup} = \frac{\text{Best speedup}}{1 + \sum p_i \times P_i}$$

Here p_i is the probability that then will be a conflict. and P_i is the penalty associated with that conflict. Figure 8.21 identifies a conflict with a penalty of two clock cycles, and a conflict with a penalty of one clock cycle. A plot of the above equation is shown in Figure 8.22. Curves are included for the case when all conflicts incur two penalty cycles and the case in which all conflicts require a single cycle. In practice, the actual penalty incurred because of conflicts in a system of this type would result from a combination of conflicts that incur both penalties, and hence a line representing the effective conflict penalty would be found between the two lines in the figure. Obviously, it is beneficial to reduce the required number of penalty cycles, and we will identify some mechanisms for doing that later in this section.

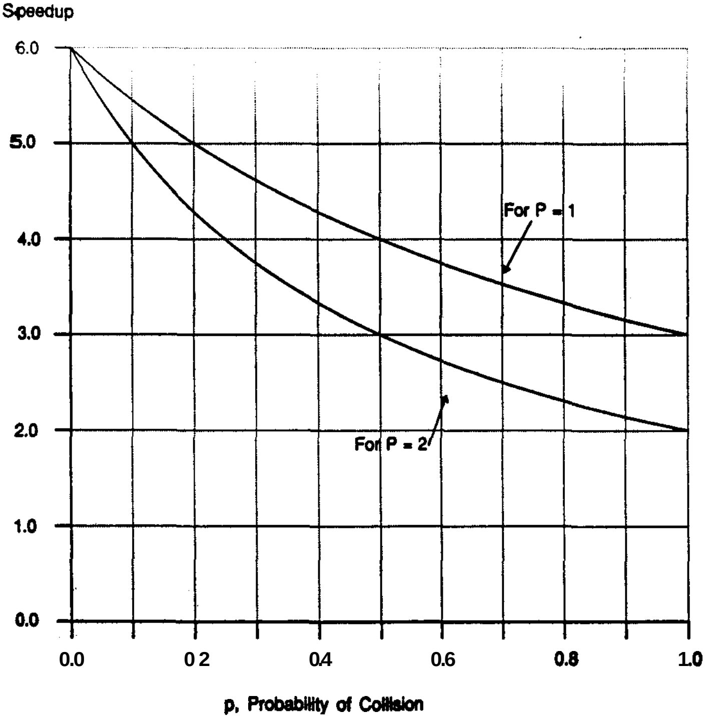


Figure 8.22. Effect of Conflicts on Speedup

The pipeline must be designed in such a way that the resources needed by each instruction are properly coordinated. Two instructions that use the same resource, such as a register, can either read or write to the resource. This gives rise to four possible orderings:

First Instruction	*Second Instruction*	*Conflict Handling*
READ	READ	No conflict; instructions need not coordinate access to resource.
READ		Fit instruction must obtain correct value before second instruction is allowed to modify it.
WRITE	READ	Second instruction must wait until first instruction has appropriately modified resource before obtaining the value.
WRITE	WRITE	Sequence conflict only. The control must assure that the value of the resource is that set by the second instruction.

The control system must examine the resources being utilized and the ordering of the instructions, and assure that the results are compatible with an implementation that does not make use of pipelining. Therefore, the control unit of the pipeline must coordinate the use of the resources identified by the instruction set; independent resources need no special care, while resources utilized by more than one instruction need to be closely monitored. Thus, the control unit becomes more complex as the amount of overlap increases, which results in a pipeline which contains more stages.

2. Like a data conflict, an *address conflict* results because of the unavailability of information. However, rather than a penalty that is the result of unavailability of data, the address conflict occurs because the system cannot generate the address of the data. In the organization of the system as shown in Figure 8.19, the generate operand address block has the responsibility of identifying the location of the data. If the location of the information is specified by values that have not yet been updated, then the system must wait until that information is available. For example, consider the following set of instructions:

```
1   MOV   R1, R9
2   MOV   constant, R8
3   INC   R0
4   MOV   (R7 i R0), R6
5   ADD   R4, R3
6   ADD   (R6), R2
```

The resource utilization diagram corresponding to these instructions is shown in Figure 8.23. Instruction 4 moves a value to R6; the location of that value is identified by indexing R7 by R0. However, instruction 3 increments R0. Therefore, the value contained in R0 cannot be used in the calculation of the address until it has been modified by instruction 3. This causes a penalty that is one greater than the data conflict penalty. The conflict between instruction 3 and instruction 4 causes a penalty of three clock times, while the conflict between instruction 4 and instruction 6 incurs a penalty of two clock times. The analysis of the effect on the overall speedup proceeds exactly as the above analysis of data conflict effects, except that the penalties are larger.

3. Branch *conflicts* occur for the *same* reasons as *other* conflicts: the information needed is not available. However, the penalty for branch conflicts is

Store						1	2	3				4	5			6	
Execute					1	2	3				4	5			6		
Opr Fetch				1	2	3				. 4	5			6			
Opr Addr			1	2	3	P1	P3	P3	4	5	P1	P2	6				
Decode		1	2	3	4	5	5	5	5	6							
Fetch	1	2	3	4	5	6	6	6	6								

Figure 8.23. **Penalties** Associated with Address Conflicts.

greater than the other types, since the fetch and decode portions of the pipe occur first. When a conditional branch occurs, the instruction to be executed next is not known until the target of the branch is properly identified. That is, since the condition on which the branch will be made is not available, it is not **certain** whether or not the branch will be taken. For example. consider the **following** instructions:

```
1   label   ADD  R3, R4
2           SUB  R3, R9
3           MOV  R4. (R7 i R1)
4           CMP  R1, R3
5           JNE  label
6           ADD  R0, R1
```

Instruction 5 determines whether the program flow returns to instruction 1 or moves on to instruction 6. However, the condition on which that decision is based is not available until the comparison of R1 and R3 **(instruction** 4) is accomplished, and the result of that **comparison** has been placed in the **status** register. The resource **utilization** for this instruction is shown in Figure 8.24 for one implementation and branch path. Other implementations will incur **different** *costs.* and **different** branch paths will result in different **resource utilizations.** As shown in the **figure,** the system does not know which **instruction** follows **instruction** 5 until instruction 4 completes. **This** results in a penalty of four clock times.

As we have seen, **conflicts** in a control **pipeline** result when **information** is not available because the action specified by prior **instructions** has not been completed. Data **conflicts** occur **because** an instruction **needs** data that will not be available until a **previous instruction** completes. **Address** conflicts occur when an instruction cannot calculate the address of a data **reference** because the **information** needed to identify an **address** will not be available until a previous instruction

Store						1	2	3	4						6		
Execute					1	2	3	4						6			
Opr Fetch				1	2	3	4						6				
Opr Addr			1	2	3	4						6					
Decode		1	2	3	4	5	P1	P3	P3	P4	6						
Fetch	1	2	3	4	5					6							

Figure 8.24. Branch Conflict Penalty for One **System Implementation.**

completes. And branch conflicts occur when the next instruction to execute will not be known until the results of a previous instruction are available.

A number of techniques have been utilized to minimize the overall effect of conflicts. and we will examine four of the methods. Each of the methods uses a different mechanism to reduce the resource utilization, but the goal is the same: minimize penalty cycles due to conflicts.

The first technique has little to do with hardware and much to do with the way that the program is configured for the machine. We have *seen* that the effect of the conflicts is minimized when the instructions are independent. Thus, one method to reduce the overall effect of conflicts is to arrange the instructions in an order that will result in the same answers, but that will execute faster. For an example of this technique, consider the simple statements:

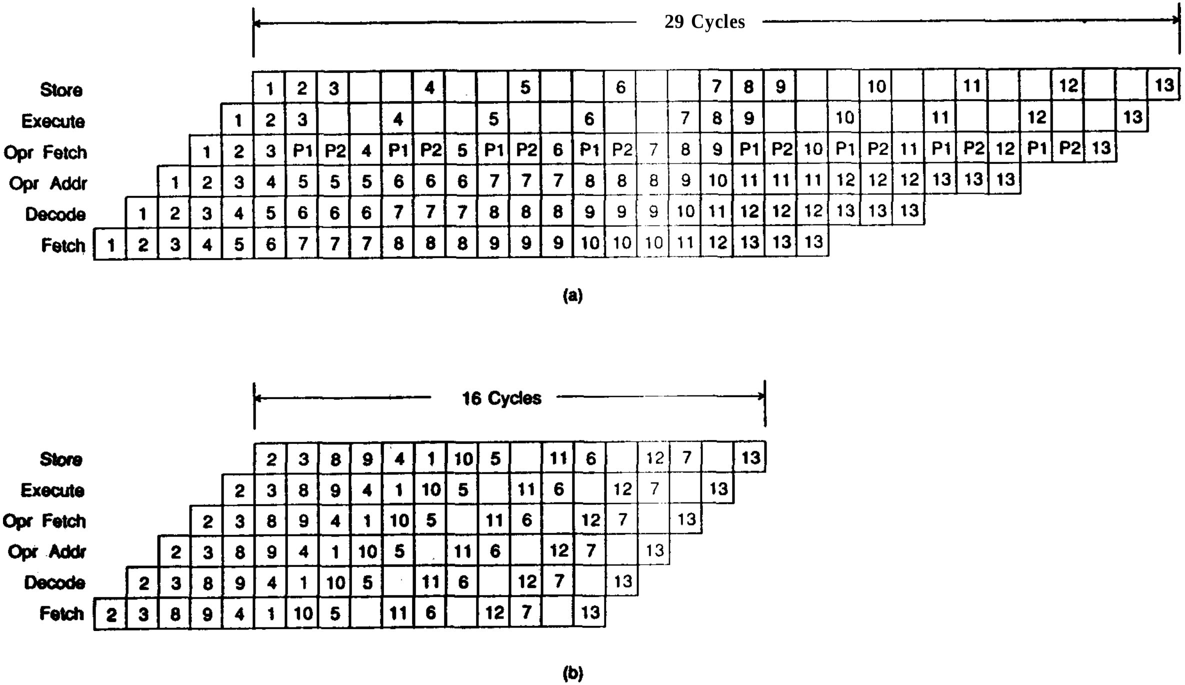
VX = VCC + (RES1 + RES2) × I1
VY = VCC + (RES3 + RES4) x 12

If we assume that these instructions are to be executed by a machine of the type that we have been discussing, then a very simple translation of the above statements into an assembly language might produce code similar to:

```
 1   MOV #<VCC>, RO       Get VCC to RO.
 2   MOV #<RES1>. R1      Get RES1 to R1.
 3   MOV #<RES2>. R2      Get RES2 to R2.
 4   ADD RI. R2           Add RES1 and RES2.
 5   MULT #<I1>. K?       Multiply by I1.
 6   ADD K0. K?           Add in VCC.
 7   MOV R2. @<VX>        Store result in VX.
 8   MOV #<RES3>, R3      Get RES3 to R3.
 9   MOV #<RES4>, R4      Get RES4 to R4.
10   ADD R3, R4           Add RES3 and RES4.
11   MULT #<I2>, R4       Multiply by I2.
12   ADD RO. R4           Add in VCC.
13   MOV R4, @<VY>        Store result in W.
```

This is a very simple set of code, yet it contains a number of data conflicts. If the code is executed as it appears above, then the resource utilization would appear as shown in Figure 8.25(a). The instructions that cause conflicts are instructions 4, 5, 6, 7, 10, 11, 12, and 13. Since all of the conflicts *are* data conflicts, they each incur a penalty of two clock times. The resulting time to complete the code segment, not including the fill time, is 29 cycles.

If some information is available about the organization of the pipeline. then appropriate choices can be made concerning the methods used by the assembly language implementations of the high level language statements like those shown above. By optimizing the order to help the conflict problem, then the time required to execute the code segment will decrease. In Figure 8.25(b), the same set of instructions is executed, but not in the order specified above. Rather, the order is specified in such a way as to guarantee that the operands are ready when needed by instructions which follow. In this way, the time required to wait fa operands is minimized. The results of the calculation will be the same as those shown in Figure 8.25(a), but the number of cycles required has been reduced to 16 cycles. Notice that independent instructions could be inserted into three spaces in the figure. If this were to be accomplished. then the pipeline would be functioning *at* maximum efficiency.



Store						1	2	3			4			5			6			7	8	9			10			11			12			13
Execute					1	2	3			4			5			6			7	8	9			10			11			12			13	
Opr Fetch				1	2	3	P1	P2	4	P1	P2	5	P1	P2	6	P1	P2	7	8	9	P1	P2	10	P1	P2	11	P1	P2	12	P1	P2	13		
Opr Addr			1	2	3	4	5	5	5	6	6	6	7	7	7	8	8	8	9	10	11	11	11	12	12	12	13	13	13					
Decode		1	2	3	4	5	6	6	6	7	7	7	8	8	8	9	9	9	10	11	12	12	12	13	13	13								
Fetch	1	2	3	4	5	6	7	7	7	8	8	8	9	9	9	10	10	10	11	12	13	13	13											

Store						2	3	8	9	4	1	10	5		11	6		12	7		13
Execute					2	3	8	9	4	1	10	5		11	6		12	7		13	
Opr Fetch				2	3	8	9	4	1	10	5		11	6		12	7		13		
Opr Addr			2	3	8	9	4	1	10	5		11	6		12	7		13			
Decode		2	3	8	9	4	1	10	5		11	6		12	7		13				
Fetch	2	3	8	9	4	1	10	5		11	6		12	7		13					

Figure 8.25. Resource Utilization for Code with Data Conflicts: (a) Simple Arrangement of Instructions; (b) Instructions Reordered to Minimize Conflicts

Example 85: Pipeline speedupsfor real systems: The effective speedup of a pipeline is a function of the **probability** of **conflictand** the penalty of **that conflict.** For the pipeline as shown in **Figure** 8.25. **is** the **formula** a reasonable **representation** of the actual speedup?

S i we an assuming ideal conditions. we will assume that execution of the instructions in a **nonpipelined system** will require **six** cycles per instruction. **Thus,** the 13 instructions of the code segment will require 78 cycles **to** complete. If the **pipeline** is **kept** full, and there are no conflicts, we would expect a speedup of six. Fmm the **code** segment and from Figure **8.25(a),** we identify that 8 **instructions** cause **conflicts,** and that the penalty of each is 2 cycles. **Thus,**

$$\text{Effective speedup} = \frac{\text{Best speedup}}{I + p \times P}$$

$$= \frac{6}{1 + (\frac{8}{13}) \times 2}$$

$$= 2.6896$$

The analytical approach says that we should see a speedup on the order of 2.69. Using the steady state number of 29 cycles, the speedup of the system that executes according to the method demonstrated by Figure 8.25(a) is:

$$\text{Effective speedup} = \frac{78}{29}$$

$$= 2.6896$$

which agrees with the calculated speedup. If the instructions **are reordered** as shown in **Figure 8.25(b), then** a **different calculation** is in order. **Here three instructions** (11, 12, **and** 13) have penalties associated with them, and the penalties an **only** *one* cycle. **Thus,**

$$\text{Effective speedup} = \frac{\textbf{Best}\text{ speedup}}{I + p \times P}$$

$$= \frac{6}{1 + (\frac{3}{13}) \times 1}$$

$$= 4.875$$

The actual time demonstrated by **Figure 8.25(b)** is 16 cycles:

$$\text{Effective speedup} = \frac{78}{16}$$

$$= 4.875$$

It is **interesting to** note that the reordering technique, **while not** modifying the hardware in any way, **resulted** in an **increase** in the **effectiveness of** the pipeline from an effective speedup of 2.689 to an effective speedup of 4.875. This is an increase of 81%.

The reordering scheme can be utilized by those who program the machine at the assembly language level. But more importantly, the technique can be used by compiler writers to generate code that will execute in a minimum amount of time. For example, one observation concerning the use of system resources in the above example is that better use could be made of the registers as temporary storage. By specifying different registers for each temporary variable, the conflicts could be minimized.

Performance enhancement can be accomplished by reordering since the technique works to organize the operations in an independent fashion. and independence leads to operation without conflicts. Another technique is to recognize that there will be conflicts in the instruction stream, and to attempt to minimize the penalty of a conflicts. One way to reduce the time required by many of the conflicts is to expand the capability of the storage function so that results are not only stored in a cycle. but they are. also made available to other stages of the pipe. That is, when the execute unit has completed an operation, the results can be supplied not only to the store unit, but they can also be provided to the other elements of the pipeline as needed. One representation of this path is shown in Figure 8.26, where the execute unit has a private data path that it can use to transfer information to other units in the pipeline.

The addition of the feedback path will reduce the penalty of many of the conflicts by one cycle. since the pipe sections need not wait until the store unit places the information into the memory or a register. The fetch operand section can obtain the data required for instructions following in the pipe. The generate operand address section can receive the information needed to specify operand addresses. And the instruction fetch/decode sections can identify the target address one cycle earlier, since the status information is made available at the same time that the status register is being updated. The overall effect is to greatly reduce the cycles consumed by all types of conflicts.

Example 8.6: Pipeline penalty reduction with internal data path: Assume that a feedback path exists in a pipelined unit as shown in Figure 8.26.

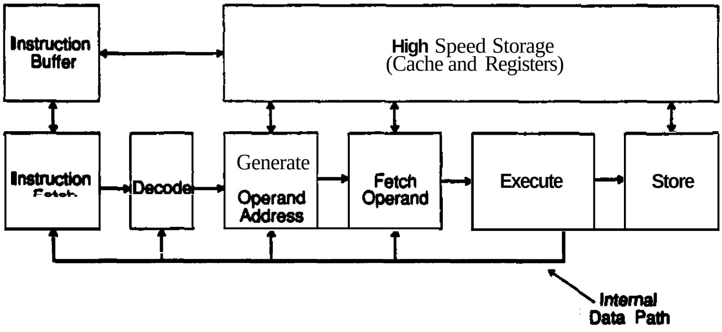


Figure 8.26. Block Diagram of Control Pipeline with **Internal Data Feedback Path.**

What will the effect be on the execution of the assembly language code used in the previous example?

The addition of the feedback path reduces the data conflicts penalty by one cycle, and this should be evident in the graph of resource utilization for the pipe. The code is repeated here for convenience:

```
 1  MOV #<VCC>, R0
 2  MOV #<RES1>, R1
 3  MOV #<RES2>, R2
 4  ADD R1, R2
 5  MULT #<I1>, R2
 6  ADD RO. R2
 7  MOV RZ, @<VX>
 8  MOV #<RES3>, R3
 9  MOV #<RES4>, R4
10  ADD R3. R4
11  MULT #<I2>, R4
12  ADD R0, R4
13  MOV R4, @<VY>
```

The graph of the resource utilization is included as Figure 8.27. As shown in part a, the simple, nonoptimized code now executes in 21 cycles, not

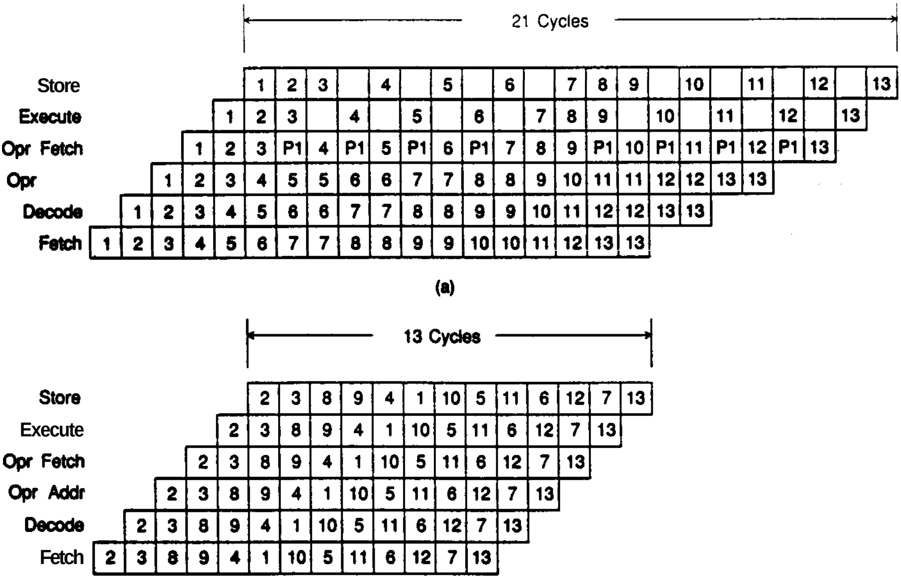

21 Cycles
Store 1 2 3 4 5 6 7 8 9 10 11 12 13
Execute 1 2 3 4 5 6 7 8 9 10 11 12 13
Opr Fetch 1 2 3 P1 4 P1 5 P1 6 P1 7 8 9 P1 10 P1 11 P1 12 P1 13
Opr 1 2 3 4 5 5 6 6 7 7 8 8 9 10 11 11 12 12 13 13
Decode 1 2 3 4 5 6 6 7 7 8 8 9 9 10 11 12 12 13 13
Fetch 1 2 3 4 5 6 7 7 8 8 9 9 10 10 11 12 13 13
(a)
13 Cycles
Store 2 3 8 9 4 1 10 5 11 6 12 7 13
Execute 2 3 8 9 4 1 10 5 11 6 12 7 13
Opr Fetch 2 3 8 9 4 1 10 5 11 6 12 7 13
Opr Addr 2 3 8 9 4 1 10 5 11 6 12 7 13
Decode 2 3 8 9 4 1 10 5 11 6 12 7 13
Fetch 2 3 8 9 4 1 10 5 11 6 12 7 13
(b)

Figure 8.27. Resource Utilization for Reduced Penalty Data Conflicts: (a) Simple Arrangement of Instructions; (b) Instructions Reordered to Minimize Conflicts.

including the fill time. Instructions4, 5, 6, 7, 10, 11, 12, and 13 still incur penalties. but now the penalties require only a one cycle delay. The resulting speedup becomes:

$$\text{Effective speedup} = \frac{\text{Best speedup}}{I + p \times P}$$

$$= \frac{6}{1 + (\frac{8}{13}) \times 1}$$

$$= 3.71$$

The reduction of the penalty from two cycles to one cycle has increased the speedup from 2.689 to 3.71, an increase of 38%. And examination of Figure 8.27(b) indicates that there are no unused cycles, so for the optimized case the speedup is at a maximum. By including the feedback path, the reordering technique in will produce results that are more effective than a system without the ability to bypass the storage function.

Reordering of instructions can be effective because of the use of independent instructions, since independent instructions do not compete for resources. Enhancing the data transfer capabilities of a pipeline to bypass the storage function reduces the penalties associated with all kind of conflicts. As we have seen, the penalties associated with the branch conflict are some of the largest penalties. so various techniques have been devised to try to minimize the overall branch conflict penalty. We will now describe one of these techniques and identify some of the additional problems created by the solution.

The main reason that branch conflicts cause delay is not that the functions cannot be performed, but rather that the machine docs not how which of the functions (instructions) are to be done. The correct "next" instruction following a conditional branch will not be identified until the condition on which the branch is based is known. However, one method to minimize the overall effect is to make a guess as to which of the instruction paths will be followed, and start execution along that path. Then. if the guess was correct, the penalty reduces to zero. However, if the guess was not correct, the time penalty will be the same as if no guess had been made.

To visualize this process, consider a set of instructions similar to those already examined. We will construct the set of instructions so that the only delay is a branch conflict, but with some problems that will demonstrate the added capabilities needed by the pipeline. The following instructions could be used to move data from one location to another.

```
I          MOV  #<FROM>, R0    R0 points to source of info.
2          MOV  #<TO>, R1      R1 points to where it goes.
3          MOV  #<1024>, R2    R2 is counter.
4   label  MOV  *R0+, *R1+     Move data, bump pointers.
5          DEC  R2             Decrement counter.
6          JNZ  label          If not zero, more data to move.
7          MOV  R5, R2         When done moving data, do this.
```

The pipeline system will make a guess as to the appropriate next instruction from 6. In this case, it is obvious that 1,023 times the next instruction is located at "label," so most of the time the correct choice will be instruction 4. If the designers of the system find that the JNZ instruction is indeed found principally at the end of a loop of this nature, then the system can be designed to assume that the branch is taken. Then the resource utilization graph for an iteration of the loop may appear as shown in Figure 8.28. The sequence of instructions indicated by the graph assumes that the next instruction after the jump will be instruction 4. Thus, that instruction is initiated, and the pipeline continues as if it were an unconditional jump. This should cause no problem until the operand fetch portion of the next instruction 4 that is to execute. Instruction 4 causes RO and R1 to increment, and if the branch is not taken, the values should not change. Therefore, a pipeline system that allows a conditional branch to follow one of the paths must be capable of flushing the pipe of the effects of the instructions if the path turns out to be the incorrect action. Thus, the operand fetch portion of the instruction circled in Figure 8.28 must not cause changes (in RO or RI) until after the validity of the path has been established.

By allowing the machine to continue execution. branch prediction techniques allow a system to minimize the time required to wait for conflicts to be resolved. This results in an overall speedup, even if the guesses are correct for only a fraction of the instructions. The larger the fraction, the greater the speedup. The cost of this speed enhancement is the additional hardware needed to allow the effects of a branch that should not have been taken to be removed from the pipe.

The final mechanism we will examine is another technique for minimizing the effect of conditional branches. This technique requires a combination of hardware and software to be effective, and hence must be applied in a system solution. That is. the hardware can provide the capability. but unless the software (compiler in conjunction with the operating system) makes use of the technique. no benefit will result.

One of the observations made earlier concerning the conditional branch penalties is that the target of the branch is not known until the condition on which the branch is based has been resolved. One approach to pipeline implementation is to cause the action of the system to stop until the condition has been determined. Since the desired effect of pipelining is to utilize the stages of the pipe as much of the time as possible, another approach is to design the pipeline in such a way that the instruction following a conditional branch is always executed. With this technique, an instruction that is always executed in the body of a loop can be placed directly after the conditional branch that determines the end of the loop, and it will produce the correct results.

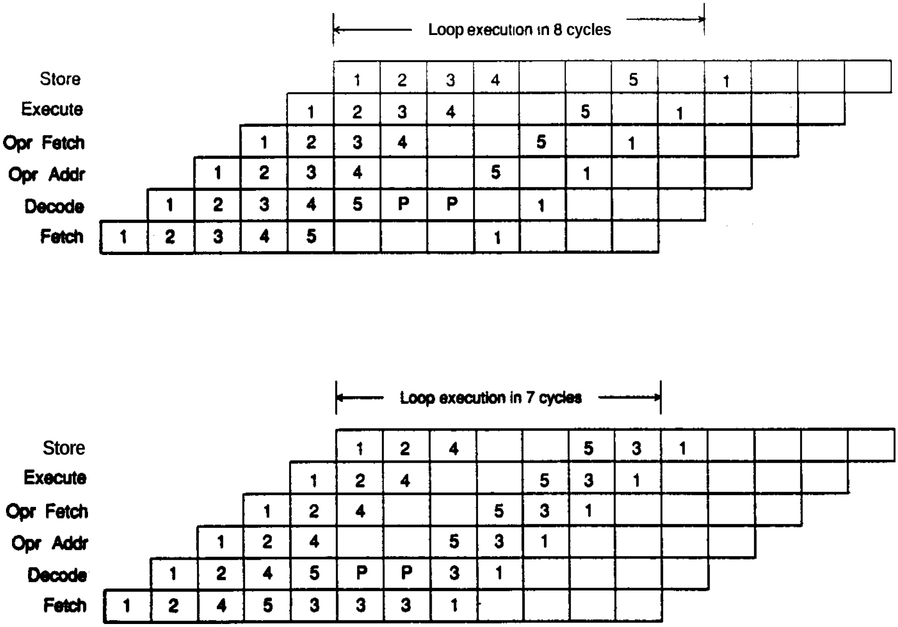
Store						1	2	3	4	5	6	4	5				
Execute					1	2	3	4	5	6	4	5					
Opr Fetch				1	2	3	4	5	6	④	5						
Opr Addr			1	2	3	4	5	6	4	5							
Decode		1	2	3	4	5	6	4	5								
Fetch	1	2	3	4	5	6	4	5									

Figure 8.28. Resource Utilization for a Pipeline with Branch Guess.

This technique results in more effective utilization of the stages of the pipeline, since an independent instruction is executed during the time required to identify the target of a conditional branch. To visualize this process wnsider the following set of instructions:

1	label	ADD R1, R3	Add two regs together.
2		ADD R2, R4	Add two other regs.
3		INC R0	Bump another reg.
4		CMP R3, R8	Do a comparison and if ...
5		BNE label	Values are equal, branch.

This code segment may result from a loop in a high level language. Note that instructions 1, 2, and 3 are executed each iteration of the loop. The result of applying this technique to the pipeline used as an example throughout this section is shown in Figure 8.29. Pan a of the figure indicates that the above loop will execute in 8 cycles. assuming that instruction 5 must wait until the execute portion of instruction 4 determines that the next instruction will k instruction number I. With this assumption, as soon as instruction 4 completes the execute section, instruction I can begin. Reordering the instructions to take advantage of the fact that an instruction following a conditional branch would result in the resource utilization shown in part b of the figure. Instruction 3 has been placed



Loop execution in 8 cycles

Store						1	2	3	4			5		1			
Execute					1	2	3	4			5		1				
Opr Fetch				1	2	3	4			5		1					
Opr Addr			1	2	3	4			5		1						
Decode		1	2	3	4	5	P	P		1							
Fetch	1	2	3	4	5				1								

Loop execution in 7 cycles

Store						1	2	4			5	3	1				
Execute					1	2	4			5	3	1					
Opr Fetch				1	2	4			5	3	1						
Opr Addr			1	2	4			5	3	1							
Decode		1	2	4	5	P	P	3	1								
Fetch	1	2	4	5	3	3	3	1									

Figure 8.29. Resource Utilization **With** and Without Using **Technique That** Always **Executes Instruction After Branch: (3Execution of** Instructions in "Normal" **Fashion;** (b) **Instruction** Execution **When** Instruction **Following** Branch Always Executes.

after the branch, and the system is designed so that the instruction will execute regardless of the result of the branch. With this technique applied to this pipeline. the resulting loop execution time is reduced by one cycle.

Studies have shown that this mechanism can be effectively utilized from 60–80% of the time, depending on the job type and the mechanisms involved. If it is determined that the instruction that follows the branch cannot be effective utilized, then a NOP (no operation) instruction is used in that slot The net result is that when the instruction following the branch is not a NOP, the branch penalty is reduced by one.

Example 8.7: RISC system pipeline: One of the available RISC systems is made by MIPS Corporation. What are the pipeline stages involved with the system? The pipeline of the MIPS system is shown in Figure 8.30. Basically, five sections are identified in the figure. The time for execution is indicated in the figure. and each section utilizes a time of one cycle, except the write back section. The first section is used for instruction fetch. and it has the responsibility of determining the real address of the instruction using the translation lookaside buffer, and then initiating the cache request for this information. This architecture calls for two separate cache systems, one for the instruction stream and one for the data stream. The instruction will not actually be provided until the first part of the decode stage.

The read/decode section obtains the instruction. decodes it, and reads any needed operands from the appropriate CPU registers. (RF stands for register fetch.) The information is then presented to the ALU stage.

The ALU stage performs any required work on operands obtained by the decode section. In the instruction is a LOAD or STORE, the TLB is interrogated to perform the virtual to real address translation to identify a spot in the data cache.

The memory section is responsible for handling LOAD and STORE instructions. The system does not allow operands used in arithmetic or logic instructions to be located in memory, so this section is responsible for moving the information needed for instructions to the appropriate system registers, and also for transferring information from registers back to memory. The addresses needed for this are generated in the half cycle preceding the memory request

The final section has the task of writing back ALU results or values loaded from the data cache to the register file.

This pipeline is capable of having five instructions executing at any one time, each in its appropriate section of the pipe. Note that TLB access is required for both the instruction address and the data address, and that these requests occur in different halves of a clock cycle. Thus, the TLB accesses will not slow the execution rate of the pipe.

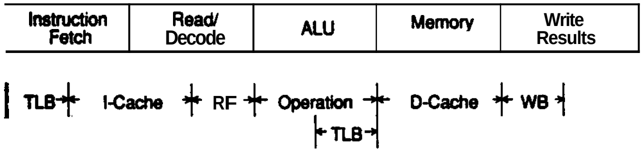


Figure 8.30. Pipeline Stages for MIPS System.

> The pipeline utilizes the delayed instruction technique for conditional branches, which we discussed above. However, the delayed availability of operands when performing load operations creates a condition similar to the conditional branch. The system utilizes the same technique with loads and stores as with branches: interlocks am not built into the hardware, but the software must be configured to assure that data not yet available is not requested for an operation. If there is no independent instruction which can be placed in the "delayed slot" after a branch or between fetching information from memory and using it, then a NOP *must* be inserted in the code. This policy makes the hardware easier to build, since conditions for halting the pipe are removed by careful attention to the software before the program runs.

Pipelines for control and instruction functions appear in virtually all high speed processors, and many microprocessors and smaller systems as well. The benefits obtained by execution of independent instructions in different sections of a pipeline justify the complexity of the system. In those systems with a possibility of using multiple system resources, interlocks must be provided to assure that the instructions will produce the proper results. Other systems, especially systems with short pipelines and systems that are RISC in nature, *use* software systems that create programs so that the results are correct, by presenting sequences of instructions that do not have resource conflicts.

8.4. Summary

The execution rate of computing systems can be increased by dividing the processing that needs to be done into small pieces and executing these pieces for different operations in the same time period. The division *process* breaks the required processing into sections each of which perform a portion of the overall function; the computational action is accomplished by passing information from one section to the next, and an operation is complete only when it has accomplished the work needed by all of the sections. Both data functions and control *or* instruction functions *can* be divided into basic sections and utilize the concept of pipelining.

With this technique, many different operations *can* be in progress at the same time, and each operation occupying a different stage of the pipeline. The highest rate for execution of operations with pipelining occurs when the pipeline is entirely full. When this is the case, each clock cycle results in another completed function, and extremely high computational rates can occur. To support this high execution rate, the data transfer mechanisms of the computational element that move information to and from memory, and to and from registers within the system, must be able to transfer operands and instructions at a rate sufficiently high to keep the system busy. If the data system is not capable of this high transfer rate. then the full benefits of pipelining will not be realized.

If the data system is capable of supporting the data transfers needed to maintain high data *rates* within a computer, then performance degradations will occur only when operations within the system are not independent. If the results of *one* operation am required by the next, then the appropriate pipeline *section* must wait until the data is available to proceed. The independence must be main*tained* in pipelines for both computational functions and control or instruction *functions.* Pipelines **can** be **required** to wait for **data** information **(data** conflicts). for **information needed** to generate operand addresses (address **conflicts),** and for status **information** needed to **identify** the target of a conditional branch **(branch** conflicts). These conditions arise when operations within the pipe are not **independent.**

Performance improvements will occur in pipeline systems **whenever** steps are taken to **reduce** the **penalties** associated with **nonindependent** operations. The four techniques presented in this chapter all seek to reduce the tine required to resolve conflicts **incurred** by *use* of common system resources. The **first** technique requires no **hardware** commitments; rather, the **software** is manipulated in such a way that the **instructions** are fed into the pipelined instruction unit in such a way that the operations are independent. This results in a higher apparent execution rate. The second technique is to provide an internal data path within the pipeline. so that operands can be obtained by sections of the pipe when the operands become available. rather than waiting for them to be stored. This **reduces** the time needed to wait for results.

Another technique presented is to allow the pipeline to identify the expected target of a conditional branch and begin execution at that point. This reduces to **zero** the **penalty** associated with the branch if the **guess** is **correct,** but incurs the cost of being able to flush from the pipe the effects of those instructions followed if the guess is incorrect. The final technique **presented** is to design the system in such a way that the instruction following a conditional branch is always executed. and relv on the users of the system (compiler writers. assembly language programmers. etc.) to create the programs in such a way that the instruction after the branch is effectively utilized.

8.5. Problems

8.1 Develop a formula for T_{TOT}, the total tine required to perform N independent arithmetic **operations** in an arithmetic pipe. Assume that the arithmetic pipe contains 6 stages. and that each **stage** executes in 100 nsec.

8.2 Design a **pipelined** floating point add unit. To **accomplish** this:

a. Give a block diagram of the floating point add operation.

b. Describe each element in the block diagram, and specify the **hardware needed** to perform the work of that block.

c. Identify the delays associated with each of the blocks in the block diagram.

d. Insert registers *at* appropriate locations in the block diagram. (What &lays are associated with the registers?)

e. **Assuming** that collision avoidance is *handled* by another piece of the system, identify the controls **needed** in **the pipelined** unit, and show how the **control** is handled

f. What is the data rate needed to keep the pipeline full?

8.3 Assume that a high speed floating point multiply **part** is available at a reasonable cost. Give r block diagram of a pipeline **used** to **provide** a divide operation. If a **ROM** is available to specify the first **coefficient** to 15 **bits,** how many stages are **required** to provide a **result** correct to 56 bits (double precision floating point)? If the multiply takes 120 nsec, and the register requires 10 nsec, what is the floating point divide rate? What must the data rate of the memory system be to sustain the highest operation rate?

8.4 Show a block diagram level design for a pipelined vector floating point add system. Include vector registers capable of holding up to 64 elements of a vector. Show on a time plot the action of each of the elements of the system for a period of 10 clock cycles.

8.5 Give a detailed logic diagram for the pipelined system of Problem 8.4. Assume that you have memories which are 64 × 8 to work with, and that these come in 20 pin packages. For this problem:

a. Use the block diagram of Problem 8.4. Remember to include whatever address registers are needed to identify the elements of the vectors.

b. Specify the devices needed to implement the system.

c. Identify the control lines needed to control the flow of data in the system.

d. Create a control system that will assert the control signals in the proper fashion to do the calculation as specified.

e. Provide the logic diagrams of the system.

8.6 Discuss possible alternatives, as well as their advantages and disadvantages, for the organization of a memory system to be used with the pipelined system of Problem 8.4, and the interconnection between the memory and the vector registers.

8.7 Identify the modifications required to add another pipelined functional unit to the system of Problem 8.4. That is, the system as specified is capable of doing a floating point add for vectors of values. What would be required to add a functional unit that would utilize the same vector registers, but that would do a floating point multiply for vectors of numbers?

8.8 A pipelined control unit has six separate stages. The various collisions that can occur in the system can produce penalties of one cycle, two cycles, three cycles, or four cycles. Develop a formula that will give the effective speedup for the system as a function of the probability of each of the four penalties. Plot the formula such that the abscissa is the effective speedup, and the ordinate is the probability of collisions. For the plot, vary only one probability at a time, leaving the other three probabilities zero.

8.9 The instruction portion of a certain computer has been broken into five distinct parts for purposes of pipelining: instruction fetch, instruction decode, operand fetch. instruction execution, and storage of results, where necessary. Assume that all instructions must take all five cycles to execute. Assume also that results are not available until after the end of the fifth section. The code segment that is to run on this fictitious machine is:

```
        MAD R1 with -512
        LOAD R2 with 4000
        LOAD R3 with 5000
        LOAD R4 with 6000
OVER    MOVE *R2, R8          * is indirection, destination is R8.
        MOW *R3, R9
        ADD R8, R9
```

```
MOVE R9, *R4
ISZ R2              ISZ is increment and skip if zero,
ISZ R3
ISZ R4
ISZ R1
JNZ OVER
```

a. If there were neva any problem with collisions, how much faster would the pipelined system run than unpipelined system?

b. How long will it take to execute the above section of code? Assume that the time step is 100 nsec.

8.10 Consider a two address computer designed with a pipelined control section. The sections are instruction fetch, decode, operand I fetch, operand 2 fetch. execute. and store. Each section does its work in one cycle. The store section can be bypassed for data needed for either operand fetch section. The status bits are available for testing directly after the execute section. The branch guess is the next instruction in line. How many cycles does it take to compute the following section of code:

```
        MOV    100, R0    100 decimal to counter.
label   MOV    R0, R2     Count to temporary.
        ADD    R3, R2     Add constant to temporary
        MOV    R2, *R0    Store in memory.
        DEC    R0         Decrement counter.
        BNZ    label      Loop til done.
```

8.11 Consider a computer with an instruction time of 8t units. Assume that conditions are ideal and that this computer can be redesigned to take advantage of pipelining, and that the pipeline consists of four equal segments. For this new machine:

a. How long will it take to start and complete a sequence of 40 instructions?

b. A time penalty is associated with collisions in the pipeline. That is. the pipeline must halt for some period of time when an applicable collision occurs. (Remember that independent instructions do not cause collisions, and that collisions result in modified data/address do not incur a penalty.) How long does it take to completely execute the following code to add two vectors? (I, J, K, L stored in registers.)

```
         10 → I      I,J,K can be wed as addresses.
         20 → J
         30 → K
        -20 → L
label:  CLA          Clear the accumulator.
        ADD M[I]     Add in one value.
        INC I        Bump the Address.
        ADD M[J]     Add in another value.
        INC J        Bump the address.
        STO M[K]     Store the result.
        INC K        Bump the address.
        ISZ L        Check to see if done.
        JMP label    If not done, go back.
        HALT         Otherwise, quit
```

8.12 Consider the instruction unit of a computer; a design team wants to pipeline the system. Currently the process of fetching, decoding, and executing instructions takes 600 nsec. By very clever work, you have been able to divide the unit into six separate actions: fetch, decode, operand 1 fetch, operand 2 fetch, execute, store result. Each of these actions will take 100 nsec, and single operand instructions do nothing during operand 2 fetch. A data collision in a register must wait for the correct value to reach the register. By being extremely clever the design team has eliminated the address collision problem.

a. If this machine executes 60 instructions, all independent so there are no collisions, how much time elapses between initiation of the first instruction and completion of the last?

b. The following set of instructions adds two vectors together and stores the final value in a new vector. The conditional jump instruction here is designed to assume that the jump will be successful. How long will it take to complete this set of instructions?

```
        300 → J  (a register)
        400 → K  (a register)
        20 → L  (a register)
LABEL:  MOVE MEM[L],N  (N is a register)
        ADD MEM[J],N
        DEC J
        STORE N,MEM[K]
        DEC K
        DEC L
        JNZ LABEL
        HALT
```

c. If the machine were to stay in the above loop (LABEL-JNZ) forever, what would the effective instruction time be?

8.6. References and Readings

[AnSp67] Anderson, D. W.. F. J. Sparacio and R M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,⁴ *IBM Jownal of Research and Developments.* Vol. 11, No. 1, January 1967, pp. 8–24.

[Baer84] Baa, J. L, "Computer Architecture," *Computer.* Vol. 17, No. 10, October 1984, pp. 77-87.

[Baer80] Baa. J. L. *Computer System Architecture.* Rockville, MD: Computer Science Press, 1980.

[BaBr68] Barnes, G. H., R M. Brown, M. Kato, a al., "The Illiac IV Computer, *IEEE Transactions on Computers.* Vol. C-17, No. 8, August 1968, pp. 746–757.

[Batc80] Batcher, K. E., "The Design of a Massively Parallel Processor," *IEEE Transactions on Computers.* Vol. C-29, No. 9, September 1980, pp. 836–840

[BeNe71] Bell, C. G. and A. Newell, *Computer Structures: Readings* and *Examples.* New York: McGraw-Hill Book Company, 1971.

[BuGo46] Burks, A. W., H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Studies, 1946, reprinted in [Swar76].

[Chen80] Chen, T. C., "Overlap and Pipeline Processing," in [Ston80], pp. 427–485.

[Chen71] Chen, T. C.. "Parallelism Pipelining, *and* Computer Efficiency," *Computer Design.* Vol. 10, No. 1, January 1971, pp. 69–74.

[Davi71] Davidson, E. S. The Design and Control of Pipelined Function Generators," *Proceedings of the IEEE International Conference on Systems Networks and Computers.* 1971, pp. 19–21.

[DRGI85] DeRosa, J., R. Glackemeyer, and T. Knight. "Design and Implementation of the VAX 8600 Pipeline," *Computer.* Vol. 18. No. 5, May 1985, pp. 38-48.

[FoME85] Fossum, T., J. B. McElroy, and W. English, "An Overview of the VAX 8600 System," *Digital Technical Journal.* Hudson, *MA:* Digital Equipment Corporation, 1985. pp. 8–23.

[BrHe82] Gross, T. R.. and I. L. Hennessy, "Optimizing Delayed Branches," *Proceedings of the 15th Annual Workshop on Microprogramming.* New York, NY: IEEE Computer Society Press, 1982. pp. 114–120.

[Hill85] Hillis, W. D.. *The Connection Machine.* Cambridge, *MA:* MIT Press, 1985.

[HwBr84] Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

[Kane87] Kane. Gerry, *MIPS R2000 RISC Architecture.* Englewood Cliffs. *NJ:* Prentice Hall. 1987.

[Kell75] Keller. R. M.. "Lookahead Processers." ACM *Computing Surveys.* Vol. 7, No. 4, 1975, pp. 177–195.

[Kogg81] Kogge, P. M. *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[KuSm86] Kunkel, S. R., and J. E. Smith, "Optimal Pipelining in Supercomputers," *Proceedings of the 13th International Symposium on Computer Architecture.* Washington, DC: IEEE Computer Society Press, 1986, pp. 404–411.

[RaLi77] Ramamoorthy, C. V., and H. F. Li, "Pipeline Architecture," *ACM Computing Surveys.* Vol. 9, No. I. March 1977, pp. 61–102.

[Russ78] Russell, R. M., "The CRAY-1 Computer System," *Communications of the ACM.* Vol. 21, No. 1, January 1978, pp. 63-72.

[Ryma82] Rymarczyk, J., "Coding Guidelines for Pipelined Processors," *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating System.* New York: ACM, 1982, pp. 12–19.

[SiBe82] Siewiorek, D. P., C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples.* New York: McGraw-Hill Book Company. 1982

[Site78] Sites, R L., "An Analysis of the CRAY-1 Computer," *Proceedings of the 5th Symposium on Computer Architecture,* New Yak: IEEE Computer Society Press 1978, pp. 101–106.

[Ston80] Stone, H. S., (Ed.), *Introduction to Computer Architecture.* Chicago, IL: Science Research Associates, 1980.

[Ston87] Star, H. S., *High-Performance Computer Architecture.* Reading *MA:* Addison-Wesley Publishing Company, 1987.

[Swar76] Swartzlander, E. E., Jr. (Ed.), *Computer Design Development: Principal Papers.* Rochelle Park, NJ: Hayden Book Company, Inc., 1976.

[Thor64] Thornton, J. E., "Parallel Operation in the Control Data 6600," *Proceedings of the Fall Joint Computer Conference.* AFIPS, Montvale, NJ:AFIPS Press, Vol. 24, 1964, pp. 33–40.

[Thur76] Thurber, K. J., *Large Scale Computer Architecture. Parallel and Associative Processers.* Rochelle Park, NJ: Hayden Book Company, Inc., 1976.

[Toma67] Tomasulo, R. M., "An Efficient Algorithm f a Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development.* Vol. 11, No. 1, January 1967, pp. 25–33.

[TrCh85] Troiani, M., S. S. Ching, N. N. Quaynor, et al., "The VAX 8600 I Box. A Pip lined Implementation of the VAX Architecture," *Digital Technical Journal.* Hudson, *MA:* Digital Equipment Corp., 1985. pp. 24–42.

[WeRo84] Wedig, R. G., and A. Rose, "The Reduction of Branch Instruction Execution Overhead Using Structured Control Flow." *Eleventh Annual International Conference on Computer Architecture,* Silver Springs, *MD:* IEEE Computer Society Press, June 1984. pp. 119–125.

[WeSm84] Weiss, S.. and J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," *Transactions on Computers.* Vol. *C-33.* No. 11, November 1984, pp. 1013–1022.