

APPENDIX I

ADDITIONAL INSTRUCTION PIPELINE TOPICS

William Stallings

Copyright 2012

I.1	PIPELINE RESERVATION TABLES	2
	Reservation Tables for Dynamic Pipelines	2
	Instruction Pipeline Example.....	8
I.2	REORDER BUFFERS	13
	In-Order Completion	13
	Out-of-Order Completion	15
I.3	TOMASULO'S ALGORITHM	18
I.4	SCOREBOARDING	24
	Scoreboard Operation.....	25
	Scoreboard Example	28

Supplement to
Computer Organization and Architecture, Ninth Edition
Prentice Hall 2012
ISBN: 013293633X
<http://williamstallings.com/ComputerOrganization>

This appendix expands on some topics referenced in Chapters 14 and 16.

I.1 PIPELINE RESERVATION TABLES

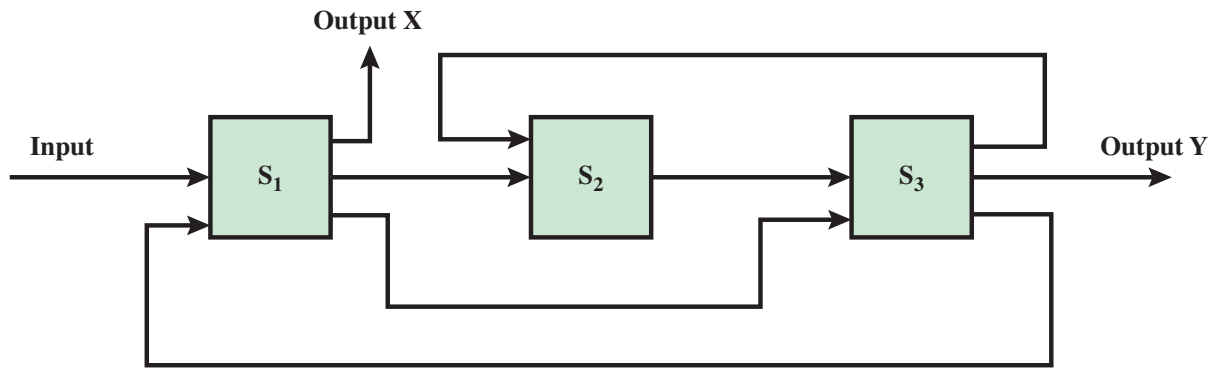
A central problem in pipeline design is that of ensuring that the pipeline can achieve the highest throughput consistent with avoiding structural hazards. That is, we would like to feed instructions into the pipeline at the highest rate that can be achieved without having collisions in the use of a shared resource. One of the earliest techniques introduced for this purpose is the reservation table ([DAVI71], [DAVI75], [PATE76]).

A reservation table is a timing diagram that shows the flow of data through a pipeline and indicates which resources are needed at each time interval by an instruction flowing through the pipeline.

Reservation Tables for Dynamic Pipelines

First, we look at the more general case of a dynamic pipeline. A dynamic pipeline is one that can be reconfigured to support different functions at different times. Further, such a pipeline might involve feedforward and feedback connections. Most instruction pipelines will not exhibit all of this flexibility, although some dynamic features may appear in some implementations. In any case, a static pipeline, which performs a fixed function without feedback or feedforward loops is just a special case for which the same principles apply.

Figure I.1a, an example taken from [HWAN93], shows a multifunction dynamic pipeline. Two reservation tables are shown in Figures I.1b and I.1c, corresponding to a function X and a function Y. In the reservation table, the rows correspond to resources (in this case pipeline stages), the columns to time units, and an entry in row i , column j indicates that station i is busy at time j .



(a) A three-stage pipeline

	time →							
	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

(b) Reservation table for function X

	time →					
	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

(c) Reservation table for function Y

Figure I.1 Pipeline with Feedforward and Feedback Connections for Two Different Functions

The number of time units (clock cycles) between two imitations of a pipeline is the **latency** between them. Any attempt by two or more initiations to use the same pipeline resource at the same time will cause a **collision**. It is readily seen that a collision will occur if two instructions are

initiated with a latency equal to the distance between two entries in a given row. From an examination of the reservation table, we can obtain a list of these forbidden latencies and build a **collision vector**:¹

$$(C_1, C_2, \dots, C_n)$$

where

$C_i = 1$ if i is a forbidden latency; that is initiating a pipeline instruction i time units after the preceding instruction results in a resource collision

$C_i = 0$ if i is a permitted latency

n largest value in the collision list

Although these examples do not show it, there may be multiple consecutive entries in a row. This corresponds to a case in which a given pipeline stage requires multiple time units. There may also be multiple entries in a column, indicating the use of multiple resources in parallel.

To determine whether a given latency is forbidden or permitted, we can take two copies of the reservation table pattern and slide one to the right and see if any collisions occur at a given latency. Figure I.2 shows that latencies 2 and 5 are forbidden. In the figure, the designation X_1 refers to the first initiation of function X , and X_2 refers to the second initiation. By inspection, we can also see that latencies 2, 4, 5, and 7 are all forbidden.

¹ The online simulation used with this book defines the collision vector beginning with C_0 . However a latency of 0 will always produce a collision, so this convention is not followed in the literature. Also, some treatments of reservation tables use the reverse order for the collision vector: (C_1, C_2, \dots, C_n) .

	1	2	3	4	5	6	7	8	9	10
S_1	X_1		X_2			X_1		X_1, X_2		X_2
S_2		X_1		X_1, X_2		X_2				
S_3			X_1		X_1, X_2		X_1, X_2		X_2	

(a) Collision with scheduling latency 2

	1	2	3	4	5	6	7	8	9	10	11
S_1	X_1					X_1, X_2		X_1			
S_2		X_1		X_1			X_2		X_2		...
S_3			X_1		X_1		X_1	X_2		X_2	

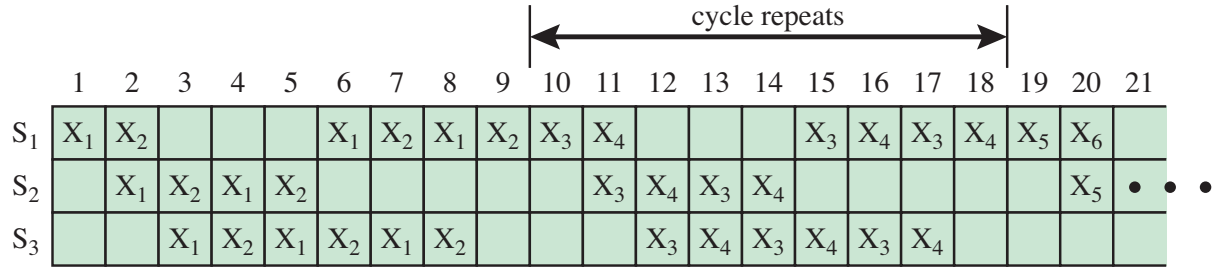
(b) Collision with scheduling latency 5

Figure I.2 Collisions with Forbidden Latencies 2 and 5 in Using the Pipeline in Figure I.1 for the Function X

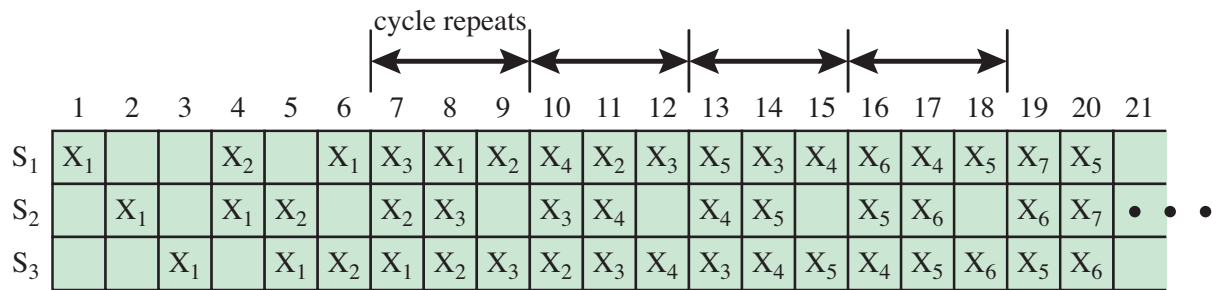
Latencies 1, 3, 6 are permitted. Any latency greater than 7 is clearly permitted. Thus, the collision vector for function X is:

$$C_X = (0101101)$$

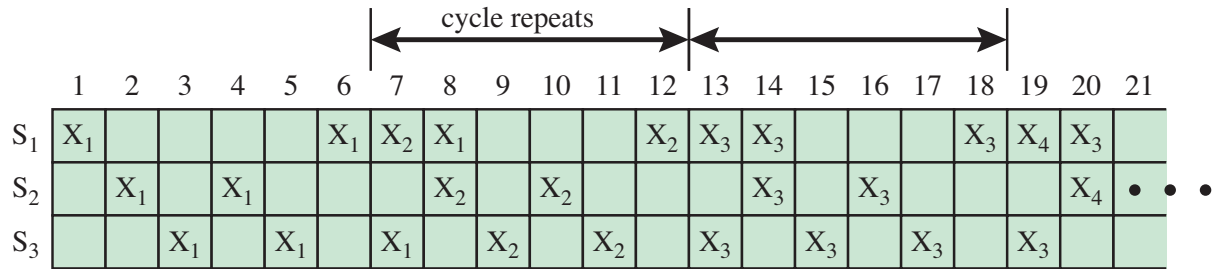
We must be careful how we interpret the collision vector. Although a latency of 1 is permitted, we cannot issue a sequence of instructions, each with a latency of 1 to the preceding instruction. To see this, consider that the third instruction would have a latency of 2 relative to the first instruction, which is forbidden. Instead, we need a sequence of latencies that are permissible with respect to all preceding instructions. A **latency cycle** is a latency sequence that repeats the same subsequence indefinitely.



(a) Latency cycle (1, 8) = 1, 8, 1, 8, ... with average latency of 4.5



(b) Latency cycle (3) = 3, 3, 3, 3, ... with average latency of 3



(c) Latency cycle (6) = 6, 6, 6, 6, ... with average latency of 6

Figure I.3 Successful Latencies for the Pipeline in Figure I.1 for the Function x

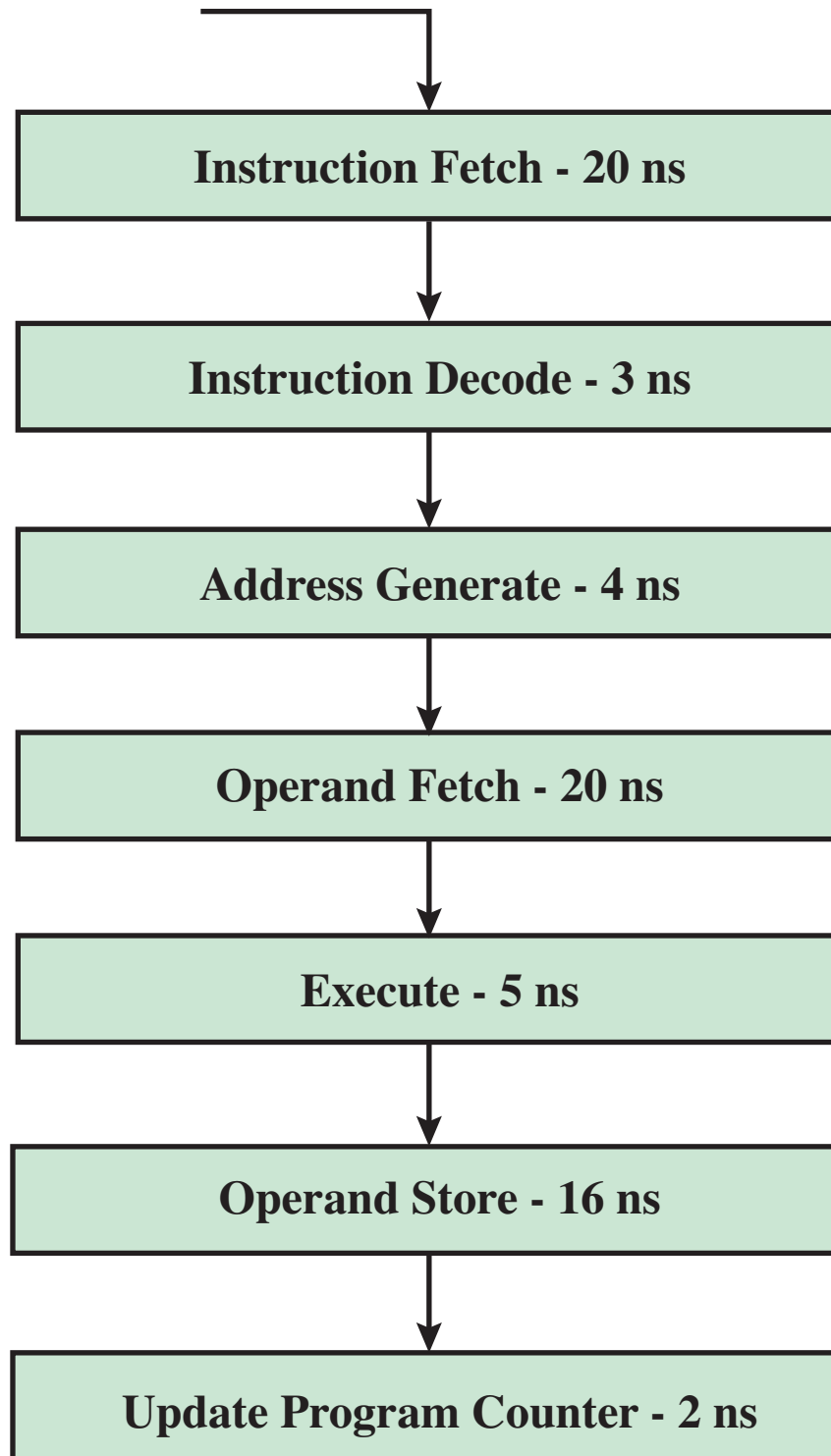


Figure I.4 An Instruction Pipeline

Figure I.3 illustrates latency cycles for our example that do not cause collisions. For example, Figure I.3a implies that successive initiations of new tasks are separated by one cycle and eight cycles alternately.

Instruction Pipeline Example

Let us look now at the construction of a reservation table for a typical pipeline. We use as an example a pipeline presented in [STON93]. Figure I.4 illustrates the pipeline and indicates the approximate amount of time for each stage. A straightforward attempt to develop a reservation table yields the result in Figure I.5a. To develop this table, we use a fixed clock rate and synchronize all processing with this fixed clock. Each stage performs its function within one or more clock periods and shifts results to the next stage on the clock edge. For this example, we choose a 20-ns clock.

However, we need to keep in mind that the rows of the table are supposed to correspond to resources, not simply pipeline stages. In Figure I.5b, we modify the reservation table to reflect that the memory operations are using the same functional unit. Here, the only permissible latency is 15.

Now suppose that we implement an on-chip cache and reduce memory access time to one clock cycle. The revised reservation table is shown in Figure I.6a. By sliding one copy of the reservation table pattern over another, we easily discover that the collision vector is 011010. The question then arises: what schedule will provide the lowest average latency. As Figure I.3 indicates, there are a number of possibilities for any given collision vector.

A convenient tool for determining when we can initiate a new process into the pipeline and avoid collisions, when some operations are already in the pipeline, is the collision vector state transition diagram. The procedure for creating the state diagram is as follows:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Instr Fetch	X	X	X	X												
Instr Dec					X											
Addr Gen						X										
Op Fetch							X	X	X	X						
Execute											X					
Op Store												X	X	X	X	
Update PC																X

(a) Reservation table: first attempt

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Mem OP	X	X	X	X			X	X	X	X		X	X	X	X	
Instr Dec					X											
Addr Gen						X										
Execute											X					
Update PC																X

(b) Reservation table: second attempt

Figure I.5 Reservation Table for an Instruction Pipeline

1. Shift the collision vector left one position, inserting a 0 in the rightmost position. Each 1-bit shift corresponds to an increase in latency by 1.
2. Continue shifting until a 0 bit emerges from the left end. When a 0 bit emerges after p shifts, this means that p is a permissible latency.
3. The resulting collision vector represents the collisions that can be caused by the instruction currently in the pipeline. The original collision vector represents the collisions that can be caused by the instruction

that we now insert into the pipeline. To represent all the collision possibilities, we create a new state by bitwise-ORing the initial collision vector with the shifted register.

4. Repeat this process from the initial state for all permissible shifts.
5. Repeat this process from all newly created states for all permissible shifts. In each case, bitwise or with the original collision vector.

Figure I.6b shows the result for the reservation table of Figure I.6a. The state diagram shows that, after we have initiated an operation into an empty pipeline, we can initiate a new operation into the pipeline after 1 cycle. However, this brings us to a state where we cannot initiate another operation until 6 more time units. This gives us an average latency of 3.5 for a seven-stage pipeline, which is not very good. Alternatively, we can wait for 4 time units after the initial operation begins, and then initiate a new operation every 4 cycles. This yields even poorer performance, with an average latency of 4.

A clever way to achieve better performance is to actually introduce a deliberate delay into the pipeline. Suppose we put a one-cycle delay after the execute stage. This results, in effect, in an 8-stage pipeline. The reservation table is shown in Figure I.7a, where the entry D indicates a delay. This yields a collision vector of 0011010. We now have latency possibilities of 1, 2, 4, and 7. Figure I.7b shows the resulting state transition diagram.² We now look for a closed loop, or cycle, through the state diagram that yields the minimum average latency.

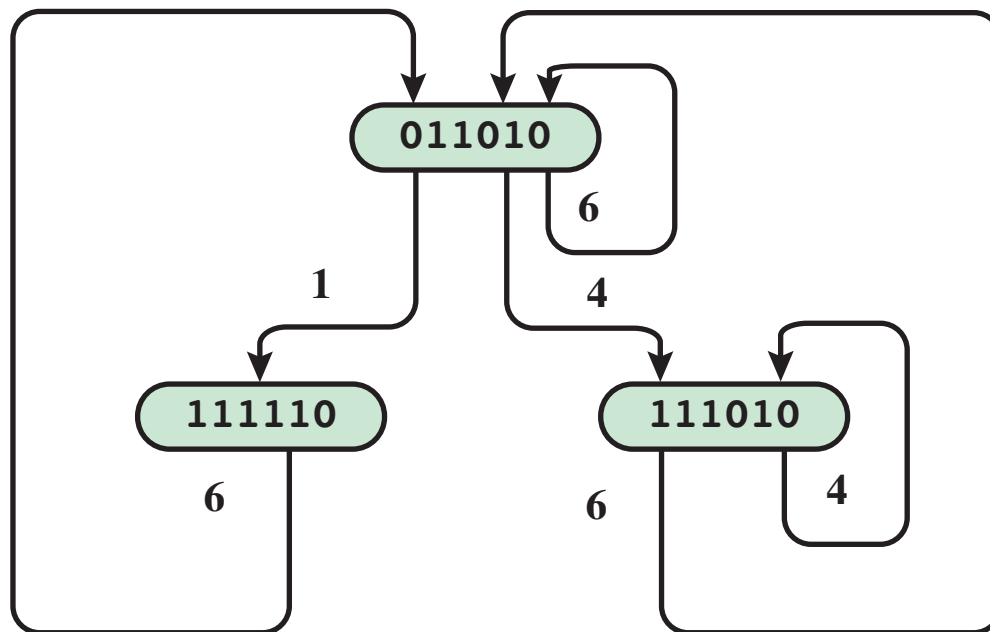
One possibility is the greedy cycle. A **greedy cycle** is one whose edges are all made with minimum latencies from their respective starting positions; that is, at each state, choose the exiting edge with the smallest latency

² All the states have an arc back to the beginning with a latency of 7, in addition to those noted. There are not shown for clarity.

						</	

Collision vector = 011010

(a) Reservation table



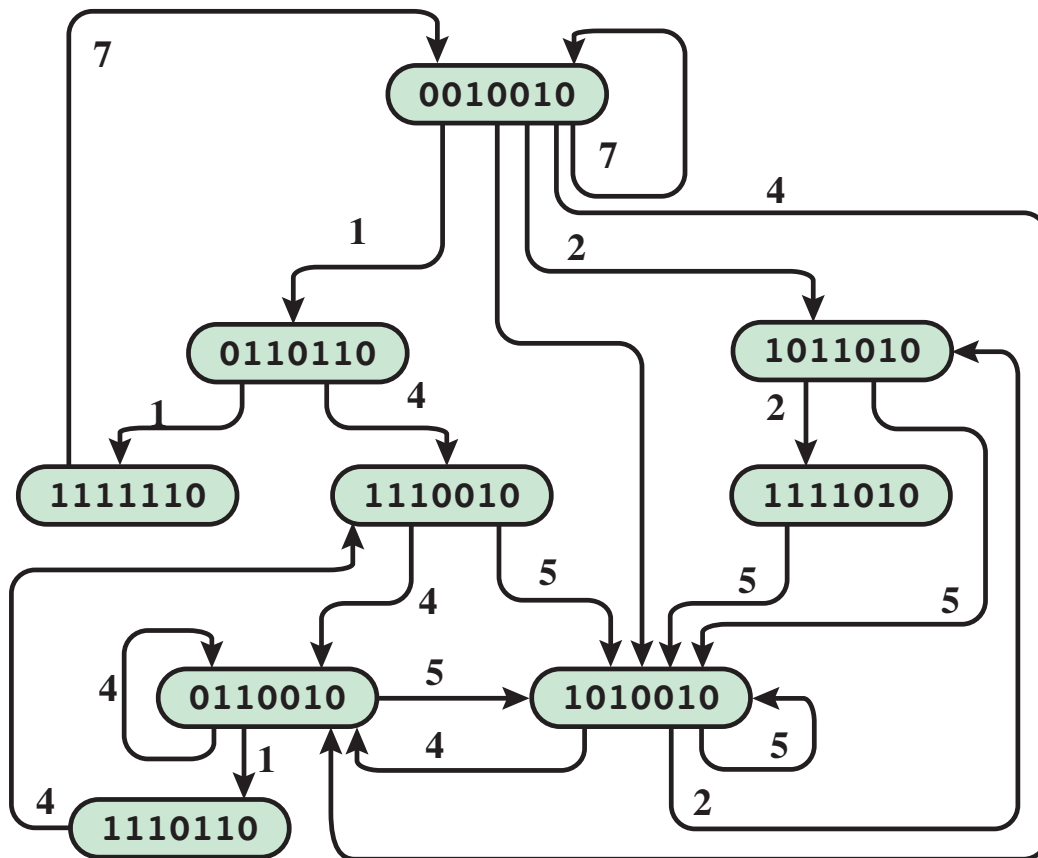
(b) State transition diagram

Figure I.6 Reservation Table and State Transition Diagram for Revised Instruction Pipeline

	time →						
	1	2	3	4	5	6	7
Mem Op	X			X			X
Instr Dec		X					
Addr Gen			X				
Execute					X	D	
Update PC							X

Collision vector = 0011010

(a) Reservation table



(b) State transition diagram

Figure I.7 Reservation Table and State Transition Diagram for Instruction Pipeline with Delay Inserted

value. In this case, the greedy cycle has latency values 1, 1, 7, so that the average latency is $9/3 = 3$. A careful study of the diagram shows that, in this case, no other cycle produces a smaller average latency.

I.2 REORDER BUFFERS

As was mentioned in Chapter 14, a common superscalar technique to support out-of-order completion is the reorder buffer [SMIT88]. The reorder buffer is temporary storage for results completed out of order that are then committed to the register file in program order.

To explain the operation of the reorder buffer, we first need to look at a technique that supports in-order completion, known as the result shift register.

In-Order Completion

With in-order completion, an instruction is dispatched and allowed to modify the machine state (register values, interrupt status) only if no preceding instruction has caused an interrupt condition. This restriction ensures that an instruction will not be retired until preceding instructions have completed any changes to the register file.

The processor controls the writing of results to registers by means of a **result shift register**.³ The result shift register is a table with as many

³ The term *shift register* is somewhat misleading. The shift is not a bitwise shift, left or right, but rather a shift of all of the entries in the data structure up one entry position, with the topmost entry shifted out of the structure.


Direction of movement	Stage	Functional Unit Source	Destination Register	Valid Bit	Program Counter
	1				
	2				
	3				
	4				
	5				
	⋮	⋮	⋮	⋮	⋮
	⋮	⋮	⋮	⋮	⋮
	⋮	⋮	⋮	⋮	⋮
	N				

Figure I.8 Result Shift Register

entries (rows) as there are pipeline stages in the longest execution pipeline (Figure I.8). Each entry, if valid, refers to a single instruction, and consists of four fields:

- **Functional unit source:** the functional unit that will be supplying the result
- **Destination register:** the destination register for the result
- **Valid bit:** indicates whether this entry currently contains valid information
- **Program counter:** location of the instruction

An instruction that takes i clock periods reserves stage i of the result shift register at the time it issues. If the valid bit of stage i is already set, then instruction issue is held until the next clock period, and stage i is

checked once again. When an entry can be made, the four fields of the entry are filled in. At the same time, all entries in the result shift register from 1 through $i - 1$ that are not in use are filled with null information but the valid bit is set. This prevents a following short-latency instruction from reserving any of the lines preceding i , and therefore ensures that no instruction is issued if it will finish execution before a logically preceding instruction.

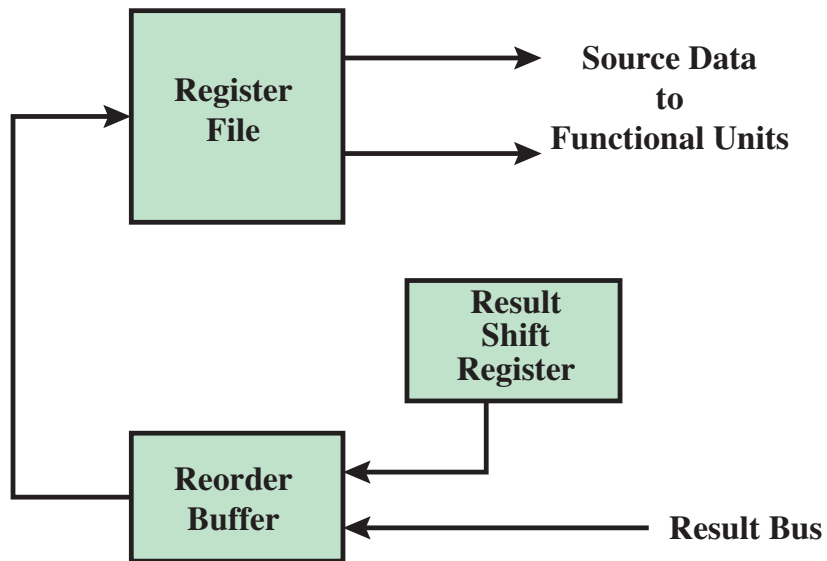
Finally, at each clock cycle, entry 1 of the register is deleted, the remaining entries are shifted up one row, and a null entry is placed in row N .

Out-of-Order Completion

The reorder buffer is used to support out-of-order completion by delaying the retiring of a completed instruction until results can be posted in the logically correct order. Figure I.9a shows the organization for this approach. A modified result shift register is still used, but this now provides input to the reorder buffer. Results from completed instructions also provide input to the reorder buffer. The update of registers with results from completed instructions takes place through the reorder buffer, which, as the name suggests, reorders results from completed instructions to assure correct execution.

Figure I.9b shows the details of the two data structures used in this approach. The **reorder buffer** is a circular buffer with head and tail pointers. Entries between the head and the tail are considered valid.⁴ When an instruction issues, the next available reorder buffer entry, pointed to by the tail pointer, is given to the issuing instruction. The entry number value for this instruction is used as a tag to identify the entry in the buffer reserved for the instruction. The tag is placed in the result shift register along with other control information. The tail pointer is then incremented,

⁴ If head greater than tail, the valid entries are from head to the end of the buffer, then from the beginning of the buffer to tail.



(a) Reorder buffer organization

	Stage	Functional Unit Source	Valid Bit	Tag
Direction of Movement ↑	1			
	2			
	3			
	4			
	5			
		⋮	⋮	⋮
	N			

Result Shift Register

	Entry Number	Dest. Reg	Result	Excep-tions	Valid Bit	Program Counter
Head →	1					
	2					
	3					
	4					
	5					
		⋮	⋮	⋮	⋮	⋮
Tail →						

Reorder Buffer

(b) Reorder buffer and associated result shift register

Figure I.9 Reorder Buffer Implementation

modulo the buffer size. The result shift register differs from the one used earlier because there is a field containing a reorder tag instead of a field specifying a destination register.

When an instruction completes, results are sent to the reorder buffer. The tag from the result shift register is used to guide them to the correct reorder buffer entry. When an entry at the head of the reorder buffer contains valid results (its instruction has completed, the results are written into the registers.

Note that the entries in the reorder buffer are in the order in which the instructions are issued. Thus the reorder buffer enforces a write to the registers in the logically correct order. That is, the reorder buffer serves as temporary storage for results that may be completed out of order and then commits these results to the register file in program order. The result shift register is not strictly necessary in this arrangement but does provide an indexing mechanism into the reorder buffer so that it is not necessary to do an associative lookup after each instruction completion.

I.3 TOMASULO'S ALGORITHM

Tomasulo's algorithm was developed for the IBM 360/91 floating-point unit [TOMA67]. It was subsequently used in a number of IBM 360/370 machines, CDC 6600/7600 machines, PowerPC implementations, and other processors. The algorithm minimizes RAW hazards by tracking when operands for instructions are available. It also minimizes WAW and WAR hazards by the renaming of registers to remove artificial dependencies and the use of buffers, known as reservation stations, that permit data to be temporarily stored, thus eliminating the need to read from the register file once the data are available.

The reservation stations fetch and store instruction operands as soon as they are available. Source operands point to either the register file or to other reservation stations. Each reservation station corresponds to one instruction. Once all source operands are available, the instruction is sent for execution, provided a functional unit is also available. Once execution is complete, the result is buffered at the reservation station. Thus, the functional unit is free to execute another instruction. The reservation station then sends the result to the register file and any other reservation station that is waiting on that result. WAW hazards are handled since only the last instruction (in program order) actually writes to the registers. The other results are buffered in other reservation stations and are eventually sent to any instructions waiting for those results. WAR hazards are handled since reservation stations can get source operands from either the register file or other reservation stations (in other words, from another instruction). In Tomasulo's algorithm, the control logic is distributed among the reservation stations.

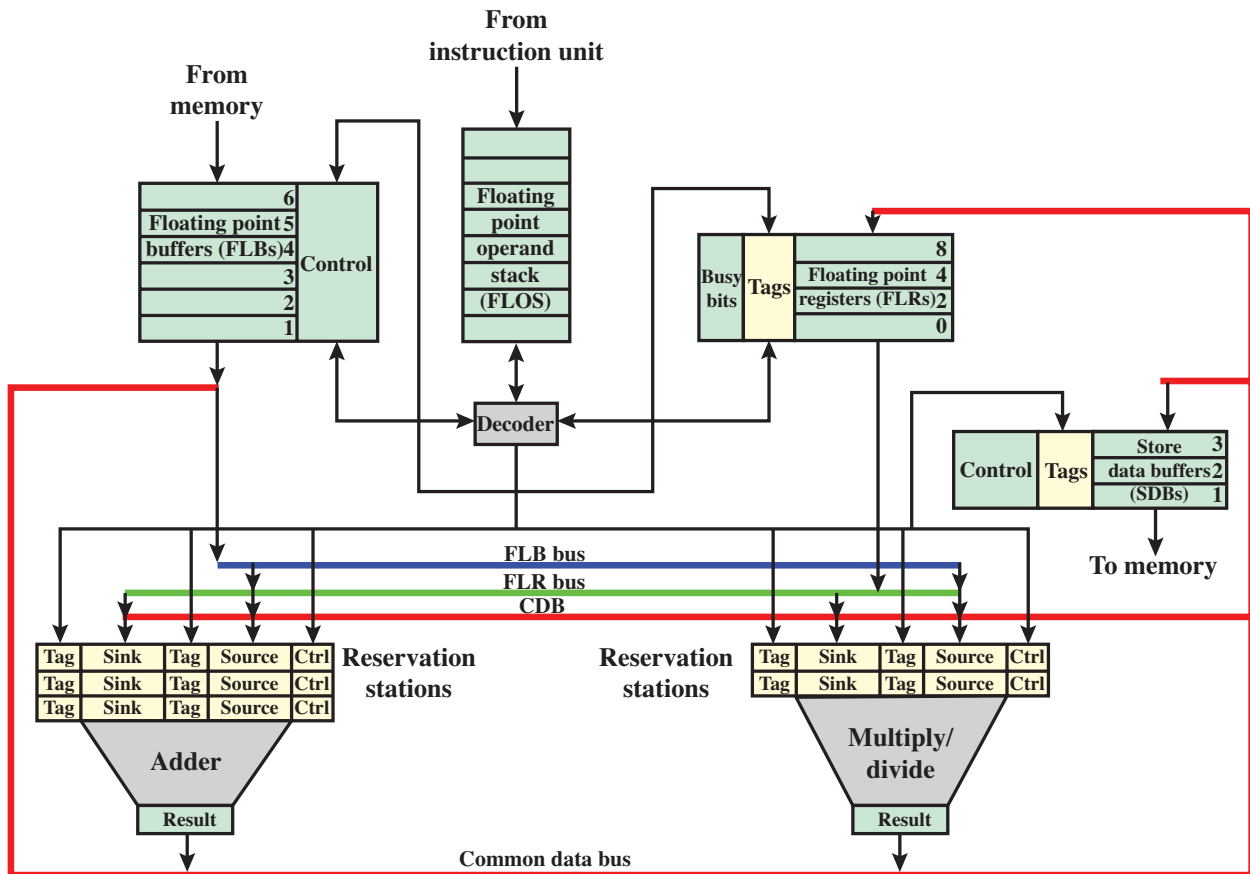


Figure I.10 Basic Structure of a Floating-Point Unit Using Tomasulo's Algorithm

An example of the organization for Tomasulo's algorithm is shown in Figure I.10. This is the organization for the original implementation of the floating point unit of the IBM 360/91.

To get a feel for the algorithm, let us follow an instruction through the stages of a typical Tomasulo processor. Note that each stage here may correspond to more than one actual pipeline stage:

- **Fetch:** Just like in a simple scalar processor, instructions are fetched from instruction memory. Unlike a simple scalar processor, instructions are likely to be fetched in batches of two or more at a time.

- **Issue:** The processor determines which reservation station to issue the instruction to, based on what type of functional unit it requires, and the availability of space in the reservation stations. Instructions can be issued to reservation stations regardless of whether or not their operands are available. If an operand is not available to be read from the register file immediately, this must be because the value associated with that register has not yet been calculated. If this is the case, then the operand will be updated with the result of an instruction that has already been issued, which instruction is therefore already assigned to a reservation station. Hence, as the issue unit issues the instruction to its reservation station, it renames any references to outstanding registers with an identifier tag which indicates which functional unit will produce the result, and which virtual register identifier the result will be identified as. The issue unit also renames any result registers associated with the new instruction to a virtual register identifier so that it can tell subsequent instructions where to find the results of this instruction.
- **Execute:** The execution stage of a processor implementing Tomasulo's algorithm consists of a number of functional units, each with their own reservation station. The reservation station holds a small number of instructions that have been issued, but cannot yet be executed. When an instruction becomes ready to be executed because of operand values becoming available (by being broadcast on the common data bus), and the functional unit is ready to accept a new instruction, the reservation station passes the instruction to the functional unit, where the instruction's real execution takes place: arithmetic operations are calculated, memory is accessed.
- **Writeback:** The final stage an instruction goes through is writeback. This is similar in many ways to a simple pipelined machine: when the result of an instruction has been calculated, the value is driven on one

of a number of common data buses, to be sent to the register file. This bus is also monitored by other parts of the machine so that the value may be used immediately by waiting instructions.

To get a feel for the algorithm, we present an example for a typical processor organization using Tomasulo's algorithm, taken from [SHEN05]. We assume that each instruction includes references to three floating-point registers as operands. Up to two instructions can be dispatched (in program order) at a time. We also assume that an instruction can begin execution in the same cycle that it dispatched to a reservation station. Latencies for floating point add and multiply operations are two and three cycles per instruction, respectively. An instruction can forward its result to dependent instructions during its last execution cycle, and a dependent instruction can begin execution in the next cycle. Tag values 1, 2, and 3 are used to identify the three reservation stations of the adder functional unit, while values 4 and 5 are used to identify the two reservation stations of the multiply/divide functional unit.

The example sequence consists of the following program fragment:

```
w:  R4 ← R0 + R8
x:  R2 ← R0 × R4
y:  R4 ← R4 + R8
z:  R8 ← R4 × R2
```

The initial values of registers R0, R2, R4, R8 are 6.0, 3.5, 10.0, and 7.8, respectively. Figure I.11 illustrates six cycles of execution of the program.

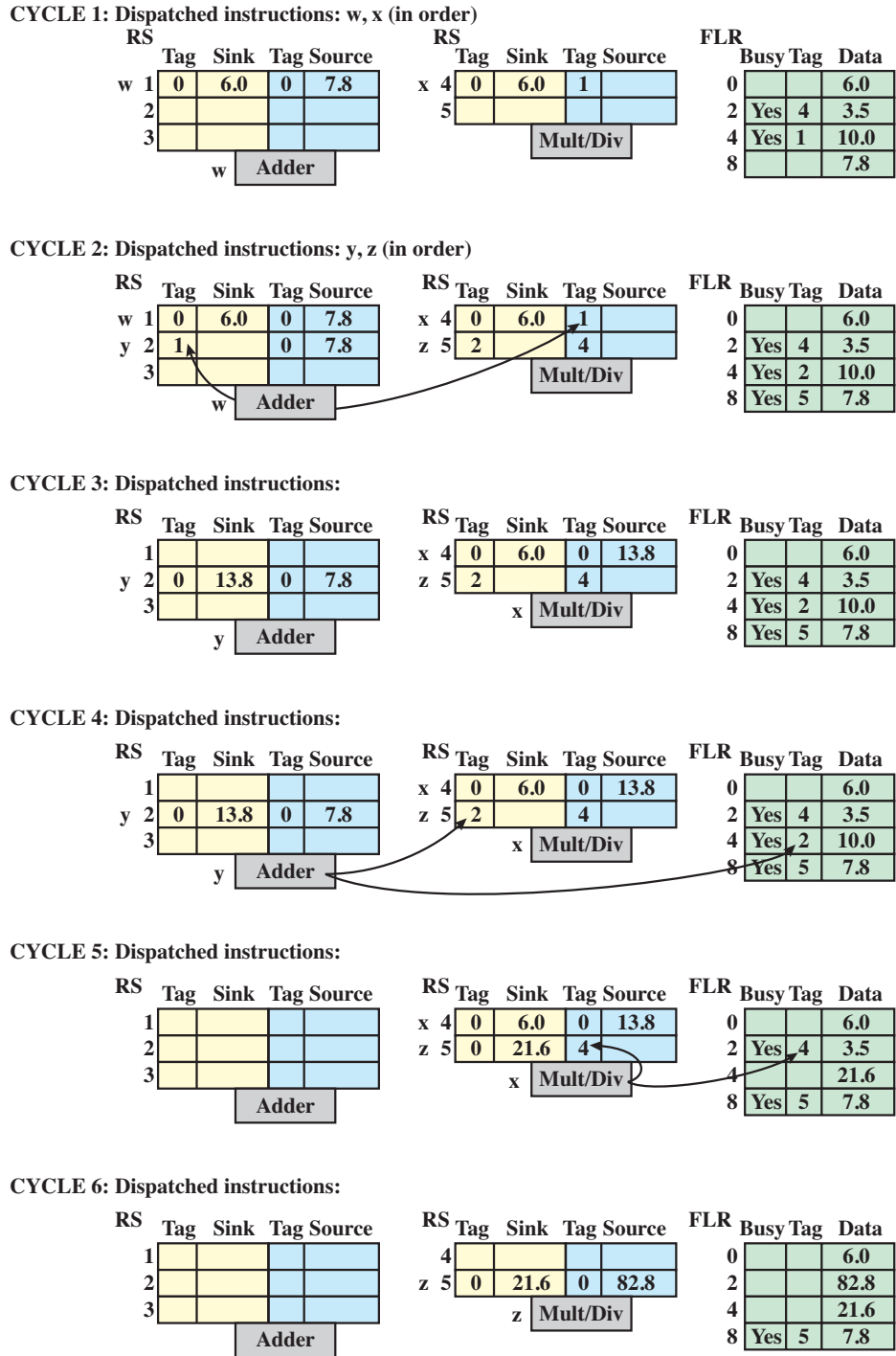


Figure I.11 Illustration of Tomasulo's Algorithm on an Example Instruction Sequence

- **Cycle 1:** Instructions w and x are dispatched (in order) to reservation stations (RSs) 1 and 4, respectively. The busy bits of the destination

registers for these instructions (R4 for w, R2 for x) are set in the floating point register (FLR) file. The tag field for R2 is set to 4, indicating that the instruction in RS 4 will produce the result for updating R2. Similarly, the tag field for R4 is set to 1, indicating that the instruction in RS a will produce the result for updating R4. Note that in the RS entries, one of the two source operands is placed in the sink field. This serves as a temporary storage location. The other operand will be combined with the value in the sink field to produce the destination operand value. Both operands for instruction w are available, so it begins execution immediately (indicated by the w next to the Adder function). Instruction x requires the result (R4) of instruction w, so it can not yet execute. Instead, the tag bit for the source operand is set to 1, referring to entry 1 in the RS file. This tag indicates that the instruction in RS 1 will produce the needed source operand.

- **Cycle 2:** Instructions y and z are dispatched (in order) to RSs 2 and 5, respectively. The busy bit for the destination register of z (R8) is set in the FLR file. Instruction y needs R8 as a source register. Because the busy bit is set for this R8, indicating that the value in the FLR is no longer valid, the tag for R4 is picked up and placed in the RS entry for that source register. Furthermore, because R4 will be updated by y, the tag for R4 is updated to the value 2, indicating that the value for R4 in the FLR file will not be valid until the completion of instruction y. Similarly, when z is dispatched, it is placed in RS entry 5. Instruction z needs R4 and R8 as sources and so the tags for these registers are put in the RS entry for z. Also, the adder function for w completes at the end of cycle 2, and the result will be fed to the two RS entries with a tag value of 1. These RS entries will pick up the result via the common data bus (CBD). This value (13.8) will show up at the beginning of cycle 3.

Instruction z will update R8, so its tag value (5) is entered in the FLR file entry for R8.

- **Cycle 3:** At the beginning of this cycle, the appropriate value for R4 (13.8) is placed in the instruction entries for y and x. Instruction y begins execution in the adder unit and instruction x begins execution in the multiply/divide unit.
- **Cycle 4:** Instruction y completes at the end of this cycle, and broadcasts its result (21.6). This result is recorded in the RS entry for instruction z and in R4 itself in the FLR. Note that this result is posted to R4 even though the preceding instruction (x) has not completed. This out-of-order completion is possible because, in effect, instructions work with renamed registers, with the register name consisting of the original register name plus the associated tag value at the time the instruction is dispatched.
- **Cycle 5:** Instruction x completes at the end of this cycle and broadcasts its result (82.8). This result is recorded in the RS entry for instruction z and the FLR entry for R2.
- **Cycle 6:** Instruction z begins execution. R8 is thus not valid until z completes.

I.4 SCOREBOARDING

Scoreboarding is hardware-based dynamic instruction scheduling technique that is used as an alternative to register renaming to achieve pipeline efficiency. In essence, scoreboarding is a bookkeeping technique that allows instructions to execute whenever they are not dependent on previous instructions and no structural hazards are present. This technique, also known as Thornton's algorithm, was initially developed for the CDC 6600 [THORN64, THORN80]. Variations on the scoreboarding technique have been implemented on a number of machines.

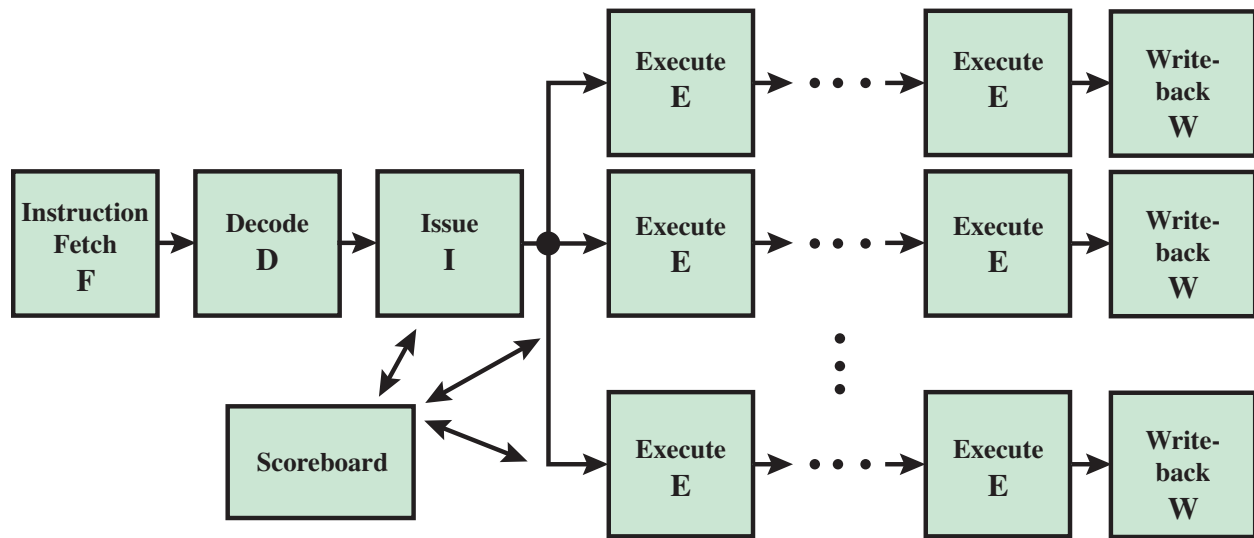


Figure I.12 Block Diagram of a CDC 6600-Style Processor

Figure I.12 is a block diagram that illustrates the role of the scoreboard in a processor organization. We assume a single instruction pipeline for the fetch, decode, and instruction issue stages. The remainder of the pipeline is a superscalar architecture with parallel pipelines for different functional units and/or replicated functional units, with each parallel pipeline including one or more execute stages and a write backstage. The scoreboard is a central unit that exchanges information with the issue stage and the execute stages. It also interacts with the register file.

Scoreboard Operation

The scoreboard can be viewed as consisting of two tables, one with an entry for each functional unit, and one with an entry for each register. Each functional unit entry indicates whether there is an instruction pending for the functional unit, and if so which instruction is pending. The entry also indicates which registers are reserved by this functional unit for input, and whether the registers are currently available for this instruction. Each

register entry tells which functional unit, if any, has this register reserved for output. As each new instruction is brought up, the conditions at the instant of issuance are set in the scoreboard. If no waiting is required, the execution of the instruction is begun immediately under control of the functional unit. If waiting is required (for example, an input operand may not yet be available in the register file), the scoreboard controls the delay, and when ready, allows the unit to begin its execution.

From the point of view of an instruction, execution proceeds as follows [THOR70]:

- 1.** Check availability of functional unit and result register. If either is already reserved, the instruction is not issued until the reserved resource becomes available. This is a resource hazard or a WAW hazard, depending on where the conflict occurred. New reservations get stalled. An example of a functional unit conflict:

$$R6 \leftarrow R1 + R2$$
$$R5 \leftarrow R3 + R4$$

Both instructions use the Add functional unit, a situation in which the second instruction must wait for the first to complete that functional unit. However, if there are multiple Add units, the instructions may proceed in parallel. An example of a result register conflict;

$$R6 \leftarrow R1 + R2$$
$$R6 \leftarrow R4 \times R5$$

Both instructions call for register R6 for the result, thus the second instruction must wait for the first to be completed.

- 2.** Enter reservations for functional unit and result register. If one or both source registers is reserved, the instruction cannot be issued, but the machine can keep entering reservations. This is a RAW hazard. For example:

$$R6 \leftarrow R1 / R2$$

$R7 \leftarrow R5 + R6$ (conflict on this instruction)

$R3 \leftarrow R2 \times R4$ (this instruction free to execute)

The second instruction is issued but held in the Divide unit until R6 is ready.

3. When source registers contain valid data, read the data and issue the instruction to the functional unit. The functional unit now executes the instruction under local control.
4. When the functional unit has completed the instruction, it checks to see if it can write its output to its result register (this is impossible if the register is reserved as a source by another functional unit, and that functional unit already has it marked as available, a WAR hazard).

For example:

$R3 \leftarrow R1 / R2$

$R5 \leftarrow R4 \times R3$ (RAW conflict on R3)

$R4 \leftarrow R0 + R6$ (WAR conflict on R4)

In this example, the WAR conflict on R4 is the direct result of a RAW conflict on R3. Because the instructions are issued on consecutive cycles and because the Add function is much faster than the Divide or Multiply, the addition is accomplished and ready for entry in the result register R4 in advance of the start of Multiply. The RAW conflict on R3 causes the Multiply to hold until that input operand is ready. This holds up the entry of R4 into the Multiply unit also. The WAR conflicts are resolved by holding the result in the functional unit until the register is available.

Scoreboard control thus directs the functional unit in starting, obtaining its operands, and storing its results. Each unit, once started, proceeds independently until just before the result is produced. The unit then sends a signal to the scoreboard requesting permission to release its results to the result register. The scoreboard determines when the path to the result

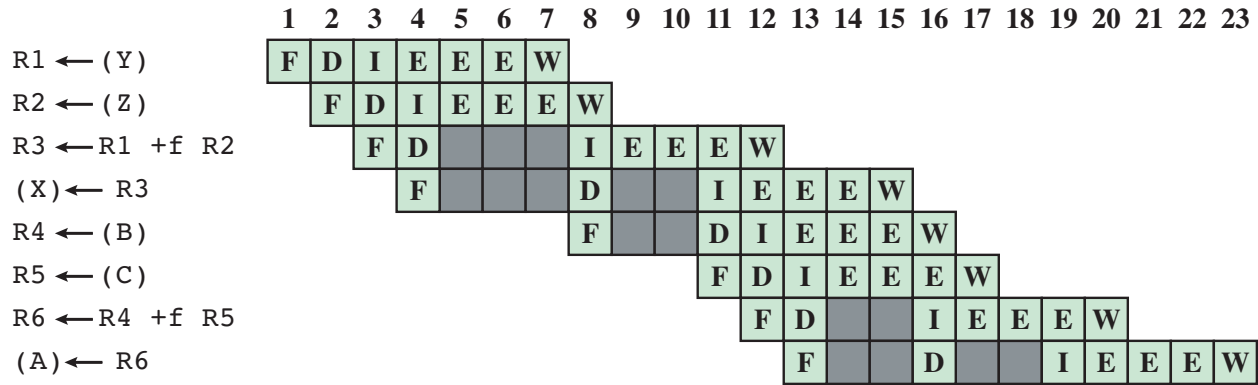
register is clear and signals the requesting unit to release its result. The releasing unit's reservations are then cleared, and all units waiting for the result are signaled to read the result for their respective computations.

Scoreboard Example

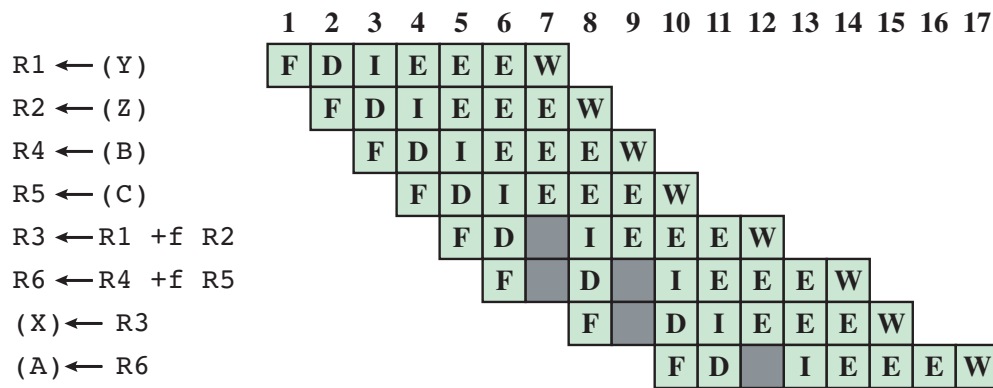
We now present an example adapted from [SMIT89]. Assume a pipeline organization with three execute stages. Consider the following program fragment:

R1 ← (Y)	Load register R1 from memory location Y
R2 ← (Z)	Load register R2 from memory location Z
R3 ← R1 +f R2	Floating add R1 and R2; store in R3
(X) ← R3	Store result in memory location X
R4 ← (B)	Load register R4 from memory location B
R5 ← (C)	Load register R5 from memory location C
R6 ← R4 *f R5	Floating multiply R4 and R5; store in R6
(A) ← R6	Store result in memory location A

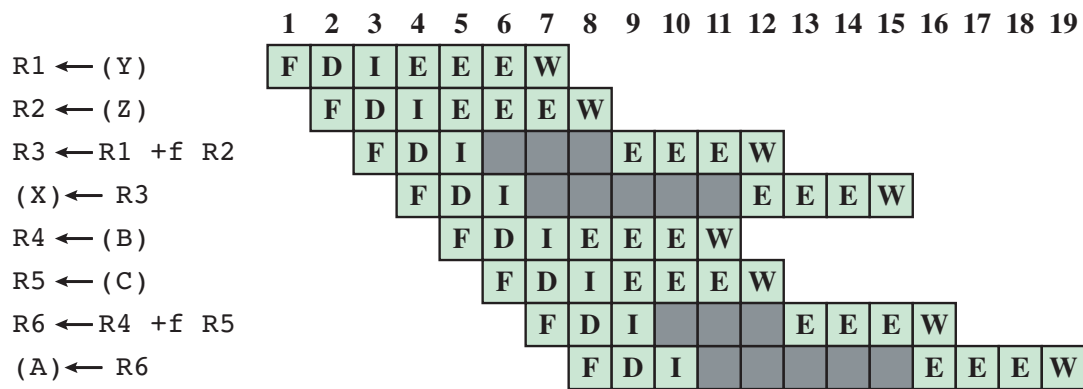
Figure I.13a shows a straightforward pipeline implementation that assumes no parallel execution units and no attempts to circumvent hazards. The diagram assumes that load and store instructions take three execution cycles, as do multiply and add instructions. Figure I.13b shows the result of instruction scheduling at compile time. With instructions appropriately reordered, six clock cycles are saved. Finally, Figure I.13c shows the result of using the hardware scoreboard rather than the compiler to improve performance. In this example, 4 cycles are saved compared to the original pipeline.



(a) In-order instruction issuing



(b) Reorder instruction issuing



(c) Hardware scoreboard for dynamic instruction scheduling

Figure I.13 Pipelined Execution Example

References

- DAVI71** Davidson, E. "The Design and Control of Pipelined Function Generators," *Proceedings, IEEE Conference on Systems, Networks, and Computers*, January 1971.
- DAVI75** Davidson, E.; Thomas, A.; Shar, L.; and Patel, J. "Effective Control for Pipelined Processors," *IEEE COMPCON*, March 1975.
- HWAN93** Hwang, K. *Advanced Computer Architecture*. New York: McGraw-Hill, 1993.
- PATE76** Patel, J., and Davidson, E. "Improving the Throughput of a Pipeline by Insertion of Delays." *Proceedings of the 3rd Annual Symposium on Computer Architecture*, 1976.
- SHEN05** Shen, J., and Lipasti, M. *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, 2005.
- SMIT88** Smith, J., and Pleszcun, A. "Implementing Precise Interrupts in Pipelined Processors." *IEEE Transactions on Computers*, May 1988.
- SMIT89** Smith, J. "Dynamic Instruction Scheduling and the Astronautics ZS-1." *Computer*, July 1989.
- STON93** Stone, H. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1993.
- THOR64** Thornton, J. "Parallel Operation in the Control Data 6600." *Proceedings of the Fall Joint Computer Conference*, 1964.
- THOR70** Thornton, J. *The Design of a Computer: The Control Data 6600*. Glenview, IL: Scott, Foreman, and Company, 1970.
- THOR80** Thornton, J. "The CDC 6600 Project." *Annals of the History of Computing*, October 1980.
- TOMA67** Tomasulo, R. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." *IBM Journal*, January 1967.