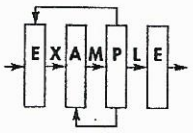
5.2.4 The Classic Tomasulo Algorithm

The design of the IBM 360/91's floating-point unit, incorporating what has come to be known as *Tomasulo's algorithm,* laid the groundwork for modern superscalar processor designs [Tomasulo, 1967]. Key attributes of most contemporary register data flow techniques can be found in the classic Tomasulo algorithm, which deserves an in-depth examination. We first introduce the original design of the floating-point unit of the IBM 360, and then describe in detail the modified design of the FPU in the IBM 360/91 that incorporated Tomasulo's algorithm, and finally illustrate its operation and effectiveness in processing an example code sequence.

The original design of the IBM 360 floating-point unit is shown in Figure 5.20. The FPU contains two functional units: one floating-point add unit and one floating-point multiply/divide unit. There are three register files in the FPU: the floating-point registers (FLRs), the floating-point buffers (FLBs), and the store data buffers (SDBs). There are four FLR registers; these are the architected floating-point registers. Floating-point instructions with storage-register or storage-storage addressing modes are preprocessed. Address generation and memory



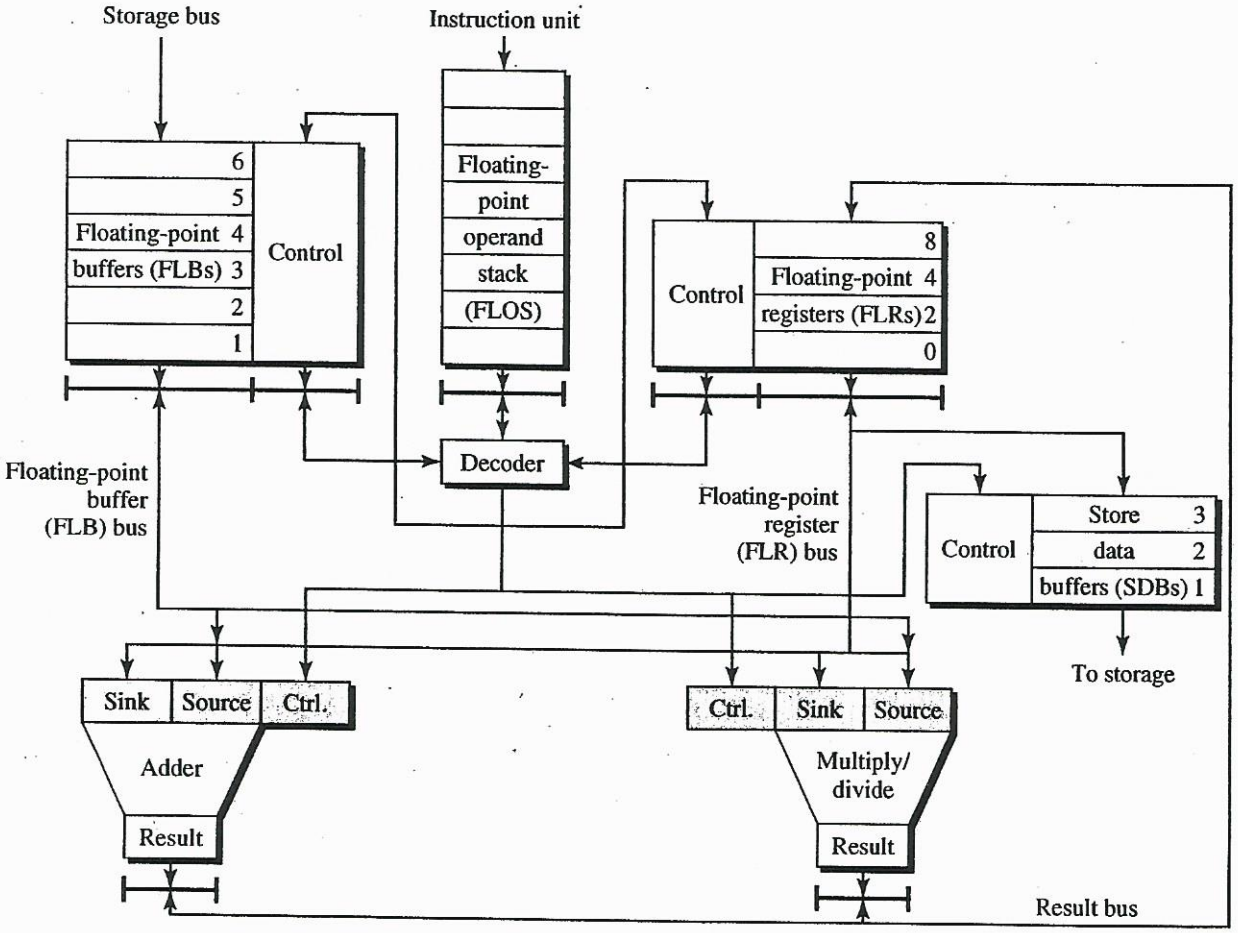


Figure 5.20
The Original Design of the IBM 360 Floating-Point Unit.

accessing are performed outside of the FPU. When the data are retrieved from the memory, they are loaded into one of the six FLB registers. Similarly if the destination of an instruction is a memory location, the result to be stored is placed in one of the three SDB registers and a separate unit accesses the SDBs to complete the storing of the result to a memory location. Using these two additional register files, the FLBs, and the SDBs, to support storage-register and storage-storage instructions, the FPU effectively functions as a register-register machine.

In the IBM 360/91, the instruction unit (IU) decodes all the instructions and passes all floating-point instructions (in order) to the floating-point operation stack (FLOS). In the FPU, floating-point instructions are then further decoded and issued in order from the FLOS to the two functional units. The two functional units are not pipelined and incur multiple-cycle latencies. The adder incurs 2 cycles for add instructions, while the multiply/divide unit incurs 3 cycles and 12 cycles for performing multiply and divide instructions, respectively.

In the mid-1960s, IBM began developing what eventually became Model 91 of the Systems 360 family. One of the goals was to achieve concurrent execution of multiple floating-point instructions and to sustain a throughput of one instruction per cycle in the instruction pipeline. This is quite aggressive considering the complex addressing modes of the 360 ISA and the multicycle latencies of the execution units. The end result is a modified FPU in the 360/91 that incorporated Tomasulo's algorithm; see Figure 5.21.

Tomasulo's algorithm consists of adding three new mechanisms to the original FPU design, namely, reservation stations, the common data bus, and register tags. In the original design, each functional unit has a single buffer on its input side to hold the instruction currently being executed. If a functional unit is busy, issuing of instructions by FLOS will stall whenever the next instruction to be issued requires the same functional unit. To alleviate this structural bottleneck, multiple buffers, called *reservation stations,* are attached to the input side of each functional unit. The adder unit has three reservation stations, while the multiply/divide unit has two. These reservation stations are viewed as virtual functional units; as long as there is a free reservation station, the FLOS can issue an instruction to that functional unit even if it is currently busy executing another instruction. Since the FLOS issues instructions in order, this will prevent unnecessary stalling due to unfortunate ordering of different floating-point instruction types.

With the availability of reservation stations, instructions can also be issued to the functional units by the FLOS even though not all their operands are yet available. These instructions can wait in the reservation station for their operands and only begin execution when they become available. The *common data bus* (CDB) connects the outputs of the two functional units to the reservation stations as well as the FLRs and SDB registers. Results produced by the functional units are broadcast into the CDB. Those instructions in the reservation stations needing the results as their operands will latch in the data from the CDB. Those registers in the FLR and SDB that are the destinations of these results also latch in the same data from the CDB. The CDB facilitates the forwarding of results directly from producer instructions to consumer instructions waiting in the reservation stations

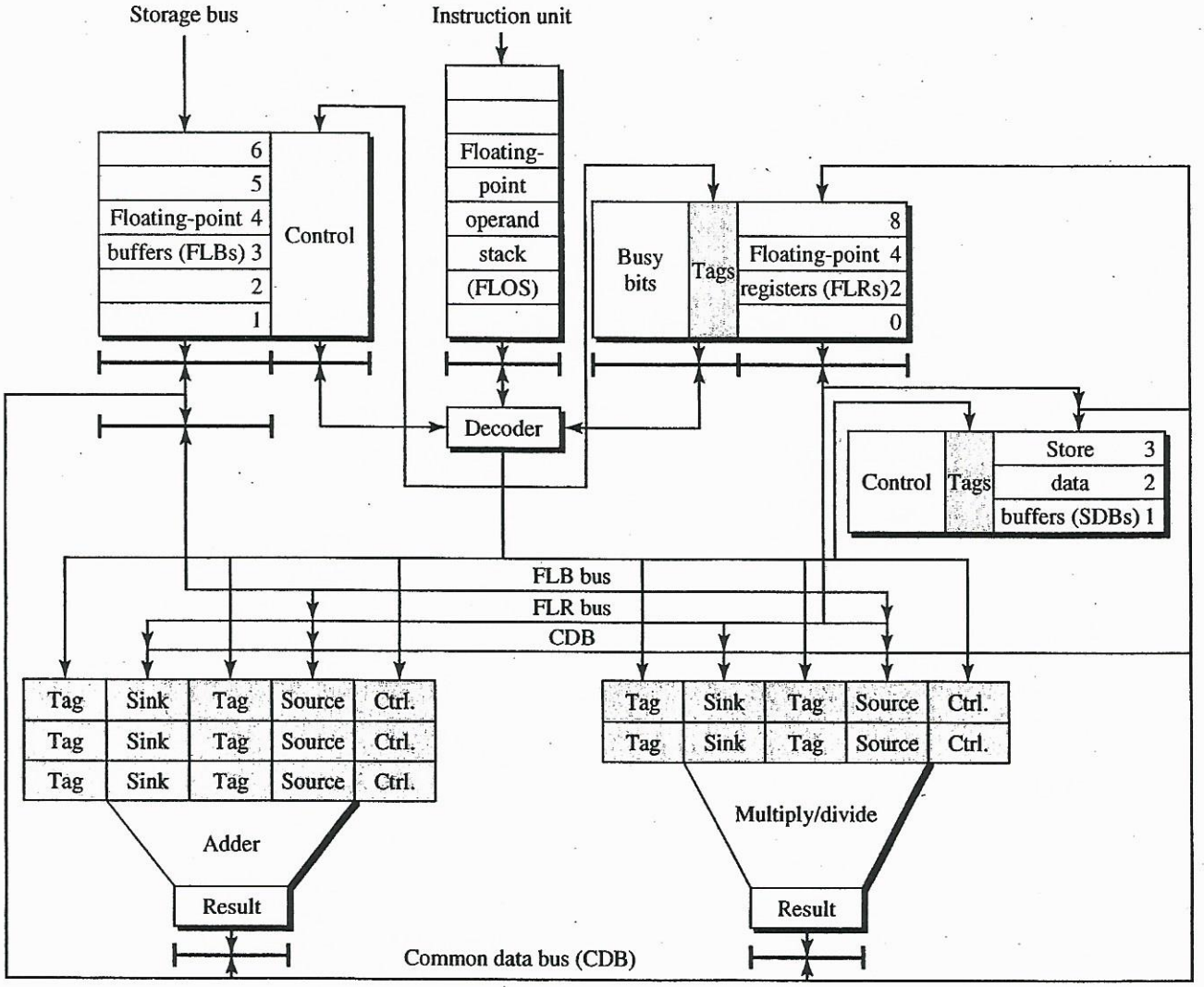
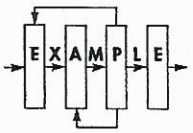


Figure 5.21
The Modified Design of the IBM 360/91 Floating-Point Unit with Tomasulo's Algorithm.



without having to go through the registers. Destination registers are updated simultaneously with the forwarding of results to dependent instructions. If an operand is coming from a memory location, it will be loaded into a FLB register once memory accessing is performed. Hence, the FLB can also output onto the CDB, allowing a waiting instruction in a reservation station to latch in its operand. Consequently, the two functional units and the FLBs can drive data onto the CDB, and the reservation station's FLRs and SDBs can latch in data from the CDB.

When the FLOS is dispatching an instruction to a functional unit, it allocates a reservation station and checks to see if the needed operands are available. If an operand is available in the FLRs, then the content of that register in the FLRs is copied to the reservation station; otherwise a tag is copied to the reservation station instead. The tag indicates where the pending operand is going to come from.

The pending operand can come from a producer instruction currently resident in one of the five reservation stations, or it can come from one of the six FLB registers. To uniquely identify one of these 11 possible sources for a pending operand, a 4-bit tag is required. If one of the two operand fields of a reservation station contains a tag instead of the actual operand, it indicates that this instruction is waiting for a pending operand. When that pending operand becomes available, the producer of that operand drives the tag along with the actual operand onto the CDB.

A waiting instruction in a reservation station uses its tag to monitor the CDB. When it detects a tag match on the CDB, it then latches in the associated operand. Essentially the producer of an operand broadcasts the tag and the operand on the CDB; all consumers of that operand monitor the CDB for that tag, and when the broadcasted tag matches their tag, they then latch in the associated operand from the CDB. Hence, all possible destinations of pending operands must carry a tag field and must monitor the CDB for a tag match. Each reservation station contains two operand fields, each of which must carry a tag field since each of the two operands can be pending. The four FLRs and the three registers in the SDB must also carry tag fields. This is a total of 17 tag fields representing 17 places that can monitor and receive operands; see Figure 5.22. The tag field at each potential consumer site is used in an associative fashion to monitor for possible matching of its content with the tag value being broadcasted on the CDB. When a tag match occurs, the consumer latches in the broadcasted operand.

The IBM 360 floating-point instructions use a two-address instruction format. Two source operands can be specified. The first operand specifier is called the *sink* because it also doubles as the destination specifier. The second operand specifier is called the *source*. Each reservation station has two operand fields, one for the sink and the other for the source. Each operand field is accompanied by a tag field. If an operand field contains real data, then its tag field is set to zero. Otherwise, its tag field identifies the source where the pending operand will be coming from, and is used to monitor the CDB for the availability of the pending operand. Whenever

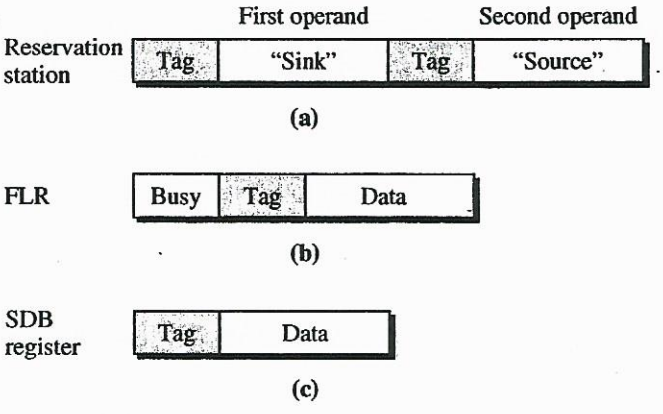


Figure 5.22
The Use of Tag Fields in (a) A Reservation Station, (b) A FLR, and (c) A SDB Register.

an instruction is dispatched by the FLOS to a reservation station, the data in the FLR corresponding to the sink operand are retrieved and copied to the reservation station. At the same time, the "busy" bit associated with this FLR is set, indicating that there is a pending update of that register, and the tag value that identifies the particular reservation station to which the instruction is being dispatched is written into the tag field of the same FLR. This clearly identifies which of the reservation stations will eventually produce the updated data for this FLR. Subsequently if a trailing instruction specifies this register as one of its source operands, when it is dispatched to a reservation station, only the tag field (called the *pseudo-operand*) will be copied to the corresponding tag field in the reservation station and not the actual data. When the busy bit is set, it indicates that the data in the FLR are stale and the tag represents the source from which the real data will come. Other than reservation stations and FLRs, SDB registers can also be destinations of pending operands and hence a tag field is required for each of the three SDB registers.

We now use an example sequence of instructions to illustrate the operation of Tomasulo's algorithm. We deviate from the actual IBM 360/91 design in several ways to help clarify the example. First, instead of the two-address format of the IBM 360 instructions, we will use three-address instructions to avoid potential confusion. The example sequence contains only register-register instructions. To reduce the number of machine cycles we have to trace, we will allow the FLOS to dispatch (in program order) up to two instructions in every cycle. We also assume that an instruction can begin execution in the same cycle that it is dispatched to a reservation station. We keep the same latencies of two and three cycles for add and multiply instructions, respectively. However, we allow an instruction to forward its result to dependent instructions during its last execution cycle, and a dependent instruction can begin execution in the next cycle. The tag values of 1, 2, and 3 are used to identify the three reservation stations of the adder functional unit, while 4 and 5 are used to identify the two reservation stations of the multiply/divide functional unit. These tag values are called the *IDs* of the reservation stations. The example sequence consists of the following four register-register instructions.

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

Figure 5.23 illustrates the first three cycles of execution. In cycle 1, instructions w and x are dispatched (in order) to reservation stations 1 and 4. The destination registers of instructions w and x are R4 and R2 (i.e., FLRs 4 and 2), respectively. The busy bits of these two registers are set. Since instruction w is dispatched to reservation station 1, the tag value of 1 is entered into the tag field of R4, indicating that the instruction in reservation station 1 will produce the result for updating R4. Similarly the tag value of 4 is entered into the tag field of R2. Both source operands of instruction w are available, so it begins execution immediately. Instruction x

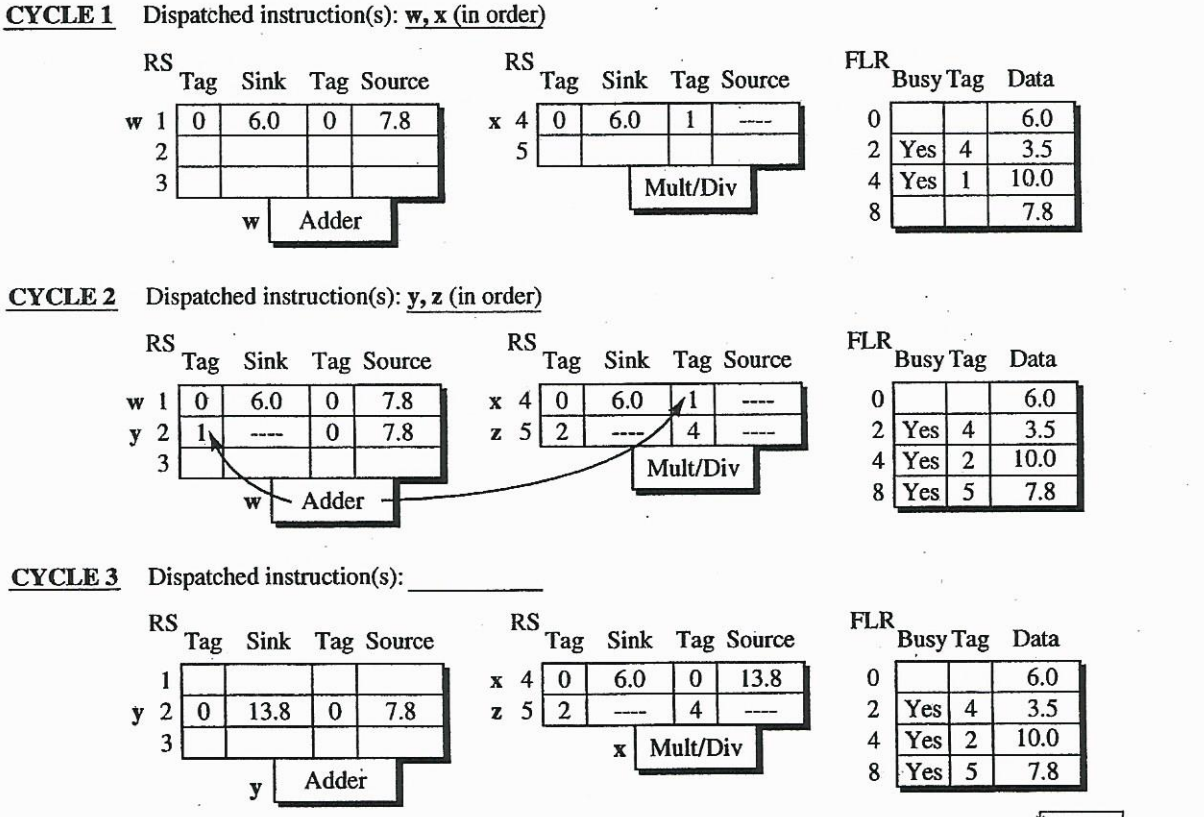
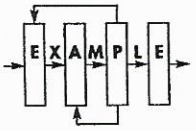


Figure 5.23
Illustration of Tomasulo's Algorithm on an Example Instruction Sequence (Part 1).



requires the result (R4) of instruction w for its second (source) operand. Hence when instruction x is dispatched to reservation station 4, the tag field of the second operand is written the tag value of 1, indicating that the instruction in reservation station 1 will produce the needed operand.

During cycle 2, instructions y and z are dispatched (in order) to reservation stations 2 and 5, respectively. Because it needs the result of instruction w for its first operand, instruction y, when it is dispatched to reservation station 2, receives the tag value of 1 in the tag field of the first operand. Similarly instruction z, dispatched to reservation station 5, receives the tag values of 2 and 4 in its two tag fields, indicating that reservation stations 2 and 4 will eventually produce the two operands it needs. Since R4 is the destination of instruction y, the tag field of R4 is updated with the new tag value of 2, indicating reservation station 2 (i.e., instruction y) is now responsible for the pending update of R4. The busy bit of R4 remains set. The busy bit of R8 is set when instruction z is dispatched to reservation station 5, and the tag field of R8 is set to 5. At the end of cycle 2, instruction w finishes execution and broadcasts its ID (reservation station 1) and its result onto the CDB. All the tag fields containing the tag value of 1 will trigger a tag match and latch in the broadcasted result. The first tag field of reservation station 2 (holding instruction y) and the second tag field of reservation station 4 (holding instruction x)

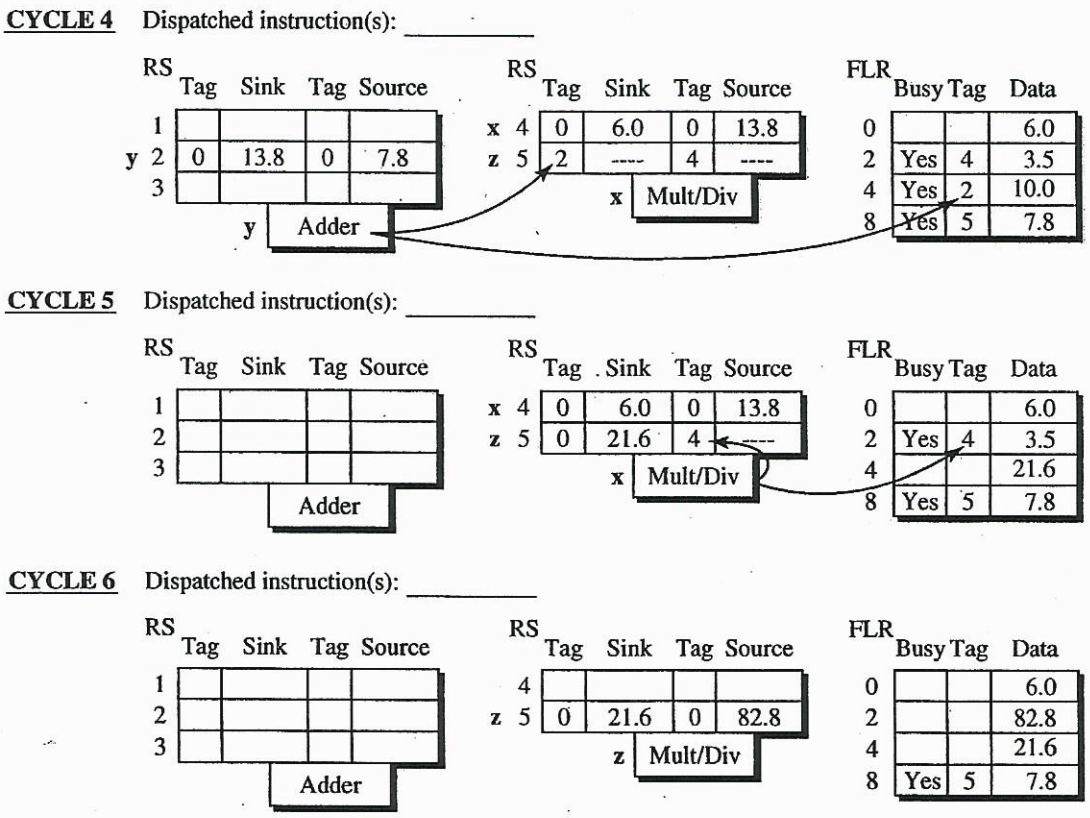
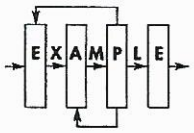


Figure 5.24
Illustration of Tomasulo's Algorithm on an Example Instruction Sequence (Part 2).



have such tag matches. Hence the result of instruction w is forwarded to dependent instructions x and y.

In cycle 3, instruction y begins execution in the adder unit, and instruction x begins execution in the multiply/divide unit. Instruction y finishes execution in cycle 4 (see Figure 5.24) and broadcasts its result on the CDB along with the tag value of 2 (its reservation station ID). The first tag field in reservation station 5 (holding instruction z) and the tag field of R4 have tag matches and pull in the result of instruction y. Instruction x finishes execution in cycle 5 and broadcasts its result on the CDB along with the tag value of 4. The second tag field in reservation station 5 (holding instruction z) and the tag field of R2 have tag matches and pull in the result of instruction x. In cycle 6, instruction z begins execution and finishes in cycle 8.

Figure 5.25(a) illustrates the data flow graph of this example sequence of four instructions. The four solid arcs represent the four true data dependences, while the other three arcs represent the anti- and output dependences. Instructions are dispatched in program order. Anti-dependences are resolved by copying an operand at dispatch time to the reservation station. Hence, it is not possible for a trailing instruction to overwrite a register before an earlier instruction has a chance to read that register. If the operand is still pending, the dispatched instruction will receive

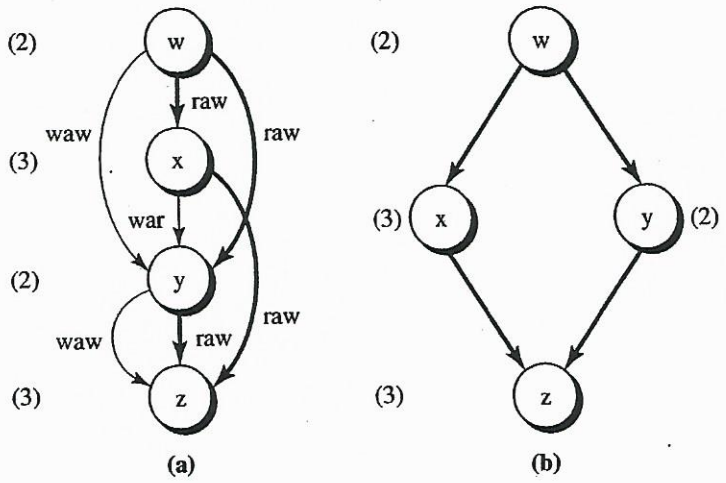
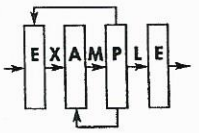


Figure 5.25
Data Flow Graphs of the Example Instruction Sequence: (a) All Data Dependences; (b) True Data Dependences.



the tag for that operand. When that operand becomes available, the instruction will receive that operand via a tag match in its reservation station.

As an instruction is dispatched, the tag field of its destination register is written with the reservation station ID of that instruction. When a subsequent instruction with the same destination register is dispatched, the same tag field will be updated with the reservation station ID of this new instruction. The tag field of a register always contains the reservation station ID of the latest updating instruction. If there are multiple instructions in flight that have the same destination register, only the latest instruction will be able to update that register. Output dependences are implicitly resolved by making it impossible for an earlier instruction to update a register after a later instruction has updated the same register. This does introduce the problem of not being able to support precise exception since the register file does not necessarily evolve through all its sequential states; i.e., a register can potentially miss an intermediate update. For example, in Figure 5.23, at the end of cycle 2, instruction w should have updated its destination register R4. However, instruction y has the same destination register, and when it was dispatched earlier in that cycle, the tag field of R4 was changed from 1 to 2 anticipating the update of R4 by instruction y. At the end of cycle 2 when instruction w broadcasts its tag value of 1, the tag field of R4 fails to trigger a tag match and does not pull in the result of instruction w. Eventually R4 will be updated by instruction y. However, if an exception is triggered by instruction x, precise exception will be impossible since the register file does not evolve through all its sequential states.

Tomasulo's algorithm resolves anti- and output dependences via a form of register renaming. Each definition of an FLR triggers the renaming of that register to a register tag. This tag is taken from the ID of the reservation station containing the instruction that redefines that register. This effectively removes false dependences from causing pipeline stalls. Hence, the data flow limit is strictly determined by the true data dependences. Figure 5.25(b) depicts the data flow graph involving only

true data dependences. As shown in Figure 5.25(a) if all four instructions were required to execute sequentially to enforce all the data dependences, including anti- and output dependences, the total latency required for executing this sequence of instructions would be 10 cycles, given the latencies of 2 and 3 cycles for addition and multiplication instructions, respectively. When only true dependences are considered, Figure 5.25(b) reveals that the critical path is only 8 cycles, i.e., the path involving instructions w, x, and z. Hence, the data flow limit for this sequence of four instructions is 8 cycles. This limit is achieved by Tomasulo's algorithm as demonstrated in Figures 5.23 and 5.24.

5.2.5 Dynamic Execution Core

Most current state-of-the-art superscalar microprocessors consist of an out-of-order execution core sandwiched between an in-order front end, which fetches and dispatches instructions in program order, and an in-order back end, which completes and retires instructions also in program order. The out-of-order execution core (also referred to as the *dynamic execution* core), resembling a refinement of Tomasulo's algorithm, can be viewed as an embedded data flow, or *micro-dataflow,* engine that attempts to approach the data flow limit in instruction execution. The operation of such a dynamic execution core can be described according to the three phases in the pipeline, namely, *instruction dispatching, instruction execution,* and *instruction completion;* see Figure 5.26.

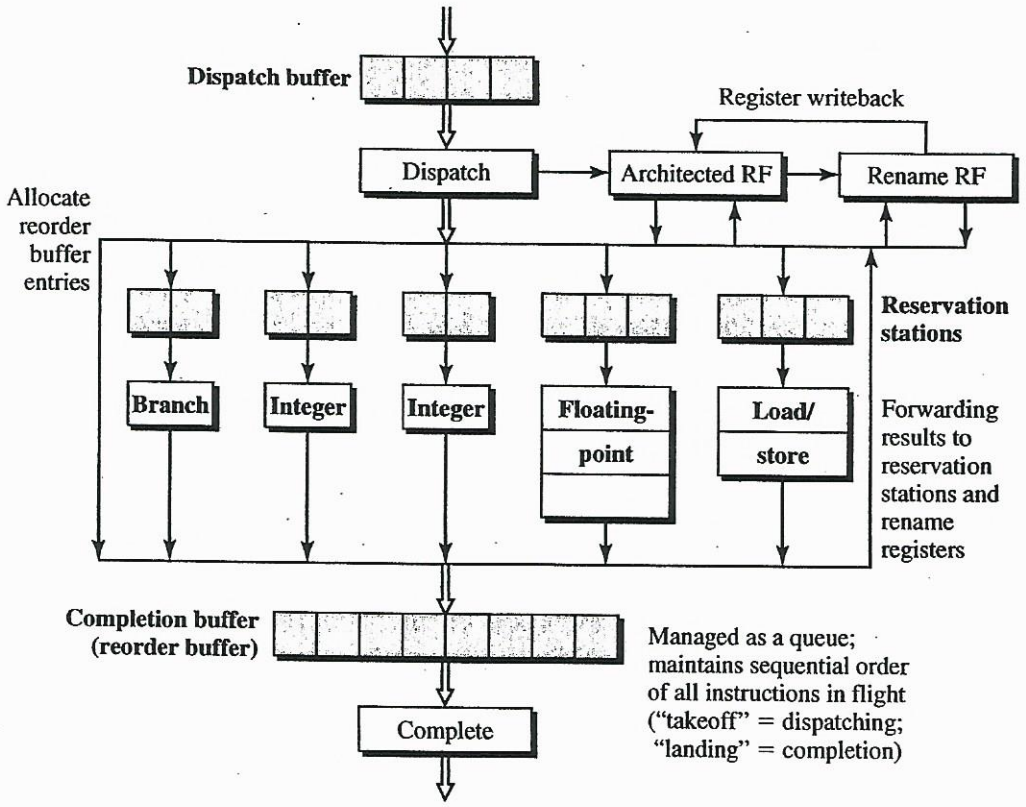


Figure 5.26
Micro-Dataflow Engine for Dynamic Execution.