Optimizing Instruction Execution in the PowerPC™ 603e Superscalar Microprocessor

by

Top Changwatchai
Skipper Smith
Nasr Ullah

Motorola RISC Microprocessor Division

System Performance Modeling and Simulation

Document 1998-014-1
Version 1.0
WTC, SS, NU

Outline

- Superscalar Microprocessor Architectures
- The PowerPC 603e Superscalar Microprocessor
- Stall Conditions
  - Dispatch and Completion Stalls
  - Execution Unit Stalls
  - Load and Store Stalls
  - Instruction Interaction Stalls
- Summary

This presentation discusses techniques for optimizing instruction execution in a superscalar microprocessor architecture such as the PowerPC™ 603e microprocessor.

Instruction execution in a superscalar processor is enhanced by allowing the parallel execution of multiple instructions. In order to enable the maximum potential of most superscalar processors, one needs to be aware of their instruction flow and execution mechanisms. Optimal performance in a microprocessor can be attained by ensuring a continuous flow of instructions through the instruction pipeline.

Being aware of the dependencies and constraints of the instruction flow mechanisms allows one to generate code that can most effectively and optimally take advantage of all the capabilities of a superscalar processor such as the PowerPC 603e microprocessor.

The 603e is a low-power implementation of the PowerPC family of reduced instruction set computer (RISC) microprocessors. The 603e is a superscalar processor capable of issuing and retiring as many as three instructions per clock. Instructions can “execute” out-of-order for increased performance, but they “retire” in-order to ensure functional correctness and well-ordered behavior.

In this paper, we first discuss the instruction flow mechanism of the PowerPC 603e microprocessor and then describe dependencies and constraints that should be avoided to reduce stalls in the instruction pipeline and maximize performance.

By closely examining the instruction flow mechanism of the 603e, a software developer will not only be able to optimize code for the 603e, but will also be able to understand some of the general principles behind superscalar microprocessors that can impact performance.

Common Superscalar Characteristics

- **Multiple instruction dispatch**
  - **ability to fetch and dispatch more than one instruction at a time**
- **Multiple functional units**
  - **ability to execute in parallel more than one instruction**
  - **out of order execution**
- **Multiple instruction retirement**
  - **Multiple instruction completion of the instructions**
- **Multiple read/write ports to register file set**
- **Mechanisms to avoid false dependencies**

There are many characteristics found in common among contemporary superscalar processors. One characteristic is the ability to execute multiple instructions in parallel. To enable this, superscalar processors contain multiple functional units, and they allow the fetching, issuing (dispatching), and retiring of multiple instructions in one clock cycle.

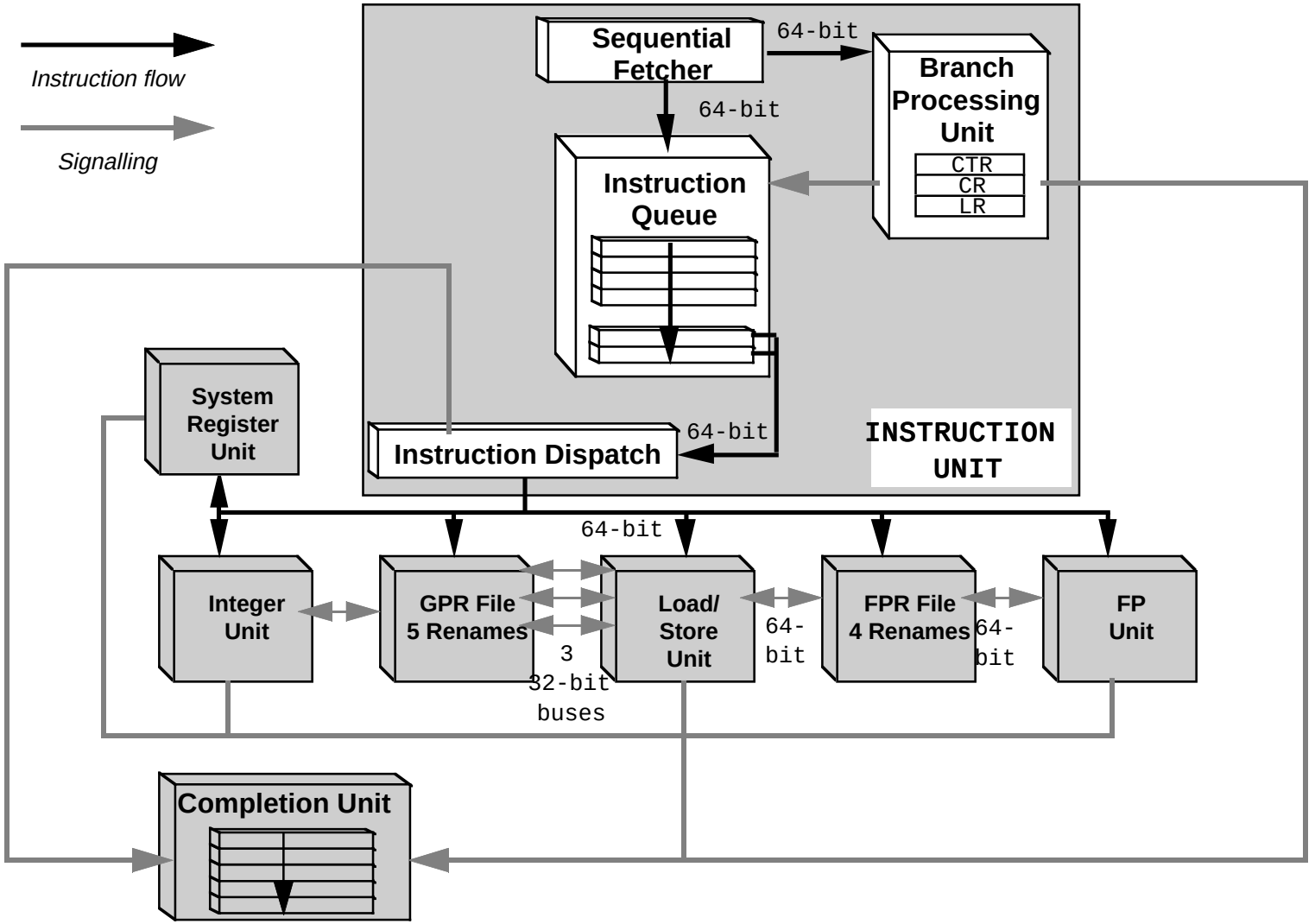
Superscalar processors typically contain a register set with multiple read/write ports to allow multiple instructions in execution to access data simultaneously. Most superscalar processors also have separate floating-point and integer register sets.

The key focus of superscalar processors is to increase overall instruction throughput by keeping the instruction pipeline free of stalls. Mechanisms exist to avoid false dependencies between instructions; instructions should be allowed to dispatch and execute until they are forced to stall due to change in instruction flow, lack of resources, or true data dependencies.

To allow the free flow of execution, most superscalar processors allow out-of-order execution of instructions. However, a mechanism must exist for bringing the instructions back in program order when they complete executing.

The primary reasons the instruction flow can stall within a superscalar processor are changes in instruction flow, resource constraints, and data dependencies. By understanding the flow mechanism, and being aware of the situations that can cause stalls, one can write code that avoids these situations and thereby executes faster.

Block Diagram of the 603e



Like most other superscalar processors, the 603e features pipelined execution flow, in which the processing of an instruction is split into discrete stages. Each stage is able to handle a different instruction, allowing multiple instructions to be in execution at once. For example, it may take three cycles for a floating-point instruction to complete (three-cycle latency), but if there are no stalls in the floating-point pipeline, then a series of floating-point instructions can have a throughput of one instruction per cycle.

The 603e processor core consists of a fetcher, a dispatcher, and five execution units: an integer unit (IU), a floating-point unit (FPU), a branch processing unit (BPU), a load/store unit (LSU), and a system register unit (SRU). An instruction queue (IQ) holds up to six instructions that are fetched in and waiting for dispatch. A completion queue (CQ) holds up to five instructions that have dispatched and are waiting to be finished and retired.

Registers and Execution Units

- **Execution Units**
  - **Integer Unit**
  - **System Register Unit**
  - **Floating-Point Unit**
  - **Branch Unit**
  - **Load/Store Unit**
- **Registers**
  - **Integer Registers**
  - **Floating Point Registers**
- **Execution Unit/Register Interaction**
  - **Rename Registers**
  - **Data Forwarding**

Execution units

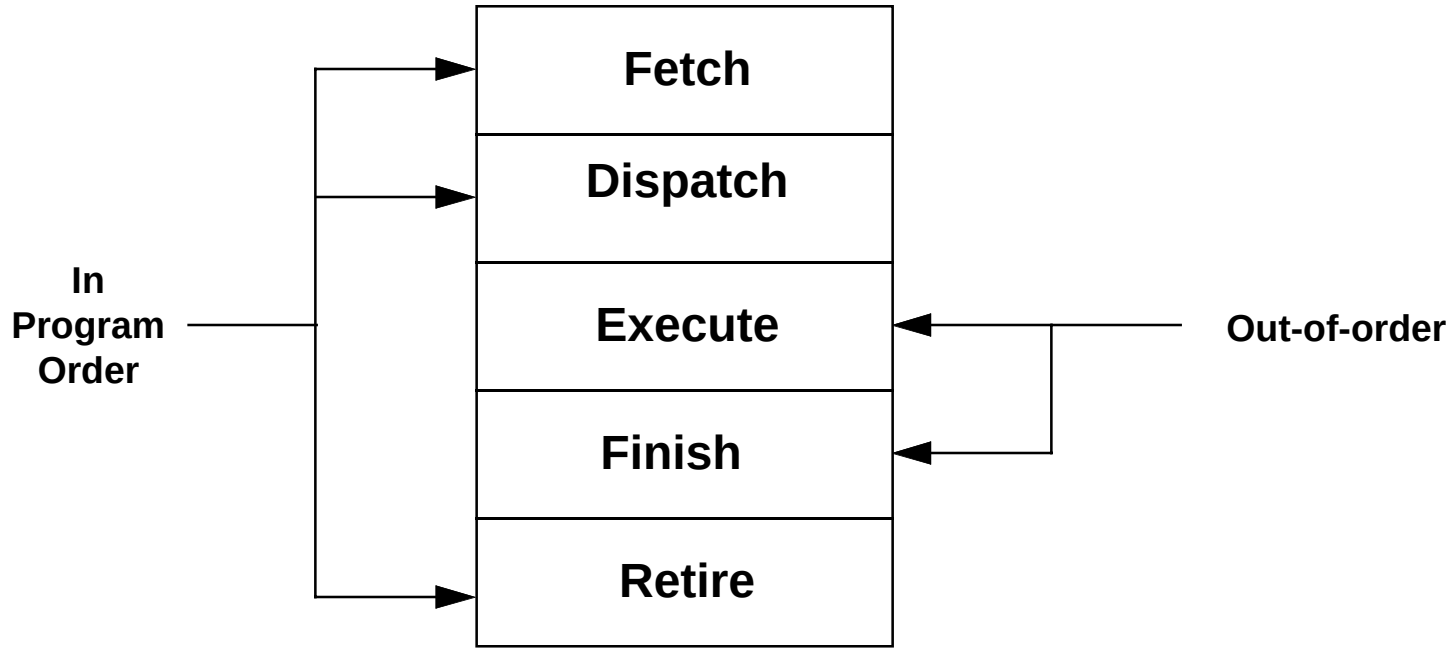
- The Integer Unit accepts all integer instructions.
- The System Register Unit accepts all synchronizing, condition register, and system register instructions. Since these instructions appear infrequently, the SRU also accepts basic add and compare instructions.
- The Floating-Point Unit accepts all instructions utilizing the FP registers (other than loads and stores).
- The Branch Processing Unit redirects instruction fetches, performs prediction and helps control speculative execution, and folds appropriate branches out of the pipeline to permit an effective branch cycle time of zero.
- The Load/Store Unit accepts instructions accessing data cache and memory.

Registers

The 603e supports 32 general-purpose registers (GPRs) that are 32 bits wide, and 32 floating-point registers (FPRs) that are 64 bits wide. These two register files are supported by rename registers which allow quick forwarding of data, in order to reduce stalls based on data dependencies. There are five GPR rename registers and four FPR rename registers.

As an example of usage, suppose we have an integer divide instruction which is computing the value of a given GPR, followed by a store instruction which must store this value to memory. When the divide is dispatched to the IU, the GPR is assigned a GPR rename register. The store is then dispatched to the LSU and must wait for the value to become valid. When the value is computed, the store immediately gets the value through the GPR rename bus, and can begin storing the value at the same time it is being written back to the GPR file.

Instruction Pipeline



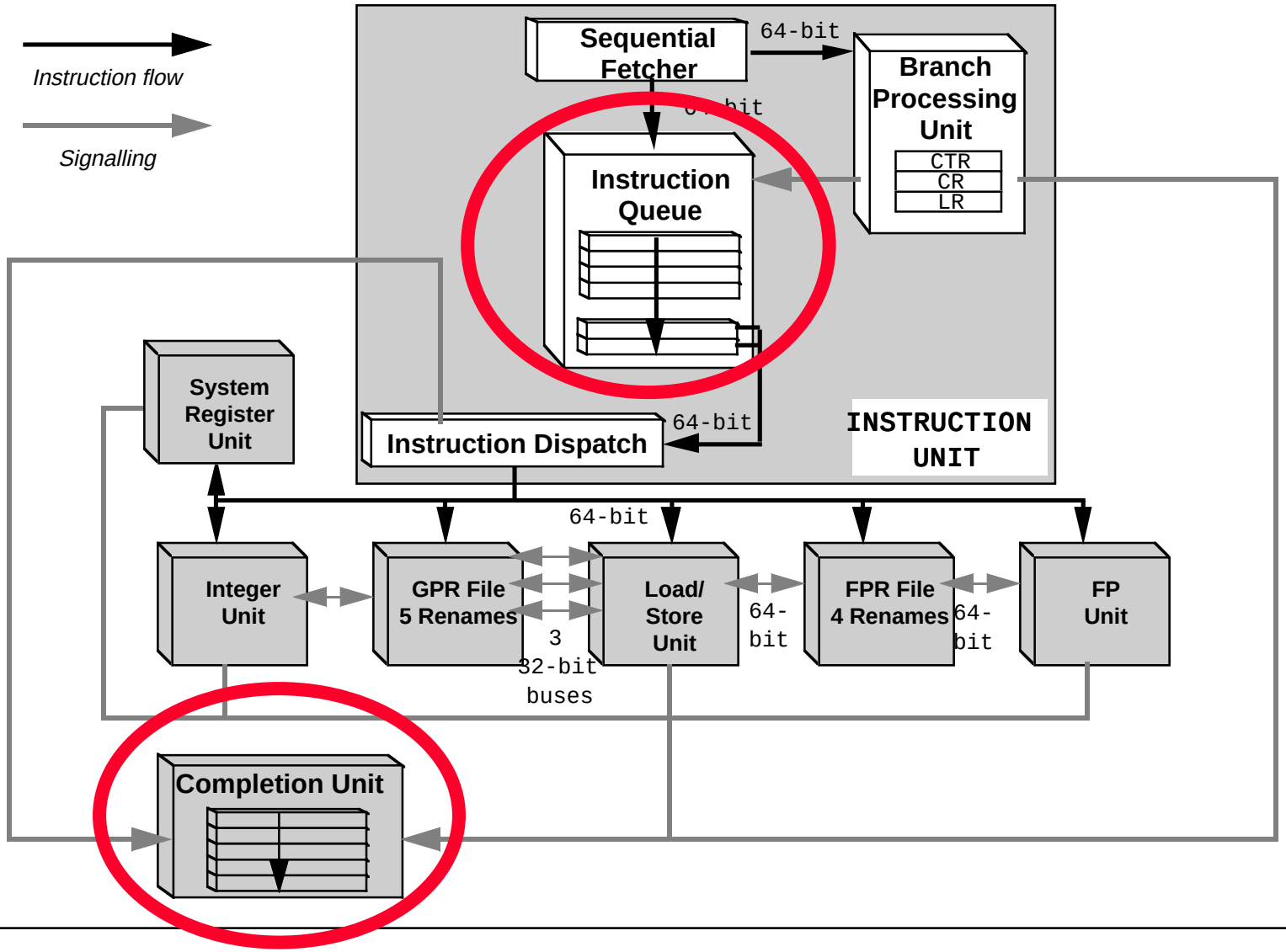
The **fetcher** fetches up to two instructions per clock into the IQ, where they appear in the lowest available slots. The **dispatcher** then dispatches up to two instructions from the IQ to the execution units (excluding the BPU, which scans instructions as they are brought in by the fetcher). The dispatcher also performs source and destination dependency checking and determines dispatch serializations. Each instruction dispatched has an entry created for it in the completion queue.

When an execution unit has finished processing an instruction, it signals the completion unit, and the instruction's entry in the completion queue is marked "finished." Up to two finished instructions per clock may be retired (removed) from the completion queue. When an instruction is retired, the architectural registers are updated.

Note that fetching, dispatching, and retiring of instructions is done in program order, but executing and finishing can be done out-of-order and in parallel.

In a pipelined architecture, anything that prevents an instruction from moving from one stage of a pipeline to the next is known as a stall. Resource checks must be performed to see if stalls will occur. The rest of this paper discusses how and where stalls can occur in the instruction pipeline.

Dispatch & Completion



The dispatcher and completion unit control the execution of instructions. Interactions between the dispatcher and completion unit and the various execution units can reduce potential stalls in the instruction pipeline. In the next few slides, we discuss these interactions.

The dispatcher is capable of buffering up to 6 instructions (in the instruction queue). However, instructions must dispatch in-order out of the dispatcher and only from the bottom two slots (an exception to this rule occurs with branch folding, discussed later). If the instruction in the bottom slot is not capable of dispatching, then the instruction in the second slot cannot, either. It is the job of the dispatcher to determine whether or not an execution unit is capable of accepting an instruction. The dispatcher will stall instruction dispatch when the instructions awaiting dispatch requires an execution unit that is unavailable, or will stall the second instruction if both instructions awaiting dispatch need the same execution resource.

The completion unit is capable of buffering up to 5 instructions (in the completion queue). The completion unit records the proper order of dispatch to enforce in-order completion. While instructions are being tracked, the completion unit also keeps a record of exceptions generated, speculation, out-of-order finishing, etc. All instructions except folded branches must be tracked in the completion unit. The completion unit assigns rename registers (up to 5 integer and 4 floating point) to the instructions as they dispatch. The completion unit will stall the dispatcher if no appropriate rename register resources are available. Additionally, if there are no slots available in the completion queue, the completion unit will order the dispatcher to stall the dispatching of instructions.

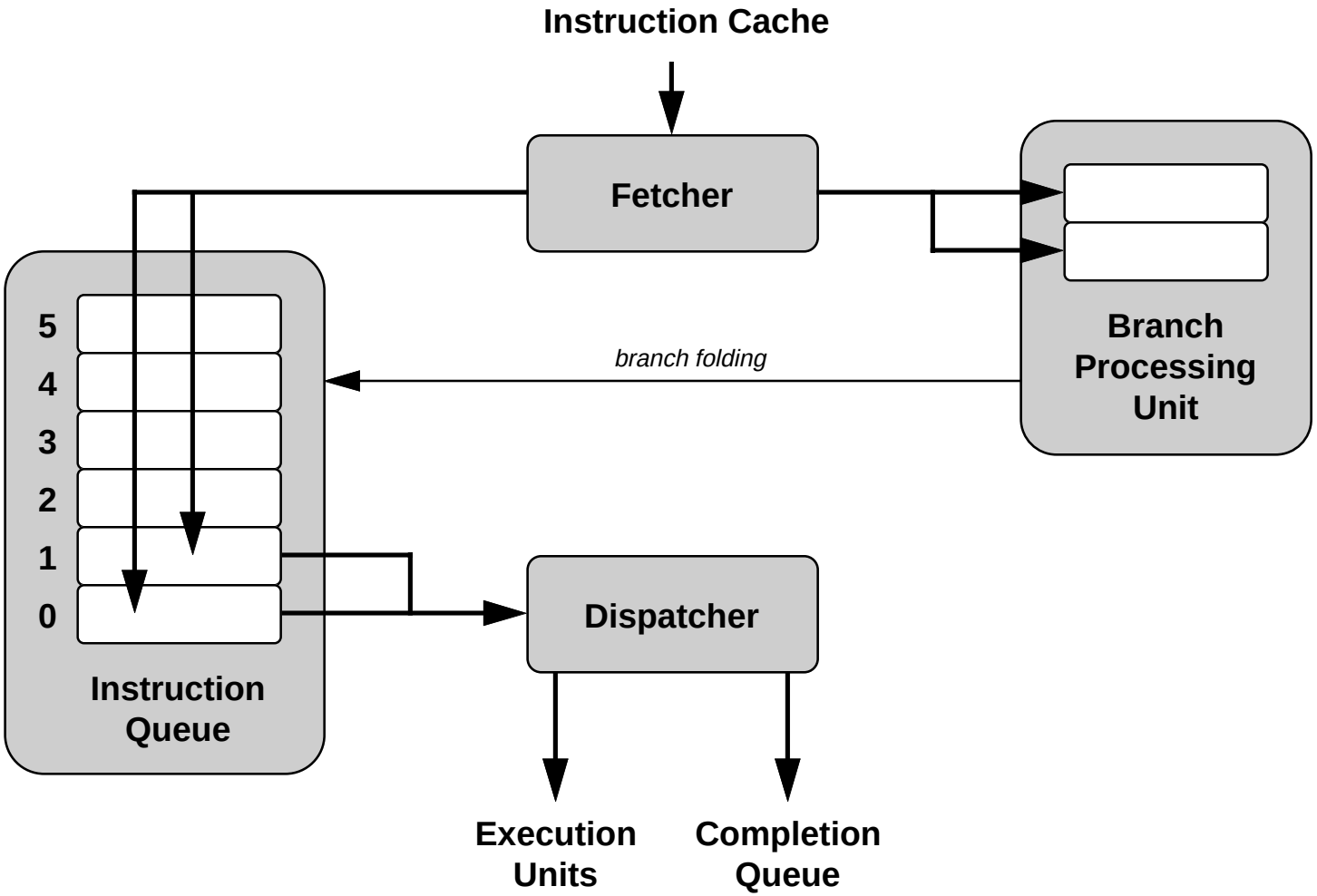
Branching

Opcode	Mnemonic	Addressing	Range
Branch Always	b ba	Relative Absolute	+/- 32MB 0 +/- 32MB
Branch Conditional	bc (condition field(s)) bca (condition field(s))	Relative Absolute	+/- 32KB 0 +/- 32KB
Branch Conditional to Count Register	bcctr (condition field(s))	count reg.	4 GB
Branch Conditional to Link Register	bclr (condition field(s))	link reg.	4 GB

The PowerPC Architecture instruction set includes of two types of branches, unconditional (branch always) and conditional. Conditional branches can depend on the contents of the condition register (CR), which can be set by compare and arithmetic instructions; on the contents of the count register (CTR), which is typically used when executing looping instructions; or on both the CR and the CTR.

Branch instructions can specify an absolute or relative target address, or they can branch to the link register (LR) or CTR. The LR is typically used for subroutine calls, and the CTR (if specified as a destination address) is typically used for absolute jumps.

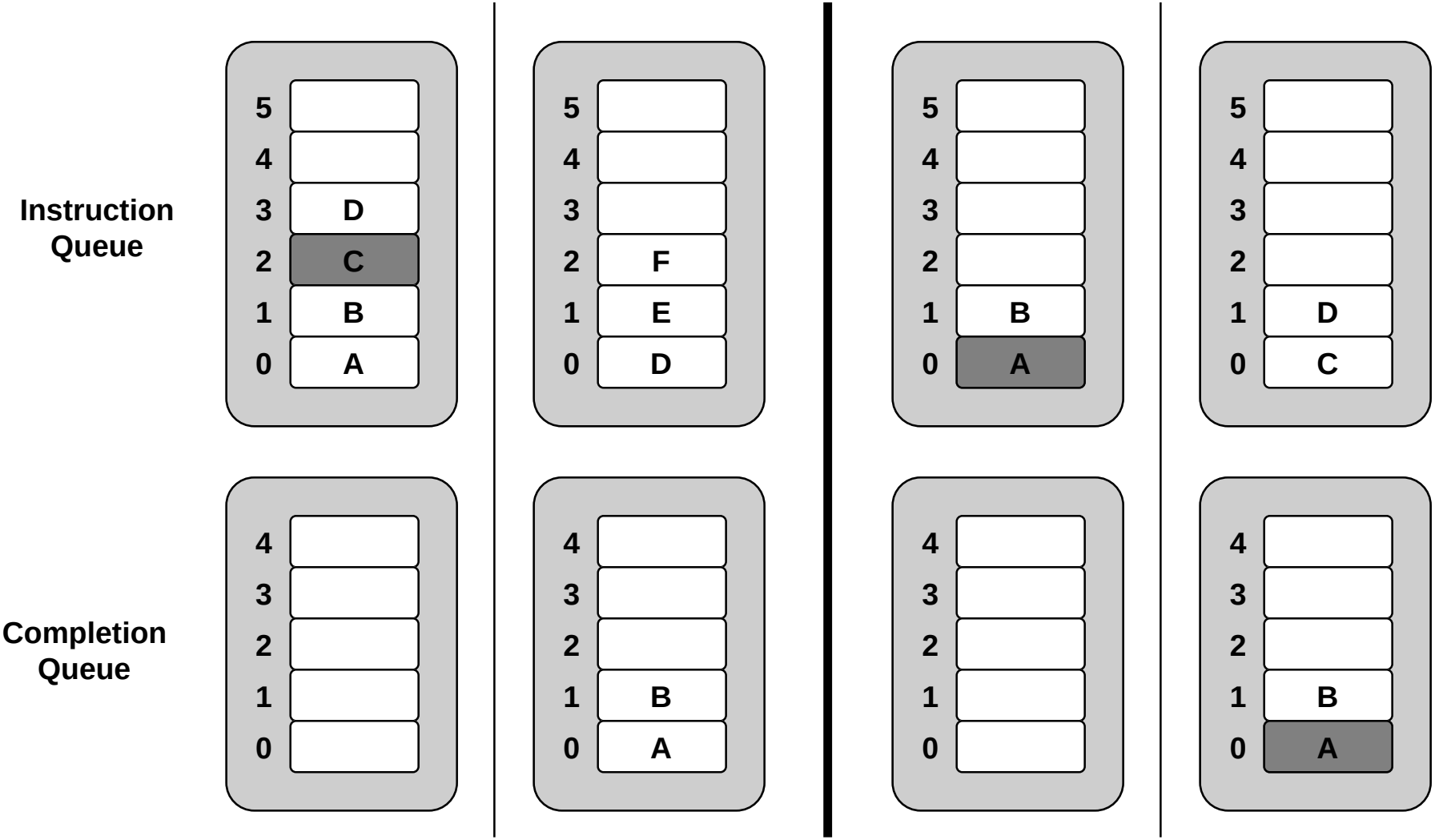
Branch Processing Unit



When the fetcher fetches instructions into the instruction queue (IQ), it also forwards them to the branch processing unit (BPU), which scans these instructions for branches. The BPU immediately begins address calculation for branches found and attempts to fold certain branches out of the instruction queue (discussed later).

Because branch instructions can change the instruction flow, they can potentially cause stalls in the instruction pipeline when new instructions must be fetched from the target address. The 603e includes two mechanisms for reducing the impact of branch instructions: branch folding and branch prediction.

Branch Folding



The branch processing unit (BPU) can fold certain branches out of the instruction queue. They are removed from the IQ before being dispatched, allowing the dispatcher to handle other instructions, and freeing space in the instruction queue and completion queue for other instructions. Frequently, instruction flow can continue as if the branch had not occurred.

The BPU can fold all unconditional branches, as well as conditional branches that do not involve the CTR or LR. Conditional branches that do involve these registers cannot be folded because the CTR and LR have corresponding rename registers which can only be tracked if branches using them get recorded in the completion queue by being dispatched.

Consider the left two columns of diagrams. We start with four instructions in the instruction queue. Instruction C is a branch. In the second column, we see that instructions A and B have been dispatched and have entries in the completion queue, and that instruction C has been folded out by the BPU. Instructions E and F have also been fetched in.

Because superscalar processors feature multiple units that are attempting to flow instructions through their pipelines as quickly as possible, race conditions between various resources can occasionally arise. One race condition occurs in the instruction queue: if the dispatcher can tag a branch for dispatch before the BPU can fold it out of the instruction queue, then the branch will not be folded; it will be dispatched and an entry created for it in the completion queue. This situation typically occurs if the IQ is empty or near-empty and the foldable branch is fetched directly into one of the bottom two slots (i.e. the slots from which instructions are dispatched). However, the performance impact of this race condition is negligible.

The right two columns illustrate the branch race condition. Instructions A and B have just been fetched into the instruction queue, with A being a branch. In this case, the dispatcher grabs A before it can be folded, and we see it in the completion queue in the next cycle.

Branch Code Stall Example

FOR...NEXT w/ bdnz

```
        li      r13,COUNT
        mtspr   CTR,r13
LOOP:
        ;Do some
        ;useful work
        bdnz    LOOP
```

FOR...NEXT w/ subi./bgt

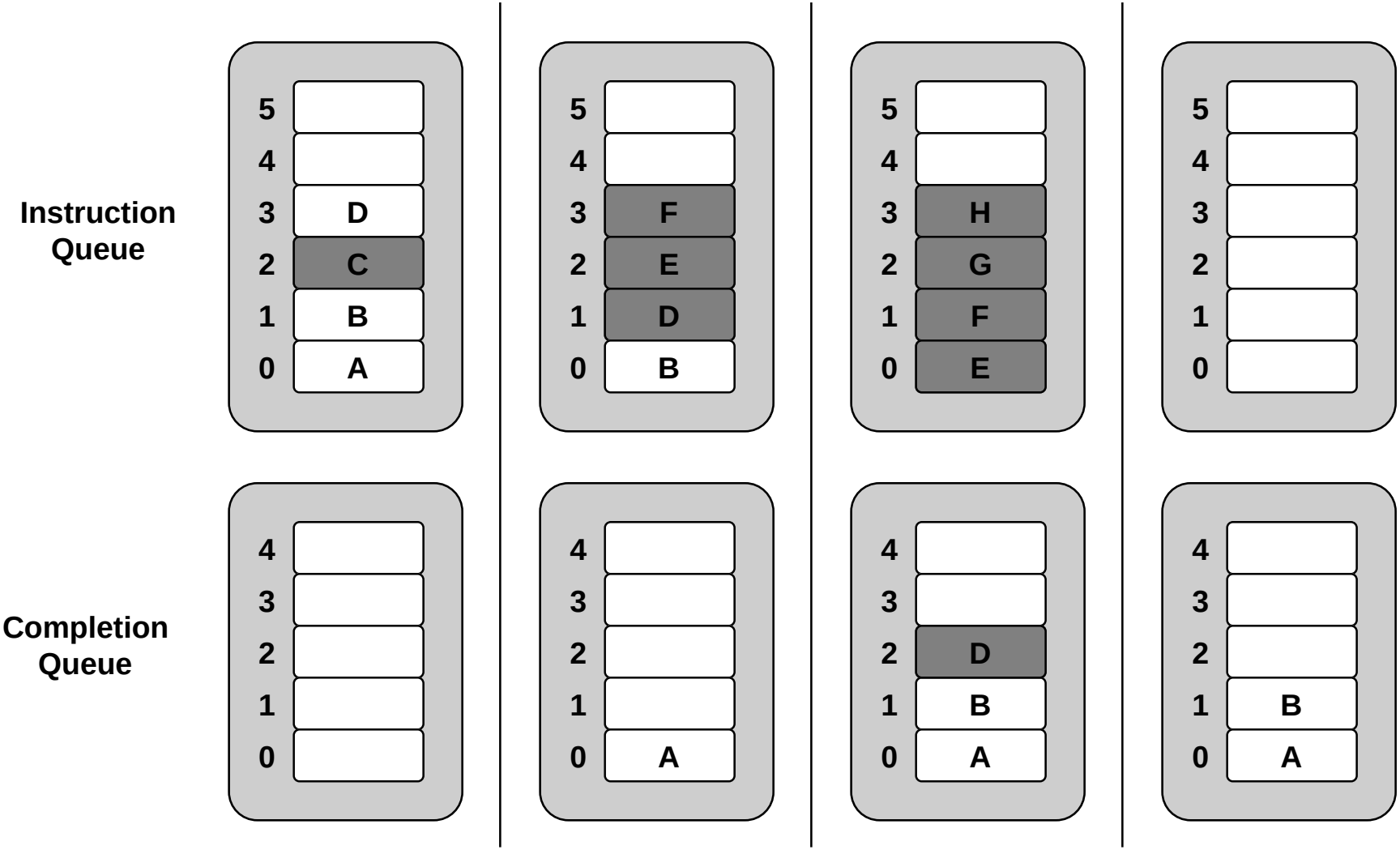
```
        li      r13,COUNT

LOOP:   subi.   r13,0x0001
        ;Do some
        ;useful work
        bgt     LOOP
```

In this slide, we depict a potential stall that can occur with branches. The code fragments demonstrate how, in some cases, one can use branches that are foldable to attain better performance than using non-foldable branches.

The two loops repeat for COUNT iterations. The first code fragment initializes the CTR and uses only one instruction to control the looping, **`bdnz`**. (**`bdnz`** is a simplified mnemonic for a conditional branch which decrements the CTR and branches if CTR is not zero.) This branch cannot be folded and must be dispatched. Since branches that dispatch are required to retire from the last stage of the completion unit, any loop involving a branch that dispatches may need an extra clock (in addition to the loop body time) to complete execution.

It is possible to avoid the additional latency by using a foldable branch instead of the **`bdnz`**. The **`bgt`** and the **`subi.`** instructions in the second code fragment can be used to obtain the same functionality as the **`bdnz`**. The **`subi.`** instruction is a single cycle instruction that can retire paired with almost any other instruction; thus in most loops, **`subi.`** adds no time to the execution of that loop. The **`bgt`** is also capable of being folded out of the pipeline and not dispatching at all. Therefore, code that uses the **`subi./bgt`** combination will likely be a clock faster each time through the loop then **`bdnz`**. However, the exact timing difference, if any, would depend on the actual composition of the loop body.

Branch Prediction and Speculative Execution



Each conditional branch instruction includes a prediction bit, which is set by the compiler or an assembly language programmer. This bit helps specify whether the branch is predicted to be taken or not taken. This is known as ***static prediction*** because the prediction behavior is encoded in the instruction. While the branch condition is waiting to be resolved, execution continues down the predicted path, and these subsequent instructions are marked as ***speculative instructions***. (Speculative instructions are not allowed to change the programming model, such as update register files or memory, and may stall until the branch is resolved and they become non-speculative.)

When the branch condition is resolved, if the prediction was correct, then the speculative instructions are marked non-speculative, and no penalty is assessed. If the prediction was incorrect, then the speculative instructions are ***flushed*** (removed from the instruction pipeline) and execution resumes along the correct execution path.

The 603e has one level of prediction, meaning that a conditional branch encountered along a speculative path cannot itself be executed speculatively. Instead, it will stall in the pipeline until the previous branch is resolved.

In the leftmost diagram, we have instructions A, B, C, and D in the instruction queue. Instruction C is a branch. In the next diagram (next cycle), instruction A was dispatched and C folded out by the BPU. However, assume that branch C cannot be resolved (perhaps it is dependent on the results of instruction A). All subsequent instructions are then marked speculative: D and the newly fetched instructions E and F.

In the next diagram, we see that B and D were dispatched to the CQ and G and H fetched into the IQ. In our example, branch C is now resolved and it turns out the branch was mispredicted. In the final diagram, the speculative instructions are flushed, and the fetcher is ready to fetch instructions from the correct input stream. If branch C had been correctly predicted, the speculative instructions would simply be marked non-speculative and no stall would occur.

Performance Impact of Branch Prediction

- Speculative execution allows instruction flow to proceed before branch conditionals have been resolved
- Correct predictions incur no performance penalty
- Incorrect predictions only incur significant performance penalties when mispredicted paths result in instruction cache misses
- Incorrect predictions may be avoided by separating the instruction that sets a branch condition from the branch that uses it

Speculative execution allows the fetcher to fetch instructions without stalling while the branch is being resolved. Prediction does not cause any pipeline stalls unless the prediction is deemed to be incorrect. If the prediction is incorrect, it is the function of the BPU to perform the necessary tasks to recover from speculation.

Branch prediction of the type used by the 603e is correct approximately 86% of the time. Due to the 603e's ability to invert the normal prediction mechanism, a smart programmer or compiler can attain greater prediction accuracy.

Mispredicted branches, which occur infrequently even using only the default speculation mechanism, only incur significant performance penalties when speculative branches also result in cache misses on the mispredicted path.

Since incorrect predictions can potentially cause many stalls, it is possible to improve performance by avoiding prediction in some code fragments. By separating the instruction that is setting the branch condition from the branch that uses it, it is possible to prevent the processor from executing speculatively altogether.

In the 603e, we can calculate the approximate separation distance by using worst case analysis for a conditional branch dependent on the CR register. Assuming that the processor dispatches 3 instructions per clock (2 instructions and a unconditional branch or nop), and assuming a worst case conditional register update time of 3 clocks, we calculate that by separating the branch condition from the condition register update instruction by 9 instructions, we will avoid speculative execution. For most code fragments, the 603e can dispatch instructions at a peak rate of 2 instructions. Additionally, most instructions (such as the COMPARE instruction) take only 1 clock to update the Condition Register. Under these conditions, one can prevent speculative execution by separating the branch condition from the condition register update instruction by only 3 instructions.

Integer Unit and System Register Unit

- **Integer Unit**
  - **No stalls caused by single-cycle instructions**
  - **Multi-cycle instructions keep the integer unit busy**
  - **Possible stalls due to dependencies minimized by allowing access to operands as soon as the source data is valid**

- **System Register Unit**
  - **Handles access to system registers**
  - **Assists the Integer unit by handling some integer unit operations**

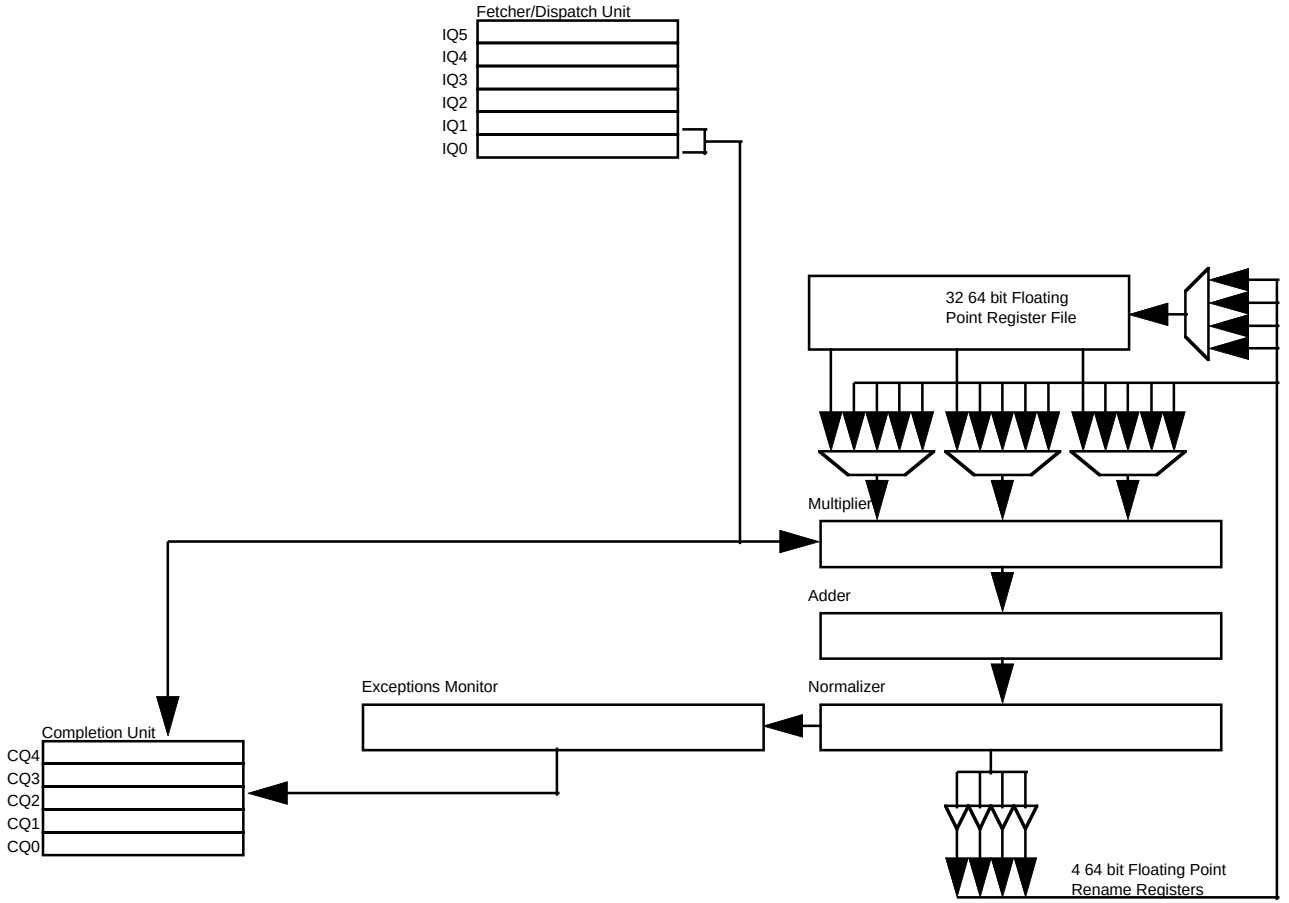
Integer Unit

Most integer instructions only take one cycle to execute; thus the integer unit does not usually stall. The only times that the integer unit stalls is if it is executing multiple-clock integer instructions such as trap, multiply, and divide, or if the instruction cannot execute because it is dependent on the results of another operation. The internal bus structure of the 603e allows an integer instruction to immediately access any operand as soon as it becomes valid.

System Register Unit

The SRU handles all of the special purpose register instructions, context synchronizing instructions, and certain integer add/compare operations. Some special purpose register instructions are also inherently context synchronizing. Context synchronization will always cause some instruction stall, but this is almost always critical to guarantee correct operation. Integer operations in the SRU take only one cycle to execute, thereby causing no stalls.

Floating Point Unit



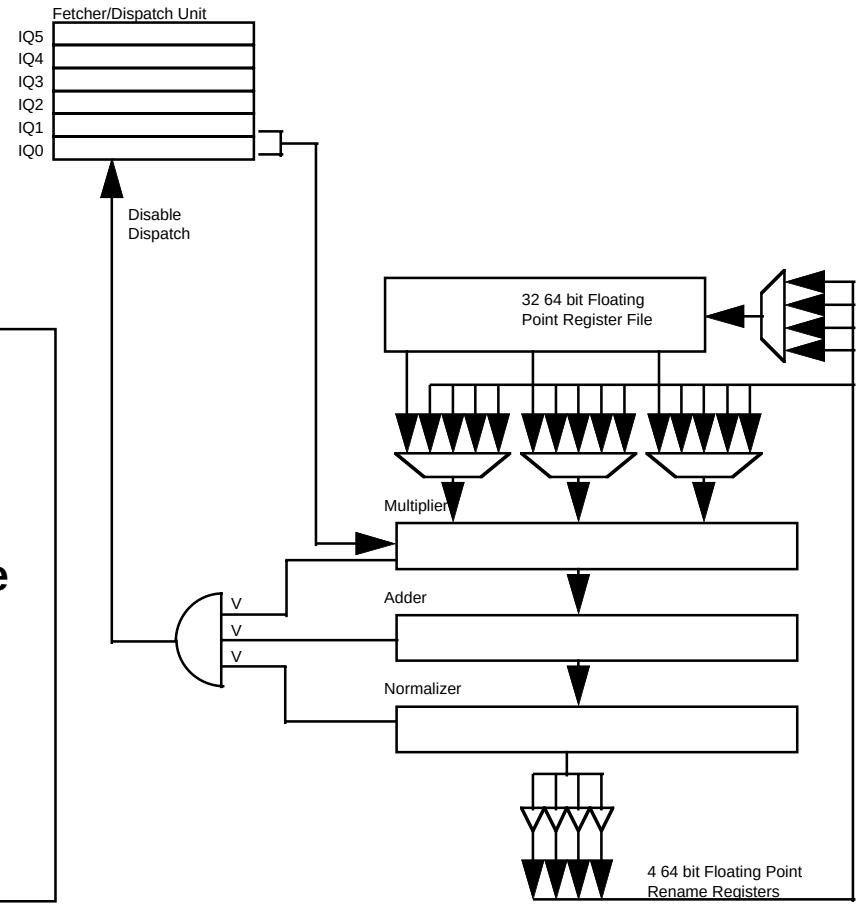
The Floating-Point Unit consists of four stages: Multiplier, Adder, Normalizer, and Exception, which are organized conceptually as shown. The Multiplier stage is a single precision multiplier that every FP instruction must pass through. No instruction can enter the FP unit if the Multiplier is occupied. Double precision operations will cause the Multiplier to be occupied for two consecutive clocks. The Adder always takes a single clock. Typically, instructions will flow through these stages without stalling unless a stall in the Normalize or Exception stage blocks the instruction pipeline flow. The FP register file only supports a single write-back port from the rename registers.

The Normalizer stage can cause delays of up to several clocks. The number of clocks that normalization takes is data-dependent. When the normalizer stalls, it prevents instructions in the Multiplier or Adder stages from stepping through.

To prevent potential speed path problems, an additional stage exists after the normalization stage. This stage is simply a holding stage and floating-point instructions use up one clock cycle to pass through it.

FPU Stall Conditions

- **Stall Conditions**
  - **Normalization blocking dispatch**
  - **Late-release of FP rename registers**
  - **Enabling exceptions**



As previously mentioned, the Normalizer stage can take multiple cycles, thereby stalling the flow of instructions within the FPU. When FP instructions occupy the Normalizer, Multiplier, and Adder stages at the same time, a signal will be sent to the dispatcher, halting dispatch of instructions to the floating-point unit. Even if normalization doesn't stall the pipeline, the distance between the Normalizer and dispatcher prevents the FPU from informing the dispatcher to resume dispatching until after it is too late to dispatch an instruction on that clock. This causes a stall after every third consecutive single cycle FP instruction.

The wait stage that exists after the normalization stage also contributes to potential stalls. The additional wait stage causes FPR rename registers to be released one cycle after the FP operation is complete. This causes a stall if a series of single-cycle FP instructions are executing in the FPU. After every fourth single cycle FP instruction, a stall will occur due to lack of FPR rename registers.

These two stall scenarios cause a series of single-cycle FP instructions to dispatch in clocks 1, 2, 3, 5, 7, 8, 9, 11, 13, 14, 15, etc. The next slide depicts the stall scenarios described above.

Finally, if exception checking is enabled in the FPU, the instruction may have to wait in the Normalizer while exceptions are checked. One can enhance performance by pre-qualifying data prior to running it and polling for possible exceptions at the last reasonable instant.

FPU Code Stalls

clock	Instruction Queue						Completion Queue					renames
0					B	A D						
1				D	C	B D					A	A
2			F	E	D	C D				B	A	A, B
3		H	G	F	E	D			C	B	A	A, B, C
4	I	H	G	F	E	D D			C	B	A F	A, B, C
5		I	H	G	F	E			D	C	B F	A, B, C, D
6	J	I	H	G	F	E D				D	C F	B, C, D
7		J	I	H	G	F D				E	D	C, D, E
8		K	J	I	H	G D			F	E	D F	D, E, F

D = Marked for dispatch

F = Marked “finished”

Series of single-cycle FP instructions

Clock 0: First two instructions (**A** and **B**) are brought into the instruction queue (IQ). **A** is marked for dispatch.

Clock 1: **A** is dispatched to the Multiplier stage of the FPU and is allocated FP rename register 0. **B** is marked for dispatch. **C** and **D** are brought into the IQ.

Clock 2: **A** steps to the Adder stage in the FPU. **B** is dispatched to the Multiplier stage and is allocated FP rename register 1. **C** is marked for dispatch. **E** is brought into the IQ. (Anything after **E** is ignored for this discussion.)

Clock 3: **A** steps to the Normalizer stage in the FPU. **B** steps to the Adder stage. **C** is dispatched to the Multiplier stage and is allocated FP rename register 2. At this point, a signal is sent to the dispatcher indicating that no instruction may be dispatched to the FPU until a stage has been freed up. This signal is negated as soon as the Normalizer stage is finished, but this will be too late to actually permit an instruction to dispatch on the next clock. **D**, therefore, stalls in the IQ (is not marked for dispatch).

Clock 4: **A** steps to a wait stage in the FPU. A signal has been sent to the completion unit indicating that **A** is finished and, since it is the oldest instruction in the completion queue, it is permitted to retire. **B** steps to the Normalizer stage. **C** steps to the Adder stage. **D** is given permission to dispatch on the next clock.

Continued on next slide

Clock 5: **A** is gone from the completion queue, but a delay on FP rename register deallocation prevents FP rename register 0 from being re-allocated. **B** is finished and permitted to retire. **C** steps to the Normalizer stage. **D** is dispatched to the FP Multiplier stage and is allocated FP rename register 3. At this point, all four FP rename registers are in use, which means **E** cannot be marked for dispatch this cycle. **E** stalls in the IQ.

Clock 6: FP rename register 0 is deallocated. **B** is gone, but its FP rename register deallocation is delayed for one clock. **C** is finished and permitted to retire. **D** moves to the Adder stage. **E** is marked for dispatch.

Clock 7: FP rename register 1 is deallocated. **C** steps the FP wait stage. **D** steps to the Normalizer stage. **E** is dispatched to the Multiplier stage and is allocated FP rename register 0. At this point, the pattern of stalls repeats.

Again, note the dispatch stall during clock 3. This is caused by all of M/A/N stages being in use.

Also note the dispatch stall during clock 5. This is caused by all of the rename registers being tied up (a rename register must be deallocated for one clock before it can be reused).

FPU and Completion Unit

```
loop:   lfsu    f22,4(r20)          ; A
        fmadd   f15,f16,f13,f28     ; B
        lfsu    f23,4(r21)          ; C
        fmadd   f18,f19,f14,f29     ; D
        lfsu    f13,4(r20)          ; E
        fmadd   f25,f24,f22,f30     ; F
        lfsu    f14,4(r21)          ; G
        bdnz    loop                ; H
```

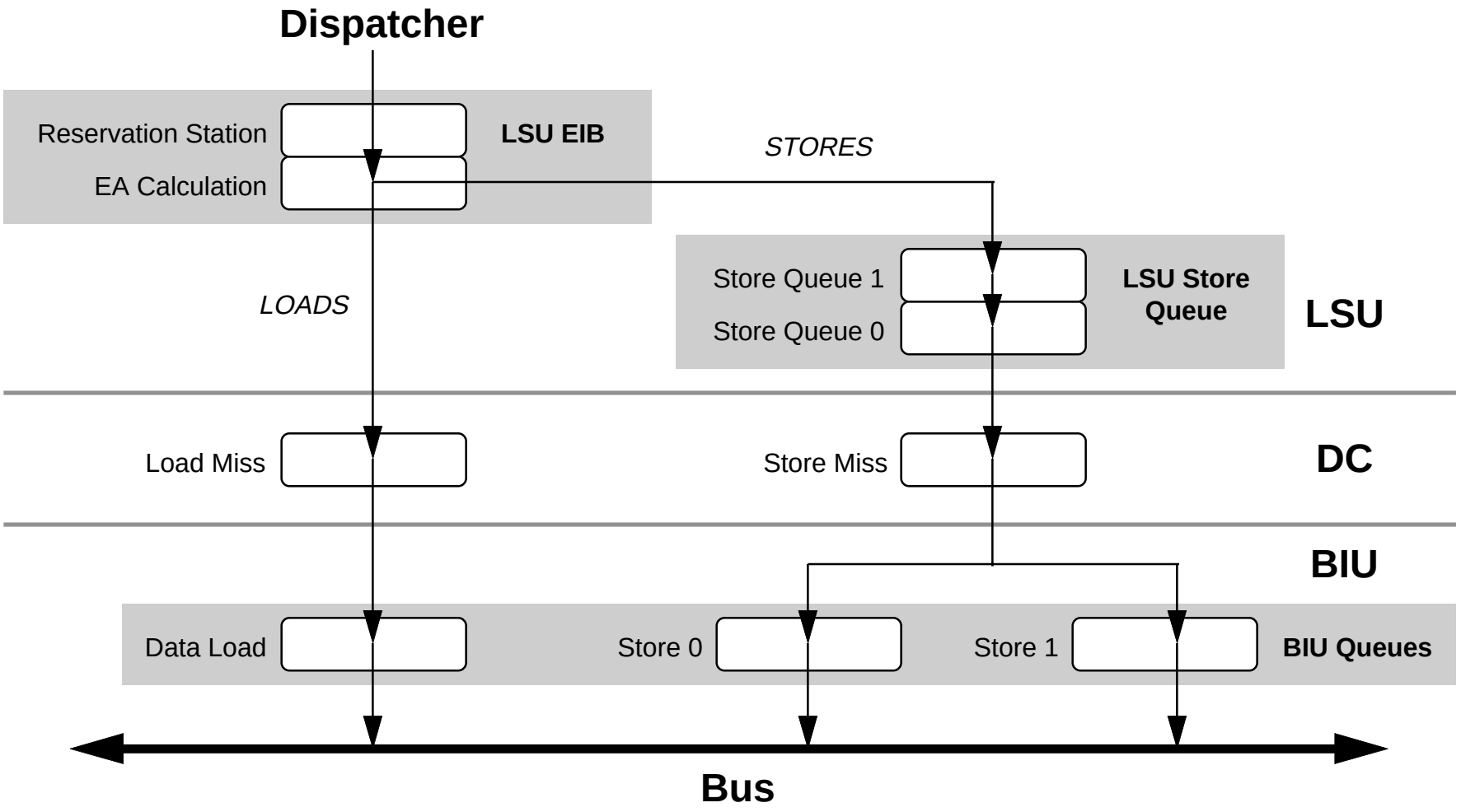
cycles

Completion Queue	1	2	3	4
			F	G
		D	E	F
	B	C	D	E
	A	B	C	D

Completion of floating-point unit instructions is a potential source of stalls. Due to the single write-back port on the floating point register file, multiple instructions trying to write back floating-point results will have to do so in a sequential manner. This will typically happen in matrix math where math operations occur in parallel with loads that initialize registers for subsequent math operations.

The code segment above depicts such a scenario. Adjacent load and fmadd instructions have no register dependencies nor do they require the same execution unit. Therefore, each pair can dispatch together, execute in parallel, and even finish (update rename registers) in parallel. However, due to the single write back port, this code has an effective throughput of only a single instruction per clock. If this code is part of a larger code segment that includes integer instructions, then it is possible to achieve a greater instruction throughput by intermixing integer instructions (from elsewhere in the code sequence) with these floating-point instructions. This will allow the integer execution and write-back to overlap with the floating-point write-back, thereby improving the overall instruction throughput on the entire code segment.

Load/Store Hierarchy



The load/store hierarchy within the PowerPC chip consists of the load/store unit (LSU), data cache (DC), and the bus interface unit (BIU). The LSU stages consist of a two-element EIB, to receive dispatched instructions and calculate effective addresses, and a two-element store queue, to hold stores waiting for the data cache. The data cache stages consist of slots for a load miss and a store miss. Only one miss can be handled at a time. The BIU stages consist of a number of one-element queues, such as the data load and store queues. Each queue can hold a separate instruction waiting for access to memory.

Instructions are first dispatched from the instruction queue (IQ) to the LSU EIB, which has two slots: the “reservation station” slot (LSU RS) and an “effective address calculation” slot (LSU EA). An instruction is held in the LSU EA slot until its address operand is available.

Normally if the LSU is available for dispatch (see below), then the instruction is dispatched directly to the LSU EA slot, if both slots are empty. If the LSU EA slot is occupied, then the instruction is dispatched to the LSU RS slot.

Once the instruction’s effective address has been calculated, its progress through the pipeline depends on whether it is a load or a store. A load would then access the data cache (DC), as described later. The load’s entry in the completion queue (CQ) is marked “finished” when the data for the load returns.

A store would pass to the first LSU store queue slot, and its entry in the CQ would be marked “finished.” Thus, a store can be considered finished and even retired from the completion queue long before its data is actually written to cache or to memory. On the next clock cycle, the store passes to the second LSU store queue slot and, on the subsequent clock, it is free to access the data cache.

Note that because a store must traverse two additional slots than a load before accessing the data cache, a load instruction may bypass preceding stores within the LSU. Also, if both a load (in the LSU EA slot) and a store (in the second LSU store queue slot) are free to access the data cache, then the load will take precedence.

Data Cache Miss Stall

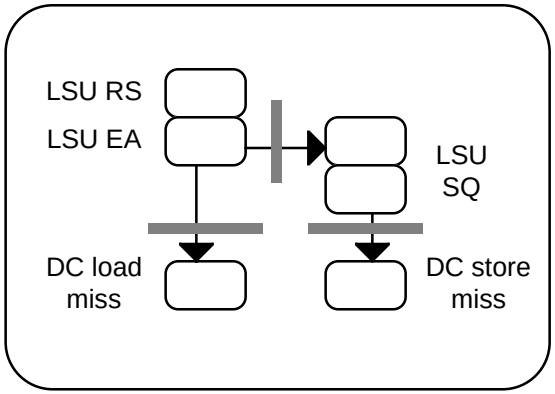


Figure 1

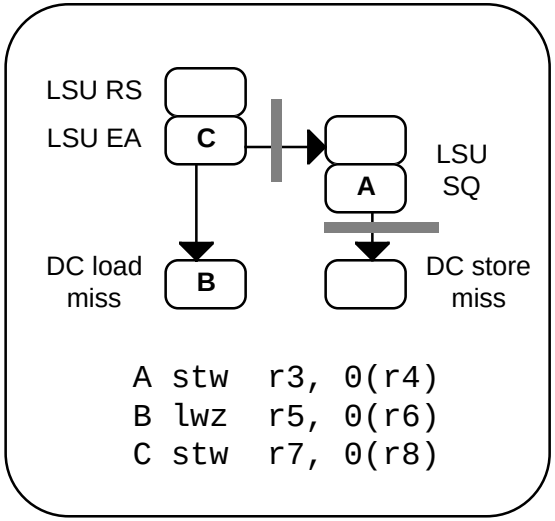


Figure 2 - Store Stalls

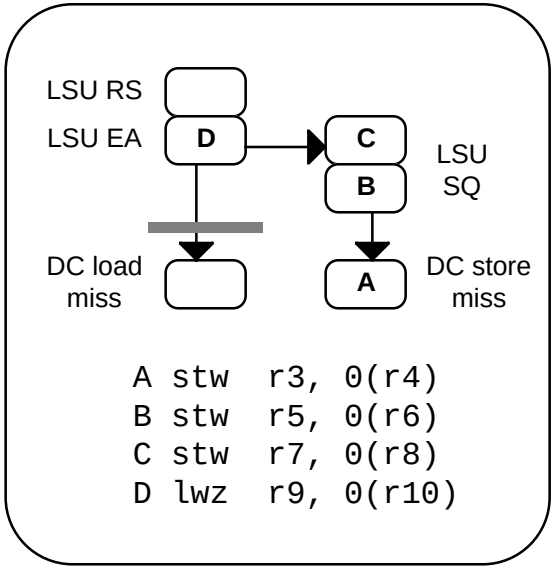


Figure 3 - Load Stall

Although superscalar architectures feature multiple execution paths, resource limitations can stall full utilization of these paths. Data cache misses are the primary cause of stalls in the LSU. The example above demonstrates how stalls can occur because the 603e data cache can only handle one miss at a time.

When a load or store misses in the data cache, the data cache asserts a busy signal that stalls subsequent instructions in the LSU, as shown in **Figure 1**. While the data cache is busy, no other instructions can access the data cache, and instructions are blocked from leaving the LSU EA stage. This prevents a store from propagating from the LSU EA stage to the LSU store queue (LSU SQ), even if the store queue is available.

For a load miss access, the data cache is busy until the data comes back from the BIU. For a store miss access, the data cache is busy until the store is able to propagate to the BIU.

Figure 2 demonstrates store stalls. While load B is waiting for its data to come back, store A may not access the data cache, and store C may not propagate to the LSU store queue. Note that load B bypassed store A in the LSU.

Figure 3 demonstrates a load stall. Load D may not access the data cache until store A propagates to the BIU. When it does, the data cache is no longer busy, and load D will bypass stores B and C.

Address Alias Stall

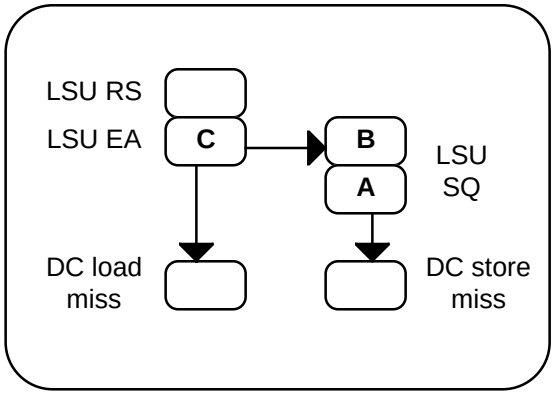


Figure 1

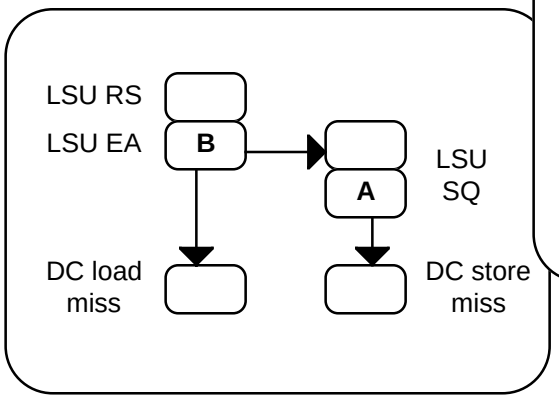


Figure 2

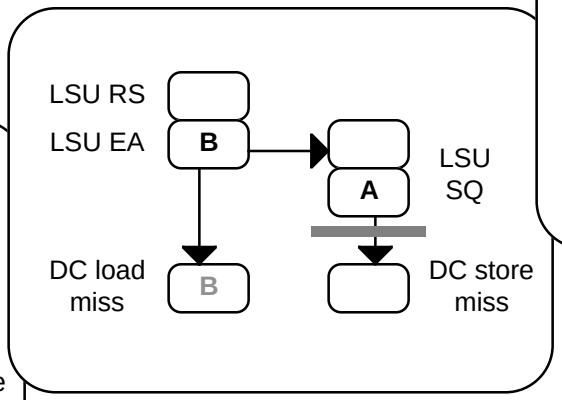


Figure 3

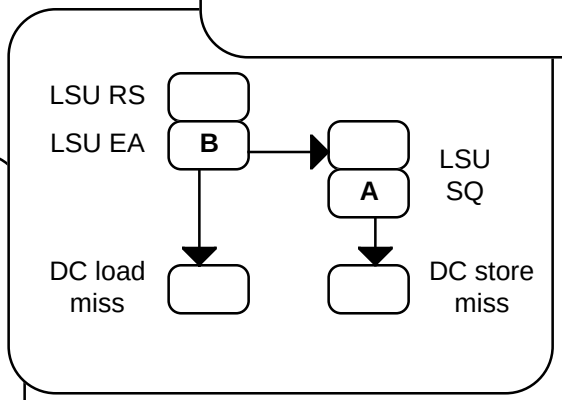


Figure 4

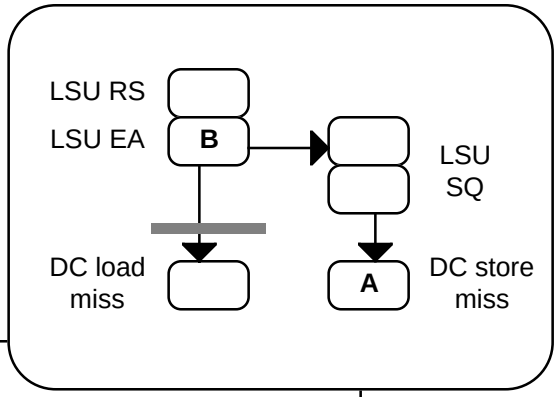


Figure 5

To understand the flow within a superscalar architecture, one cannot ignore instruction-specific details. For example, consider **Figure 1**, in which load C would ordinarily bypass stores A and B. However, if the data address of C can potentially collide (***alias***) with the data address of A or B, then C will stall in the LSU EA slot until the aliasing store passes out of the LSU store queue.

Address translation may occur after alias checking. Since only the lower 12 bits remain constant through translation, these are the only bits that can be checked. In addition, the addresses are checked with word granularity (four bytes, mask = 0xffc) if the sizes of both load and store are less than or equal to four bytes, or with double-word granularity (eight bytes, mask = 0xff8) otherwise. For instance, 0x2000 and 0x3003 would alias to each other, but 0x2000 and 0x2020 would not.

Note that it is possible to have an alias stall even if the load and store do not actually access the same location, because only the lower 12 bits of the address can be compared.

In a superscalar architecture, other stalls may occur due to timing considerations. For example, if a load which aliases a store has spent only one cycle in the LSU EA stage, then the LSU circuitry is not fast enough to prevent the load from bypassing the store in accessing the data cache. Since this aliased load should not access the cache before the store, the LSU must cancel the load in the subsequent cycle. **Figures 2-5** depict this situation.

In **Figure 2**, load B and store A have aliasing addresses. If B has been in the LSU EA stage for more than one cycle (due to some other stall), then there is time to prevent it from accessing the data cache, and the next cycle A will access the data cache. However, if B has only been in the LSU EA for one cycle, the alias check comes too late to prevent the cache access shown in **Figure 3**. A is stalled and cannot access the cache.

In the next cycle (**Figure 4**), the load is canceled, and in **Figure 5** the store propagates to the data cache. Note that in this example, the store also misses in the cache and blocks the load from accessing the data cache the next cycle.

Completion Queue Stall

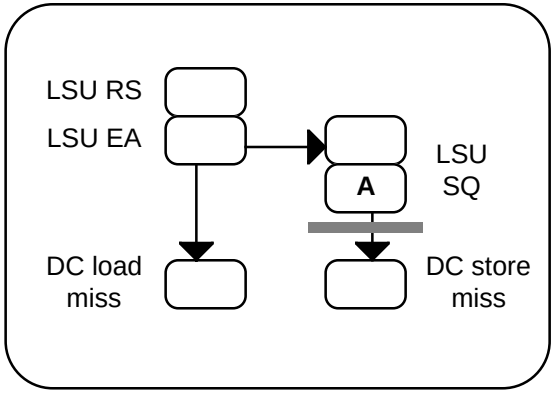


Figure 1

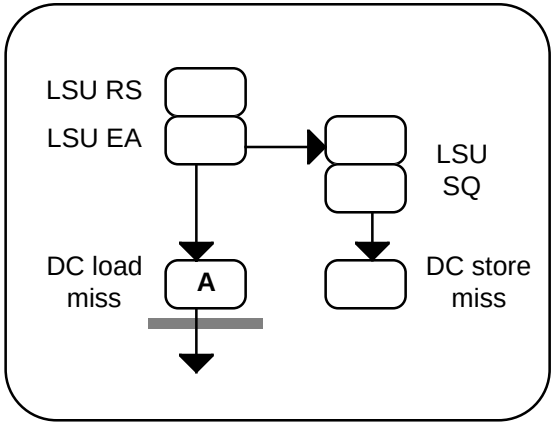


Figure 2

Superscalar architectures frequently signal between different parts of the architecture, in order to coordinate various aspects of the units. Diagrams may not always show all the signals that are shared between units with the system. These interactions can also cause stalls. We discuss a case in which the state of the completion queue can affect instruction flow in the LSU.

Since the 603e allows out-of-order execution, instructions will frequently dispatch to the LSU (as well as other execution units) before previous instructions have finished executing. If one of these previous instructions generates an exception, then all subsequent instructions (including the LSU instruction) must be canceled from the instruction flow (***flushed***). Various parts of the processor, including the LSU, must be careful to stall instructions that could be canceled before they permanently change the processor state.

On the 603e, if a load or store's entry in the completion queue is not in the bottom slot, then there are preceding instructions that could potentially generate exceptions which may cancel the load or store. The instruction must be stalled before it reaches a state that cannot be canceled.

Figures 1-2 depict this situation, in which instruction A is stalled because its entry in the completion queue is not in the bottom slot. In **Figure 1**, store A is stalled in the second slot of the LSU store queue, since writing to the data cache would incur too much of a penalty to undo.

In **Figure 2**, load A is stalled in the data cache miss slot if it is accessing guarded memory. ***Guarded memory*** is typically used to prevent out-of-order loads to I/O devices, which may produce undesired results otherwise. Note that even if load A were at the bottom of the completion queue, the 603e would stall the load for one cycle before making its request to the BIU.

LSU EA Stall

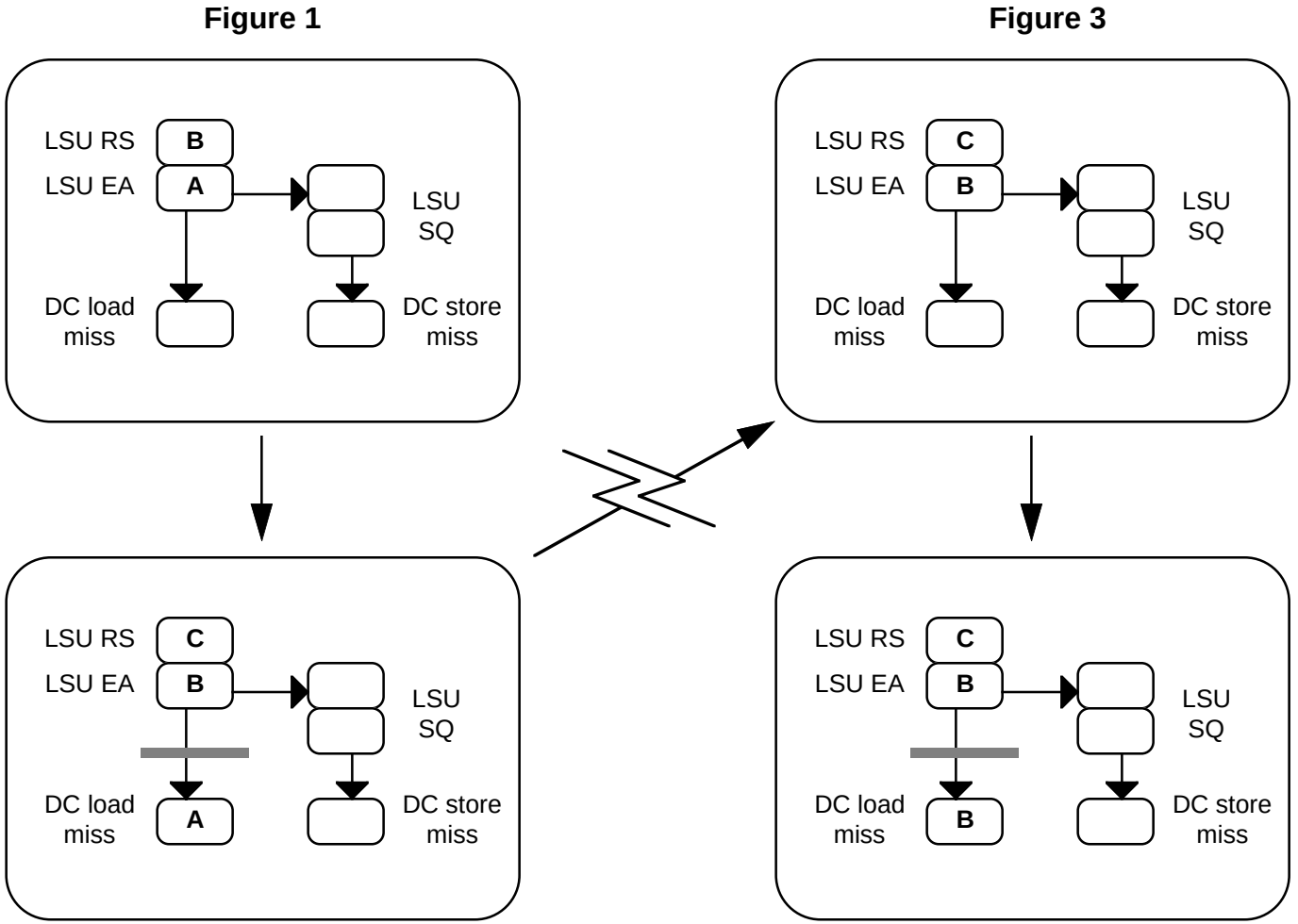


Figure 1
LSU RS
B
LSU EA
A
LSU SQ
DC load miss
DC store miss
Figure 3
LSU RS
C
LSU EA
B
LSU SQ
DC load miss
DC store miss
Figure 2
LSU RS
C
LSU EA
B
LSU SQ
DC load miss
A
DC store miss
Figure 4
LSU RS
C
LSU EA
B
LSU SQ
DC load miss
B
DC store miss

The fast timing requirements of superscalar processors sometimes lead to unusual types of stalls.

If a load has spent only one cycle in the LSU EA slot before accessing the data cache, then it is removed from the LSU after this access (assuming that the access is not canceled). However, if a load spends more than one cycle in the LSU EA slot, then it will appear to remain in this slot (blocking subsequent LSU instructions) even after the load has accessed the data cache. This block will remain until the data becomes available (and the load is marked “finished” in the completion queue).

In **Figure 1**, A flows into the LSU EA slot and flows out in **Figure 2**. This allows C to be dispatched to the LSU. However, because B is stalled in the LSU EA slot, in **Figure 3** when B accesses the data cache it keeps its entry in the LSU EA slot in **Figure 4**. This stalls C and stalls dispatch of any subsequent load/store instruction until the data for B returns from the BIU.

Misaligned Address Stall

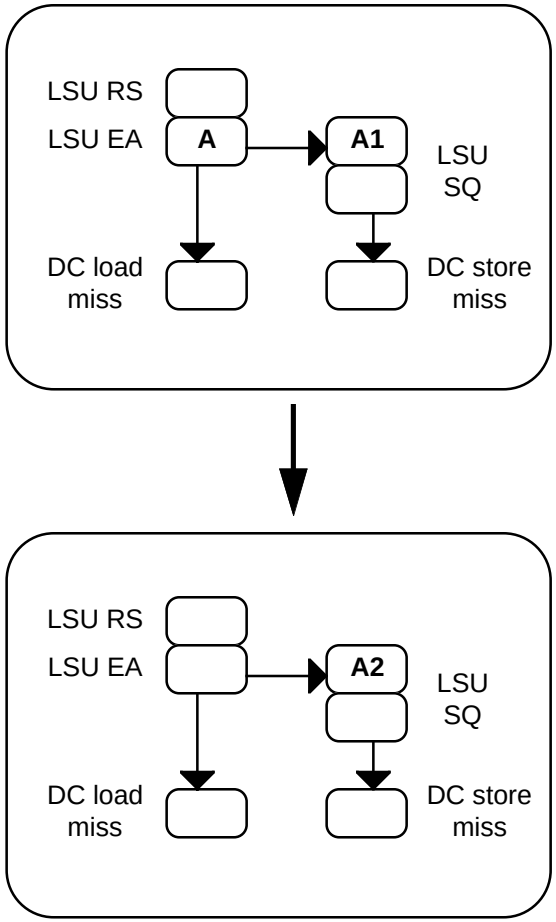


Figure 1 - Misaligned Store

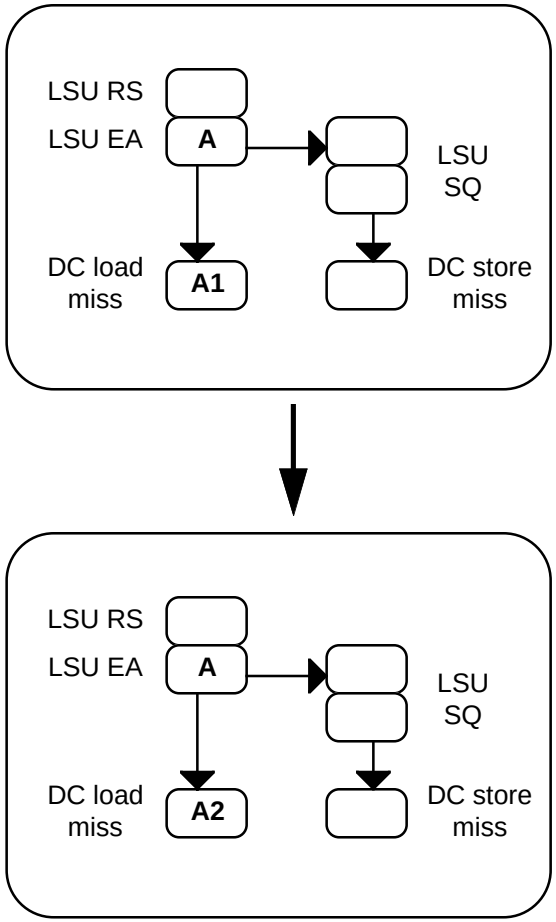


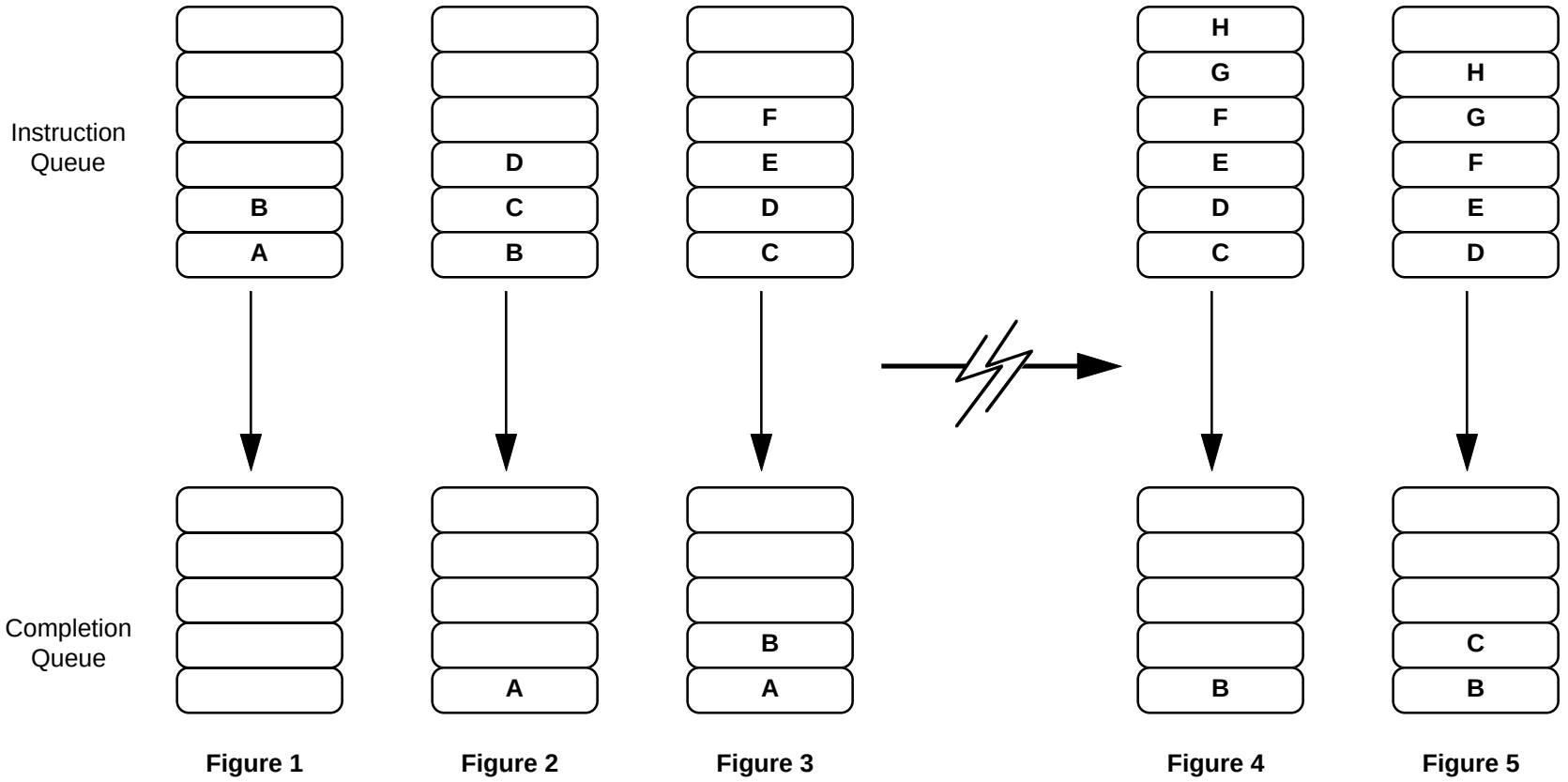
Figure 2 - Misaligned Load

Accessing misaligned addresses can result in significant performance penalties in most RISC superscalar microarchitectures.

On the 603e, a data address is aligned if it falls on a multiple of the access size. Thus a word (4-byte) access is aligned at 0x0000 and 0x0004 but not at 0x0002; a doubleword (8-byte) access is aligned at 0x0000 and 0x0008 but not at 0x0004.

If the data address of a load or store in the LSU EA slot is not aligned, then it is split into two aligned accesses. **Figure 1** shows a misaligned store. A is first split into the aligned store A1, then on the next clock it is split into the aligned store A2 and the LSU EA entry removed. In **Figure 2** we have a misaligned load, in which A is split into aligned loads A1 and A2. Note that because the load stayed in the LSU EA slot for more than one cycle, it remains in this slot until its data comes back (see LSU EA stall above).

Instruction Interactions
Rename Register Stall



As with the other execution units, there may also be stalls due to contention for the rename registers. **Figures 1-5** show the interaction between the IQ and CQ for a series of **`lwzu`**'s which are fetched into the instruction queue two at a time. Each **`lwzu`** uses two general purpose register (GPR) rename registers, one for the address operand and one for the data operand. The 603e has five GPR rename registers available (and four FPR rename registers).

In **Figure 3**, instruction C cannot dispatch because A and B have already taken four GPR rename registers and there is only one available.

Later, when A retires and releases its rename registers (**Figure 4**), C has the resources it needs to dispatch. It dispatches the following cycle (**Figure 5**).

Instruction Interactions Dependency Stall

ORIGINAL SEQUENCE

```
A lwzx   r13,r14,r15
B add    r26,r27,r13
C lis    r20,0xDEAD
D stwu   r16,4(r17)
E ori    r20,r20,0xBEEF
F cmpw   r26,r20
```

REORDERED SEQUENCE

```
A lwzx   r13,r14,r15
B lis    r20,0xDEAD
C stwu   r16,4(r17)
D ori    r20,r20,0xBEEF
E add    r26,r27,r13
F cmpw   r26,r20
```

clock cycle

Completion Queue

1	2	3	4	5
	stw	ori		
	lis	stw	cmp	
add	add	lis	ori	
lwz	lwz	add	stw	cmp

clock cycle

Completion Queue

1	2	3	4	5
	ori			
	stw	add		
lis	lis	ori	cmp	
lwz	lwz	stw	add	

The mix of instructions in an instruction sequence can result in a variety of stalls. Dependency stalls are the most common. A dependency occurs if one instruction uses as its source data the results from another instruction. Such a dependency will cause a stall if the two instructions are placed right next to each other. The 603e reduces the impact of most of these situations through use of the rename registers and forwarding of results. However, in some situations, stalls can happen as follows.

Two orderings of a code sequences are shown. In both sequences, the **add** instruction uses as its source the results of the **lwzx** load instruction. In the original code, the **add** occurs right after the **lwzx**. In the reordered sequence, the **add** is separated from the **lwzx** by moving it down three instructions.

Analysis of original code sequence:

Assuming the **lwzx** hits in the data cache, its data will return in 2 clocks. Although both the **add** and the **lwzx** can be dispatched to the completion queue in the same clock, the **add** cannot begin execution until the data from the **lwzx** returns. Therefore it cannot retire with the **lwzx** and is stalled by one clock. The **lis** dispatches to the SRU, executes, and is ready to retire with the **add** in cycle 3. In cycle 4, the **stwu** and **ori** can also retire together. Then in cycle 5, the **cmpw** retires alone. Total time: 5 clocks.

Analysis of reordered code sequence:

The **lwzx** (cache hit) takes 2 clocks. Since the **lis** is not dependent on the **lwzx**, it can retire with the **lwzx** in clock 2. The **stwu** and **ori** can also retire together on the next clock (clock 3). Finally, in clock 4, the **add** and **cmpw** retire together. Total time: 4 clocks.

Thus by separating the generation of a result from the subsequent use of that result, we were able to prevent a stall. It is normally a good practice to provide this separation; however in some cases the benefit gained in one place is lost in another place.

Summary

- Scheduling code around superscalar microprocessor resource constraints reduces code stall conditions
- Code stalls can occur in instruction issue/completion control logic
  - Availability of instruction and completion buffers
  - Availability of rename registers
  - Number of register file write ports
- Code stalls can occur within execution units
  - Aliasing between loads and stores
  - Misaligned accesses
- Code stalls can occur due to instruction mixes
  - Dependencies between instructions

By being able to process multiple instructions at the same time, superscalar microprocessors like the 603e enable systems to attain extremely high levels of performance. However, there are many aspects of a superscalar architecture that can cause code stalls in the instruction flow. By being aware of constraints that cause code stall conditions, one can generate code that can will execute with minimal latency in a superscalar processor. This paper has discussed the aspects of PowerPC 603e that can cause stalls. Although this paper is specific to the 603e, the lessons learned can be applied to most contemporary superscalar microprocessors.