+

# William Stallings Computer Organization and Architecture 10th Edition

# + Chapter 16

## Instruction-Level Parallelism and Superscalar Processors

# Superscalar

## Overview

Term first coined in 1987

Refers to a machine that is designed to improve the performance of the execution of scalar instructions

In most applications the bulk of the operations are on scalar quantities

Represents the next step in the evolution of high-performance general-purpose processors

Essence of the approach is the ability to execute instructions independently and concurrently in different pipelines

Concept can be further exploited by allowing instructions to be executed in an order different from the program order

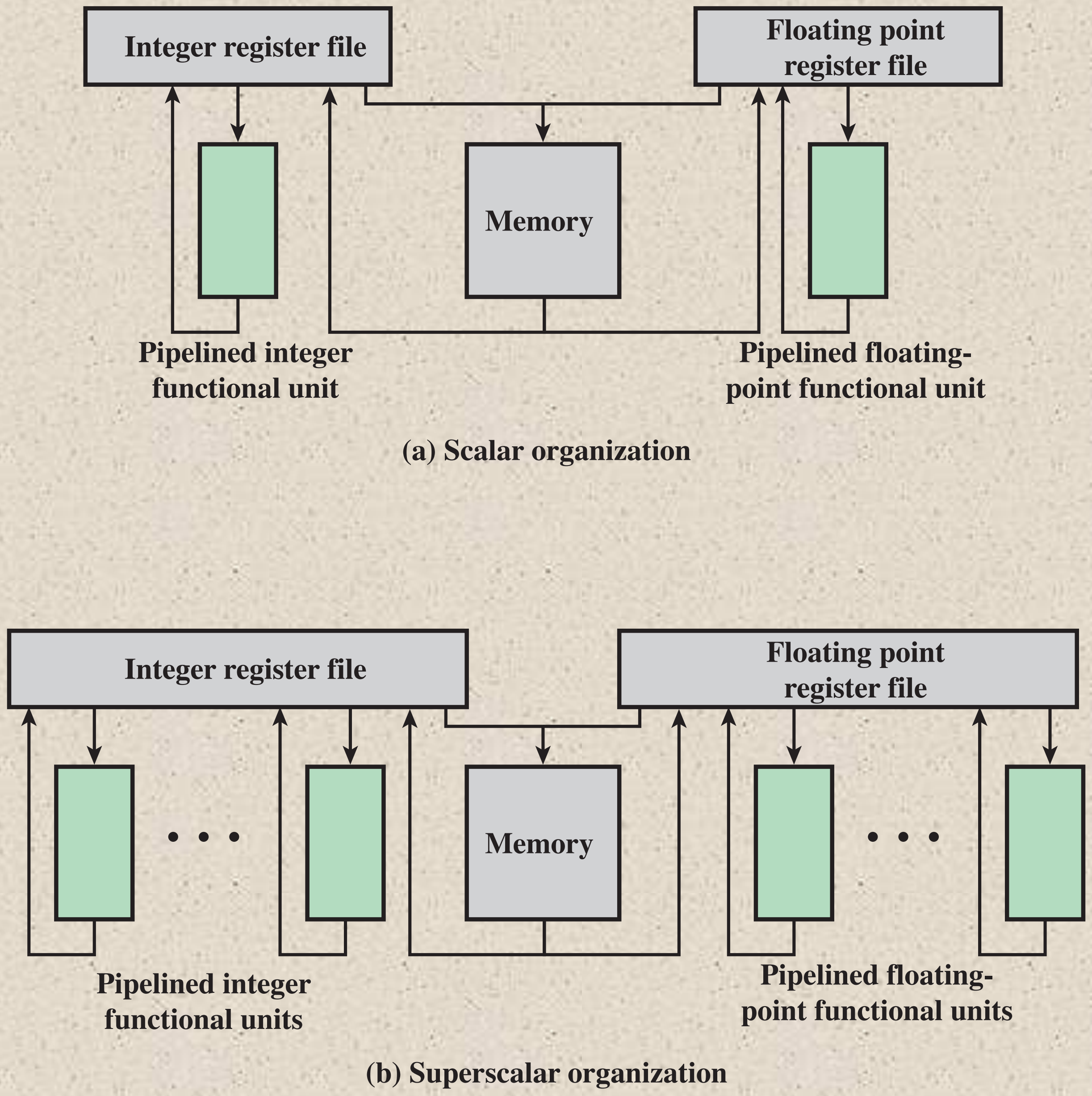

Integer register file
Floating point register file
Memory
Pipelined integer functional unit
Pipelined floating-point functional unit
(a) Scalar organization
Integer register file
Floating point register file
• • •
Memory
• • •
Pipelined integer functional units
Pipelined floating-point functional units
(b) Superscalar organization

**Figure 16.1 Superscalar Organization Compared to Ordinary Scalar Organization**

## Table 16.1 Reported Speedups of Superscalar-Like Machines

| Reference | Speedup |
|---|---|
| [TJAD70] | 1.8 |
| [KUCK77] | 8 |
| [WEIS84] | 1.58 |
| [ACOS86] | 2.7 |
| [SOHI90] | 1.8 |
| [SMIT89] | 2.3 |
| [JOUP89b] | 2.2 |
| [LEE91] | 7 |

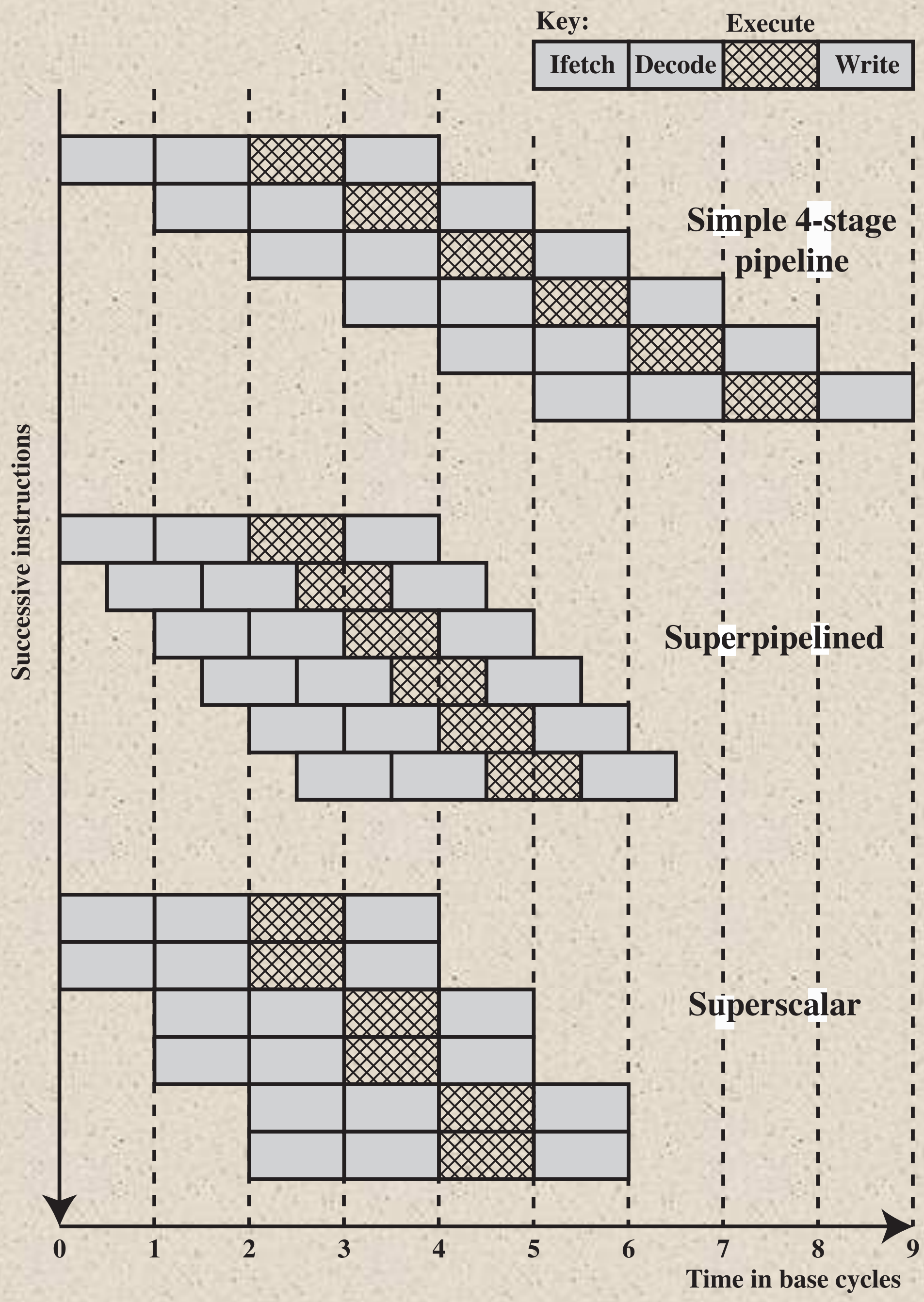


**Figure 16.2 Comparison of Superscalar and Superpipeline Approaches**

## Constraints

- Instruction level parallelism
  - Refers to the degree to which the instructions of a program can be executed in parallel
  - A combination of compiler based optimization and hardware techniques can be used to maximize instruction level parallelism
- Limitations:
  - True data dependency
  - Procedural dependency
  - Resource conflicts
  - Output dependency
  - Antidependency

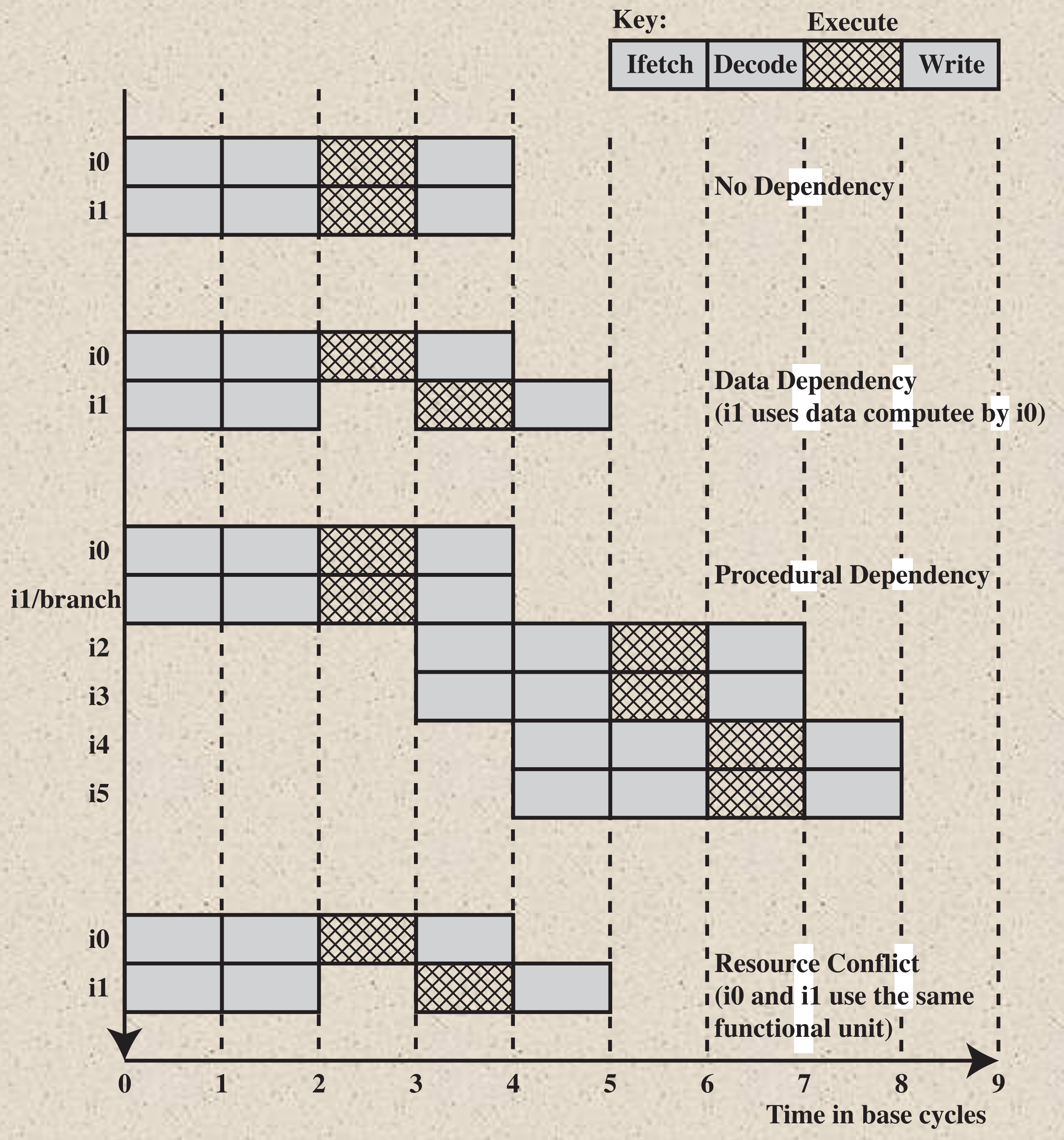


**Figure 16.3 Effect of Dependencies**

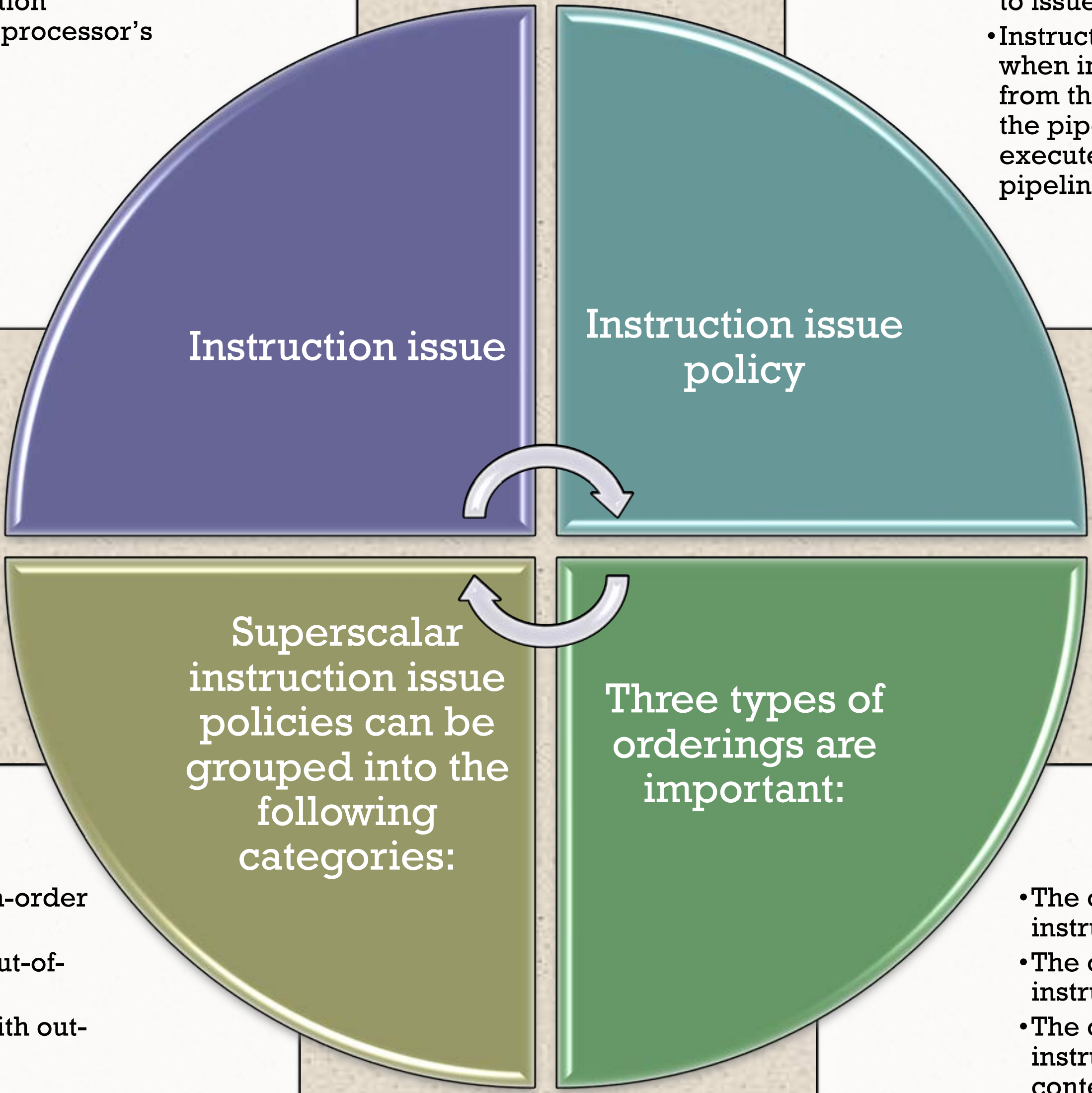
+

## Design Issues

### Instruction-Level Parallelism and Machine Parallelism

- Instruction level parallelism
  - Instructions in a sequence are independent
  - Execution can be overlapped
  - Governed by data and procedural dependency
- Machine Parallelism
  - Ability to take advantage of instruction level parallelism
  - Governed by number of parallel pipelines

# Instruction Issue Policy



Instruction issue
Instruction issue policy
Superscalar instruction issue policies can be grouped into the following categories:
Three types of orderings are important:

- Refers to the process of initiating instruction execution in the processor's functional units

- Refers to the protocol used to issue instructions
- Instruction issue occurs when instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline

- In-order issue with in-order completion
- In-order issue with out-of-order completion
- Out-of-order issue with out-of-order completion

- The order in which instructions are fetched
- The order in which instructions are executed
- The order in which instructions update the contents of register and memory locations

| Decode | | Execute | | | Write | | Cycle |
|---|---|---|---|---|---|---|---|
| I1 | I2 | | | | | | 1 |
| I3 | I4 | I1 | I2 | | | | 2 |
| I3 | I4 | I1 | | | | | 3 |
| | I4 | | | I3 | I1 | I2 | 4 |
| I5 | I6 | | | I4 | | | 5 |
| | I6 | | I5 | | I3 | I4 | 6 |
| | | | I6 | | | | 7 |
| | | | | | I5 | I6 | 8 |

**(a) In-order issue and in-order completion**

| Decode | | Execute | | | Write | | Cycle |
|---|---|---|---|---|---|---|---|
| I1 | I2 | | | | | | 1 |
| I3 | I4 | I1 | I2 | | | | 2 |
| | I4 | I1 | | I3 | I2 | | 3 |
| I5 | I6 | | | I4 | I1 | I3 | 4 |
| | I6 | | I5 | | I4 | | 5 |
| | | | I6 | | I5 | | 6 |
| | | | | | I6 | | 7 |

**(b) In-order issue and out-of-order completion**

| Decode | | Window | Execute | | | Write | | Cycle |
|---|---|---|---|---|---|---|---|---|
| I1 | I2 | | | | | | | 1 |
| I3 | I4 | *I1,I2* | I1 | I2 | | | | 2 |
| I5 | I6 | *I3,I4* | I1 | | I3 | I2 | | 3 |
| | | *I4,I5,I6* | | I6 | I4 | I1 | I3 | 4 |
| | | *I5* | | I5 | | I4 | I6 | 5 |
| | | | | | | I5 | | 6 |

**(c) Out-of-order issue and out-of-order completion**

**Figure 16.4 Superscalar Instruction Issue and Completion Policies**

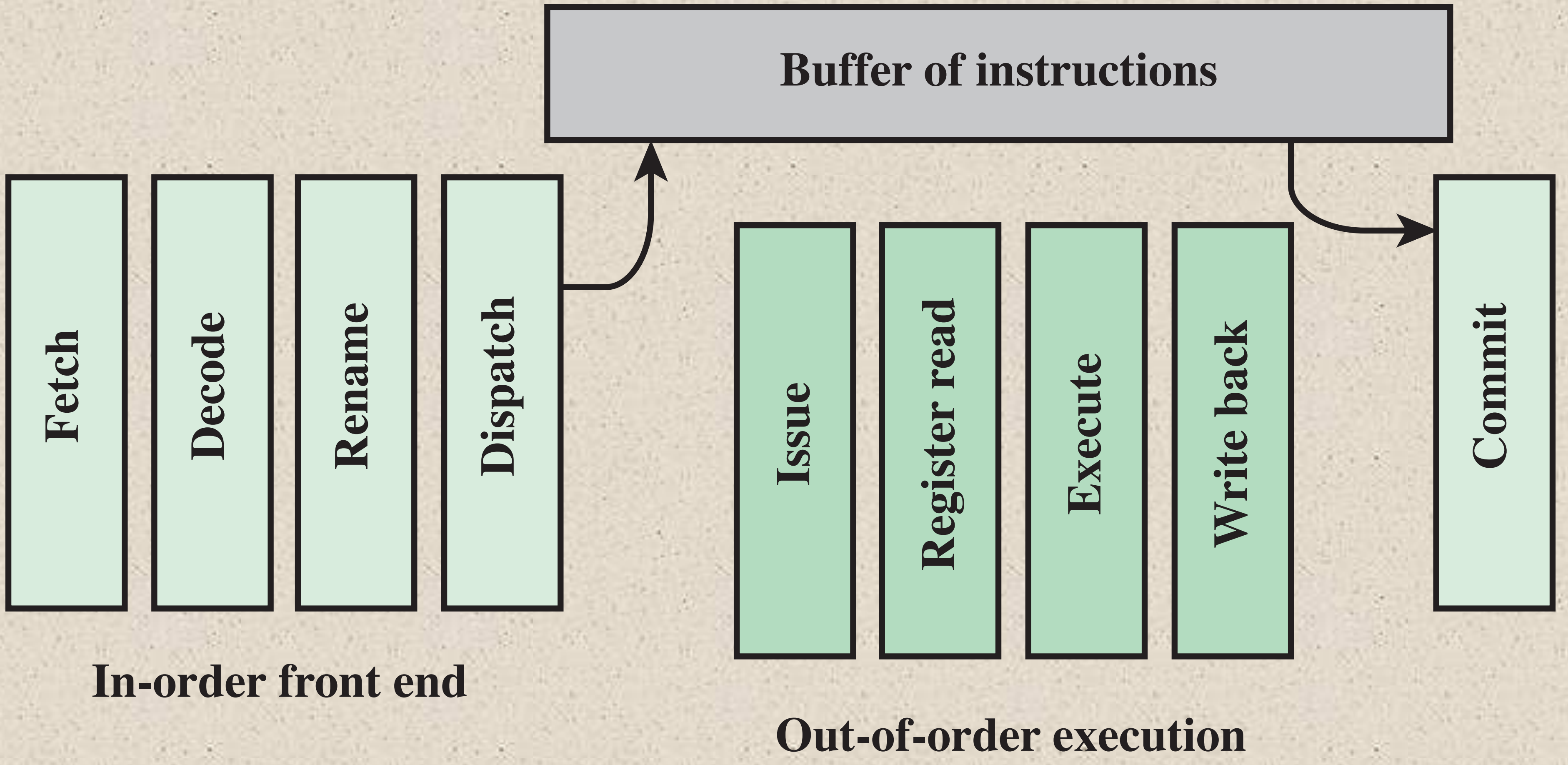


**Figure 16.5 Organization for Out-of-Order Issue with Out-of-Order Completion**

## Register Renaming

Output and antidependencies occur because register contents may not reflect the correct ordering from the program

May result in a pipeline stall

Registers allocated dynamically

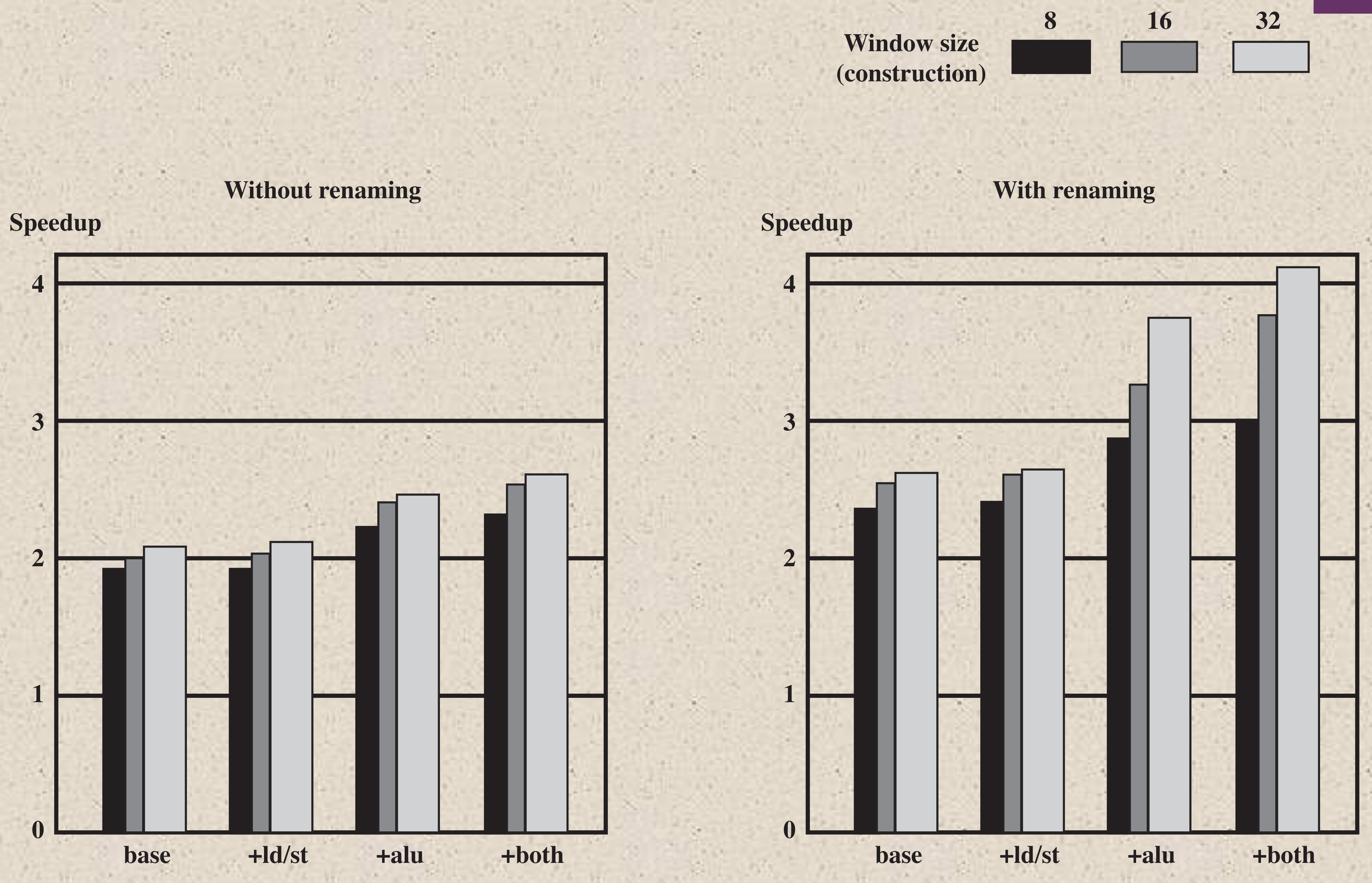
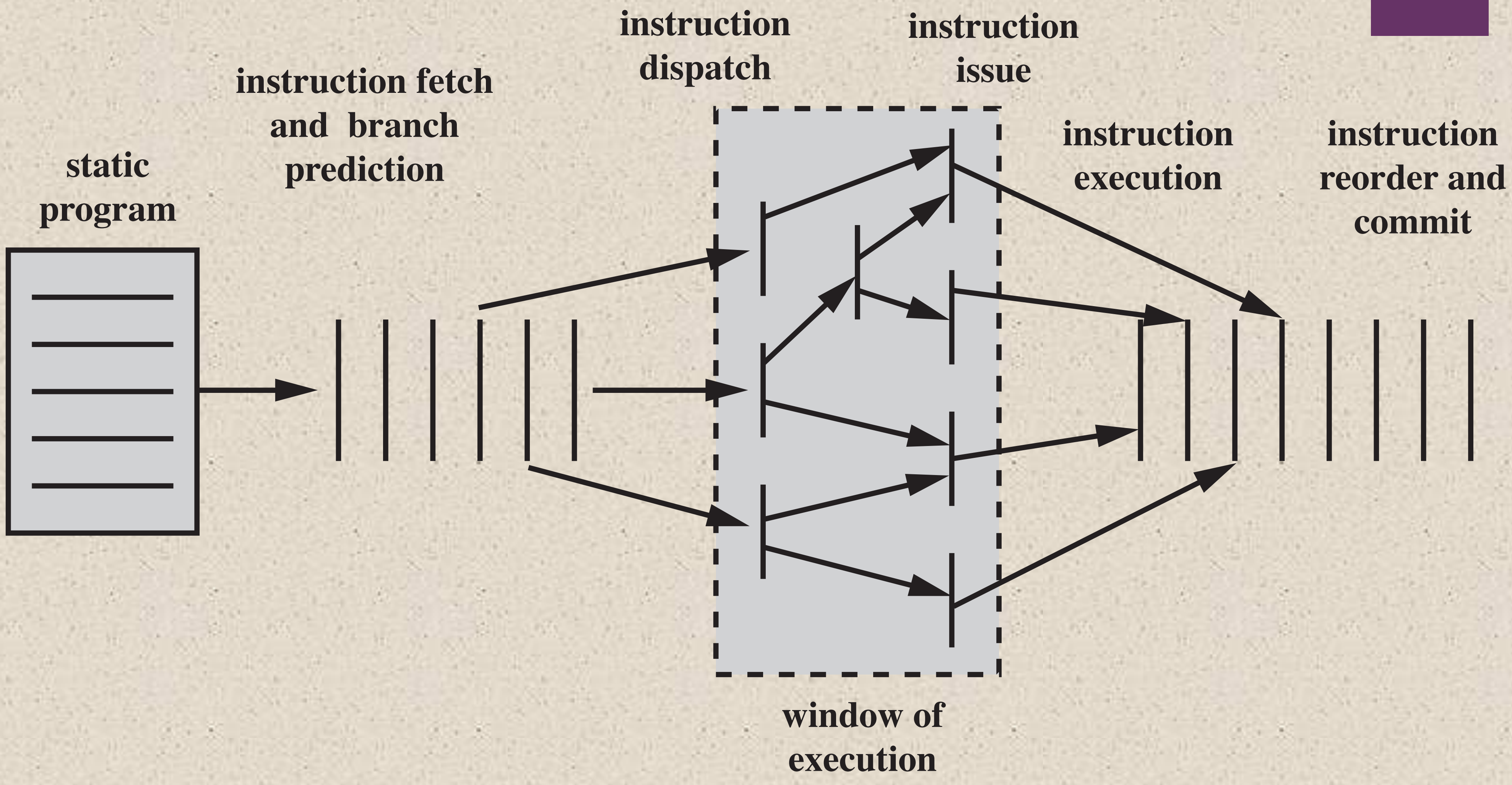


**Figure 16.6 Speedups of Various Machine Organizations Without Procedural Dependencies**

+ 
## Branch Prediction

- Any high-performance pipelined machine must address the issue of dealing with branches
- Intel 80486 addressed the problem by fetching both the next sequential instruction after a branch and speculatively fetching the branch target instruction
- RISC machines:
  - Delayed branch strategy was explored
  - Processor always executes the single instruction that immediately follows the branch
  - Keeps the pipeline full while the processor fetches a new instruction stream
- Superscalar machines:
  - Delayed branch strategy has less appeal
  - Have returned to pre-RISC techniques of branch prediction



## Figure 16.7 Conceptual Depiction of Superscalar Processing

## Superscalar Implementation

**Key elements:**

- Instruction fetch strategies that simultaneously fetch multiple instruction
- Logic for determining true dependencies involving register values, and mechanisms for communicating these values to where they are needed during execution
- Mechanisms for initiating, or issuing, multiple instructions in parallel
- Resources for parallel execution of multiple instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references
- Mechanisms for committing the process state in correct order

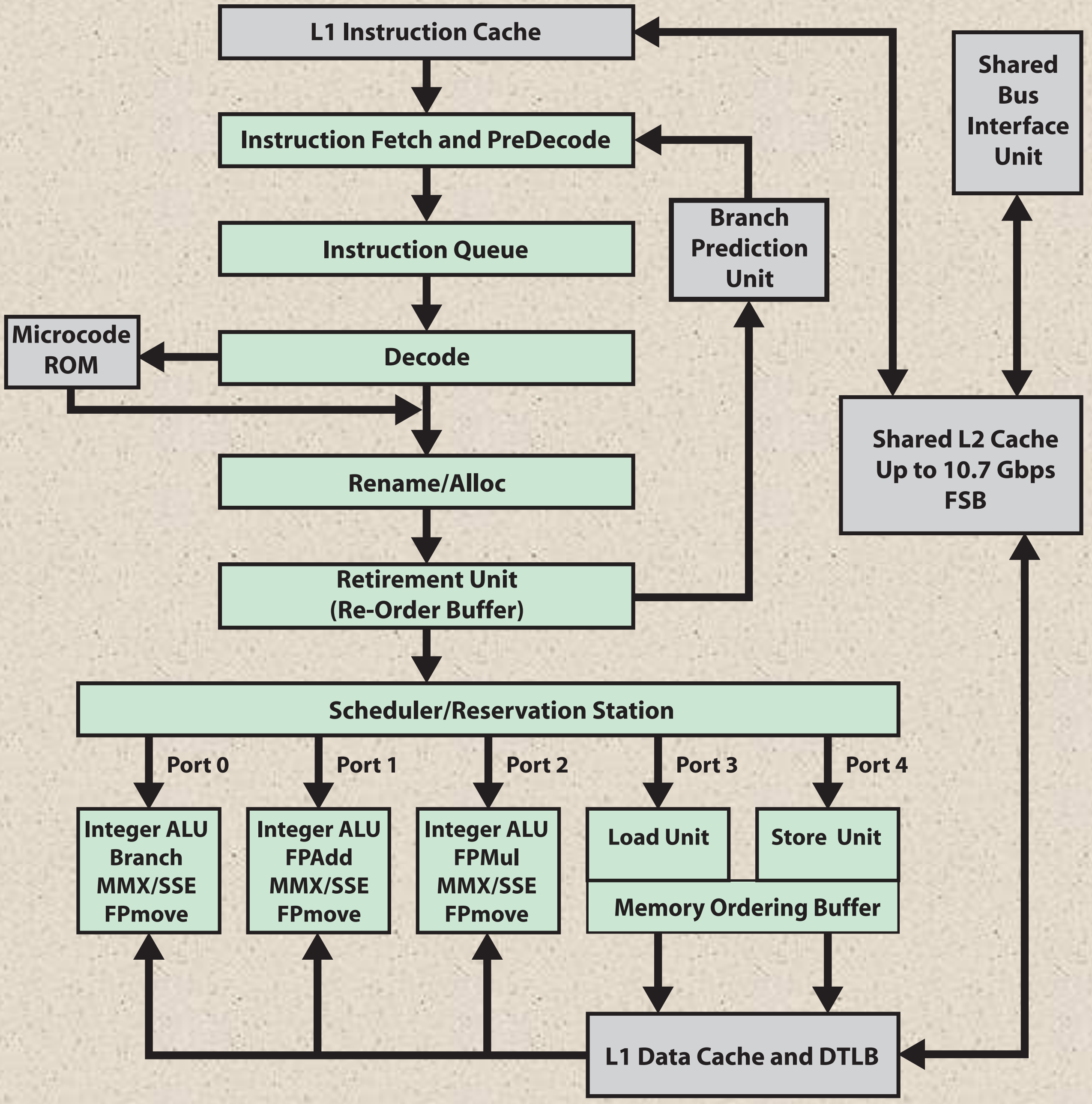

L1 Instruction Cache
Instruction Fetch and PreDecode
Instruction Queue
Decode
Microcode ROM
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Branch Prediction Unit
Shared Bus Interface Unit
Shared L2 Cache Up to 10.7 Gbps FSB
Scheduler/Reservation Station
Port 0
Port 1
Port 2
Port 3
Port 4
Integer ALU Branch MMX/SSE FPmove
Integer ALU FPAdd MMX/SSE FPmove
Integer ALU FPMul MMX/SSE FPmove
Load Unit
Store Unit
Memory Ordering Buffer
L1 Data Cache and DTLB

**Figure 16.8 Intel Core Microarchitecture**

## Table 16.2

## Cache/Memory Parameters and Performance of Processors Based on Intel Core Microarchitecture

**(a) Cache Parameters**

| Cache Level | Capacity | Associativity (ways) | Line Size (bytes) | Write Update Policy |
|---|---|---|---|---|
| L1 data | 32 kB | 8 | 64 | Writeback |
| L1 instruction | 32 kB | 8 | N/A | N/A |
| L2 (shared)[1] | 2, 4 MB | 8 or 16 | 64 | Writeback |
| L2 (shared)[2] | 3, 6 MB | 12 or 24 | 64 | Writeback |
| L3 (shared)[2] | 8, 12, 16 MB | 15 | 64 | Writeback |

*Notes:*
1. Intel Core Microarchitecture
2. Enhanced Intel Core Microarchitecture

**(b) Load/Store Performance**

| Data Locality | Load | | Store | |
|---|---|---|---|---|
| | Latency | Throughput | Latency | Throughput |
| L1 data cache | 3 clock cycles | 1 clock cycles | 2 clock cycles | 3 clock cycles |
| L1 data cache of the other core in modified state | 14 clock cycles + 5.5 bus cycles | 14 clock cycles + 5.5 bus cycles | 14 clock cycles + 5.5 bus cycles | N/A |
| L2 cache | 14 | 3 | 14 | 3 |
| Memory | 14 clock cycles + 5.5 bus cycles + memory latency | Depends on bus read protocol | 14 clock cycles + 5.5 bus cycles + memory latency | Depends on bus read protocol |

(Table can be found on page 592 in the textbook.)

## Front End

Consists of three major components:

- Branch prediction unit (BPU)
- Instruction fetch and predecode unit
- Instruction queue and decode unit

+
## Branch Prediction Unit

- Helps the instruction fetch unit fetch the most likely instruction to be executed by predicting the various branch types:
  - Conditional
  - Indirect
  - Direct
  - Call
  - Return
- Uses dedicated hardware for each branch type
- Enables the processor to begin executing instructions long before the branch outcome is decided
- A branch target buffer (BTB) is maintained that caches information about recently encountered branch instructions

## Instruction Fetch and Predecode Unit

**Comprises:**

- The instruction translation lookaside buffer (ITLB)
- An instruction prefetcher
- The instruction cache
- The predecode logic

**The predecode unit accepts the sixteen bytes from the instruction cache or prefetch buffers and carries out the following tasks:**

- Determine the length of the instructions
- Decode all prefixes associated with instructions
- Mark various properties of instruction for the decoders

**Predecode unit can write up to six instructions per cycle into the instruction queue**

- If a fetch contains more than six instructions, the predecoder continues to decode up to six instructions per cycle until all instruction in the fetch are written to the instruction queue
- Subsequent fetches can only enter predecoding after the current fetch completes

## Instruction Queue and Decode Unit

- Fetched instructions are placed in an instruction queue
  - From there the decode unit scans the bytes to determine instruction boundaries
  - The decoder translates each machine instruction into from one to four micro-ops
    - Each of which is a 118-bit RISC instruction
- A few instructions require more than four micro-ops so they are transferred to microcode ROM, which contains the series of micro-ops (five or more) associated with a complex machine instruction
- The resulting micro-op sequence is delivered to the rename/allocator module

## Out-of-Order Execution Logic

- This part of the processor reorders micro-ops to allow them to execute as quickly as their input operands are ready
- Allocate stage
  - Allocates resources required for execution
  - Performs the following functions:
    - If a needed resource is unavailable for one of the three micro-ops arriving at the allocator during a clock cycle, the allocator stalls the pipeline
    - Allocates a reorder buffer (ROB) entry which tracks the completion status of one of the 126 micro-ops that could be in process at any time
    - Allocates one of the 128 integer or floating-point register entries for the result data value of the micro-op, and possibly a load or store buffer used to track one of the 48 loads or 24 stores in the machine pipeline
    - Allocates an entry in one of the two micro-op queues in front of the instruction schedulers

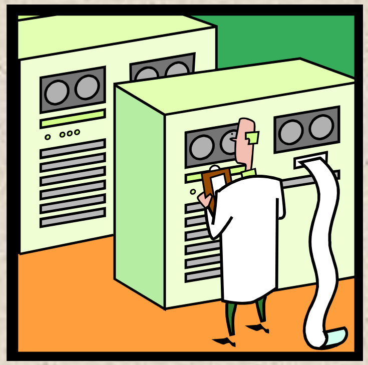
## Reorder Buffer (ROB)

Circular buffer that can hold up to 126 micro-ops and also contains the 128 hardware registers

Each buffer entry consists of the following fields:

- **State**
  - Indicates whether this micro-op is scheduled for execution, has been dispatched for execution, or has completed execution and is ready for retirement
- **Memory address**
  - The address of the Pentium instruction that generated the micro-op
- **Micro-op**
  - The actual operation
- **Alias register**
  - If the micro-op references one of the 16 architectural registers, this entry redirects that reference to one of the 128 hardware registers

+



- Register renaming
  - The rename stage remaps references to the 16 architectural registers into a set of 128 physical registers
- Micro-op queuing
  - After resource allocation and register renaming, micro-ops are placed in one of two micro-op queues, where they are held until there is room in the schedulers
- Micro-op scheduling and dispatching
  - Schedulers are responsible for retrieving micro-ops from the micro-op queues and dispatching these for execution
- Integer and floating-point execution units
  - The execution units retrieve values from the register files as well as from the L1 data cache

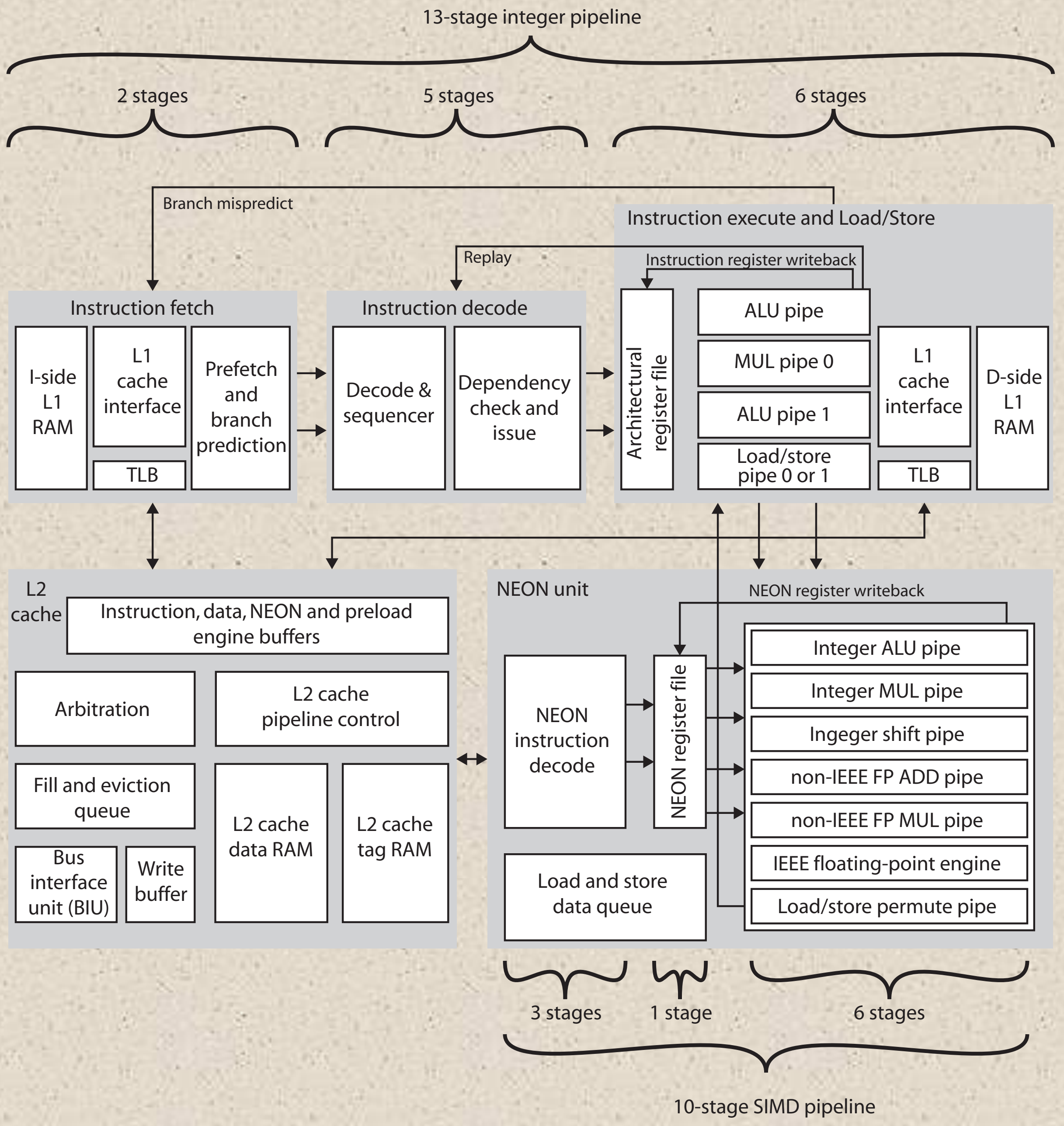
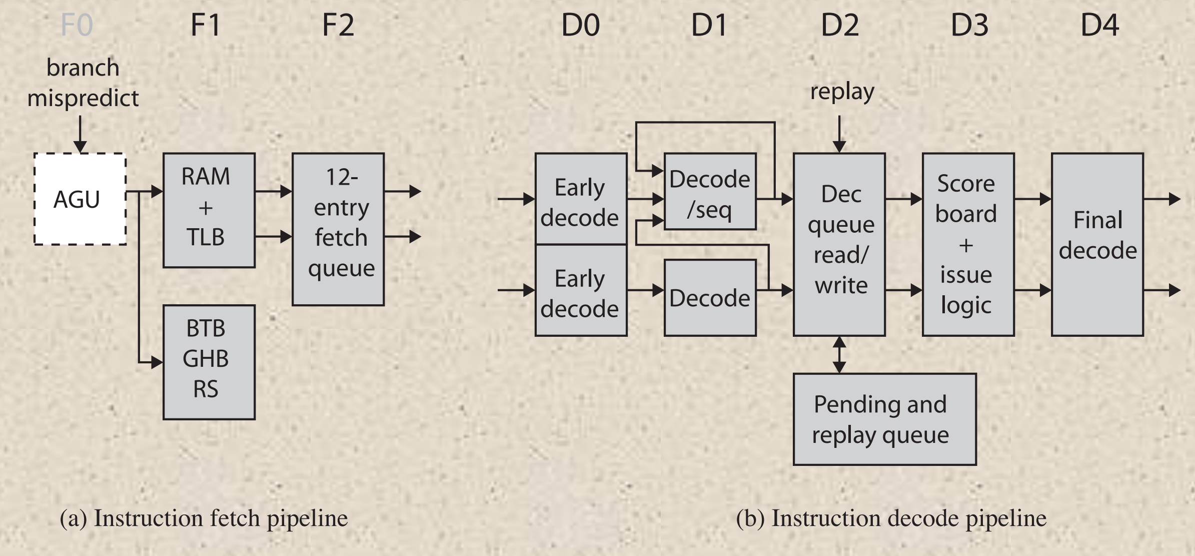


**Figure 16.9 Architectural Block Diagram of ARM Cortex-A8**



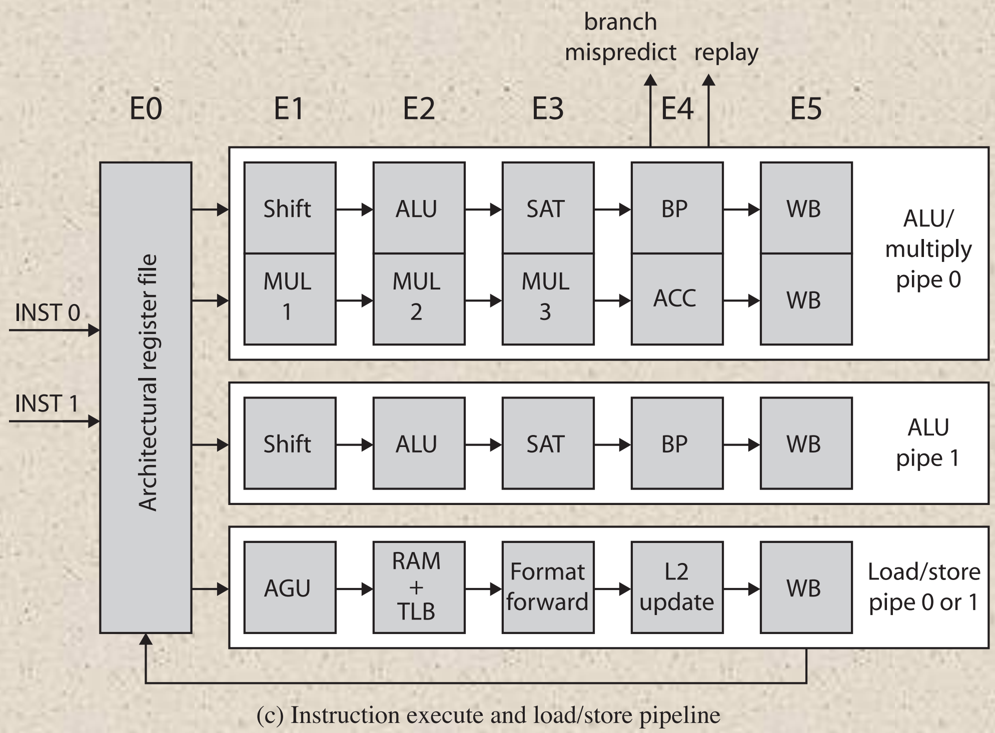


**Figure 16.10 ARM Cortex-A8 Integer Pipeline**

## Instruction Fetch Unit

- Predicts instruction stream
- Fetches instructions from the L1 instruction cache
- Places the fetched instructions into a buffer for consumption by the decode pipeline
- Also includes the L1 instruction cache
- Speculative (there is no guarantee that they are executed)
- Branch or exceptional instruction in the code stream can cause a pipeline flush
- Can fetch up to four instructions per cycle

- F0
  - Address generation unit (AGU) generates a new virtual address
  - Not counted as part of the 13-stage pipeline
- F1
  - The calculated address is used to fetch instructions from the L1 instruction cache
  - In parallel, the fetch address is used to access branch prediction arrays
- F3
  - Instruction data are placed in the instruction queue
  - If an instruction results in branch prediction, new target address is sent to the address generation unit

+

## Instruction Decode Unit

- Decodes and sequences all ARM and Thumb instructions
- Dual pipeline structure, *pipe0* and *pipe1*
  - Two instructions can progress at a time
  - Pipe0 contains the older instruction in program order
  - If instruction in pipe0 cannot issue, instruction in pipe1 will not issue
- All issued instructions progress in order
- Results written back to register file at end of execution pipeline
  - Prevents WAR hazards
  - Keeps track of WAW hazards and recovery from flush conditions straightforward
- Main concern of decode pipeline is prevention of RAW hazards

## Instruction Processing Stages

D0

- Thumb instructions decompressed and preliminary decode is performed

D1

- Instruction decode is completed

D2

- Writes instructions into and read instructions from pending/replay queue

D3

- Contains the instruction scheduling logic
- Scoreboard predicts register availability using static scheduling
- Hazard checking is done

D4

- Final decode for control signals for integer execute load/store units

## Table 16.3
## Cortex-A8 Memory System Effects on Instruction Timings

| Replay event | Delay | Description |
|---|---|---|
| Load data miss | 8 cycles | **1.** A load instruction misses in the L1 data cache.<br>**2.** A request is then made to the L2 data cache.<br>**3.** If a miss also occurs in the L2 data cache, then a second replay occurs. The number of stall cycles depends on the external system memory timing. The minimum time required to receive the critical word for an L2 cache miss is approximately 25 cycles, but can be much longer because of L3 memory latencies. |
| Data TLB miss | 24 cycles | **1.** A table walk because of a miss in the L1 TLB causes a 24-cycle delay, assuming the translation table entries are found in the L2 cache.<br>**2.** If the translation table entries are not present in the L2 cache, the number of stall cycles depends on the external system memory timing. |
| Store buffer full | 8 cycles plus latency to drain fill buffer | **1.** A store instruction miss does not result in any stalls unless the store buffer is full.<br>**2.** In the case of a full store buffer, the delay is at least eight cycles. The delay can be more if it takes longer to drain some entries from the store buffer. |
| Unaligned load or store request | 8 cycles | **1.** If a load instruction address is unaligned and the full access is not contained within a 128-bit boundary, there is a 8-cycle penalty.<br>**2.** If a store instruction address is unaligned and the full access is not contained within a 64-bit boundary, there is a 8-cycle penalty. |

(Table can be found on page 601 in the textbook.)

# Table 16.4
# Cortex-A8 Dual-Issue Restrictions

| Restriction type | Description | Example | Cycle | Restriction |
|---|---|---|---|---|
| Load/store resource hazard | There is only one LS pipeline. Only one LS instruction can be issued per cycle. It can be in pipeline 0 or pipeline 1 | LDR r5, [r6]<br>STR r7, [r8]<br>MOV r9, r10 | 1<br>2<br>2 | <br>Wait for LS unit<br>Dual issue possible |
| Multiply resource hazard | There is only one multiply pipeline, and it is only available in pipeline 0. | ADD r1, r2, r3<br>MUL r4, r5, r6<br>MUL r7, r8, r9 | 1<br>2<br>3 | <br>Wait for pipeline 0<br>Wait for multiply unit |
| Branch resource hazard | There can be only one branch per cycle. It can be in pipeline 0 or pipeline 1. A branch is any instruction that changes the PC. | BX r1<br>BEQ 0x1000<br>ADD r1, r2, r3 | 1<br>2<br>2 | <br>Wait for branch<br>Dual issue possible |
| Data output hazard | Instructions with the same destination cannot be issued in the same cycle. This can happen with conditional code. | MOVEQ r1, r2<br>MOVNE r1, r3<br>LDR r5, [r6] | 1<br>2<br>2 | <br>Wait because of output dependency<br>Dual issue possible |
| Data source hazard | Instructions cannot be issued if their data is not available. See the scheduling tables for source requirements and stages results. | ADD r1, r2, r3<br>ADD r4, r1, r6<br>LDR r7, [r4] | 1<br>2<br>4 | <br>Wait for r1<br>Wait two cycles for r4 |
| Multi-cycle instructions | Multi-cycle instructions must issue in pipeline 0 and can only dual issue in their last iteration. | MOV r1, r2<br>LDM r3, {r4-r7}<br>LDM (cycle 2)<br>LDM (cycle 3)<br>ADD r8, r9, r10 | 1<br>2<br>3<br>4<br>4 | Wait for pipeline 0, transfer r4<br>Transfer r5, r6<br>Transfer r7<br>Dual issue possible on last transfer |

(Table can be found on page 602 in the textbook.)

# Integer Execute Unit

- Consists of:
  - Two symmetric arithmetic logic unit (ALU) pipelines
  - An address generator for load and store instructions
  - The multiply pipeline
- The instruction execute unit:
  - Executes all integer ALU and multiply operations, including flag generation
  - Generates the virtual addresses for loads and stores and the base write-back value, when required
  - Supplies formatted data for stores and forwards data and flags
  - Processes branches and other changes of instruction stream and evaluates instruction condition codes

- For ALU instructions, either pipeline can be used, consisting of the following stages:
  - E0
    - Access register file
    - Up to six registers for two instructions
  - E1
    - Barrel shifter if needed.
  - E2
    - ALU function
  - E3
    - If needed, completes saturation arithmetic
  - E4
    - Change in control flow prioritized and processed
  - E5
    - Results written back to register file

## Load/Store Pipeline

- Runs parallel to integer pipeline
- E1
  - Memory address generated from base and index register
- E2
  - Address applied to cache arrays
- E3
  - Load -- data are returned and formatted
  - Store -- data are formatted and ready to be written to cache
- E4
  - Updates L2 cache, if required
- E5
  - Results are written back into the register file

| Cycle | Program Counter | Instruction | Timing Description |
|---|---|---|---|
| 1 | 0x00000ed0 | BX r14 | Dual issue pipeline 0 |
| 1 | 0x00000ee4 | CMP r0,#0 | Dual issue in pipeline 1 |
| 2 | 0x00000ee8 | MOV r3,#3 | Dual issue pipeline 0 |
| 2 | 0x00000eec | MOV r0,#0 | Dual issue in pipeline 1 |
| 3 | 0x00000ef0 | STREQ r3,[r1,#0] | Dual issue in pipeline 0, r3 not needed until E3 |
| 3 | 0x00000ef4 | CMP r2,#4 | Dual issue in pipeline 1 |
| 4 | 0x00000ef8 | LDRLS pc,[pc,r2,LSL #2] | Single issue pipeline 0, +1 cycle for load to pc, no extra cycle for shift since LSL #2 |
| 5 | 0x00000f2c | MOV r0,#1 | Dual issue with 2nd iteration of load in pipeline 1 |
| 6 | 0x00000f30 | B {pc}+8 | #0xf38 dual issue pipeline 0 |
| 6 | 0x00000f38 | STR r0,[r1,#0] | Dual issue pipeline 1 |
| 7 | 0x00000f3c: | LDR pc,[r13],#4 | Single issue pipeline 0, +1 cycle for load to pc |
| 8 | 0x0000017c | ADD r2,r4,#0xc | Dual issue with 2nd iteration of load in pipeline 1 |
| 9 | 0x00000180 | LDR r0,[r6,#4] | Dual issue pipeline 0 |
| 9 | 0x00000184 | MOV r1,#0xa | Dual issue pipeline 1 |
| 12 | 0x00000188 | LDR r0,[r0,#0] | Single issue pipeline 0: r0 produced in E3, required in E1, so +2 cycle stall |
| 13 | 0x0000018c | STR r0,[r4,#0] | Single issue pipeline 0 due to LS resource hazard, no extra delay for r0 since produced in E3 and consumed in E3 |
| 14 | 0x00000190 | LDR r0,[r4,#0xc] | Single issue pipeline 0 due to LS resource hazard |
| 15 | 0x00000194 | LDMFD r13!,{r4-r6,r14} | Load multiple: loads r4 in 1st cycle, r5 and r6 in 2nd cycle, r14 in 3rd cycle, 3 cycles total |
| 17 | 0x00000198 | B {pc}+0xda8 | #0xf40 dual issue in pipeline 1 with 3rd cycle of LDM |
| 18 | 0x00000f40 | ADD r0,r0,#2 ARM | Single issue in pipeline 0 |
| 19 | 0x00000f44 | ADD r0,r1,r0 ARM | Single issue in pipeline 0, no dual issue due to hazard on r0 produced in E2 and required in E2 |

## Table 16.5

## Cortex-A8 Example Dual Issue Instruction Sequence for Integer Pipeline

(Table can be found on page 603 in the textbook.)

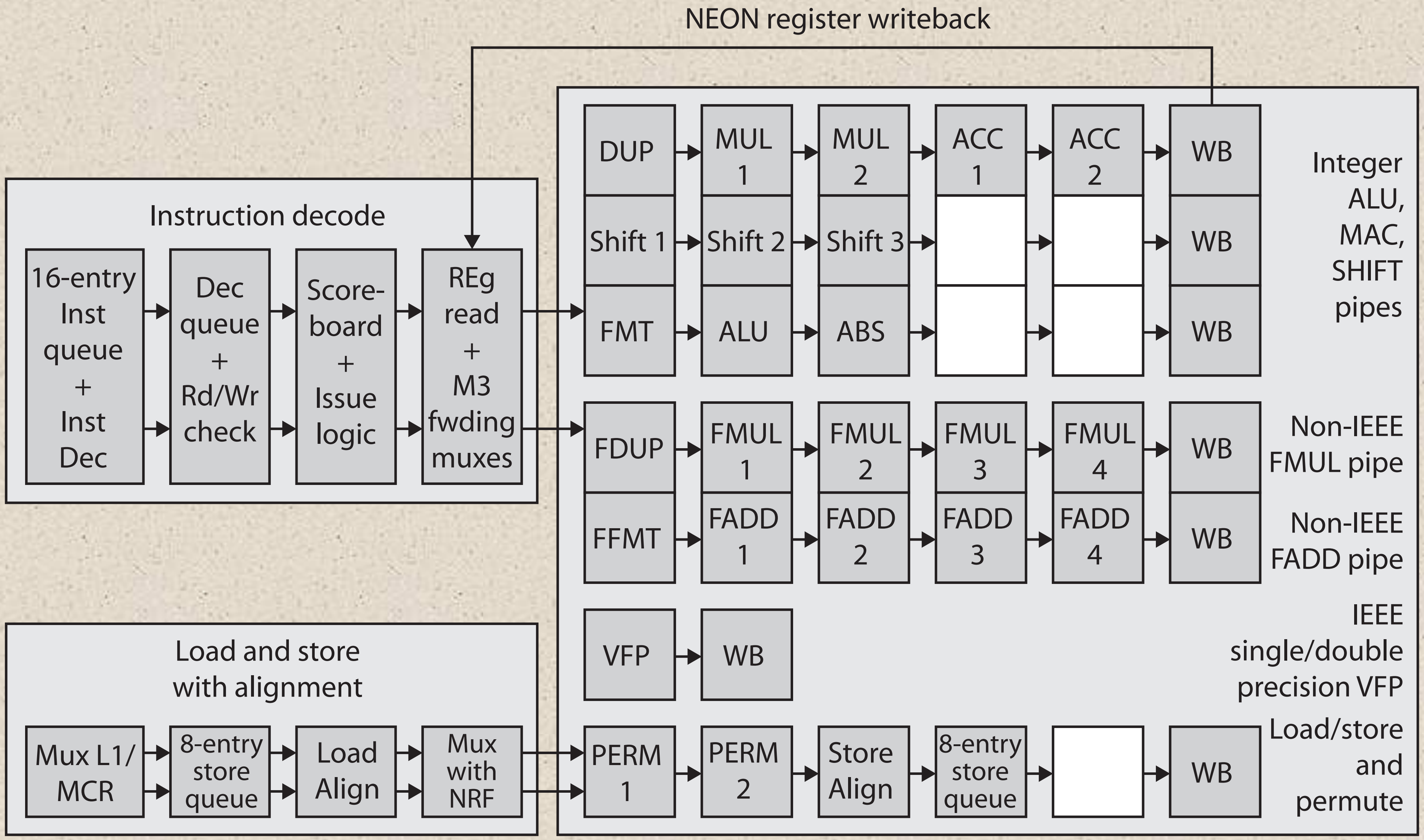


**Figure 16.11 ARM Cortex-A8 NEON and Floating-Point Pipeline**

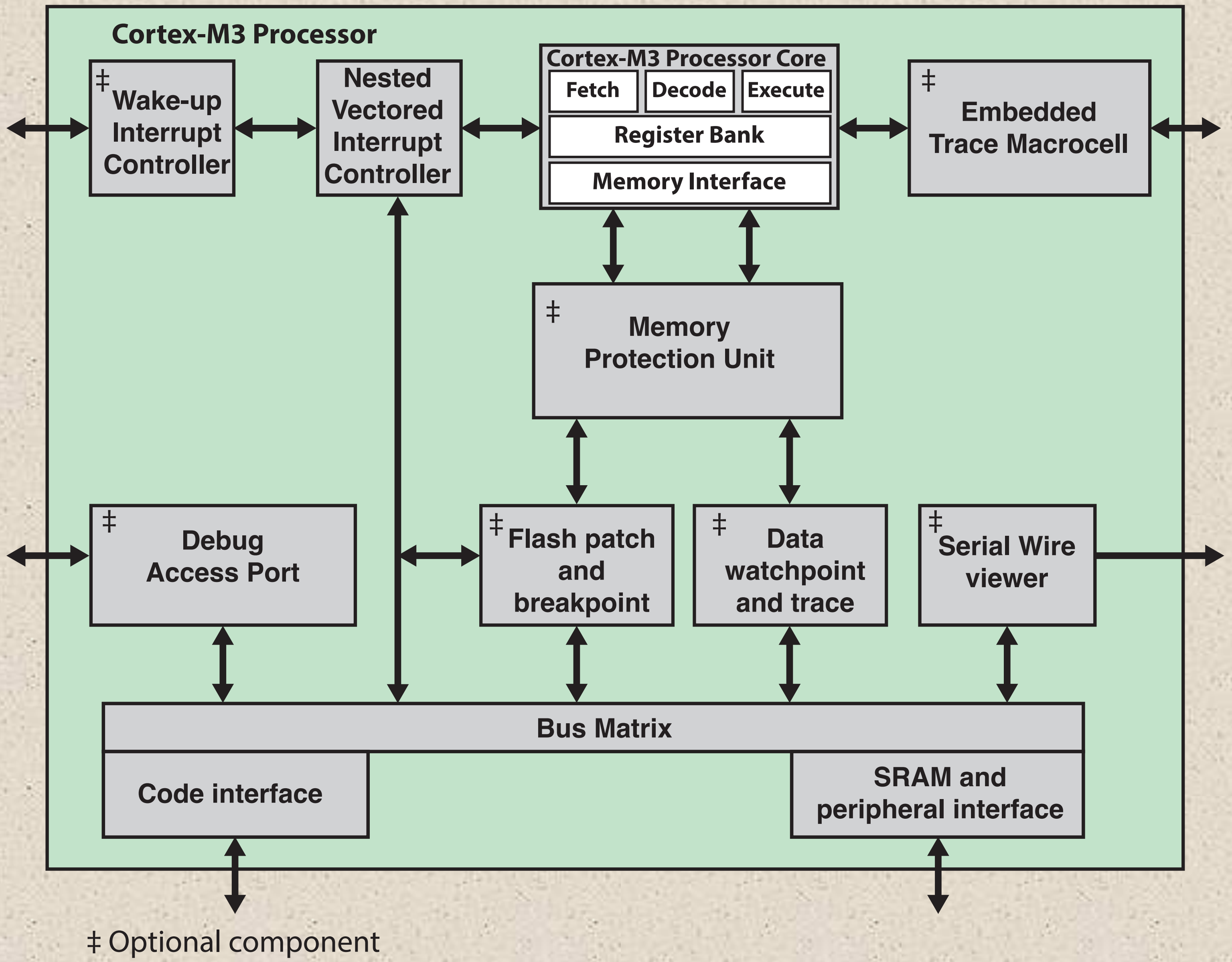
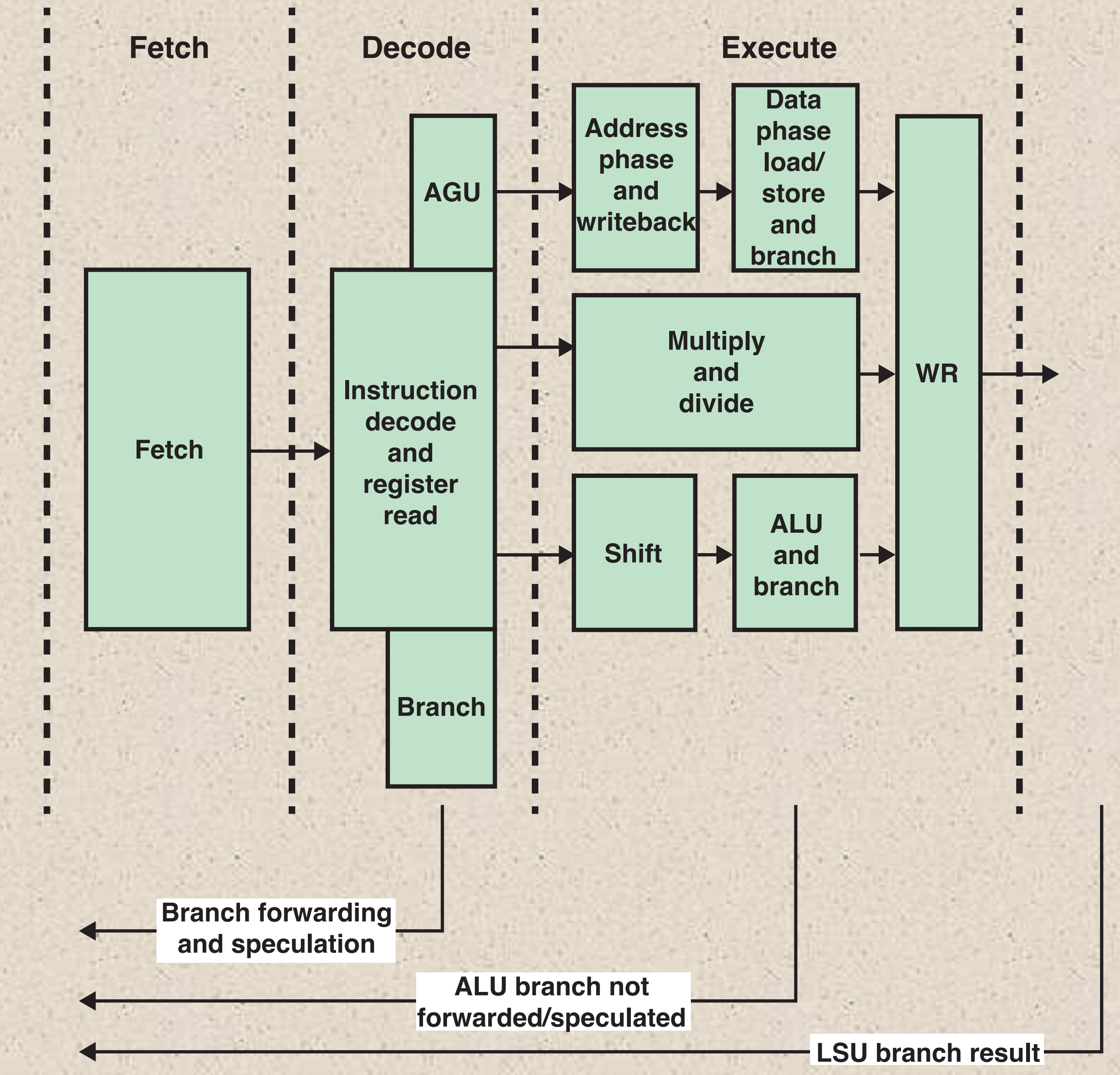


**Figure 16.12 ARM Cortex-M3 Block Diagram**


Fetch
Decode
Execute
Fetch
AGU
Instruction decode and register read
Branch
Address phase and writeback
Data phase load/ store and branch
Multiply and divide
Shift
ALU and branch
WR
Branch forwarding and speculation
ALU branch not forwarded/speculated
LSU branch result

AGU = address generation unit

Figure 16.13 ARM Cortex-M3 Pipeline

+ Summary

# Chapter 16

## Instruction-Level Parallelism and Superscalar Processors

- Superscalar versus Superpipelined
  - Constraints
- Design issues
  - Instruction-level parallelism
  - Machine parallelism
  - Instruction issue policy
  - Register renaming
  - Branch prediction
  - Superscalar execution
  - Superscalar implementation
- Intel core microarchitecture
  - Front end
  - Out-of-order execution logic
  - Integer and floating-point execution units
- ARM Cortex-A8
  - Instruction fetch unit
  - Instruction decode unit
  - Integer execute unit
  - SIMD and floating-point pipeline
- ARM Cortex-M3
  - Pipeline structure
  - Dealing with branches