

William Stallings
Computer Organization
and Architecture
8th Edition

Chapter 14
Instruction Level Parallelism
and Superscalar Processors

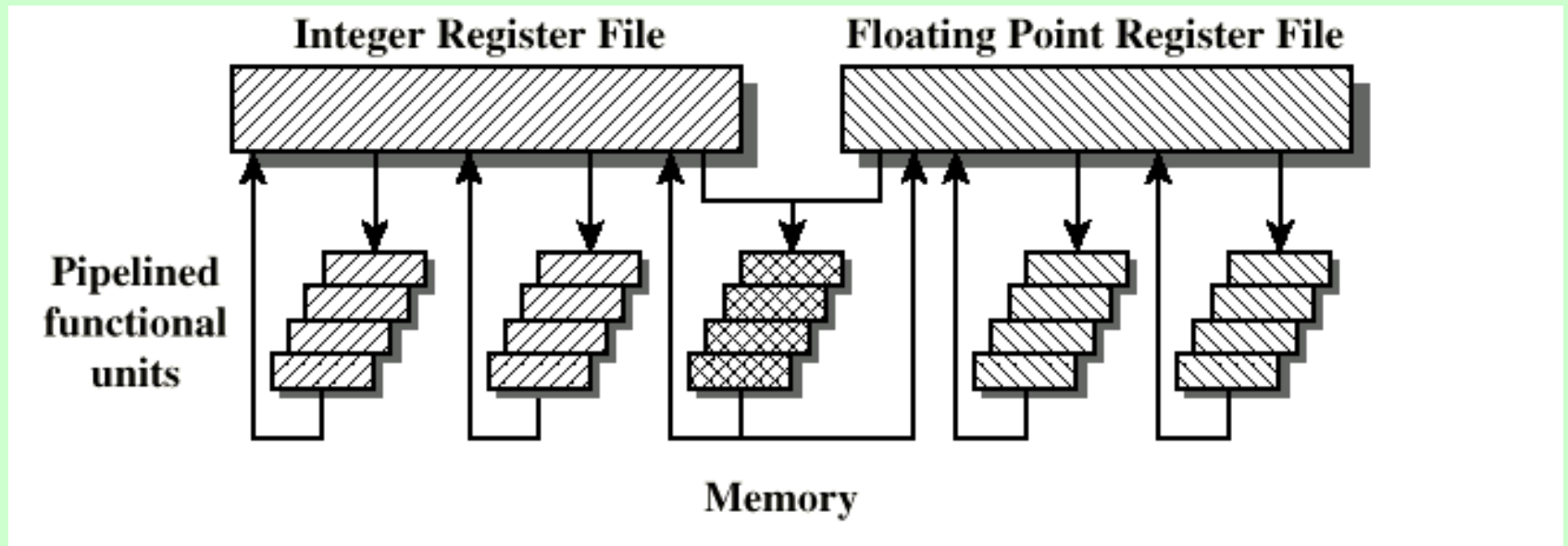
What is Superscalar?

- Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently
- Equally applicable to RISC & CISC
- In practice usually RISC

Why Superscalar?

- Most operations are on scalar quantities (see RISC notes)
- Improve these operations to get an overall improvement

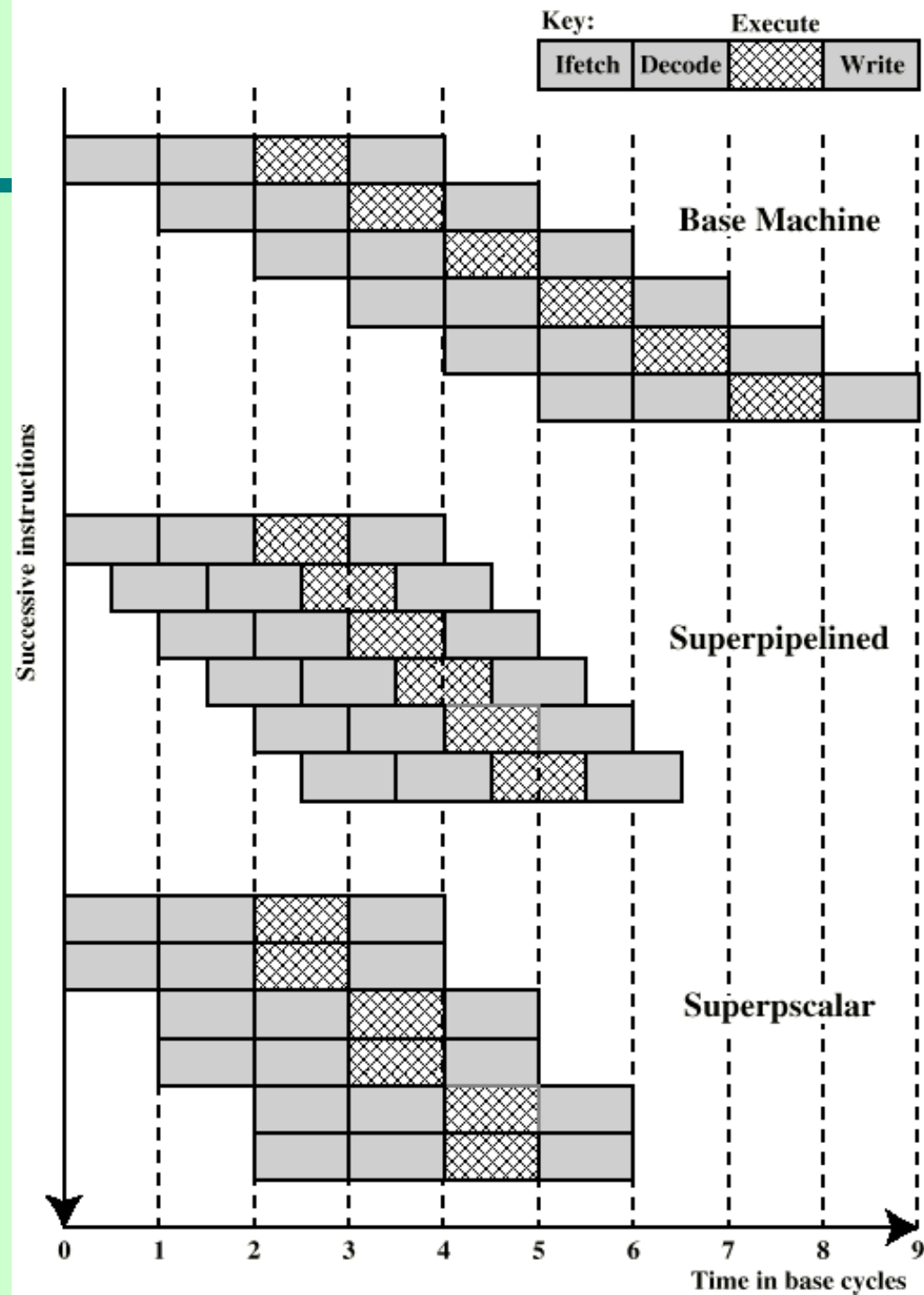
General Superscalar Organization



Superpipelined

- Many pipeline stages need less than half a clock cycle
- Double internal clock speed gets two tasks per external clock cycle
- Superscalar allows parallel fetch execute

Superscalar v Superpipeline



Limitations

- Instruction level parallelism
- Compiler based optimisation
- Hardware techniques
- Limited by
 - True data dependency
 - Procedural dependency
 - Resource conflicts
 - Output dependency
 - Antidependency

True Data Dependency

- ADD r1, r2 (r1 := r1+r2;)
- MOVE r3,r1 (r3 := r1;)
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

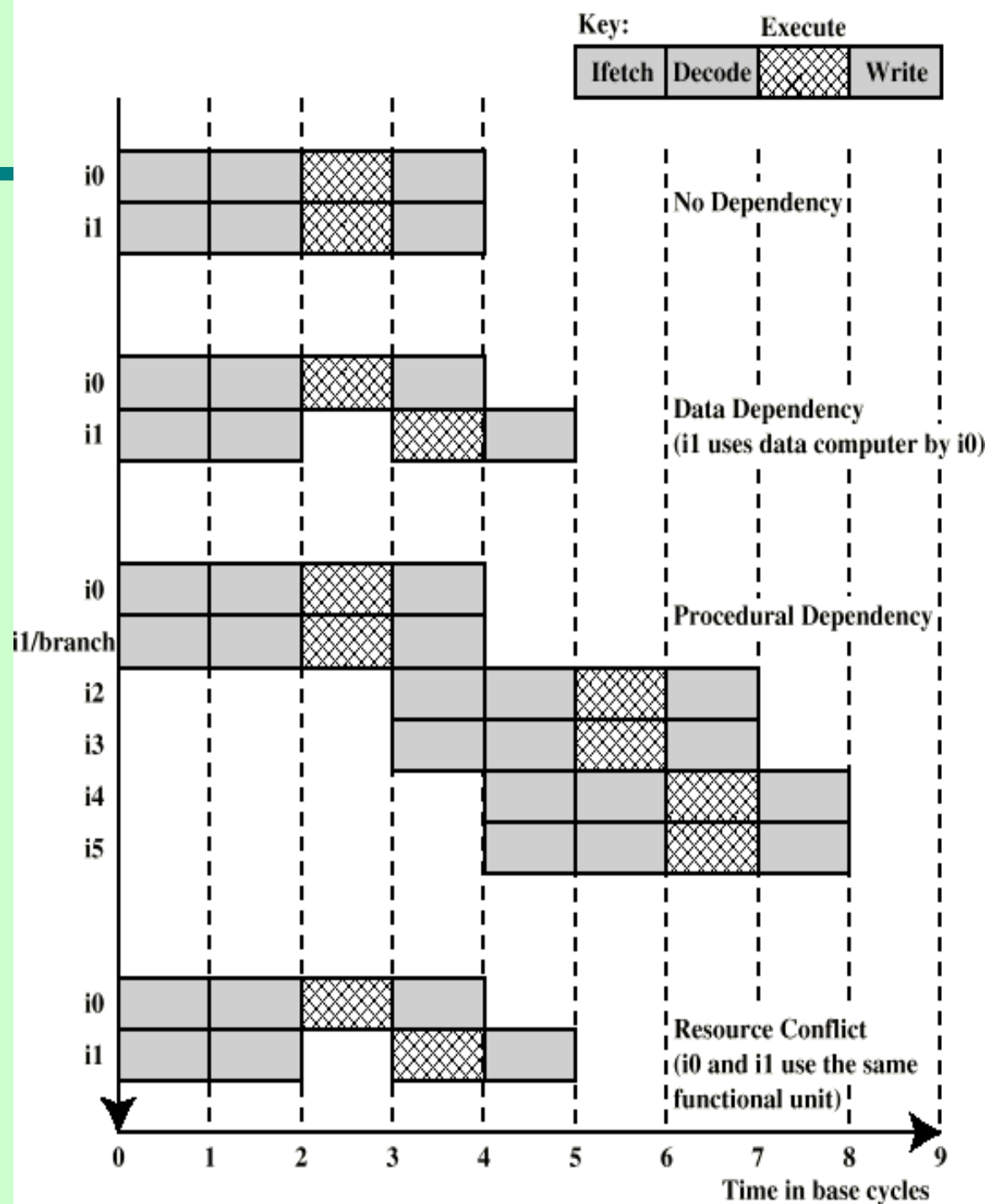
Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch
- Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
 - e.g. two arithmetic instructions
- Can duplicate resources
 - e.g. have two arithmetic units

Effect of Dependencies



Design Issues

- 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

- Order in which instructions are fetched
- Order in which instructions are executed
- Order in which instructions change registers and memory

In-Order Issue

In-Order Completion

- Issue instructions in the order they occur
- Not very efficient
- May fetch >1 instruction
- Instructions must stall if necessary

In-Order Issue In-Order Completion (Diagram)

Decode		Execute			Write		Cycle
11	12						1
13	14	11	12				2
13	14	11					3
	14			13	11	12	4
15	16			14			5
	16		15		13	14	6
			16				7
					15	16	8

In-Order Issue

Out-of-Order Completion

- Output dependency
 - $R3 := R3 + R5$; (I1)
 - $R4 := R3 + 1$; (I2)
 - $R3 := R5 + 1$; (I3)
 - I2 depends on result of I1 - data dependency
 - If I3 completes before I1, the result from I1 will be wrong - output (read-write) dependency

In-Order Issue Out-of-Order Completion (Diagram)

Decode		Execute			Write		Cycle
11	12						1
13	14	11	12				2
	14	11		13	12		3
15	16			14	11	13	4
	16		15		14		5
			16		15		6
					16		7

Out-of-Order Issue

Out-of-Order Completion

- Decouple decode pipeline from execution pipeline
- Can continue to fetch and decode until this pipeline is full
- When a functional unit becomes available an instruction can be executed
- Since instructions have been decoded, processor can look ahead

Out-of-Order Issue Out-of-Order Completion (Diagram)

Decode		Window	Execute			Write		Cycle
11	12							1
13	14	11,12	11	12				2
15	16	13,14	11		13	12		3
		14,15,16		16	14	11	13	4
		15		15		14	16	5
						15		6

Antidependency

- Write-write dependency
 - $R3 := R3 + R5;$ (I1)
 - $R4 := R3 + 1;$ (I2)
 - $R3 := R5 + 1;$ (I3)
 - $R7 := R3 + R4;$ (I4)
 - I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3

Reorder Buffer

- Temporary storage for results
- Commit to register file in program order

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
 - i.e. registers are not specifically named

Register Renaming example




- $R3b := R3a + R5a$ (I1)
- $R4b := R3b + 1$ (I2)
- $R3c := R5a + 1$ (I3)
- $R7b := R3c + R4b$ (I4)
- Without subscript refers to logical register in instruction
- With subscript is hardware register allocated
- Note R3a R3b R3c
- Alternative: Scoreboarding
 - Bookkeeping technique
 - Allow instruction execution whenever not dependent on previous instructions and no structural hazards

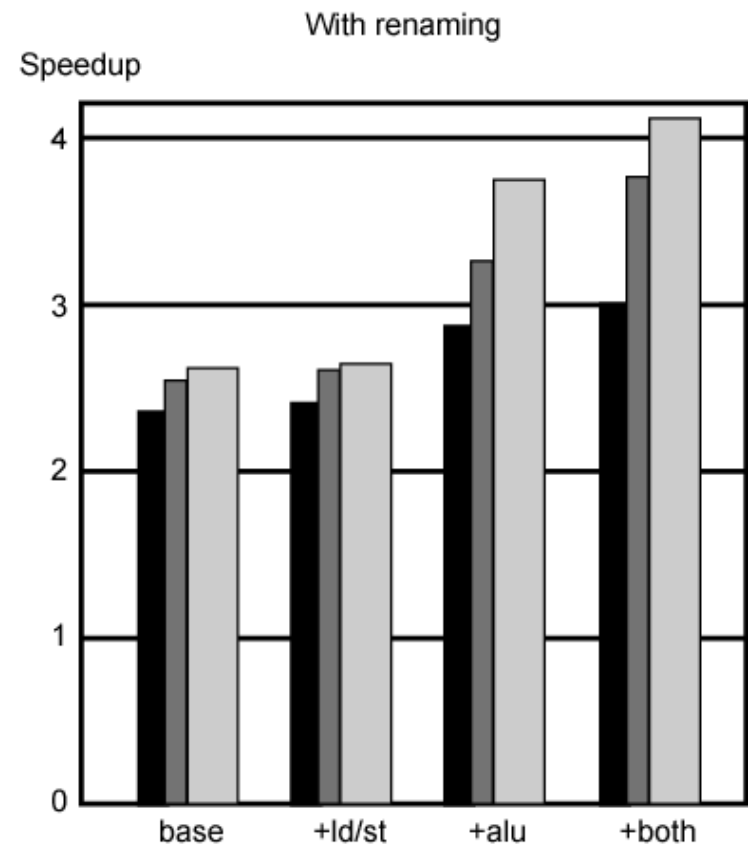
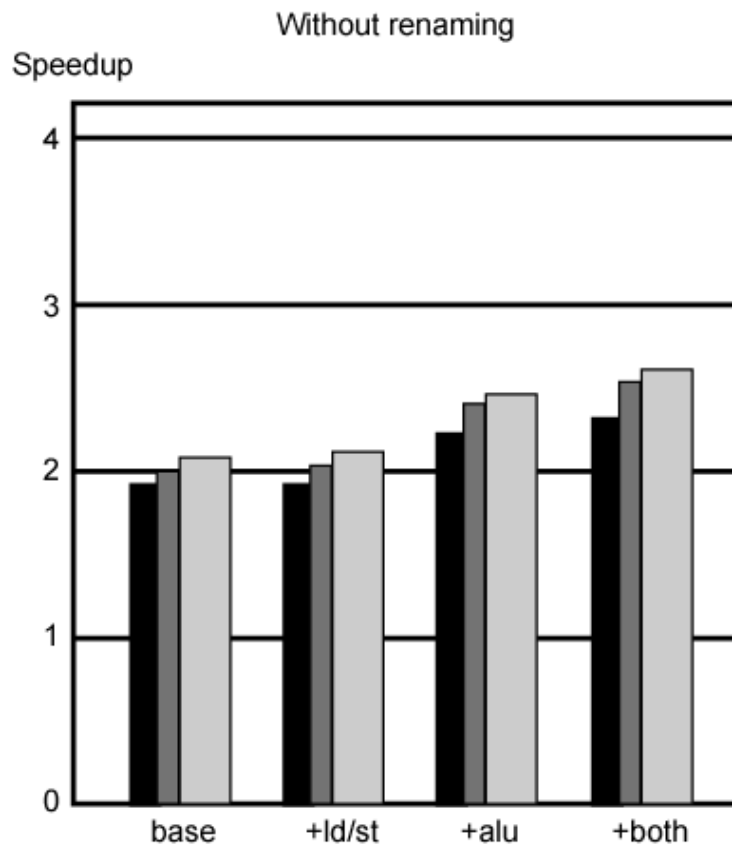
Machine Parallelism

- Duplication of Resources
- Out of order issue
- Renaming
- Not worth duplication functions without register renaming
- Need instruction window large enough (more than 8)

Speedups of Machine Organizations Without Procedural Dependencies

Window size (construction)

8	16	32
		



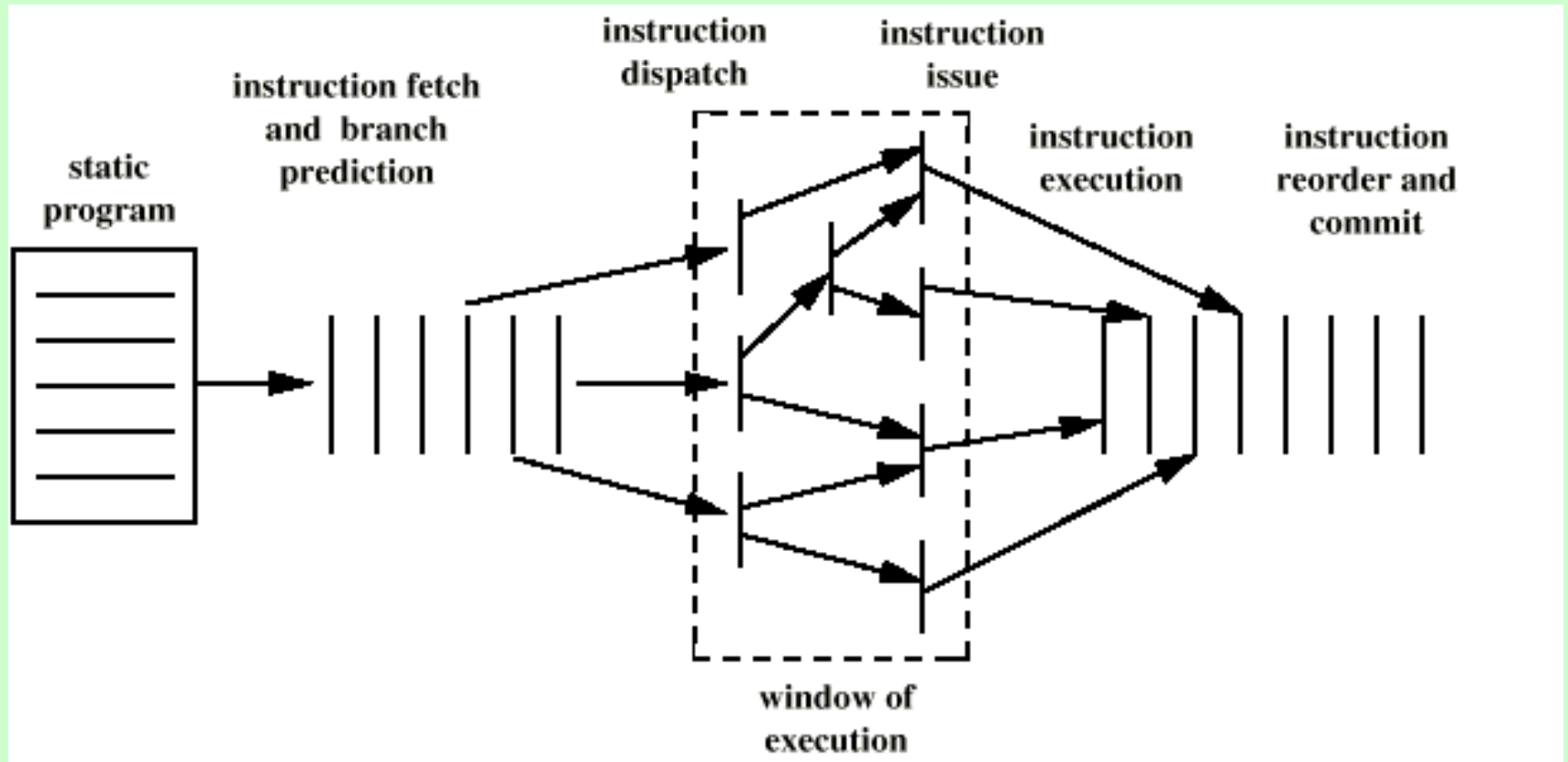
Branch Prediction

- 80486 fetches both next sequential instruction after branch and branch target instruction
- Gives two cycle delay if branch taken

RISC - Delayed Branch

- Calculate result of branch before unusable instructions pre-fetched
- Always execute single instruction immediately following branch
- Keeps pipeline full while fetching new instruction stream
- Not as good for superscalar
 - Multiple instructions need to execute in delay slot
 - Instruction dependence problems
- Revert to branch prediction

Superscalar Execution



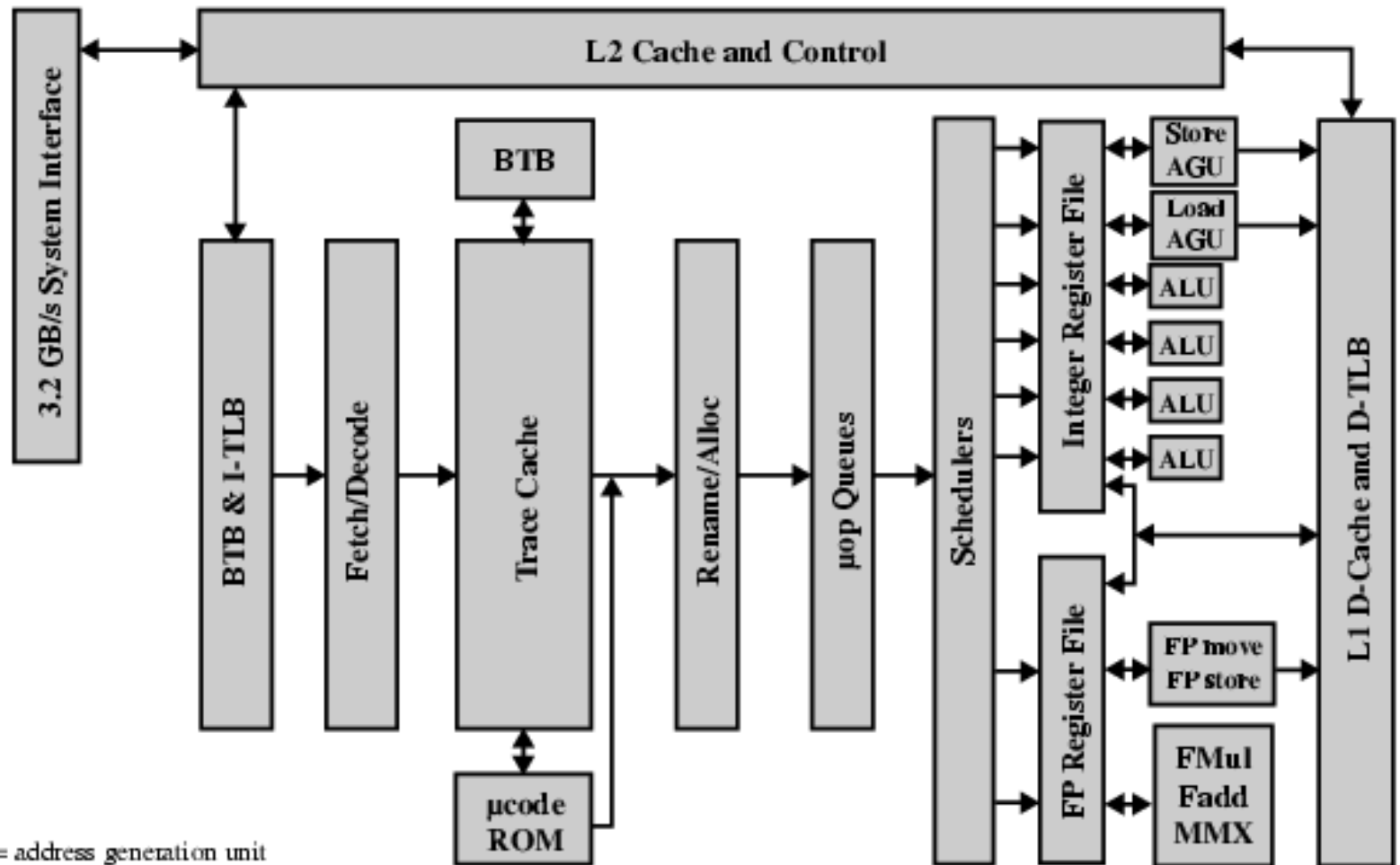
Superscalar Implementation

- Simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values
- Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order

Pentium 4

- 80486 - CISC
- Pentium – some superscalar components
 - Two separate integer execution units
- Pentium Pro – Full blown superscalar
- Subsequent models refine & enhance superscalar design

Pentium 4 Block Diagram



AGU = address generation unit

BTB = branch target buffer

D-TLB = data translation lookaside buffer

I-TLB = instruction translation lookaside buffer

Pentium 4 Operation

- Fetch instructions from memory in order of static program
- Translate instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
 - micro-ops may be executed out of order
- Commit results of micro-ops to register set in original program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages
 - Some micro-ops require multiple execution stages
 - Longer pipeline
 - c.f. five stage pipeline on x86 up to Pentium

Pentium 4 Pipeline



TC Next IP = trace cache next instruction pointer

TC Fetch = trace cache fetch

Alloc = allocate

Rename = register renaming

Que = micro-op queuing

Sch = micro-op scheduling

Disp = Dispatch

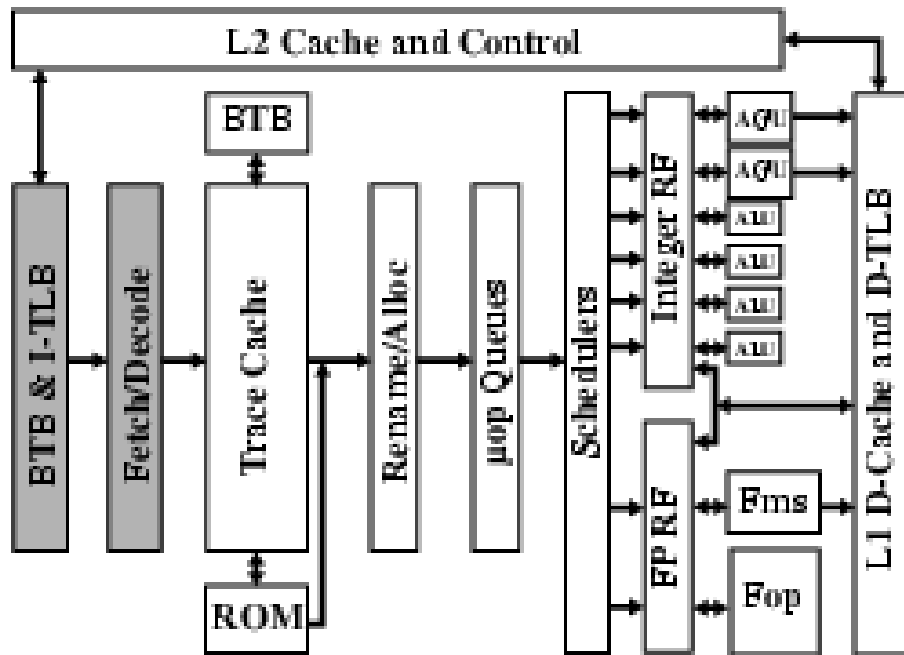
RF = register file

Ex = execute

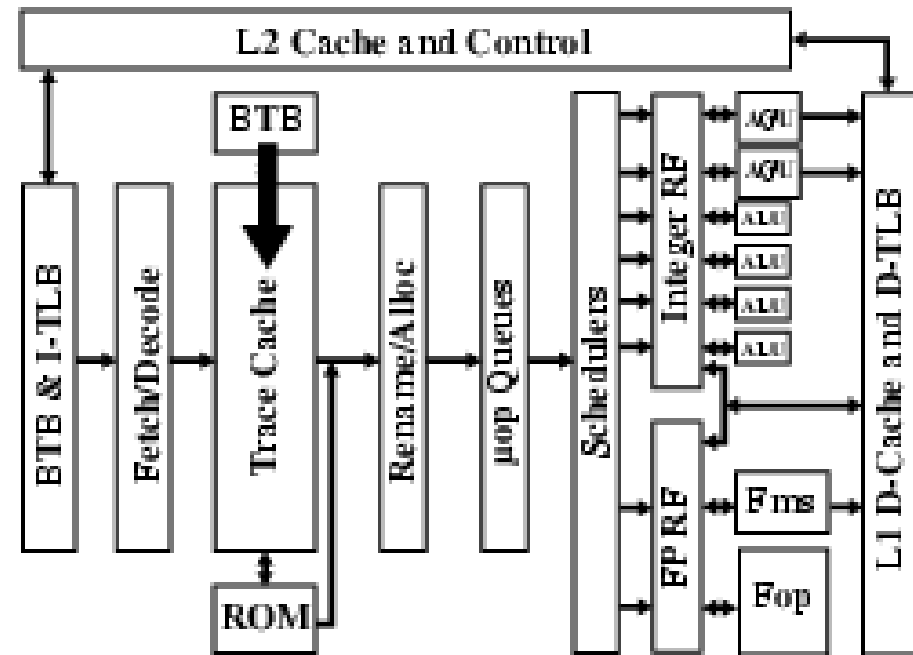
Flgs = flags

Br Ck = branch check

Pentium 4 Pipeline Operation (1)

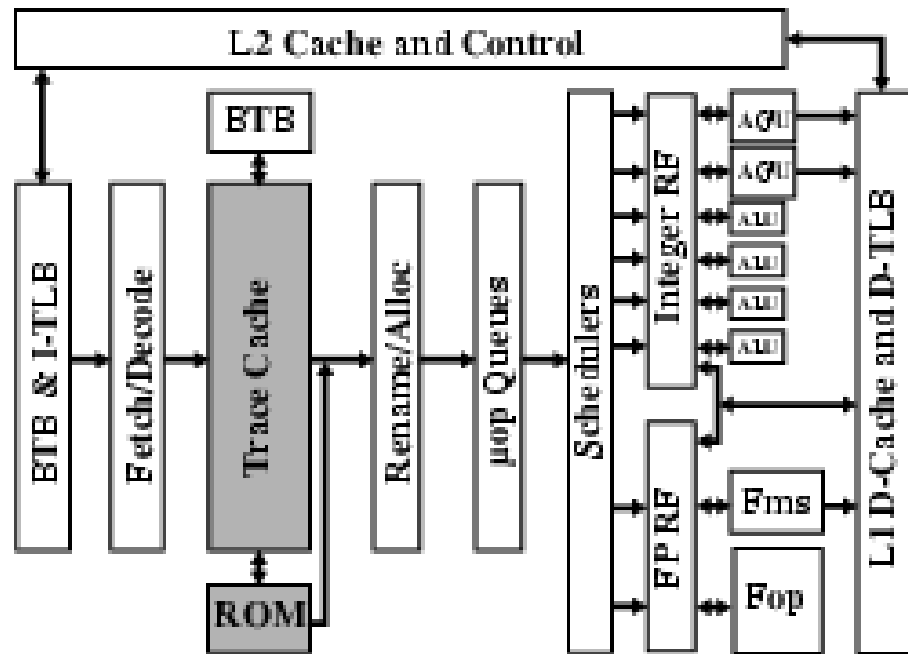


(a) Generation of micro-ops

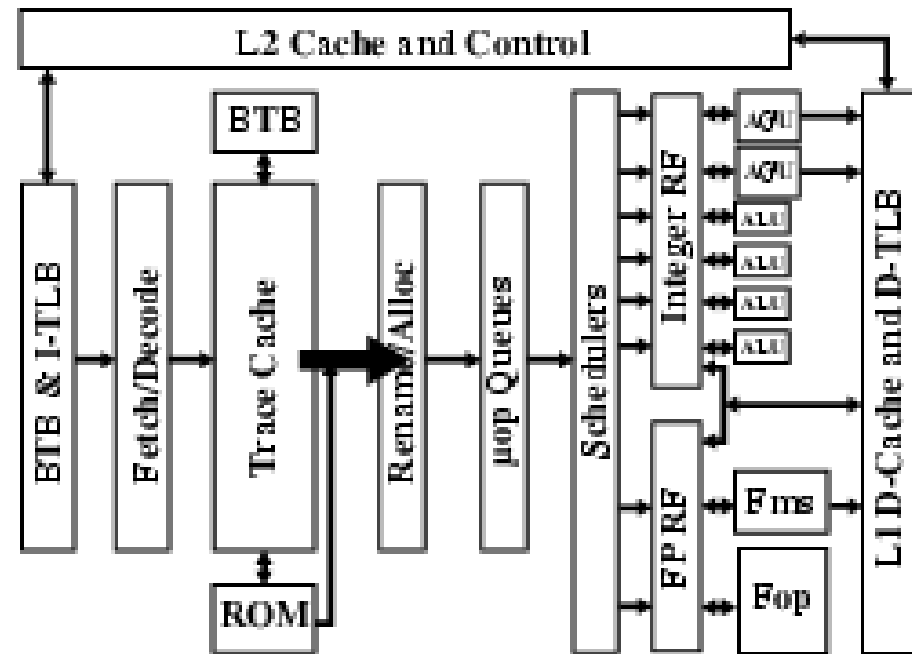


(b) Trace cache next instruction pointer

Pentium 4 Pipeline Operation (2)

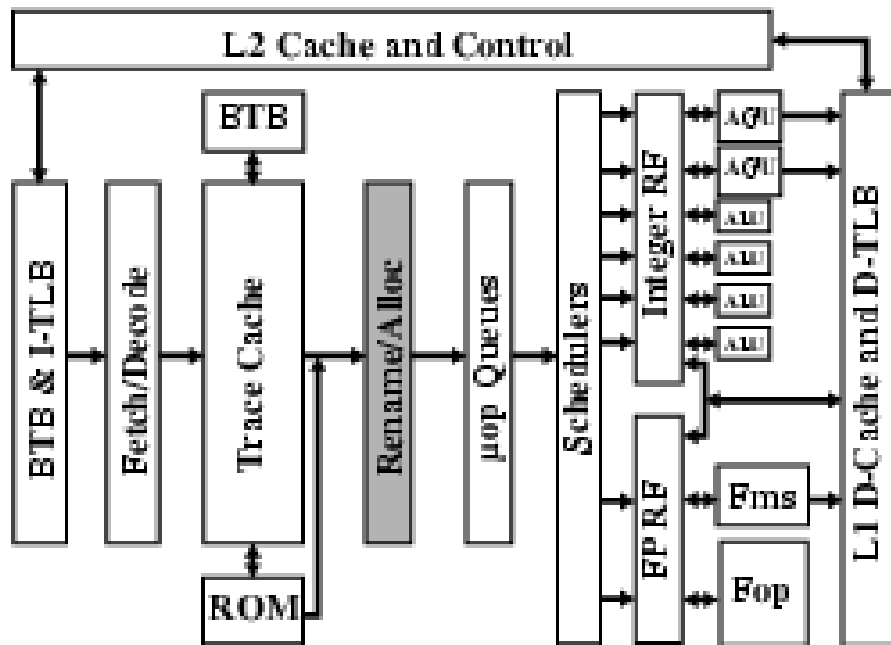


(c) Trace cache fetch

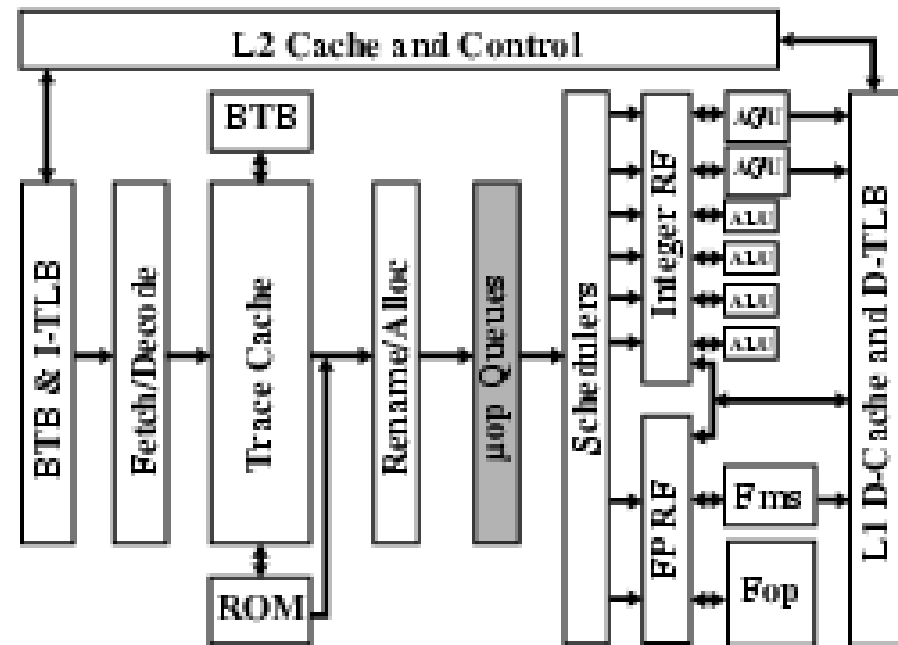


(d) Drive

Pentium 4 Pipeline Operation (3)

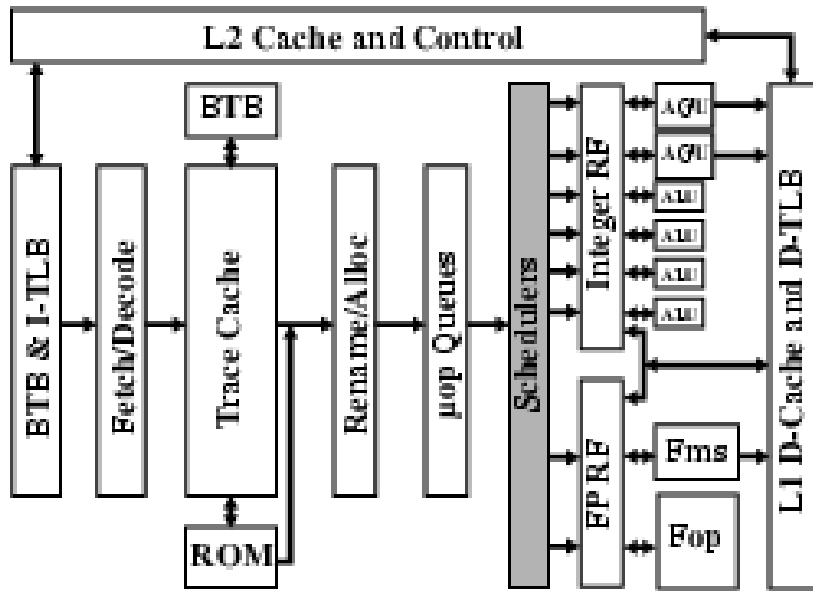


(e) Allocate; Register renaming

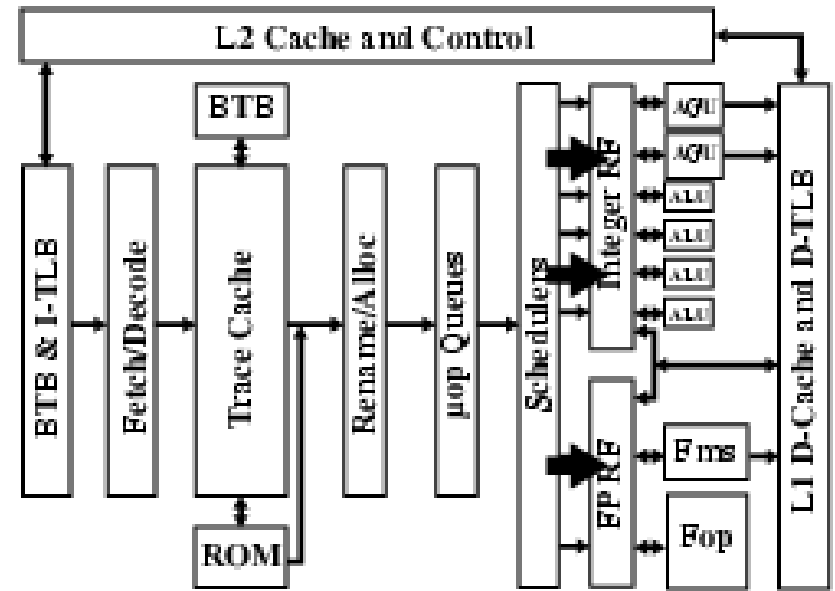


(f) Micro-op queuing

Pentium 4 Pipeline Operation (4)

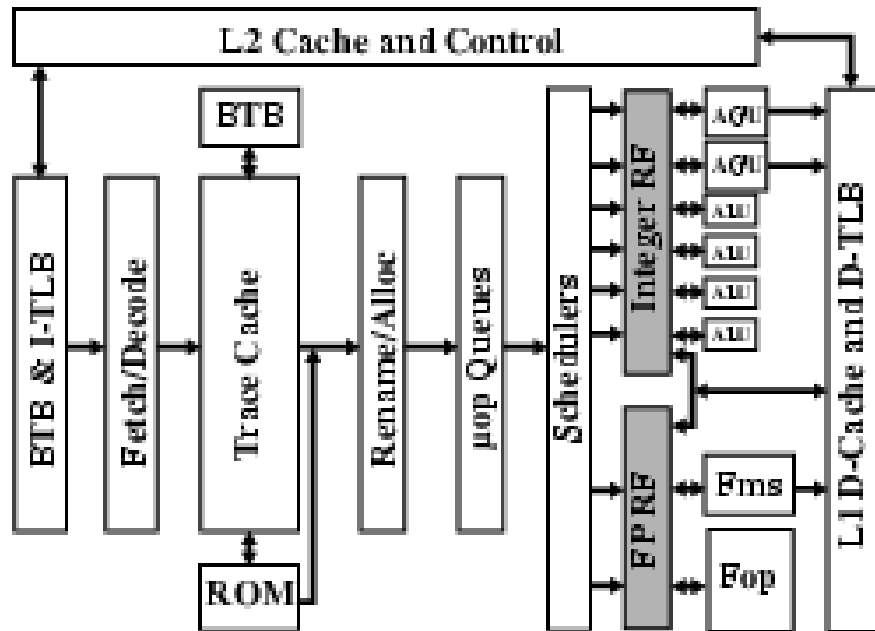


(g) Micro-op scheduling

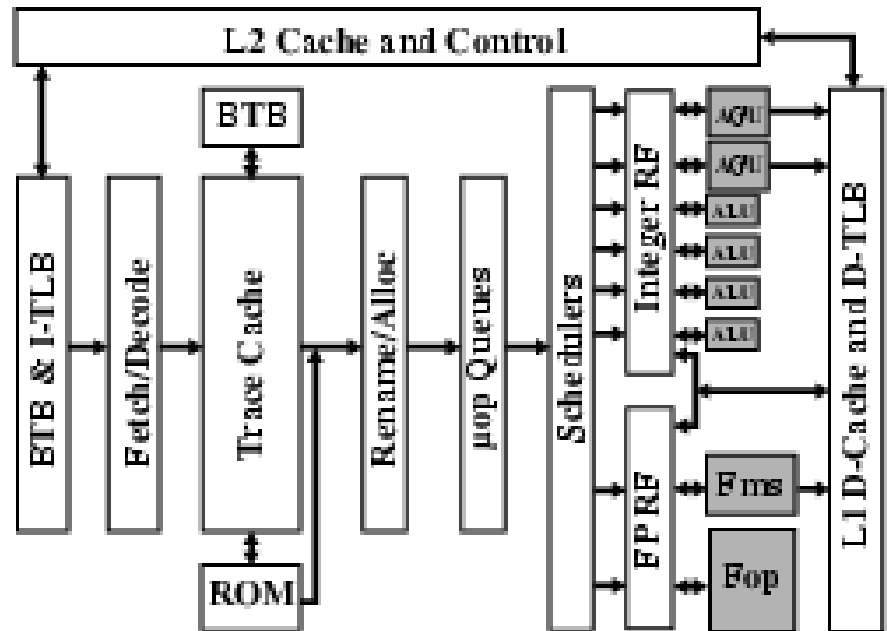


(h) Dispatch

Pentium 4 Pipeline Operation (5)

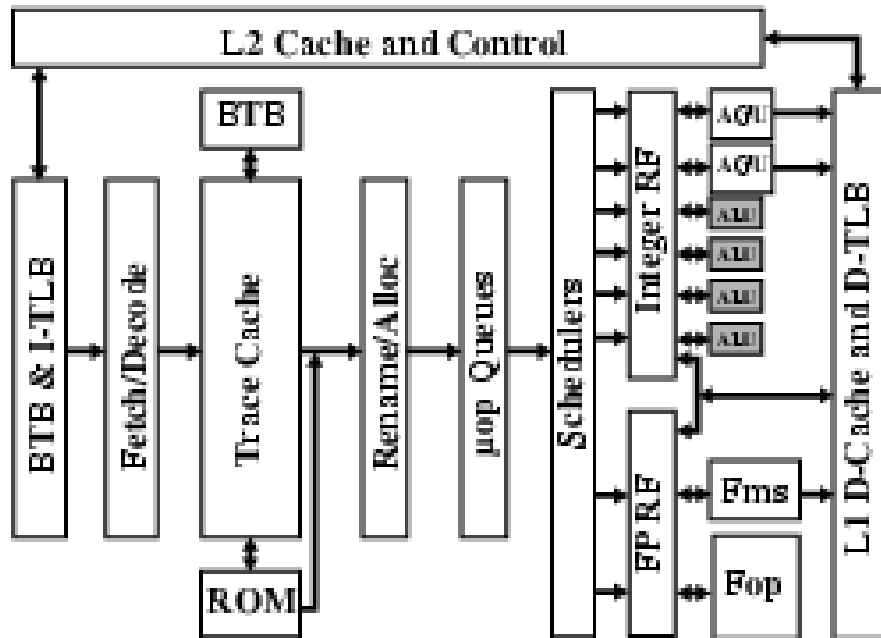


(i) Register file

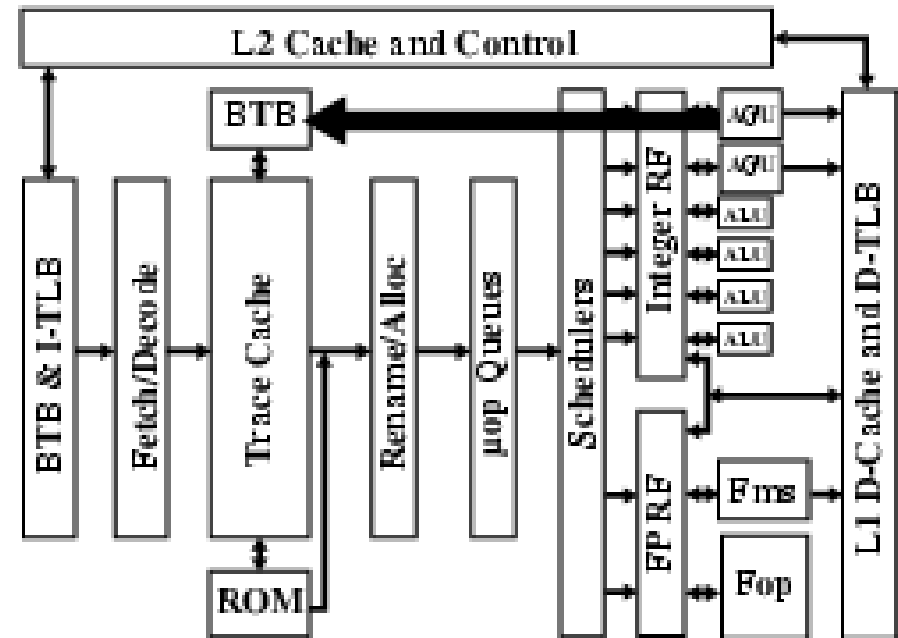


(j) Execute; flags

Pentium 4 Pipeline Operation (6)



(k) Branch check

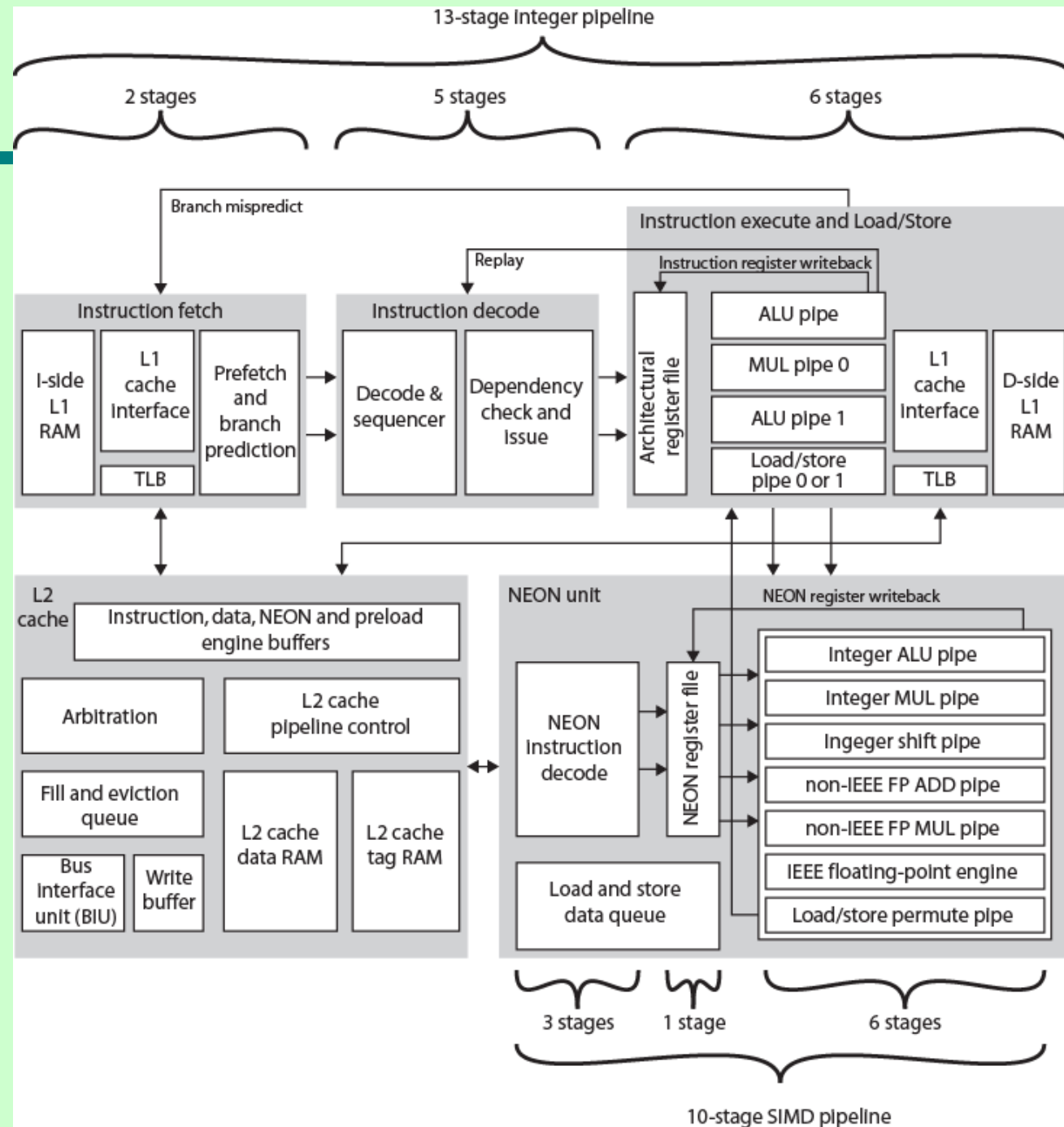


(l) Branch check result

ARM CORTEX-A8

- ARM refers to Cortex-A8 as application processors
- Embedded processor running complex operating system
 - Wireless, consumer and imaging applications
 - Mobile phones, set-top boxes, gaming consoles
automotive navigation/entertainment systems
- Three functional units
- Dual, in-order-issue, 13-stage pipeline
 - Keep power required to a minimum
 - Out-of-order issue needs extra logic consuming extra power
- Figure 14.11 shows the details of the main Cortex-A8 pipeline
- Separate SIMD (single-instruction-multiple-data) unit
 - 10-stage pipeline

ARM Cortex-A8 Block Diagram



Instruction Fetch Unit

- Predicts instruction stream
- Fetches instructions from the L1 instruction cache
 - Up to four instructions per cycle
- Into buffer for decode pipeline
- Fetch unit includes L1 instruction cache
- Speculative instruction fetches
- Branch or exceptional instruction cause pipeline flush
- Stages:
- F0 address generation unit generates virtual address
 - Normally next sequentially
 - Can also be branch target address
- F1 Used to fetch instructions from L1 instruction cache
 - In parallel fetch address used to access branch prediction arrays
- F3 Instruction data are placed in instruction queue
 - If branch prediction, new target address sent to address generation unit
- Two-level global history branch predictor
 - Branch Target Buffer (BTB) and Global History Buffer (GHB)
- Return stack to predict subroutine return addresses
- Can fetch and queue up to 12 instructions
- Issues instructions two at a time

Instruction Decode Unit

- Decodes and sequences all instructions
- Dual pipeline structure, *pipe0* and *pipe1*
 - Two instructions can progress at a time
 - Pipe0 contains older instruction in program order
 - If instruction in pipe0 cannot issue, pipe1 will not issue
- Instructions progress in order
- Results written back to register file at end of execution pipeline
 - Prevents WAR hazards
 - Keeps tracking 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 Write instruction to and read instructions from pending/replay queue
- D3 Contains the instruction scheduling logic
 - Scoreboard predicts register availability using static scheduling
 - Hazard checking
- D4 Final decode for control signals for integer execute load/store units

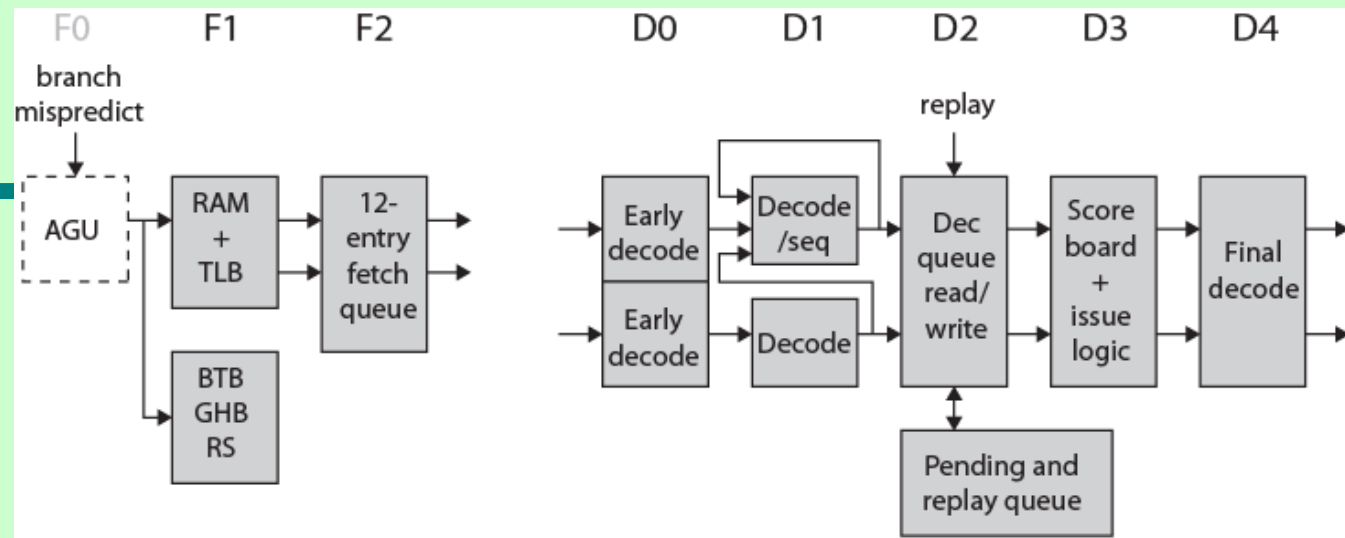
Integer Execution Unit

- Two symmetric (ALU) pipelines, an address generator for load and store instructions, and multiply pipeline
- Pipeline 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
- Multiply unit instructions routed to pipe0
 - Performed in stages E1 through E3
 - Multiply accumulate operation in E4

Load/store pipeline

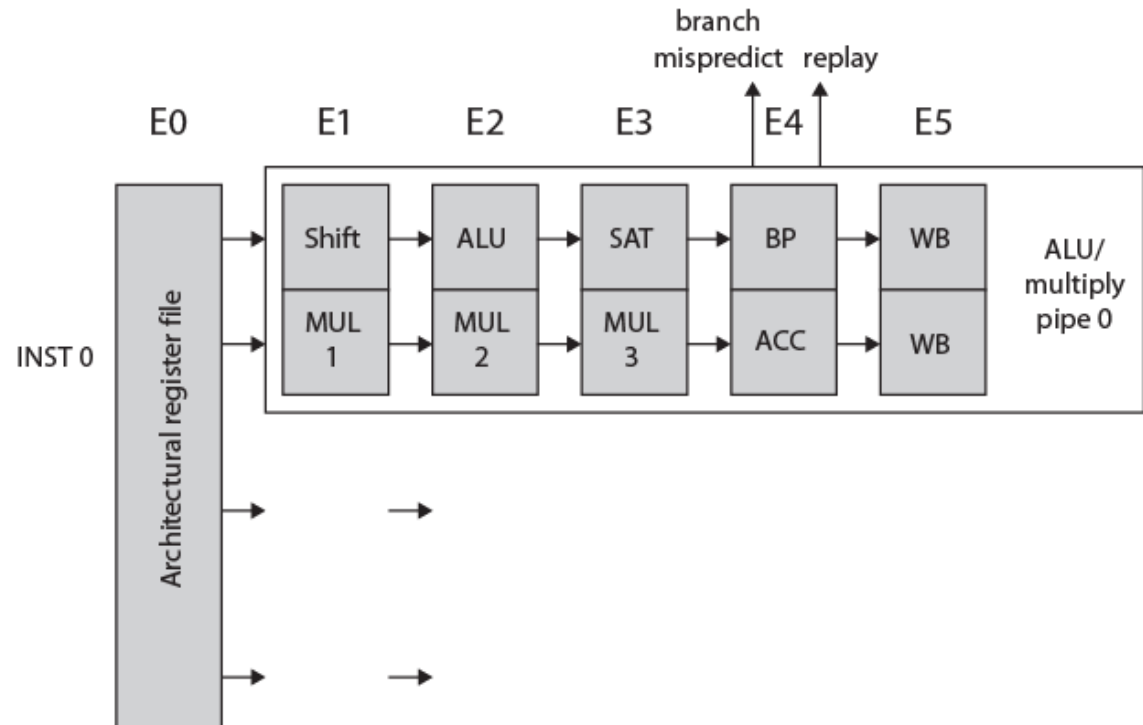
- Parallel to integer pipeline
- E1 Memory address generated from base and index register
- E2 address applied to cache arrays
- E3 load, data returned and formatted
- E3 store, data are formatted and ready to be written to cache
- E4 Updates L2 cache, if required
- E5 Results are written to register file

ARM Cortex-A8 Integer Pipeline



(a) Instruction fetch pipeline

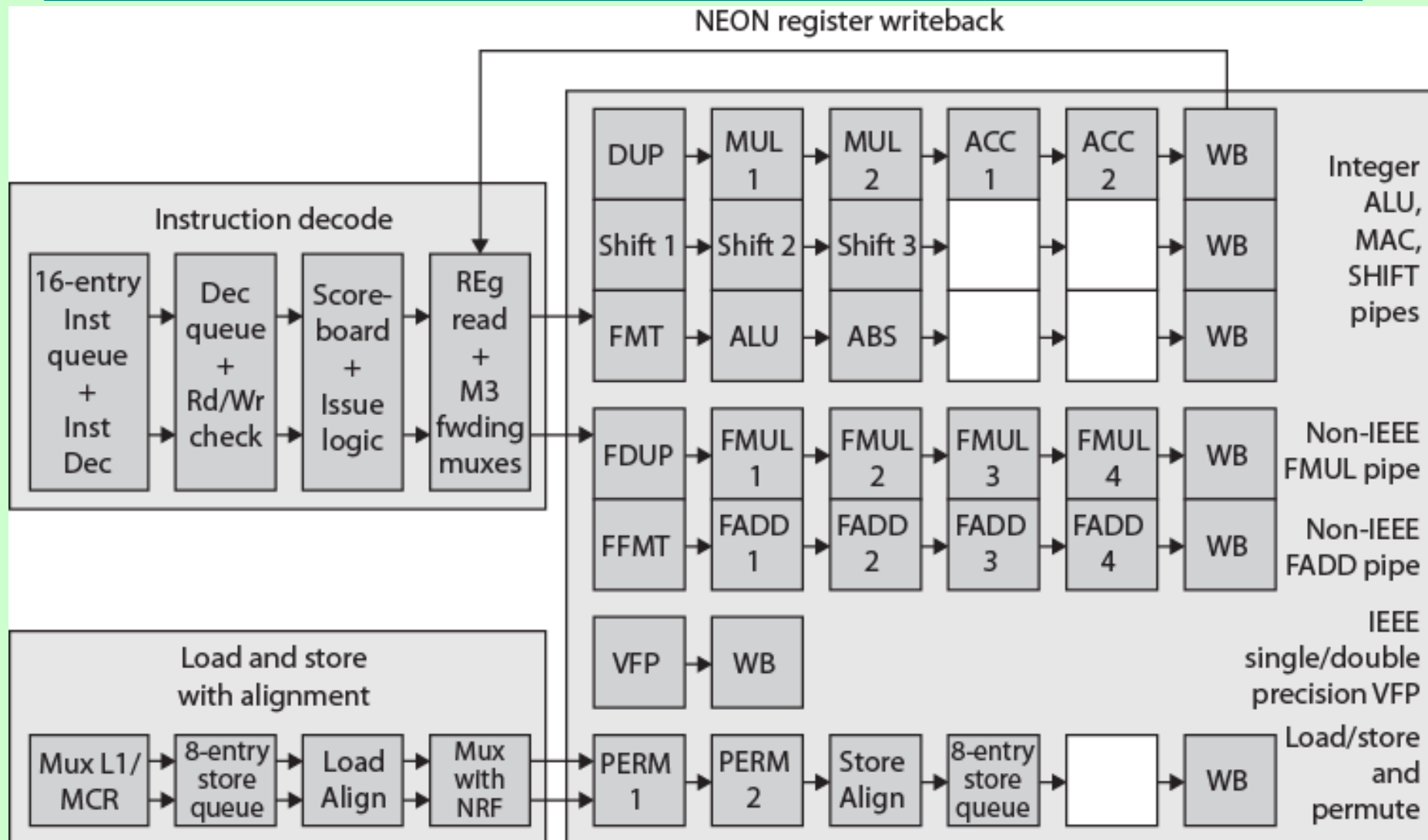
(b) Instruction decode pipeline



SIMD and Floating-Point Pipeline

- SIMD and floating-point instructions pass through integer pipeline
- Processed in separate 10-stage pipeline
 - NEON unit
 - Handles packed SIMD instructions
 - Provides two types of floating-point support
- If implemented, vector floating-point (VFP) coprocessor performs IEEE 754 floating-point operations
 - If not, separate multiply and add pipelines implement floating-point operations

ARM Cortex-A8 NEON & Floating Point Pipeline



Required Reading

- Stallings chapter 14
- Manufacturers web sites
- IMPACT web site
 - research on predicated execution