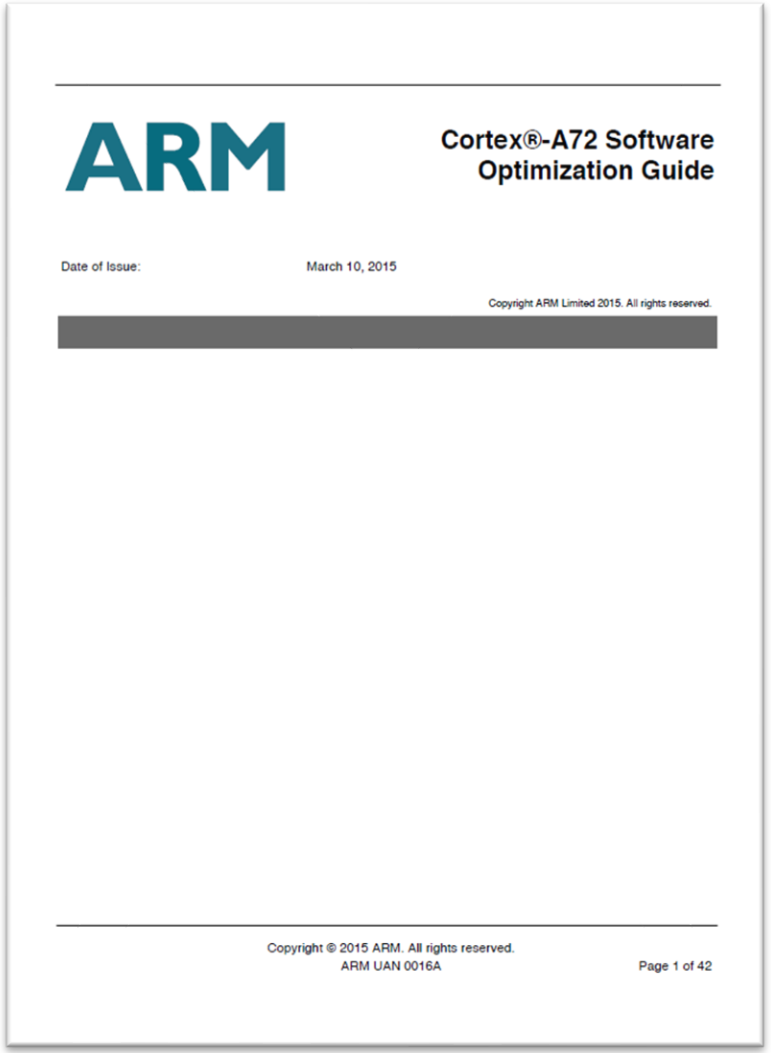
Optimizing for the Cortex-A72

Overview

- Processor Architecture and Microarchitecture
- Pipelines and Instruction Latencies
- Sources:
  - **Cortex-A72 MPCore Technical Reference Manual** (ARM 100095_0003_05_en)
  - **Cortex-A72 Software Optimisation Guide** (UAN 0016A)
  - **ARM Cortex-A Series Programmer's Guide** (DEN0013D)


ARM

Cortex®-A72 Software Optimization Guide

Date of Issue: March 10, 2015

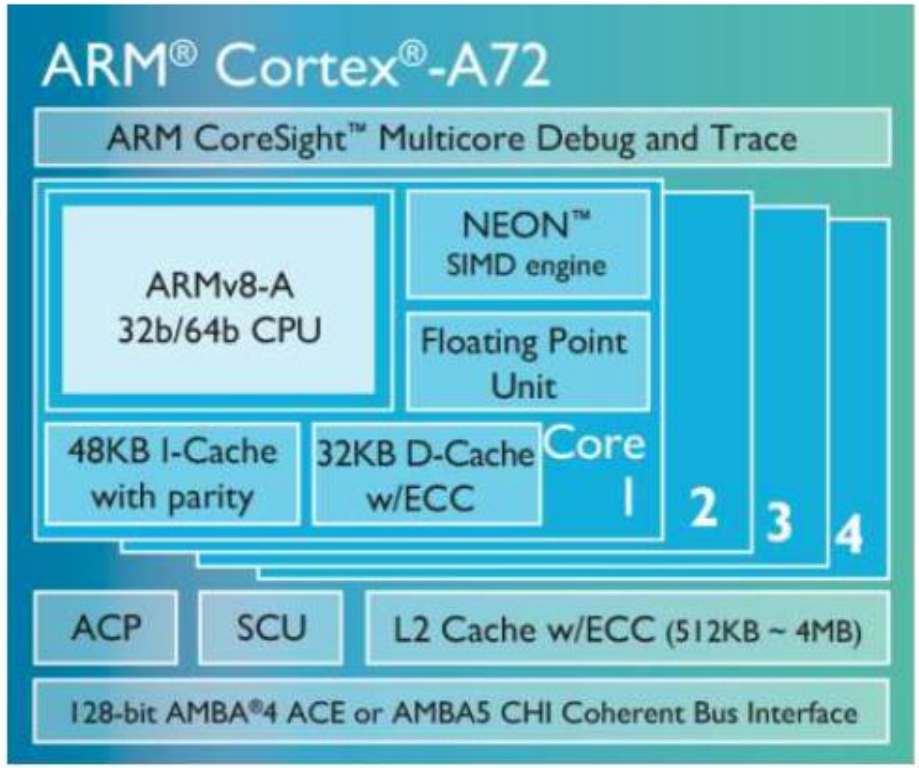
Copyright ARM Limited 2015. All rights reserved.

Copyright © 2015 ARM. All rights reserved.
ARM UAN 0016A

Page 1 of 42

Cortex-A72: Increased Performance ***and*** Reduced Power

- **Compelling single-threaded performance**
  - Large performance increase across all workloads including integer, memory-intensive, crypto, floating point, etc.
  - Baseline microarchitecture similar to Cortex-A57
- **Significant advancements in power efficiency**
  - Re-optimized every logical block from Cortex-A57
  - Power reduction enables sustained operation at Fmax
  - Area reduction lowers costs and static power
- **Feature support for enterprise and mobile SoCs**

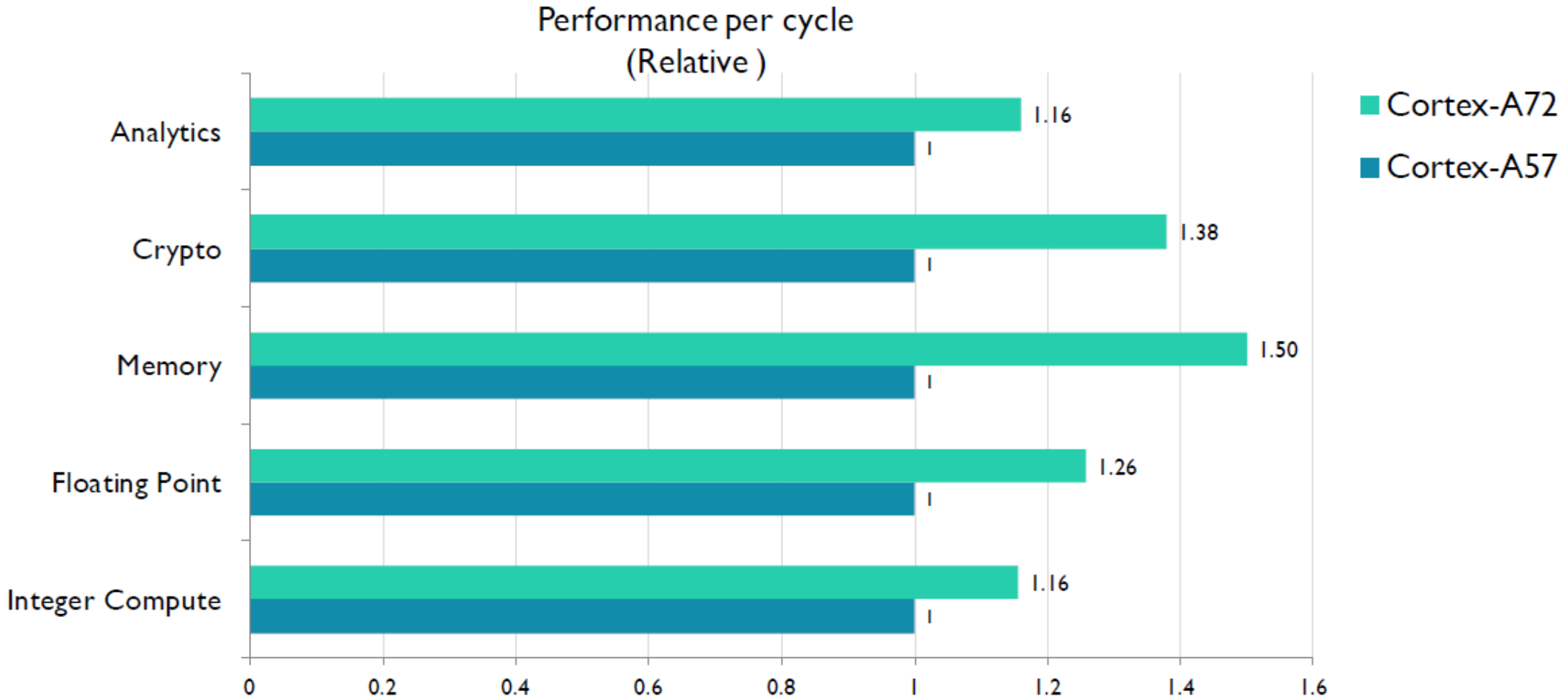


Embargo until 10pm BST 23 April

ARM

Cortex-A72: Next-Generation Performance

ARM CORTEX
Processor Technology



Performance measure on same frequency, same process and identical memory system interfaces

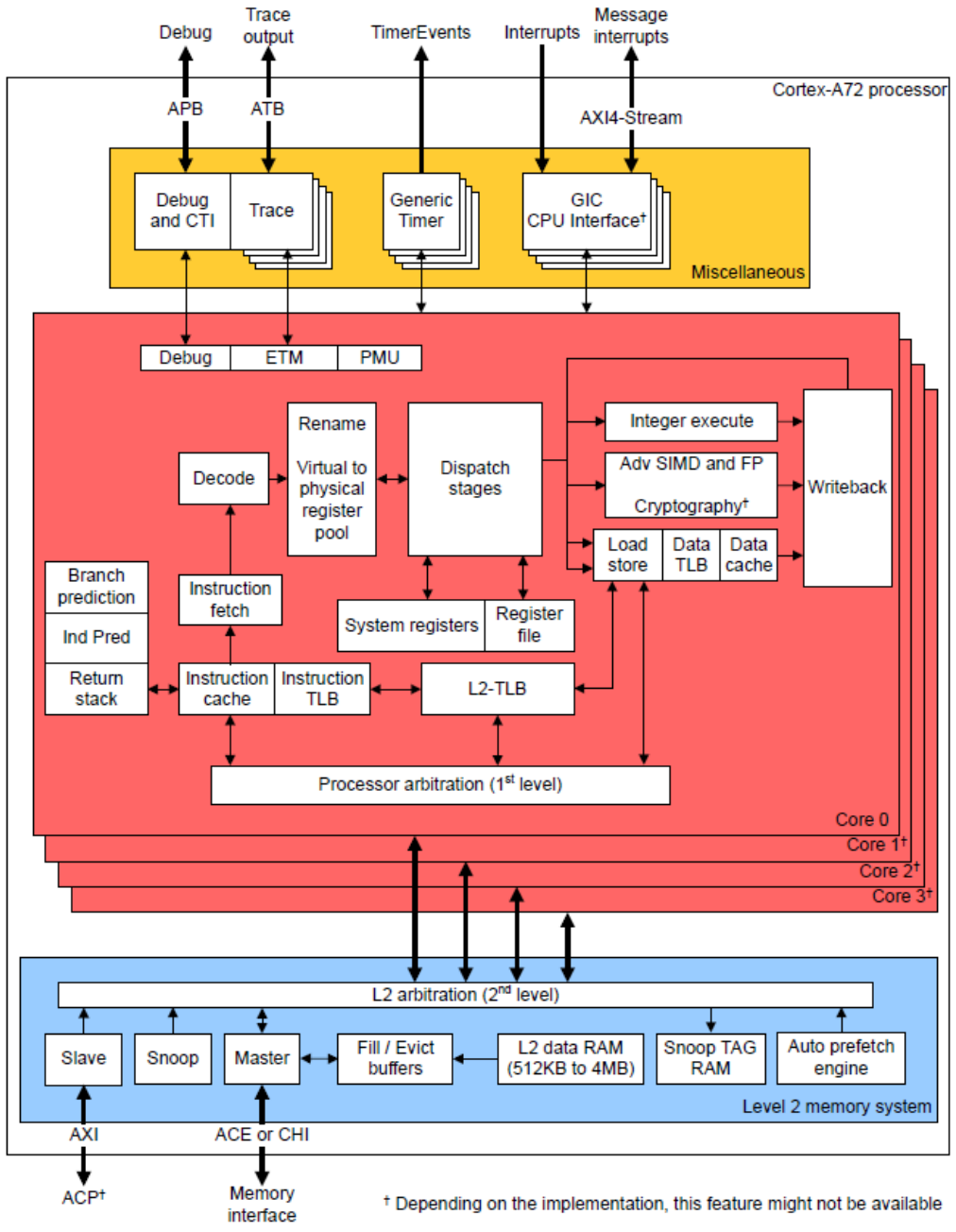
Workloads include: SPECint06, SPECfp06, Stream, LMbench, Geekbench, Antutu, Minebench, AES/SHA/CRC kernels, and other targetted kernels

Embargo until 10pm BST 23 April

ARM

Cortex-A72 Overview

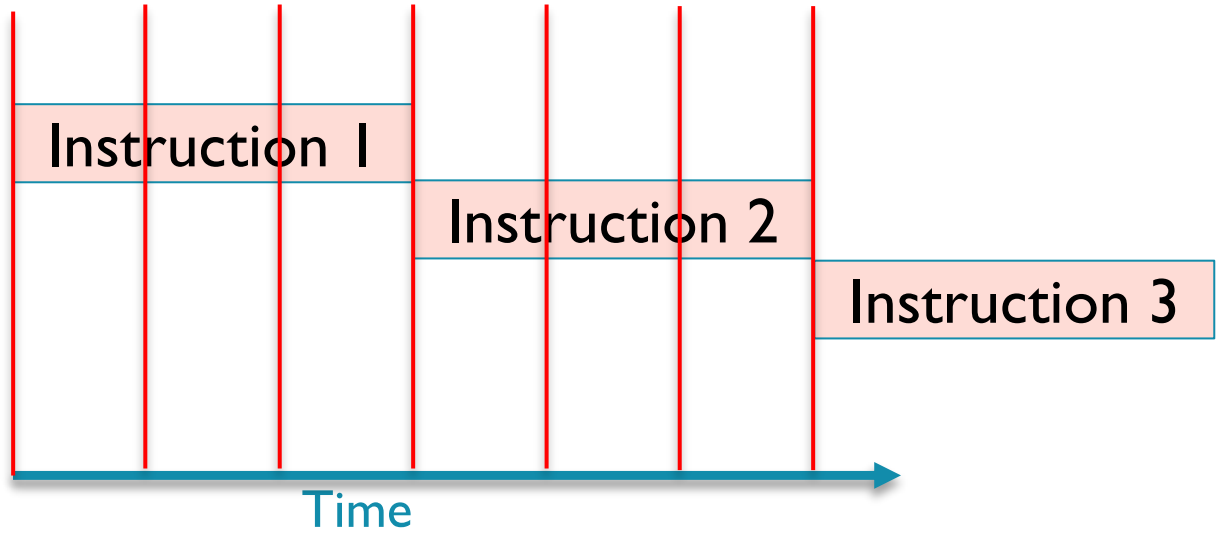
- 4 cores
- Per-core L1 caches
  - 48 kB instruction cache
  - 32 kB data cache
- Shared L2 cache
  - Unified I+D cache
  - 512 kB – 4 MB

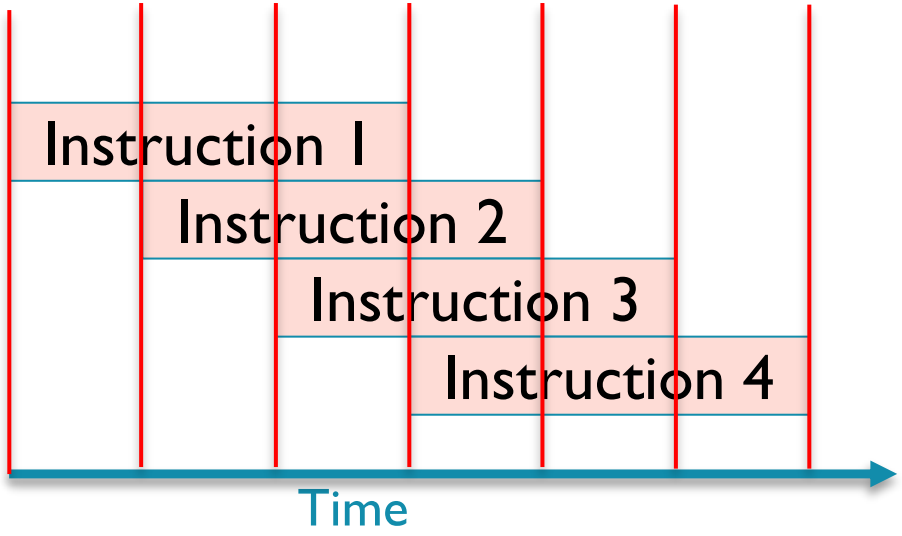


Pipelines and Their Consequences

Pipelined Instruction Execution

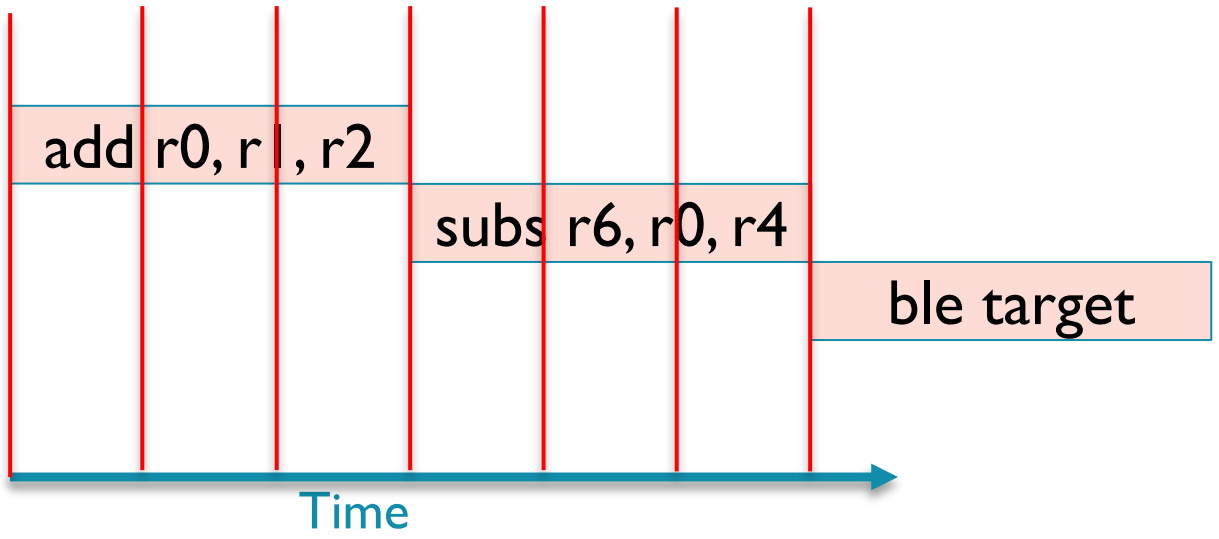
- Why use a pipeline?
  - Increase clock speed to finish a set of instructions sooner
- How? *Overlap instruction execution*
  - Start executing instruction N+1 before finishing instruction N
  - Use latches to hold intermediate results between pipeline stages
- How much performance improvement?
  - Maximum theoretical speedup is number of pipeline stages



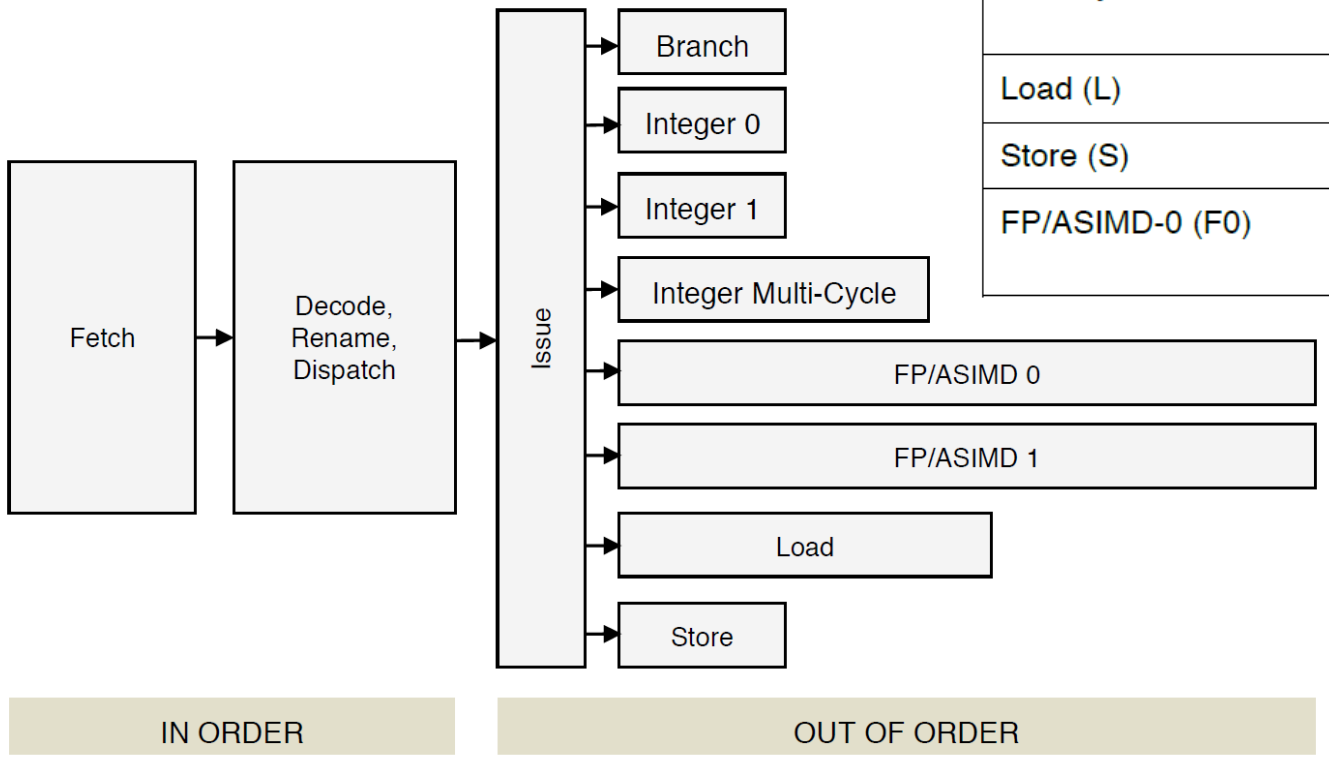


Speedup Limited by…

- Design issues
  - Latch latency
  - Increasing difficulty of splitting logic into equal-duration pipeline stages
- Hazards which stall pipeline
  - Data-flow hazards – need data which hasn't been calculated yet
    - subs uses r0, which was defined by add
  - Control-flow hazards – conditional branch causes flow of program to depend on a result which is not available yet
    - ble uses condition codes set by subs
    - Which instruction executes next? Instruction after branch, or branch target?



Cortex-A72: **Eight** Instruction Execution Pipelines



Pipeline (mnemonic)	Supported functionality
Branch (B)	Branch μops
Integer 0/1 (I)	Integer ALU μops
Multi-cycle (M)	Integer shift-ALU, multiply, divide, CRC and sum-of-absolute-differences μops
Load (L)	Load and register transfer μops
Store (S)	Store and special memory μops
FP/ASIMD-0 (F0)	ASIMD ALU, ASIMD misc, ASIMD integer multiply, FP convert, FP misc, FP add, FP multiply, FP divide, crypto μops

Cortex-A72 µArchitecture

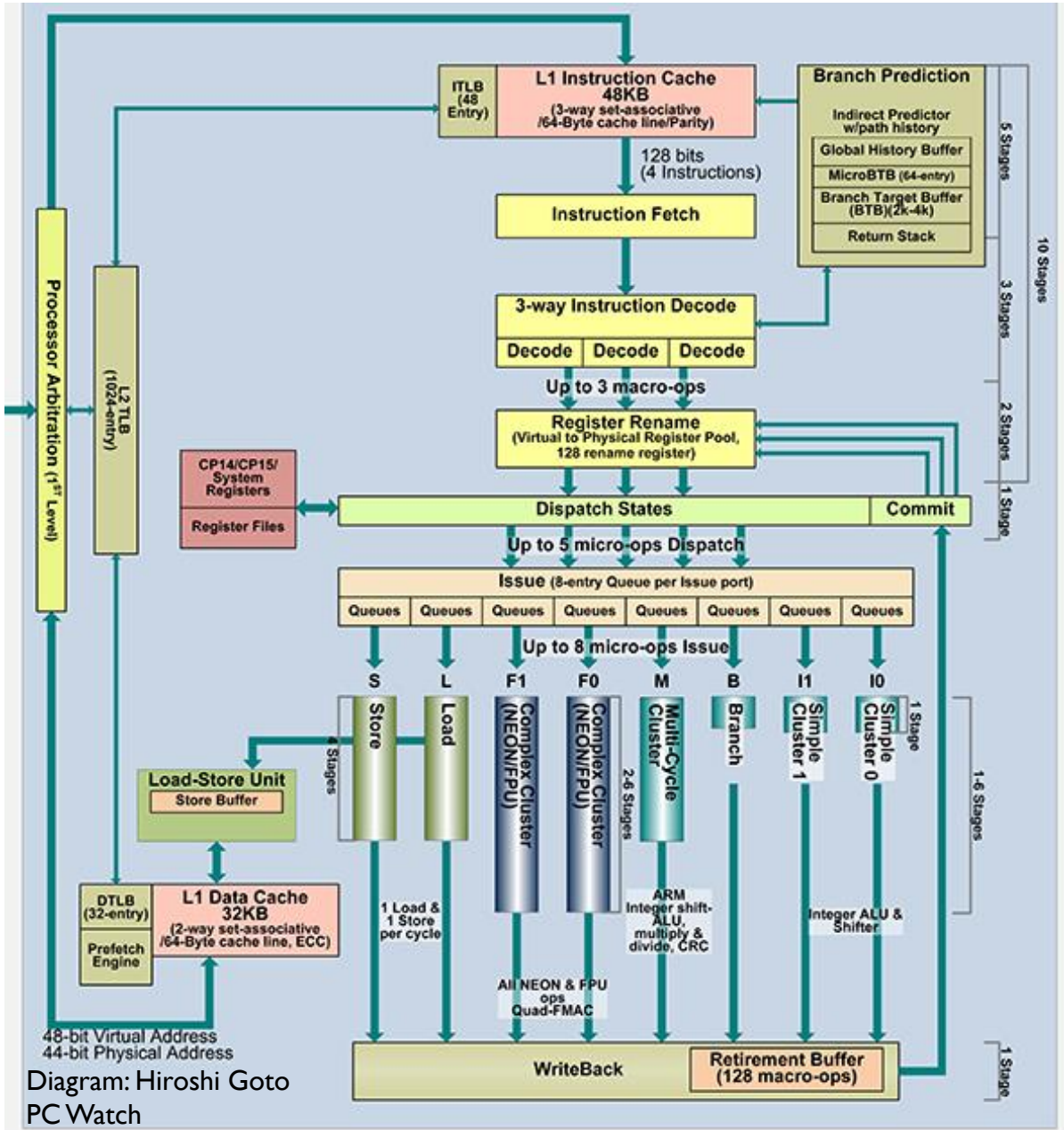


Diagram: Hiroshi Goto
PC Watch

- Fetch instruction(s)
- Decode into internal µops
- µop register renaming
- µop dispatch to await operands in issue queues
- Issue µop out-of-order to an execution pipeline
- Execute in pipeline

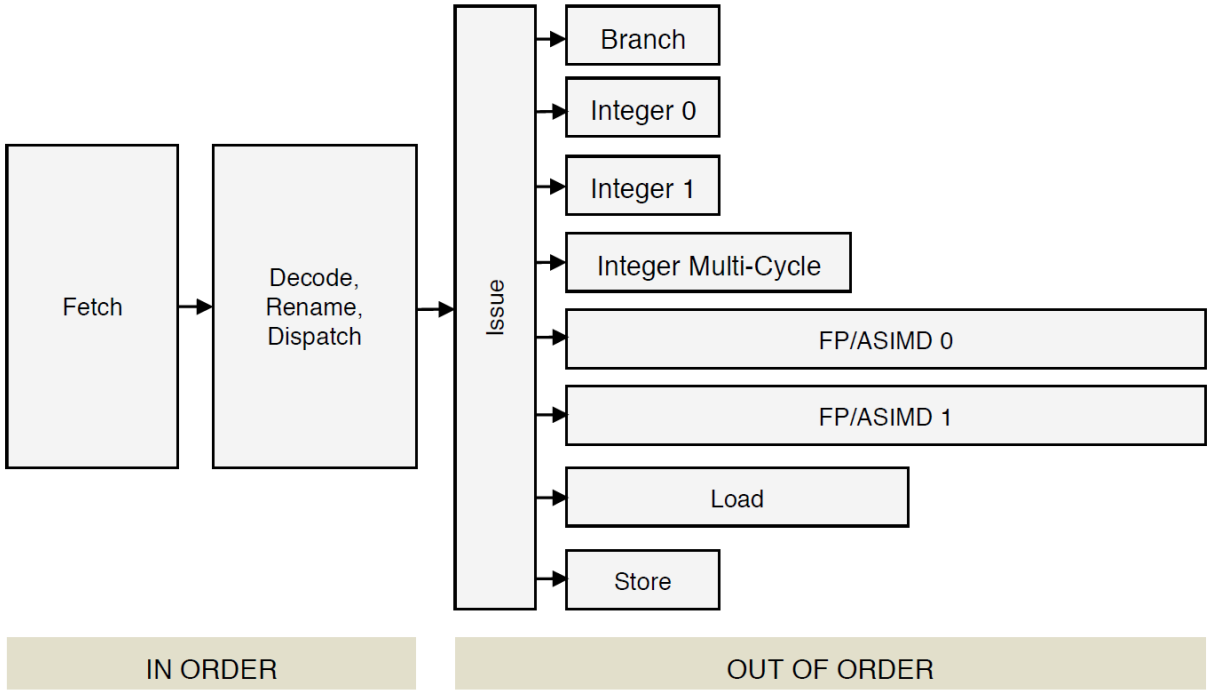
	Cortex-A15	Cortex-A57	Cortex-A72
ARM ISA	ARMv7 (32-bit)	ARMv8 (32/64-bit)	
Decoder Width	3 ops		
Maximum Pipeline Length	19 stages		16 stages
Integer Pipeline Length	14 stages		
Branch Mispredict Penalty	15 cycles		
Integer Add	2		
Integer Mul	1		
Load/Store Units	1 + 1 (Dedicated L/S)		
Branch Units	1		
FP/NEON ALUs	2x64-bit	2x128-bit	
L1 Cache	32KB I$ + 32KB D$	48KB I$ + 32KB D$	
L2 Cache	512KB - 4MB	512KB - 2MB	512KB - 4MB

Cortex-A72 Instruction Timing

- Note from Cortex-A72 TRM regarding instruction timing

> - The out-of-order design of the Cortex-A72 processor pipeline makes it impossible to provide accurate timing information for complex instructions. The timing of an instruction can be affected by factors such as:
>   — Other concurrent instructions.
>   — Memory system activity.
>   — Events outside the instruction flow.
> - Timing information has been provided in the past for some ARM processors to assist in the hand tuning of performance critical code sequences or in the development of an instruction scheduler within a compiler. This timing information is not required for producing optimized instruction sequences on the Cortex-A72 processor. The out-of-order pipeline of the processor can schedule and execute the instructions in an optimal fashion without any instruction reordering required.

- Important instruction characteristics
  - Latency
    - *Minimum* latency seen by a dependent operation
  - Throughput
    - Maximum number of active instructions of this type per clock cycle
    - If not pipelined, will be less than 1



ARM

Cortex®-A72 Software Optimization Guide

Date of Issue: March 10, 2015

Contents

AArch32 Instruction Performance for Cortex-A72

Instruction Type	Latency	Thrghpt
Branch, Br. & Link, Compare & Br.	1	1
Arithmetic, Logic w/o shift	1	2
Arithmetic, Logic w/ shift	2	1
Move, Shift	1-2	1-2
Integer Divide (blocking, early-out)	4-12	<1
Multiply, MAC, Long Multiply	3-4	½-1
Saturating and Parallel Arithmetic	2-5	½-1
Load (L1 hit)	4-5	1
Store	1-3	1
Misc Data Processing	1-4	1-2

FP Instruction Type	Latency	Thrghpt
FP Multiply	4	2
FP Multiply/Accumulate	7	2
FP Divide	6-18	<1
FP Square Root	6-32	<1
FP Load	5	1
FP Store	1	1
FP Move (see 3.11)	3-8	1-2
FP Misc. Data Processing	3-6	1/6-2

ASIMD Instruction Type	Latency	Thrghpt
ASIMD Integer	3-5	½-2
ASIMD FP	3-7	½-2
ASIMD Load	5-9	½-1
ASIMD Store	1-4	¼-1
ASIMD Misc.	3-8	½-2

Special Considerations

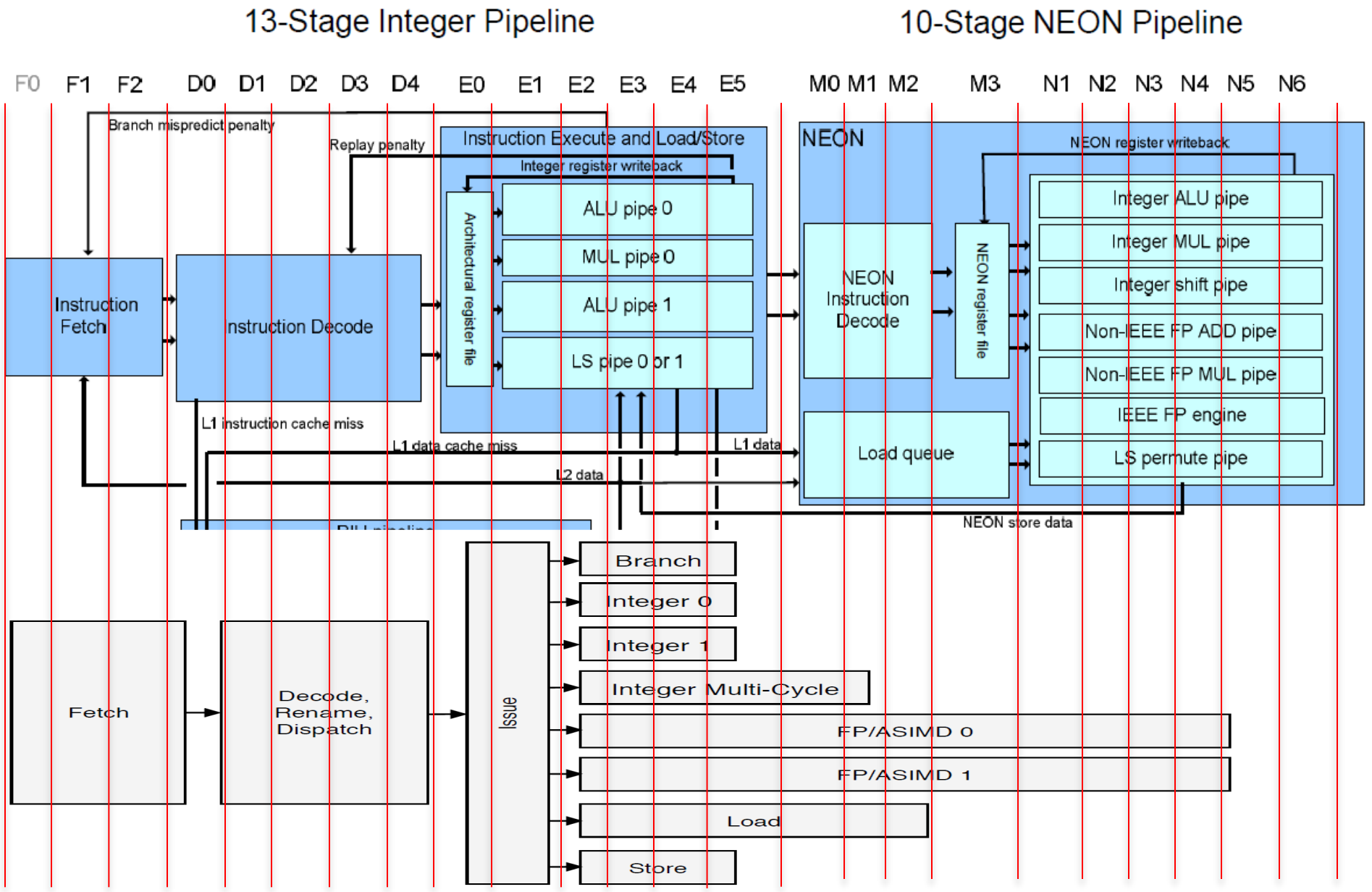
- Refer to **Cortex-A72 Software Optimisation Guide** (UAN 0016A)

- Dispatch Constraints
- Conditional Execution
- Conditional ASIMD
- Register Forwarding Hazards
- Load/Store Throughput
- Load/Store Alignment
- Branch Alignment
- Setting Condition Flags
- Special Register Access
- AES Encryption/Decryption
- Fast literal generation
- PC-relative address calculation
- FPCR self-synchronization

Control-Flow Hazards and Branch Prediction

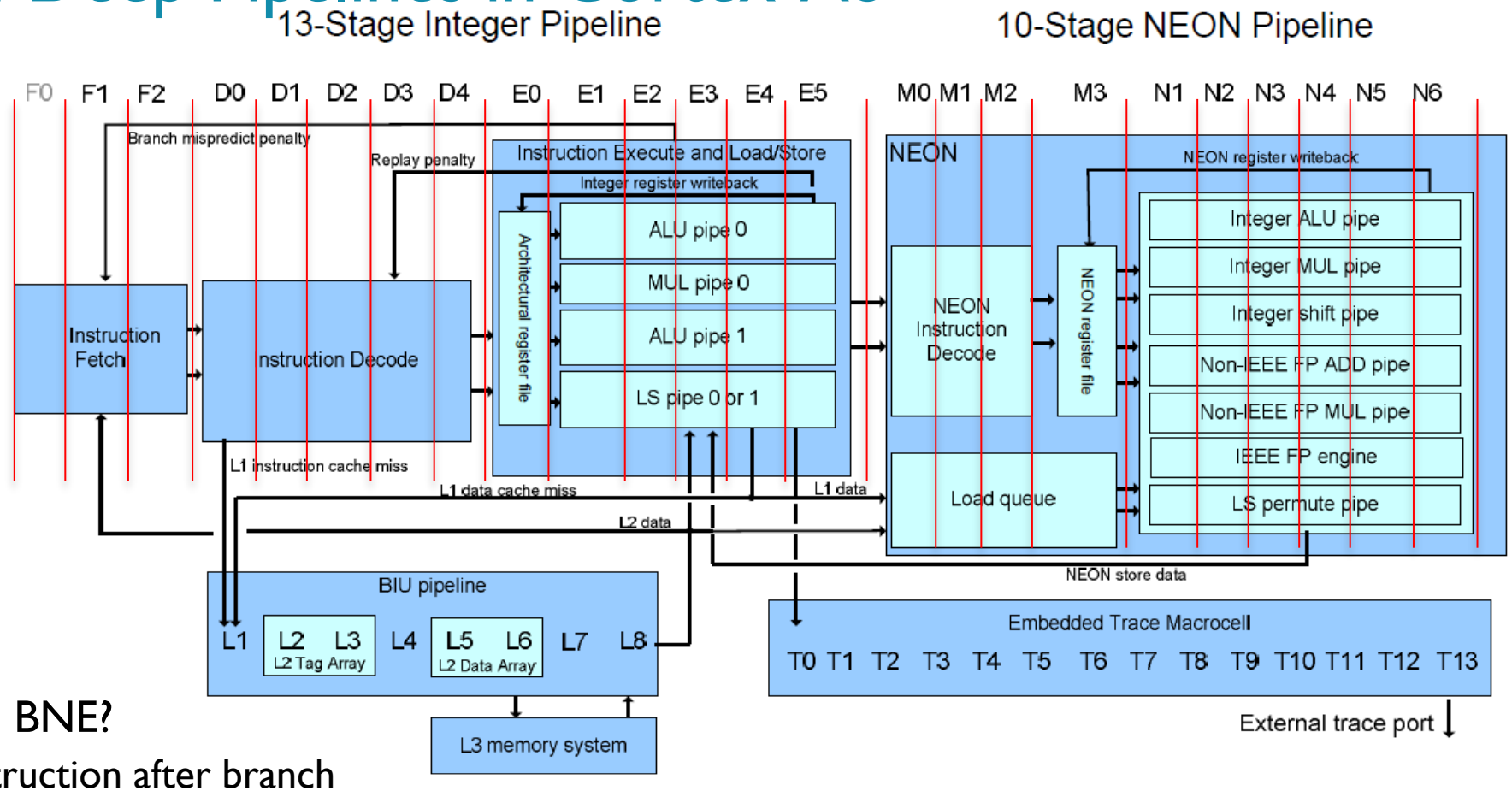
Pipeline Comparison: Cortex-A8 vs. Cortex-A72

- Cortex-A8: Two sequential pipelines
  - Integer instructions complete at end of stage E5
  - NEON and floating-point instructions
    - Flow through integer pipeline
    - Are decoded and execute in NEON pipeline (ASIMD)
    - Complete at end of stage N6
- Cortex-A72: Integrated, parallel pipelines
  - All execution pipelines start at same stage
    - Integer, floating point and ASIMD



Example: Impact of Deep Pipelines in Cortex-A8

```
loop
   …
   SUBS r1,r1,#1
   BNE loop
   CMP   r3,#0
```



- Which instruction executes after the conditional branch BNE?
  - Branch target (loop), or instruction after branch (CMP)?
  - Pipe stages with newer instructions ***will stall*** until condition controlling branch is resolved, ***reducing performance***

- Penalties (assuming L1 instruction cache hit)
  - **Integer comparison**: 13 cycles
  - **Floating-point comparison**: 23 cycles

Cortex-A72 Program Flow Prediction

- 15-cycle program flow mispredict penalty, so try to predict correctly
  - Predicted instructions
    - Conditional, unconditional, and indirect branches
    - Arm/Thumb interworking switch
    - Instructions with PC as destination
  - Unpredicted instructions
    - Anything capable of changing privilege mode or security state

- Details: Cortex-A72 TRM, Section 6.5

Program flow prediction

The Cortex-A72 processor contains program flow prediction hardware, also known as *branch prediction*.

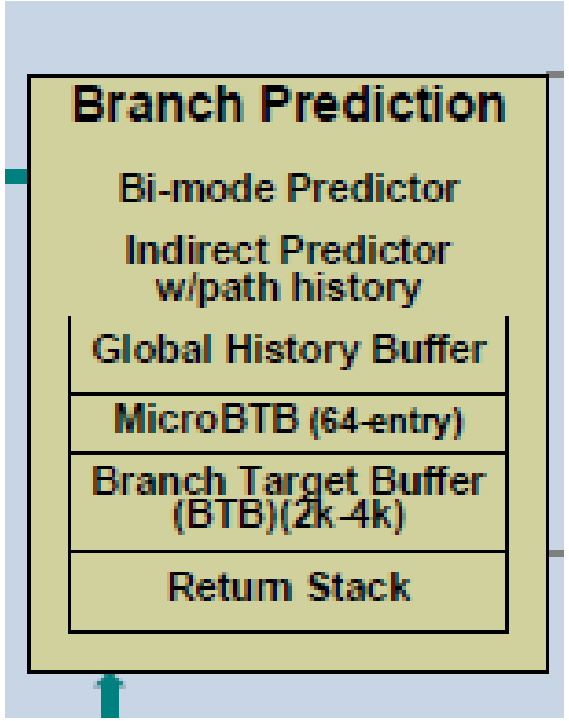
With program flow prediction disabled, all taken branches incur a penalty associated with flushing the pipeline. To avoid this penalty, the branch prediction hardware operates at the front of the instruction pipeline. The branch prediction hardware consists of:

- A *Branch Target Buffer* (BTB) to identify branches and provide targets for direct branches.
- 2-level global history-based direction predictor.
- Indirect predictor to provide targets for indirect branches.
- Return stack.
- Static predictor.

The combination of global history-based direction predictor and BTB are called *dynamic predictor*.

This section contains the following subsections:

Predictors Used



- Static predictor – helps out before dynamic predictor warms up. Predicts as taken:
  - Unconditional direct branches
  - Unconditional direct call-type branches BL immediate (call).
    - Return address is pushed to Return Address stack.
  - Unconditional return branches.
    - Target popped from Return Address stack
- Dynamic predictor
  - Branch Target Buffer
  - 2-level global history-based direction predictor
- Indirect branch predictor
  - Stores branch address, state to predict target
- Return stack predictor
  - Pushes address from LR on BL or BLX
  - Pops address
    - BX lr; MOV pc,lr; LDMIA sp!, {..pc}; LDR pc, [sp], #4
  - Exception returns not predicted