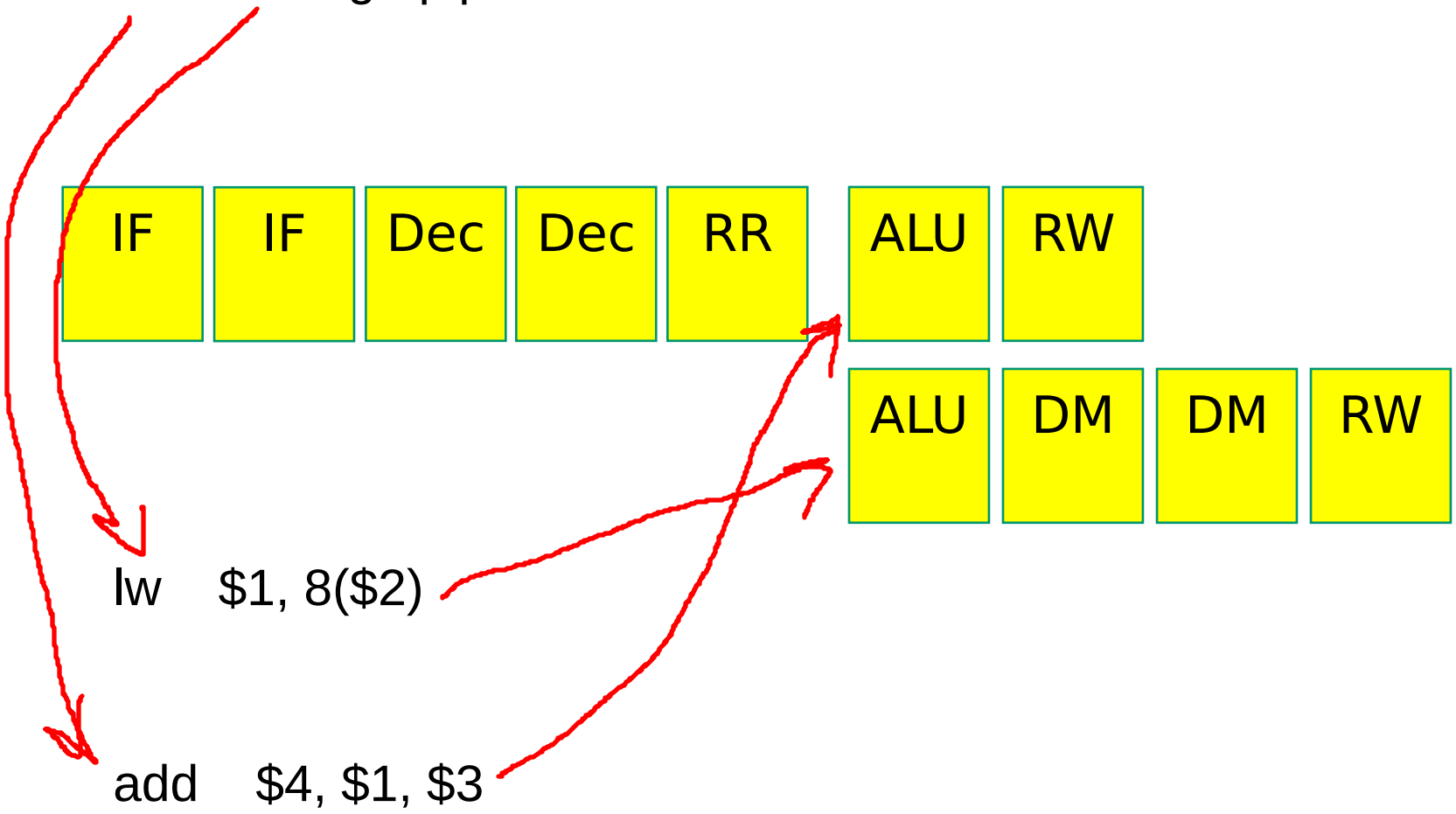
Lecture 19: Pipelining

- Today's topics:
  - Hazards and instruction scheduling
  - Branch prediction
  - Out-of-order execution

Problem 4

A 7 or 9 stage pipeline



Problem 4

แก้โดยการ ชะลอ decode ไปเรื่อยๆ
(Trick เขียนจุดเริ่ม แล้วจุดที่การกระทำพึ่งกับอีกอันควรจะอยู่
แล้วเติมตรงกลาง ถ้าเหลือก้เติม Decode)

Without bypassing: 4 stalls

IF:IF:DE:DE:RR:AL:DM:DM:RW

IF: IF :DE:DE:DE:DE:DE :DE:RR:AL:RW

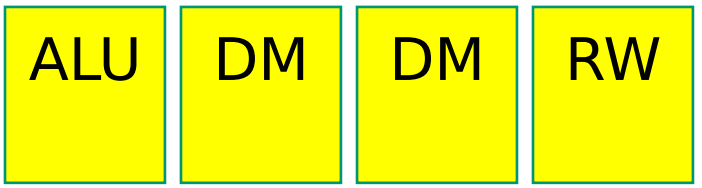
With bypassing: 2 stalls

IF:IF:DE:DE:RR:AL:DM:DM:RW

IF: IF :DE:DE:DE:DE:RR :AL:RW

lw $1, 8($2)

add $4, $1, $3



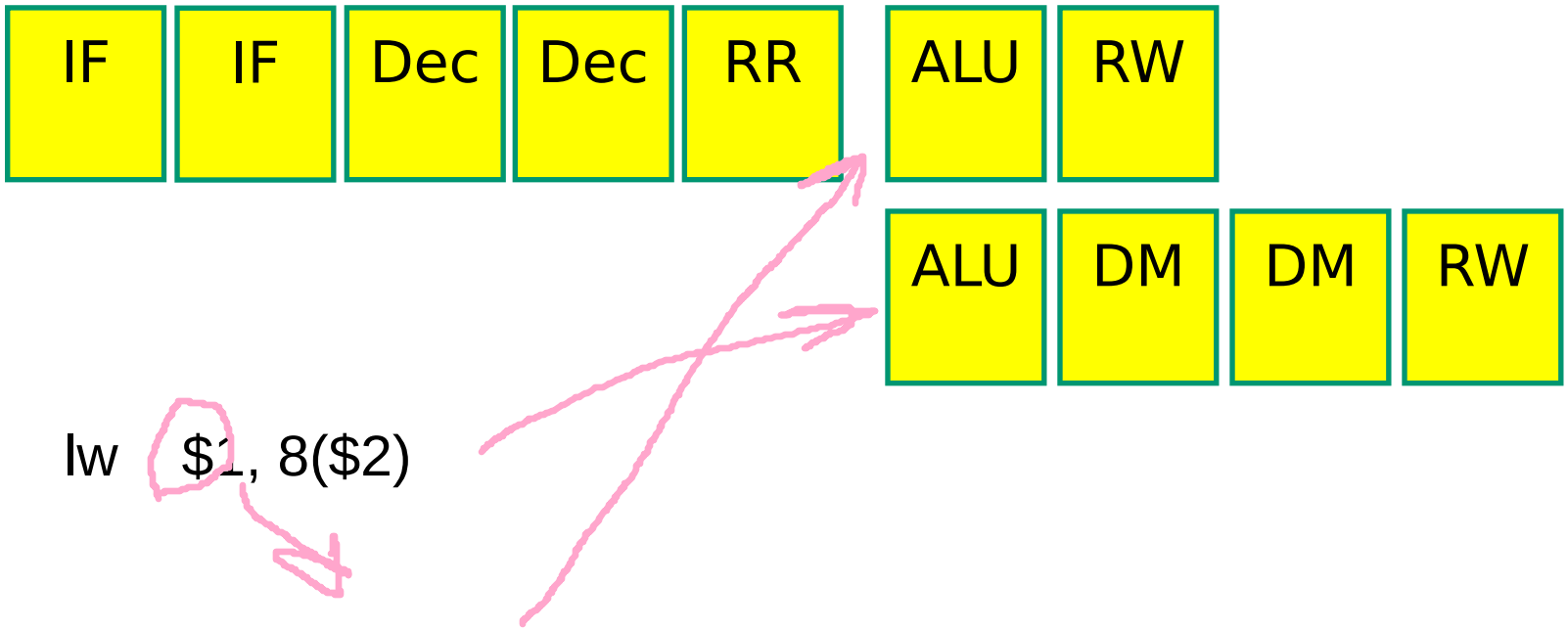
Lecture 19: Pipelining

Today's topics:

- Hazards and instruction scheduling
- Branch prediction
- Out-of-order execution

Problem 4

A 7 or 9 stage pipeline



Problem 4

Without bypassing: 4 stalls

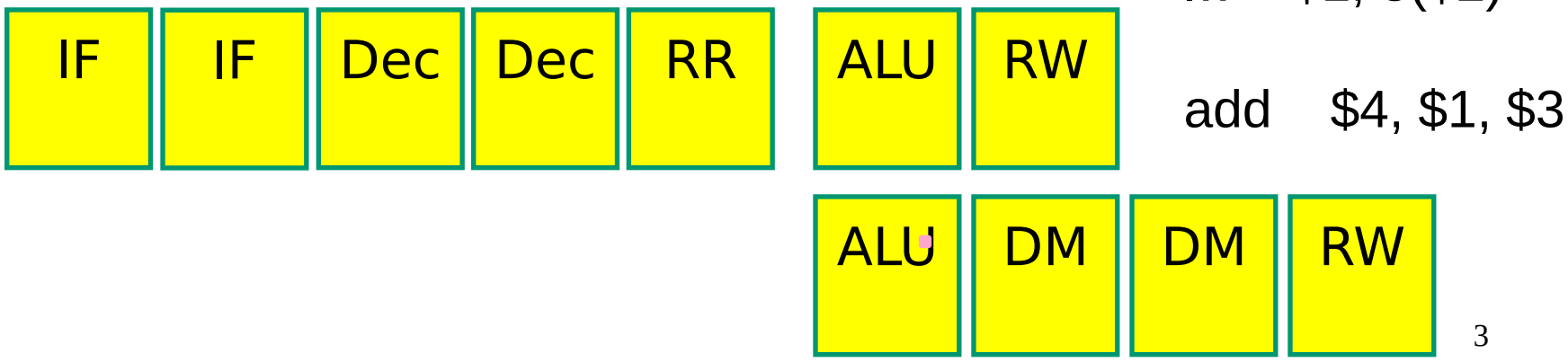
IF:IF:DE:DE:RR:AL:DM:DM:RW

IF: IF :DE:DE:DE:DE:DE :DE:RR:AL:RW

With bypassing: 2 stalls

IF:IF:DE:DE:RR:AL:DM:DM:RW

IF: IF :DE:DE:DE:DE:RR :AL:RW



Control Hazards

เสนอวิธีแก้

Simple techniques to handle control hazard stalls:

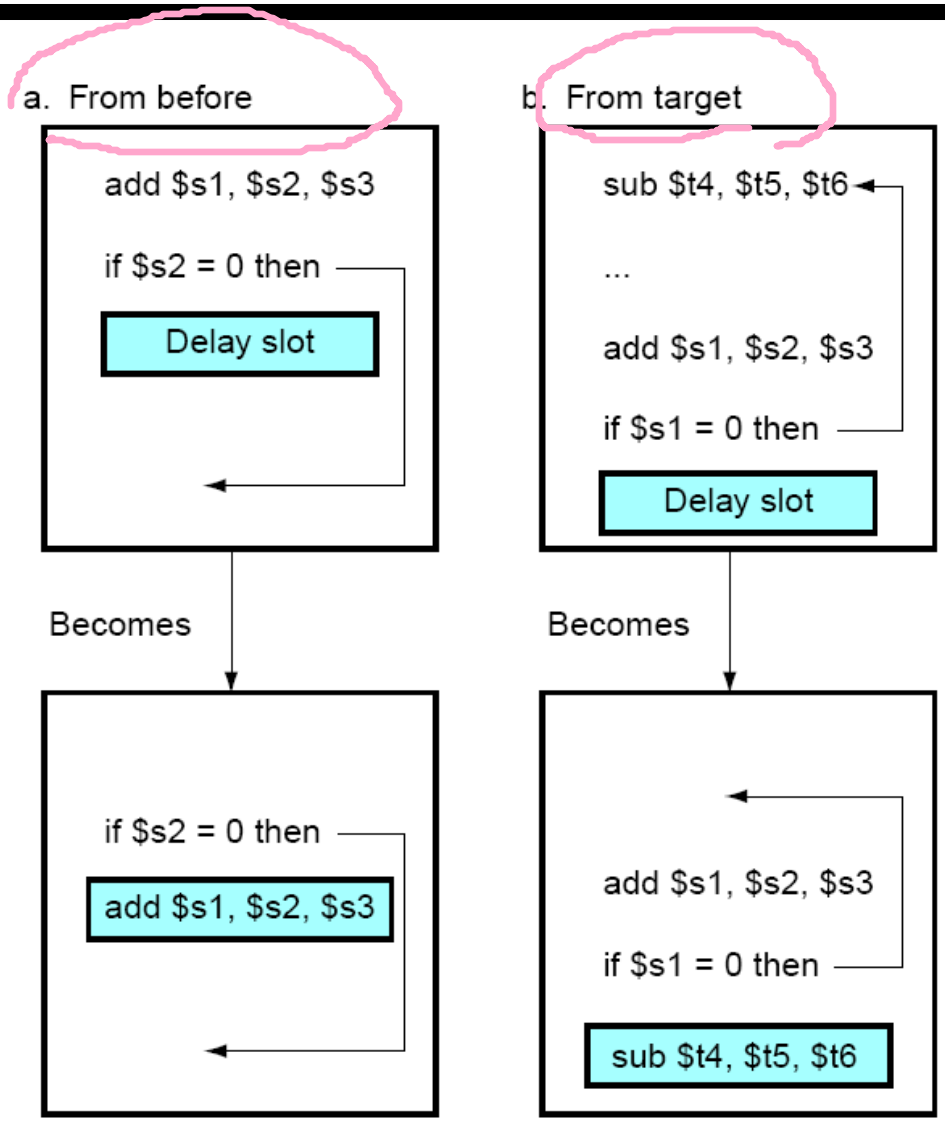
- for every branch, introduce a stall cycle (note: every 6th instruction is a branch!)
- assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instruction
- fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost
- make a smarter guess and fetch instructions from the expected target

คอมไพเลอหาอะไรที่มีประโยชน์มาทำรอใน branch delay slot

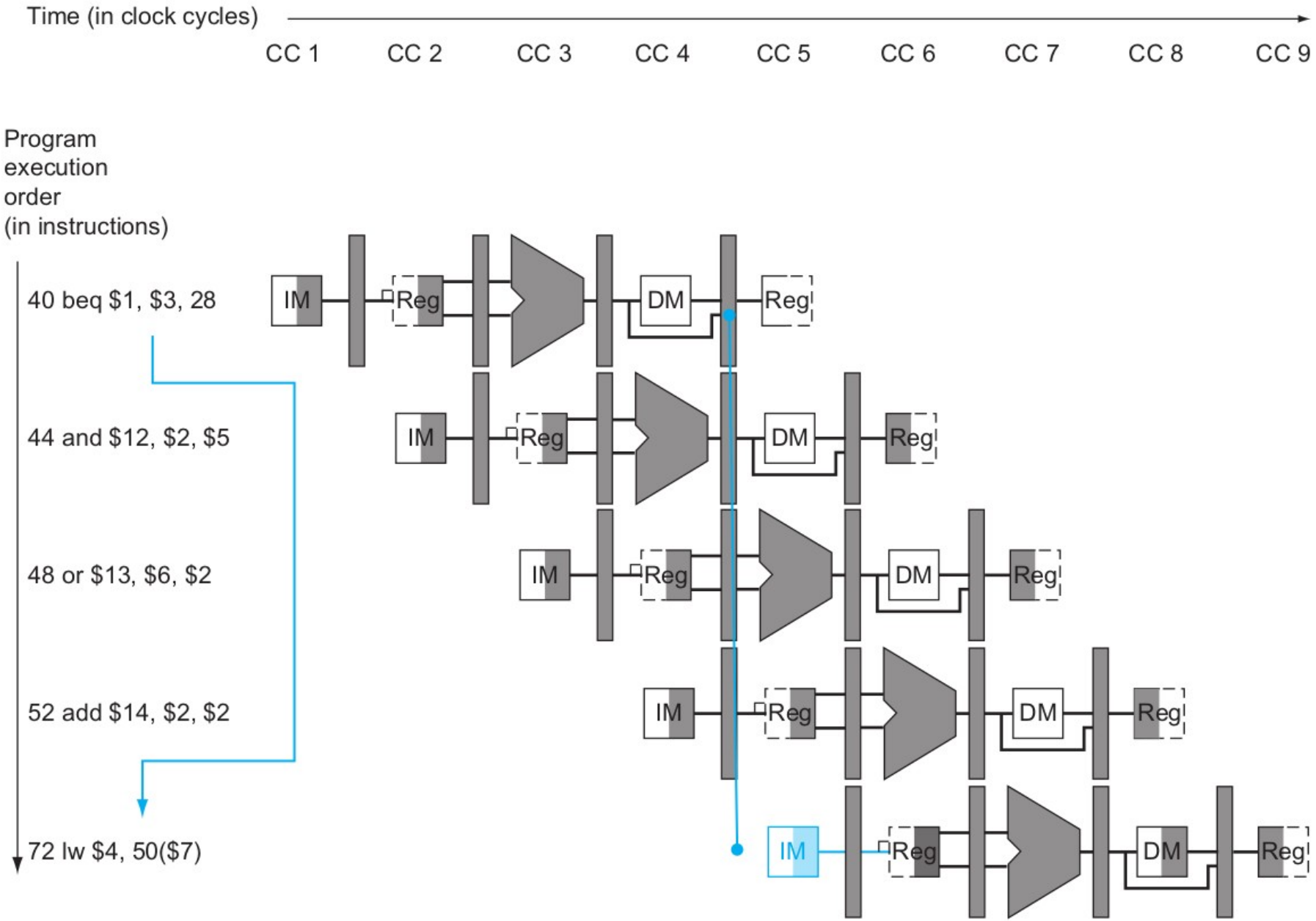
ให้โปรแกรมมั่วแบบฉลาด

Branch Delay Slots

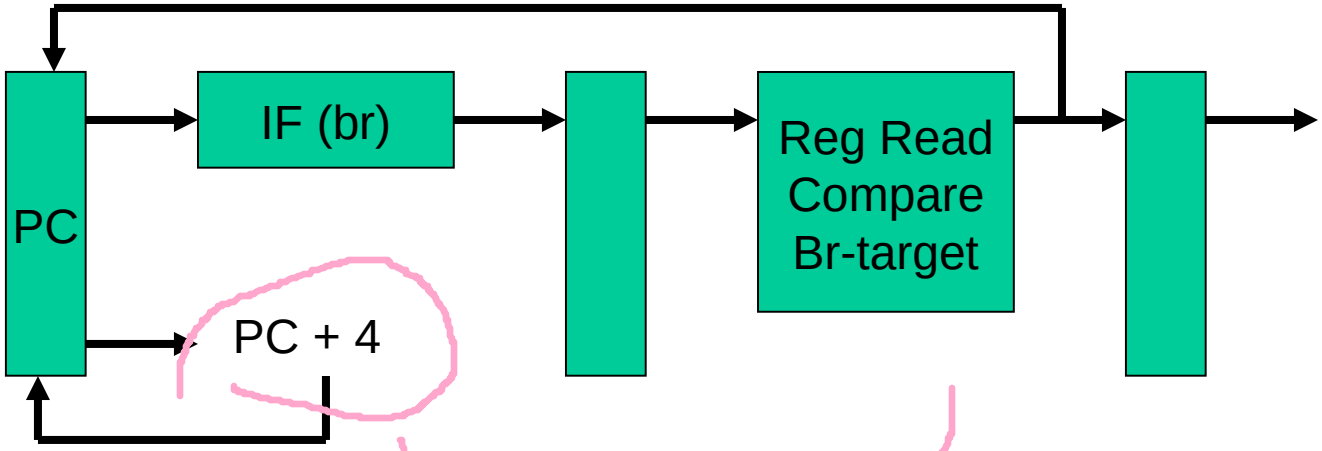
ให้ compiler หาอะไรที่มีประโยชน์หรือคาดการว่าจะมีประโยชน์มาทำระหว่าง Branch ประมวลเสร็จว่าต้องไปไหนต่อ



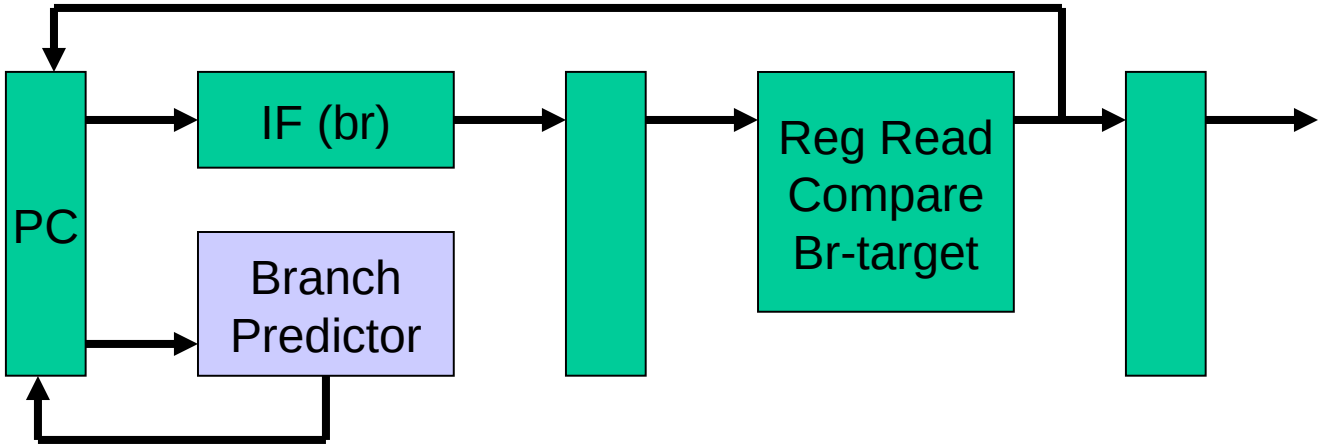
Source: H&P textbook


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
40 beq $1, $3, 28
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)
IM
Reg
DM
Reg

Pipeline without Branch Predictor



Pipeline with Branch Predictor



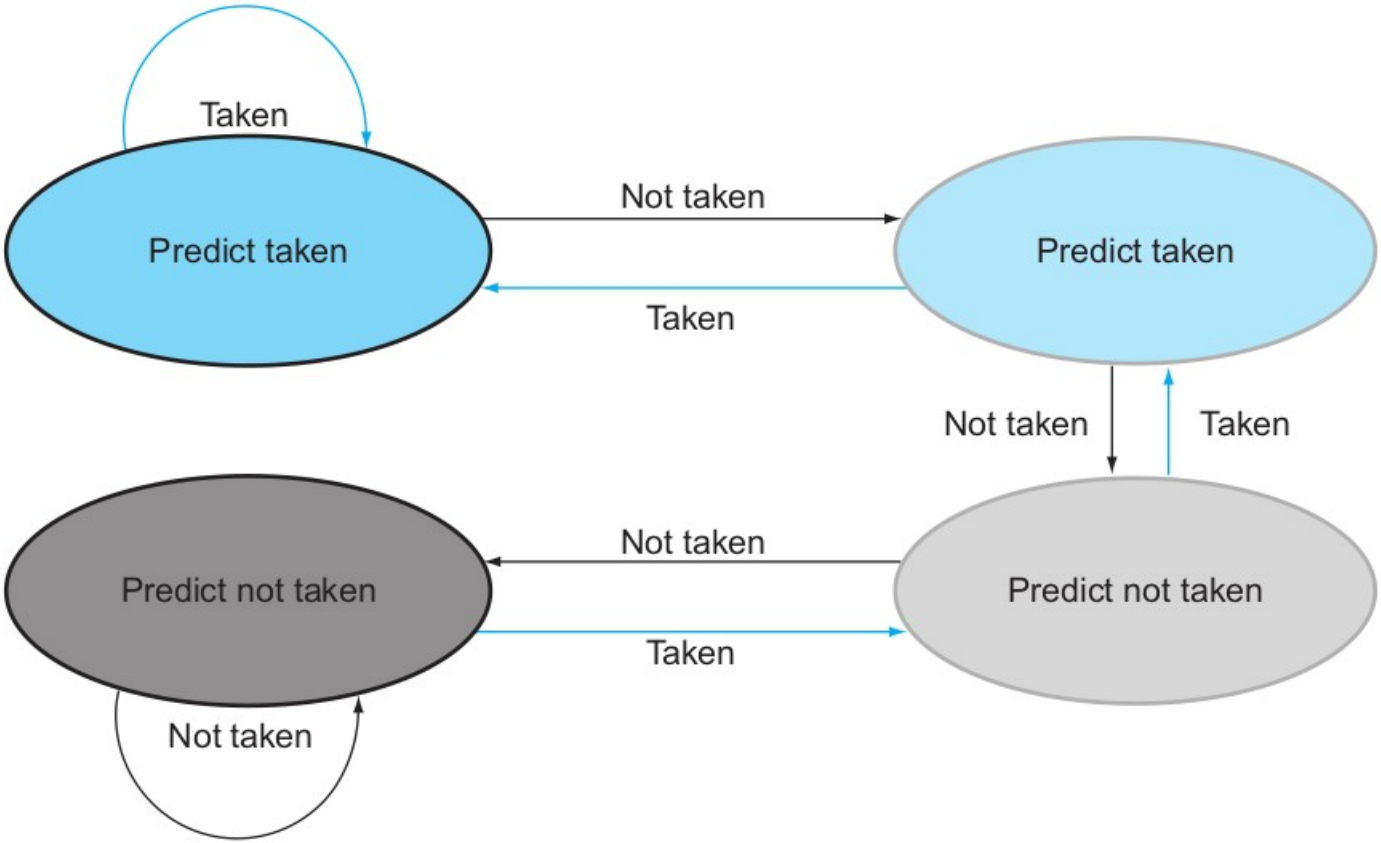
2-Bit Prediction

For each branch, maintain a 2-bit saturating counter:
if the branch is taken: counter = min(3,counter+1)
if the branch is not taken: counter = max(0,counter-1)
… sound familiar?

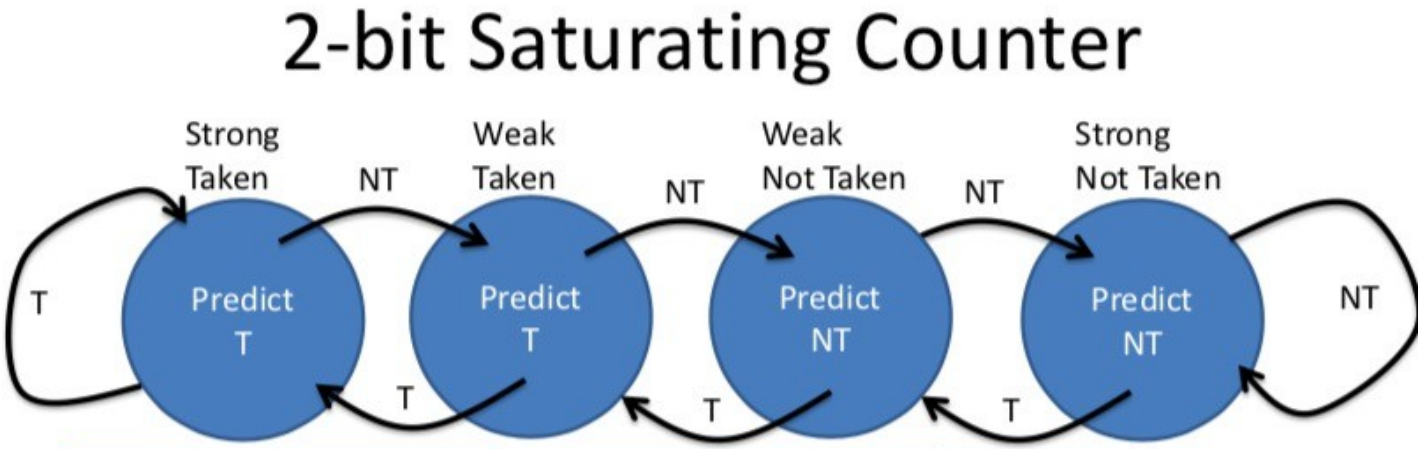
If (counter >= 2), predict taken, else predict not taken

The counter attempts to capture the common case for each branch

2-Bit Prediction



2-Bit Prediction

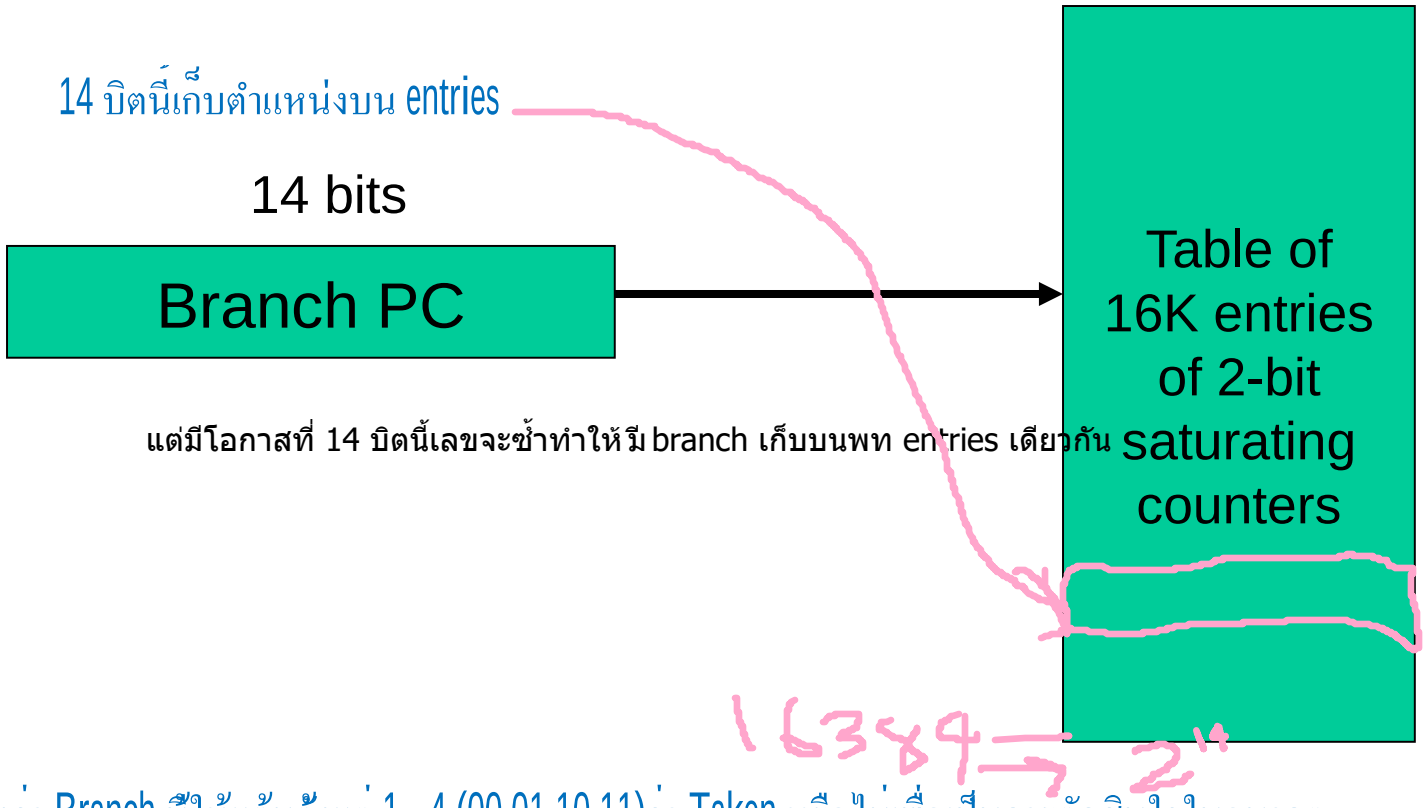


Iteration	Prediction	Actual	Mispredict?	State
1	NT	T	Y	Strong NT
2	NT	T	Y	Weak NT
3	T	T		Weak T
4	T	NT	Y	Strong T
...				
1	T	T		Weak T
2	T	T		Strong T
3	T	T		Strong T
4	T	NT	Y	Strong T

Only 1 Mispredict

19

Bimodal Predictor



ให้เก็บค่าสองบิทว่า Branch นี้ให้แต้มตั้งแต่ 1 - 4 (00 01 10 11)ว่า Taken หรือไม่เพื่อเป็นการตัดสินใจในอนาคต
Train ตัวเอง ตรวจดูจาก bit 1 ตัวซ้าย ถ้าเป็น 1 จะเทค 0 ไม่เทค เช่น 10 เทค แต่ 01 ไม่เทค

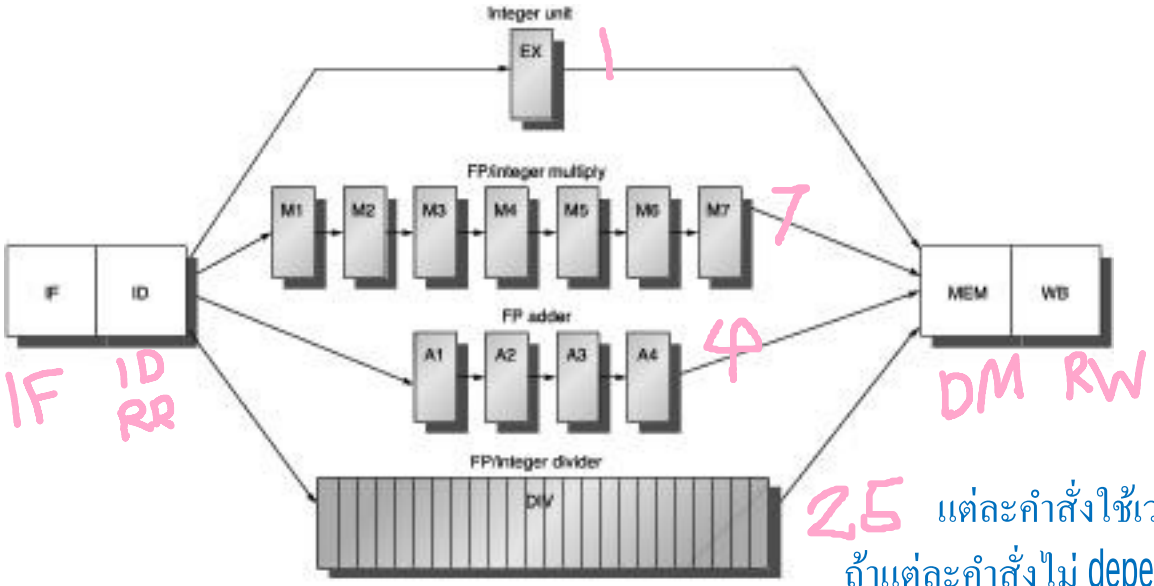
Slowdowns from Stalls

Perfect pipelining with no hazards an instruction completes every cycle (total cycles ~ num instructions)
 speedup = increase in clock speed = num pipeline stages

With hazards and stalls, some cycles (= stall time) go by during which no instruction completes, and then the stalled instruction completes

Total cycles = number of instructions + stall cycles

Multicycle Instructions นี่คือ in order 5 stage pipeline แบบเพิ่มให้รองรับคำสั่งมากขึ้น

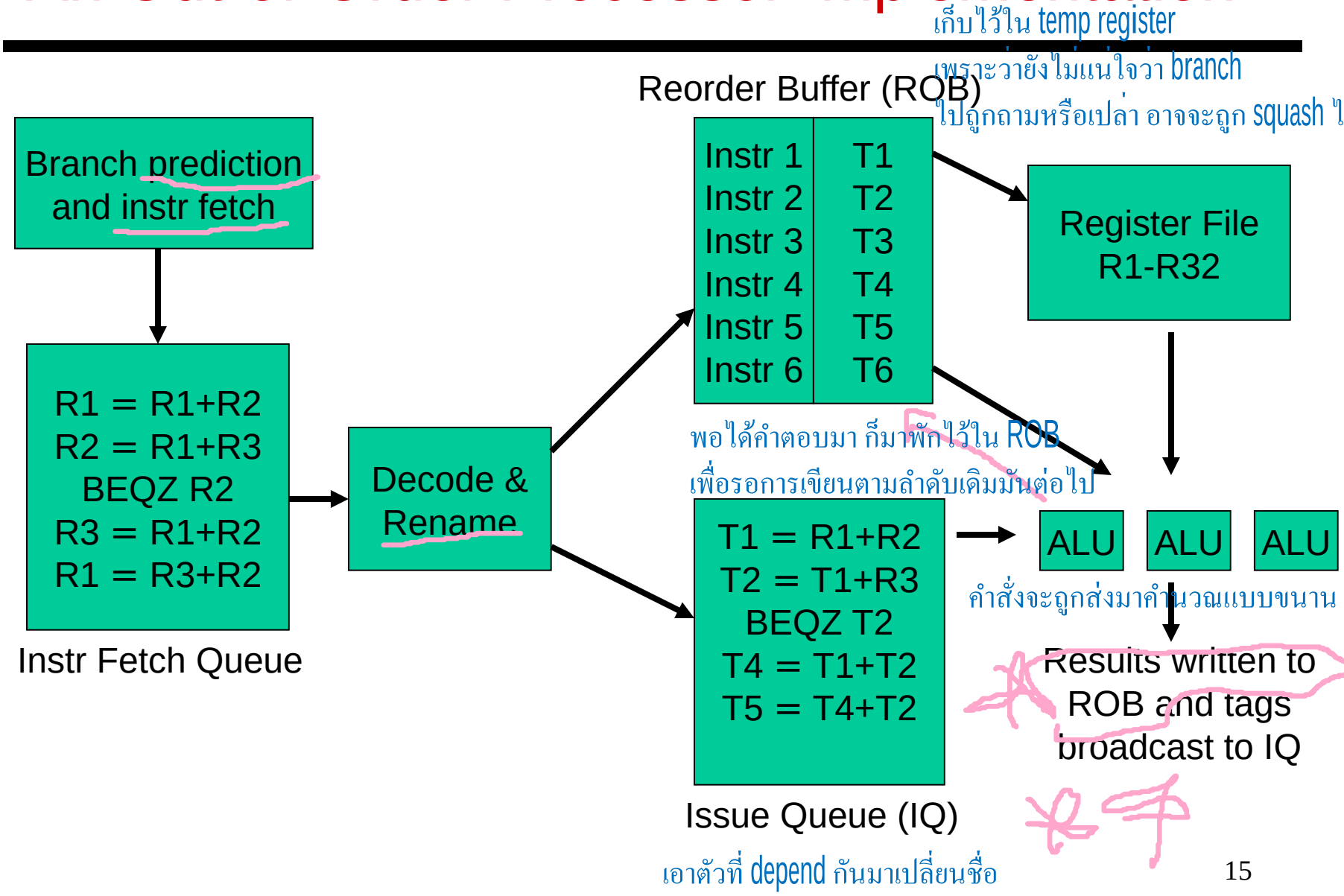


แต่ละคำสั่งใช้เวลาต่างกัน
ถ้าแต่ละคำสั่งไม่ depend กันก้ไม่มีปัญหา แต่ถ้ามีก็จะเกิดปัญหาได้ แก้โดยมี buffer มาคำตอบไว้รอเขียนลงตามลำดับคำสั่งจริงๆของตัวนั้นๆ

Multiple parallel pipelines – each pipeline can have a different number of stages

Instructions can now complete out of order – must make sure that writes to a register happen in the correct order

An Out-of-Order Processor Implementation



Slowdowns from Stalls

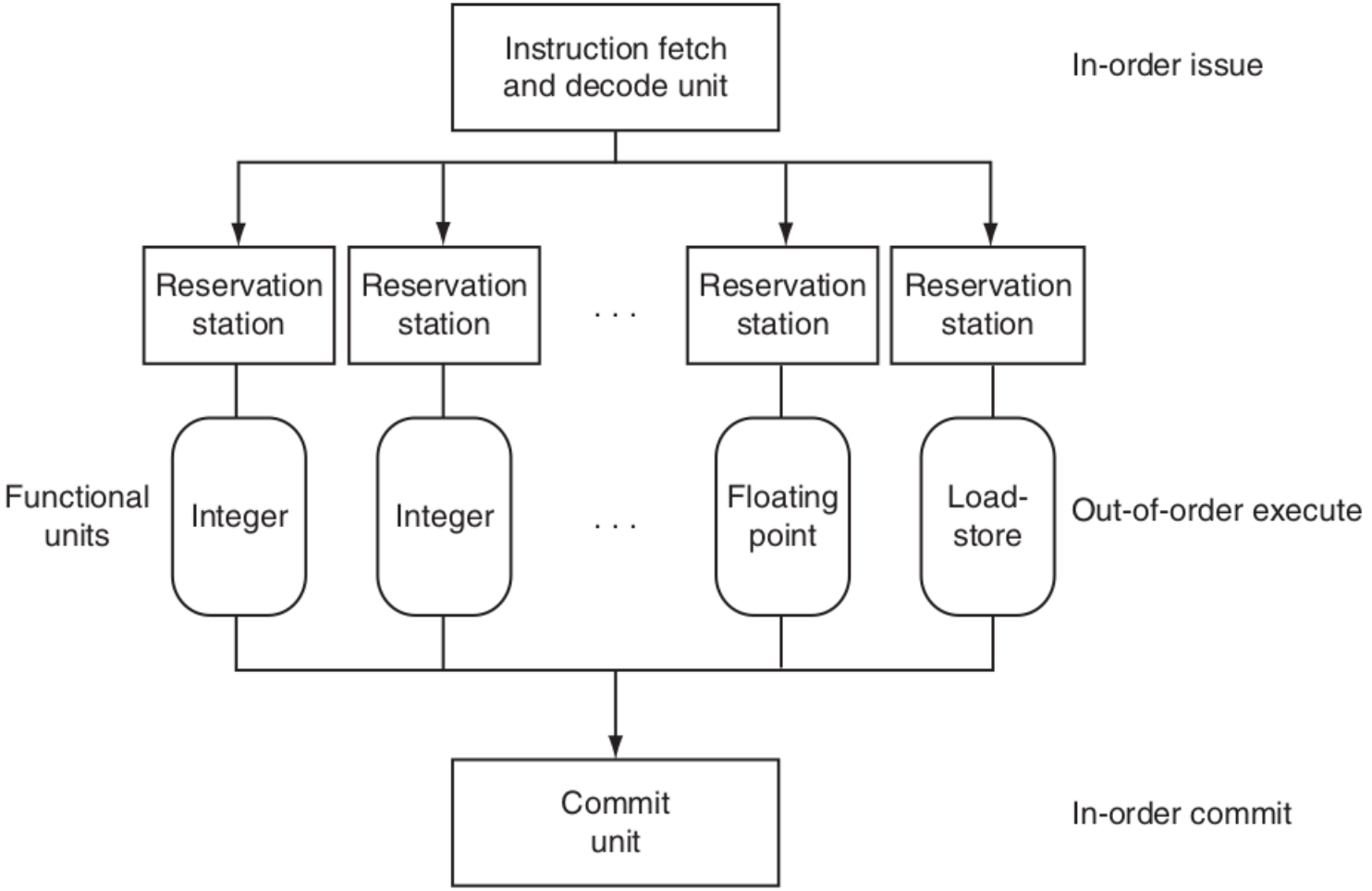
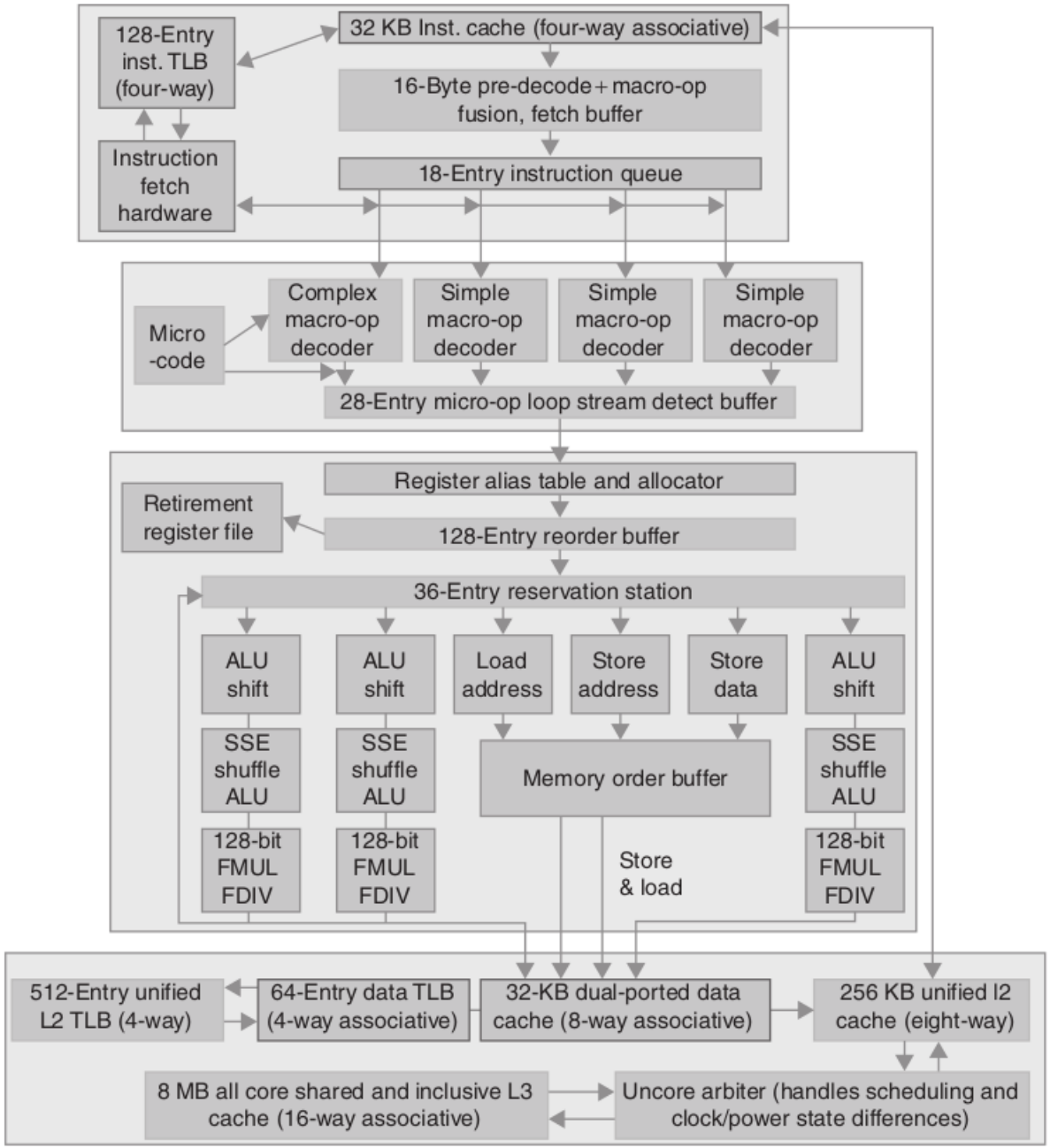


FIGURE 4.72 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.



Intel
Core i7 920

Core i7 920 SPEC2006 Integer

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

Core i7 920 SPEC2006 Integer

The formula for the geometric mean is

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

where Execution time ratio$_i$ is the execution time, normalized to the reference computer, for the ith program of a total of n in the workload, and

$$\prod_{i=1}^{n} a_i \text{ means the product } a_1 \times a_2 \times \ldots \times a_n$$

Core i7 920 SPEC2006 Integer

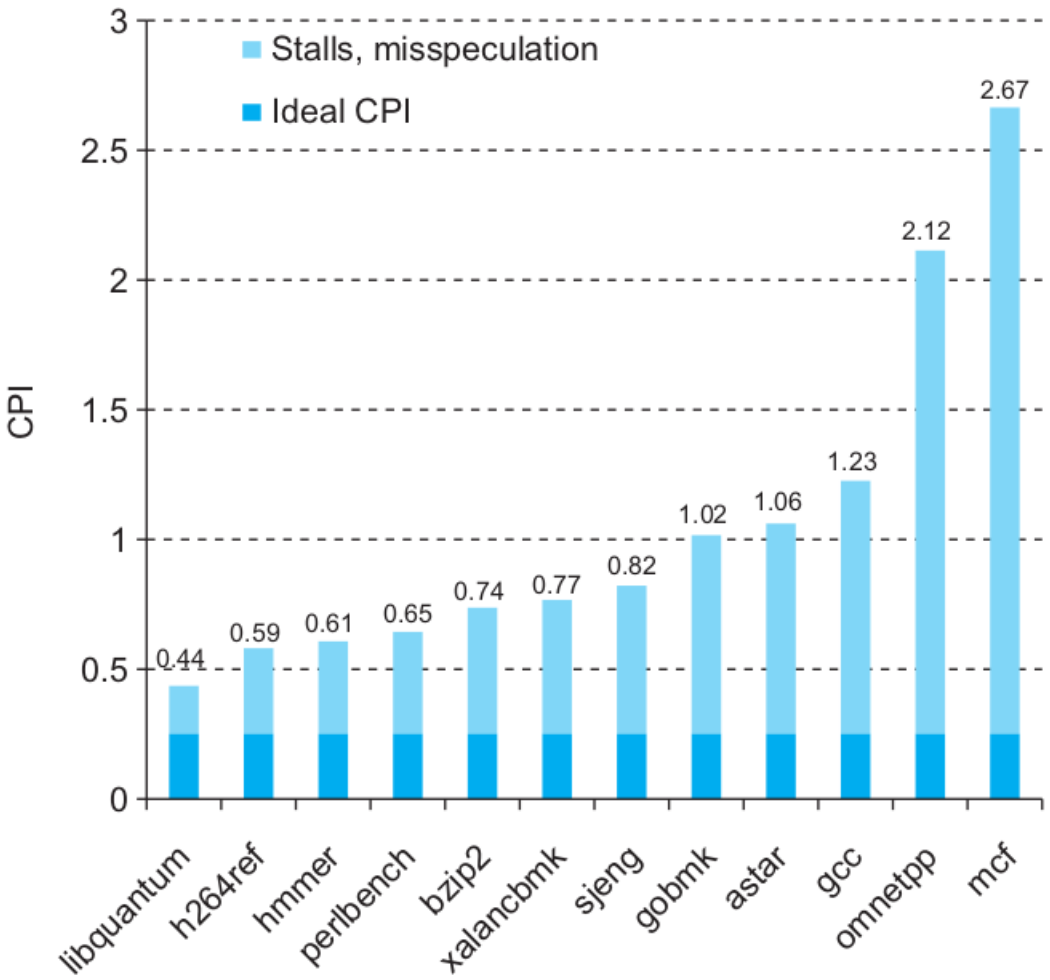


FIGURE 4.78 CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

Core i7 920 SPEC2006 Integer

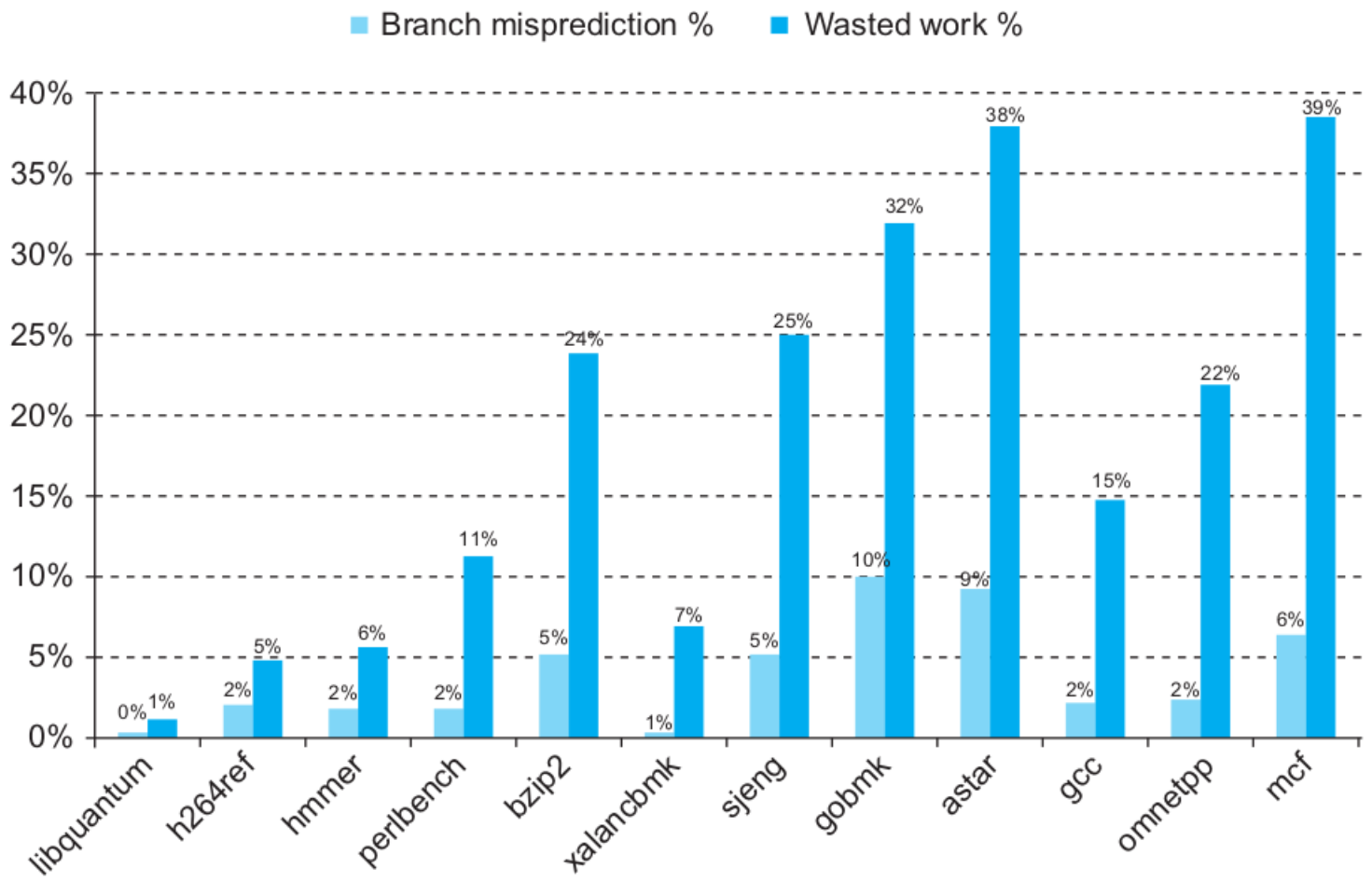
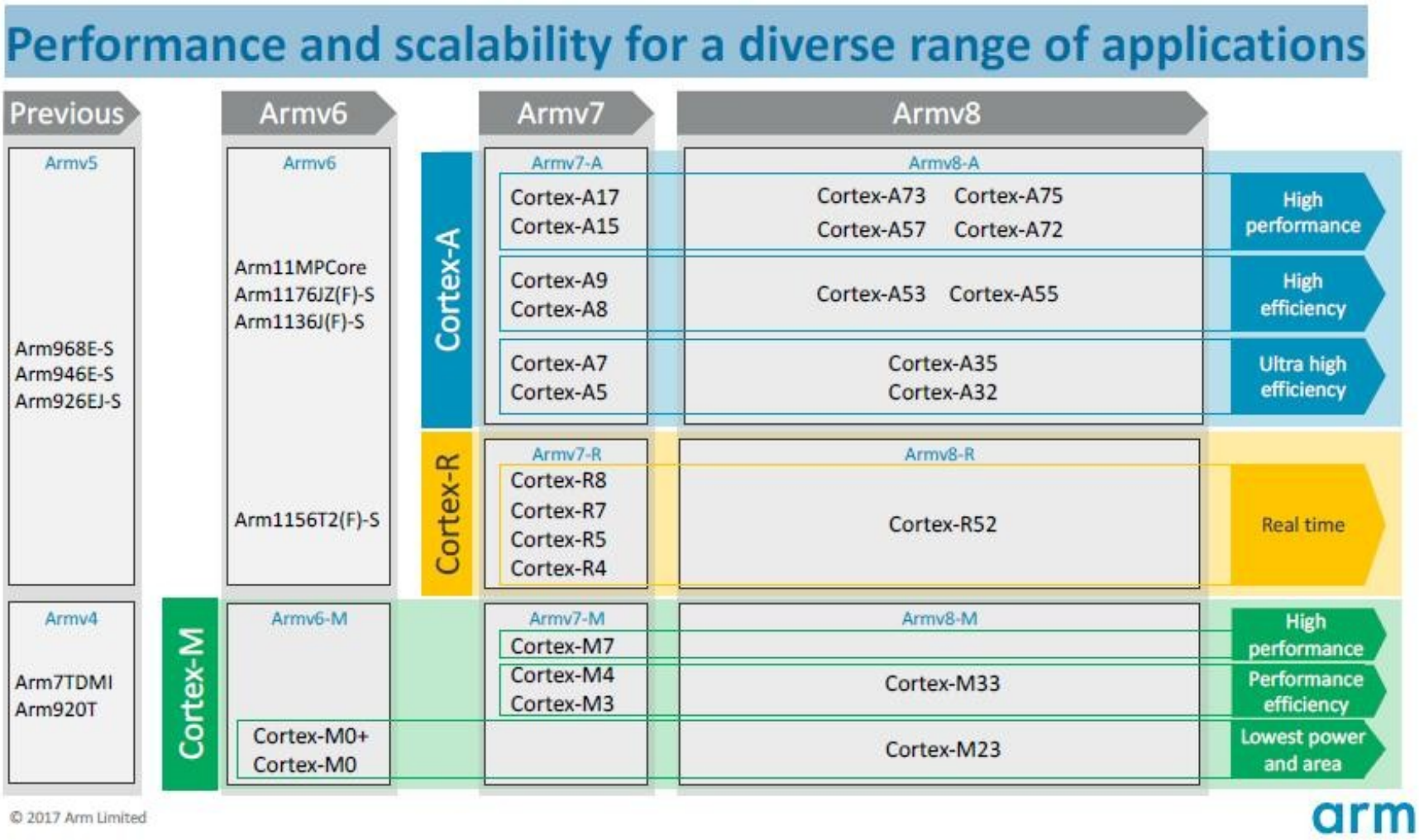


FIGURE 4.79 Percentage of branch mispredictions and wasted work due to unfruitful speculation of Intel Core i7 920 running SPEC2006 integer benchmarks.

ARM Cortex Roadmap



Cortex A7 Pipeline In-Order Execution

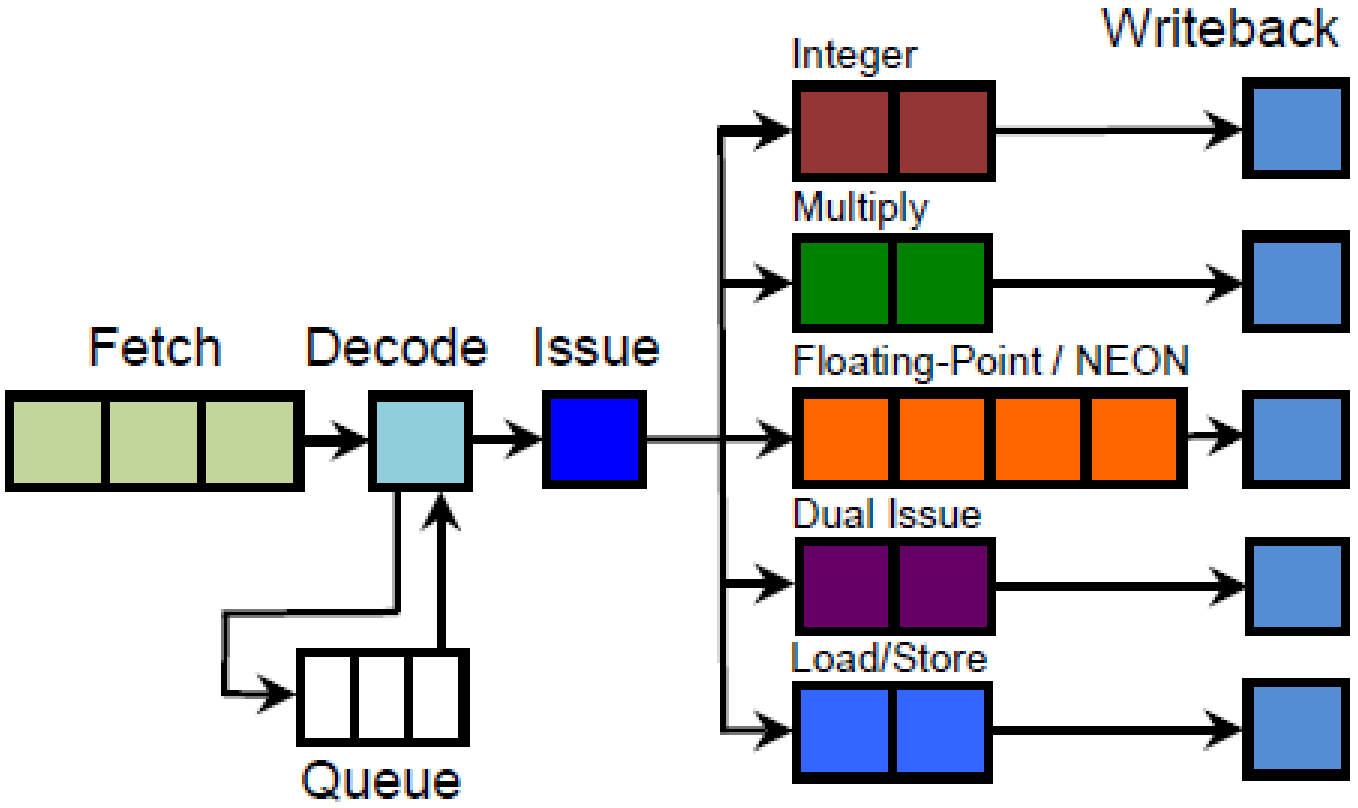
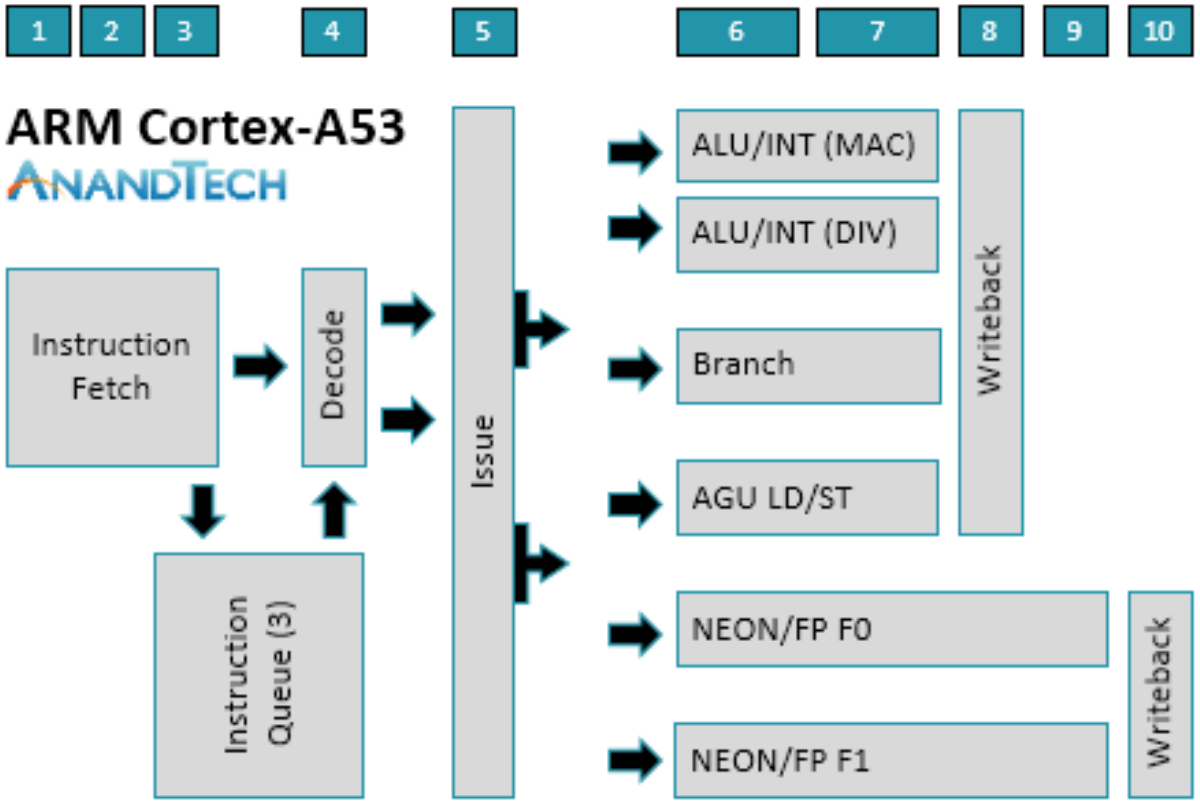


Figure 1 Cortex-A7 Pipeline

Cortex A53 Pipeline In-Order Execution



Cortex A8 Pipeline In-Order Execution

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	--	2 - 8 MiB

FIGURE 4.74 Specification of the ARM Cortex-A8 and the Intel Core i7 920.

Cortex A8 Pipeline In-Order Execution

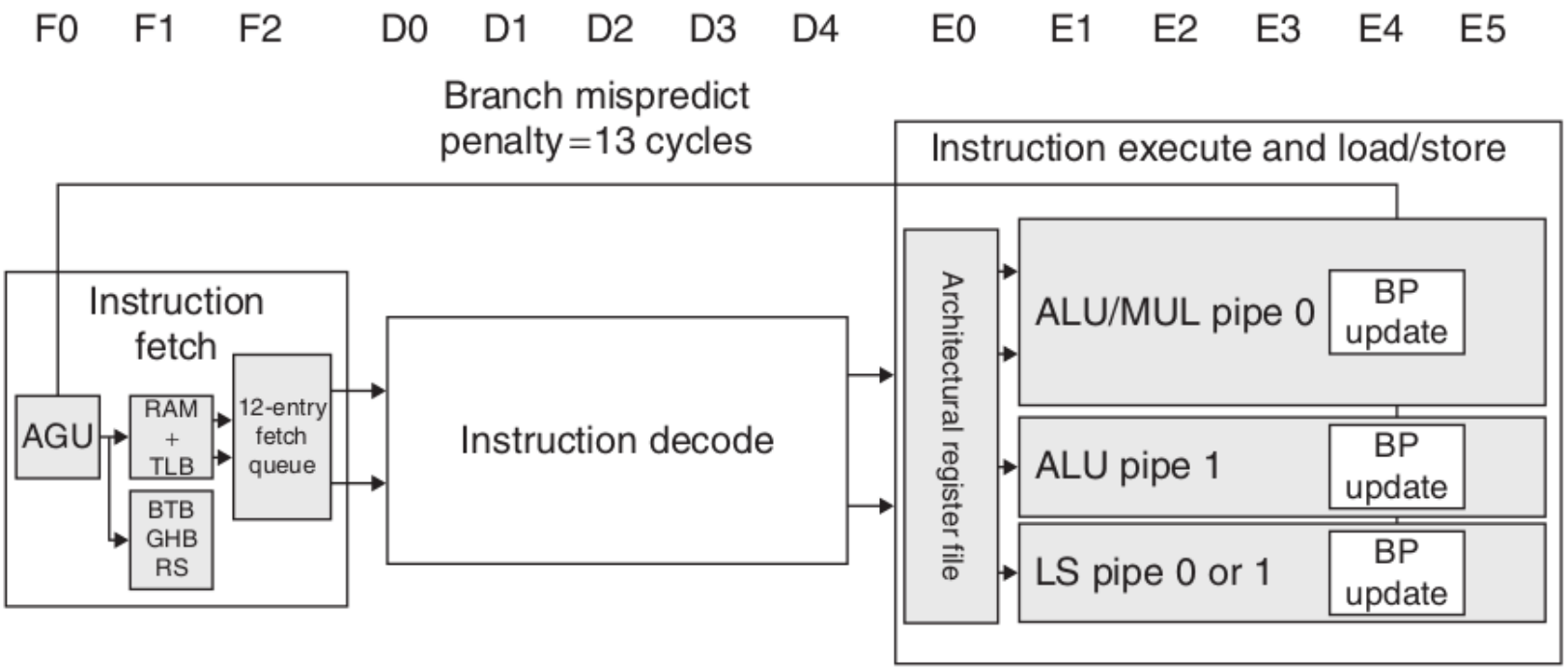


FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The *Address Generation Unit* (AGU) uses a *Branch Target Buffer* (BTB), *Global History Buffer* (GHB), and a *Return Stack* (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.

CPI ARM Cortex A8 SPEC2000

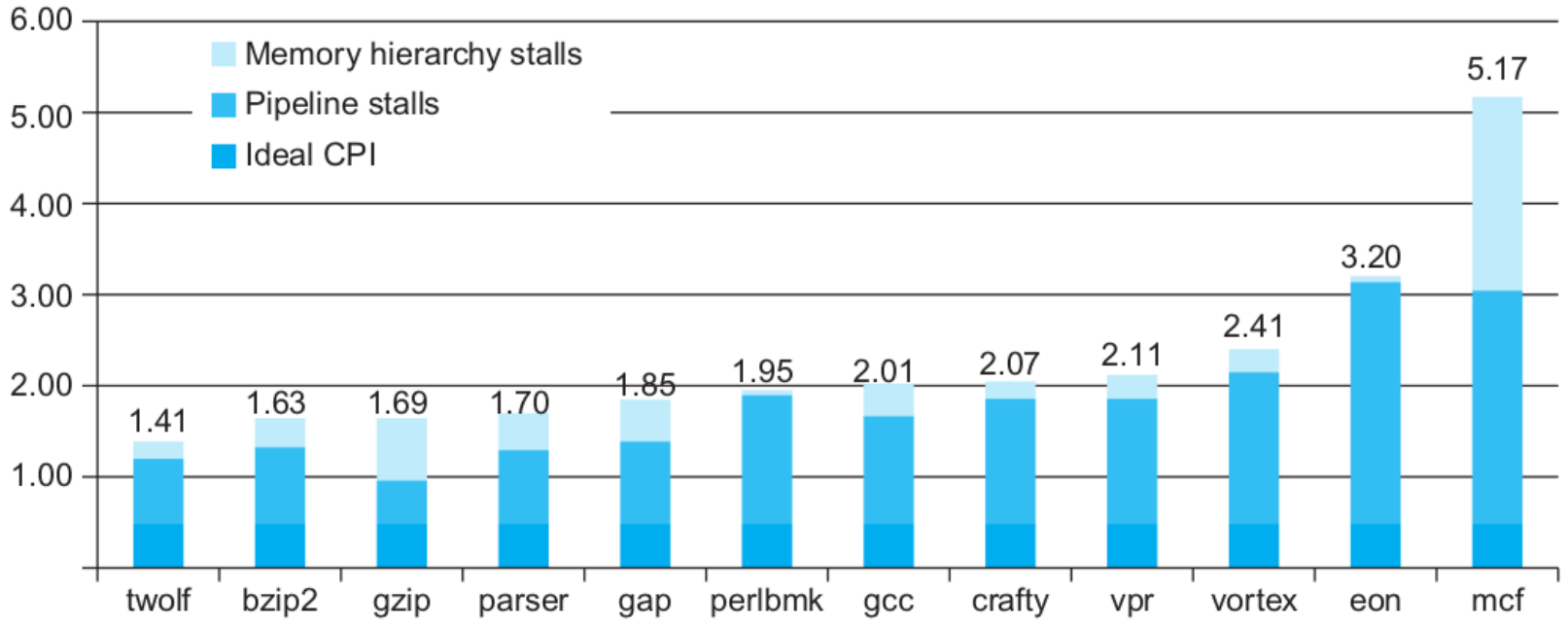


FIGURE 4.76 CPI on ARM Cortex A8 for the Minnespec benchmarks, which are small versions of the SPEC2000 benchmarks. These benchmarks use the much smaller inputs to reduce running time by several orders of magnitude. The smaller size significantly *underestimates* the CPI impact of the memory hierarchy (See Chapter 5).

Cortex A15 Pipeline Out-Order Execution

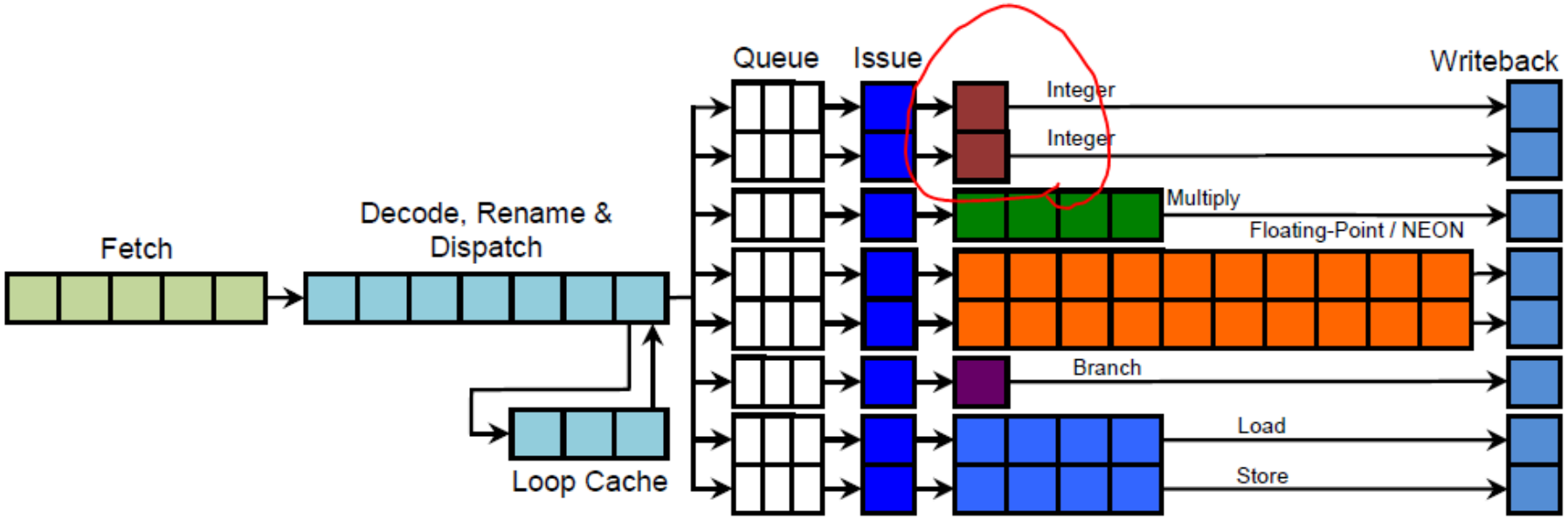
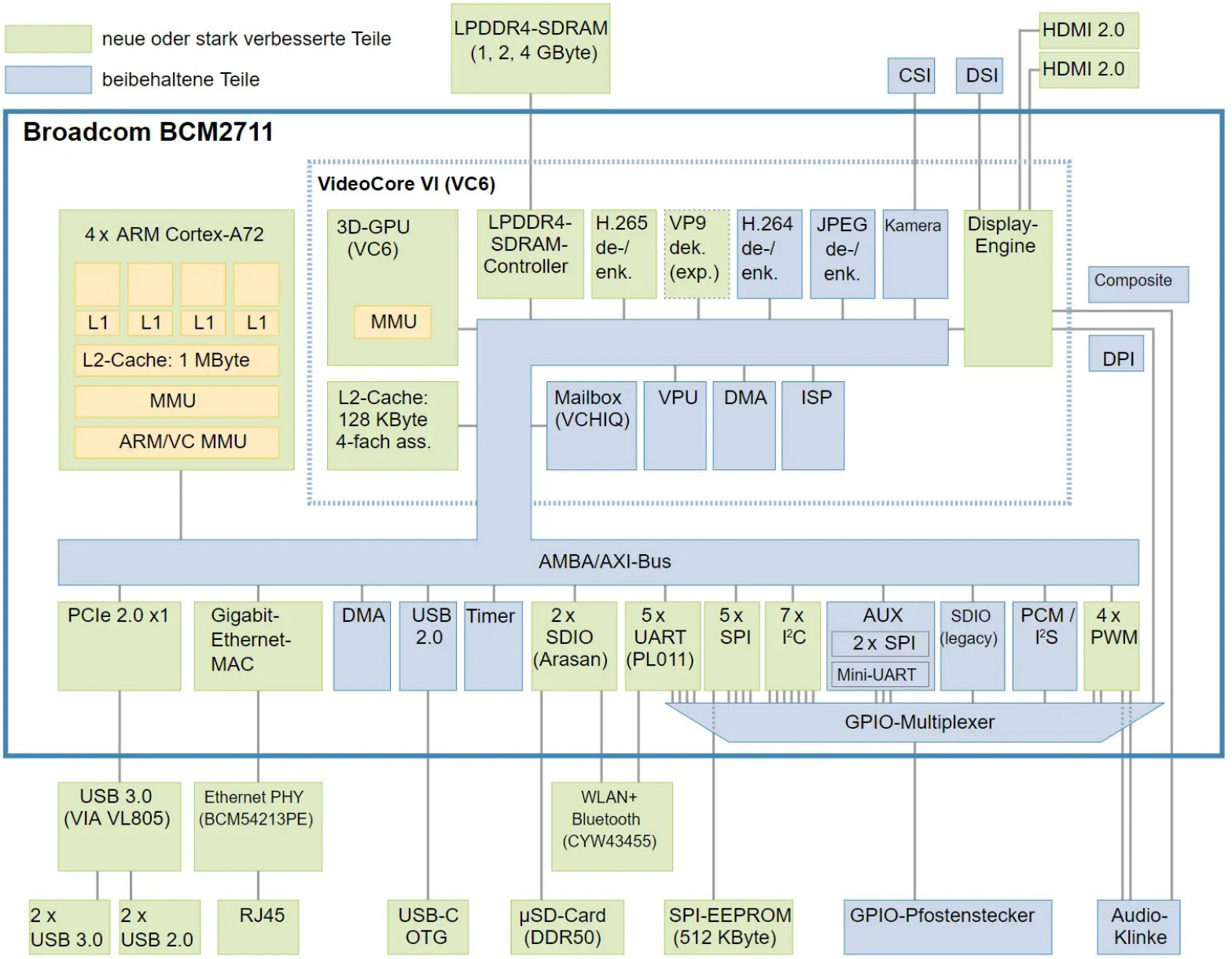


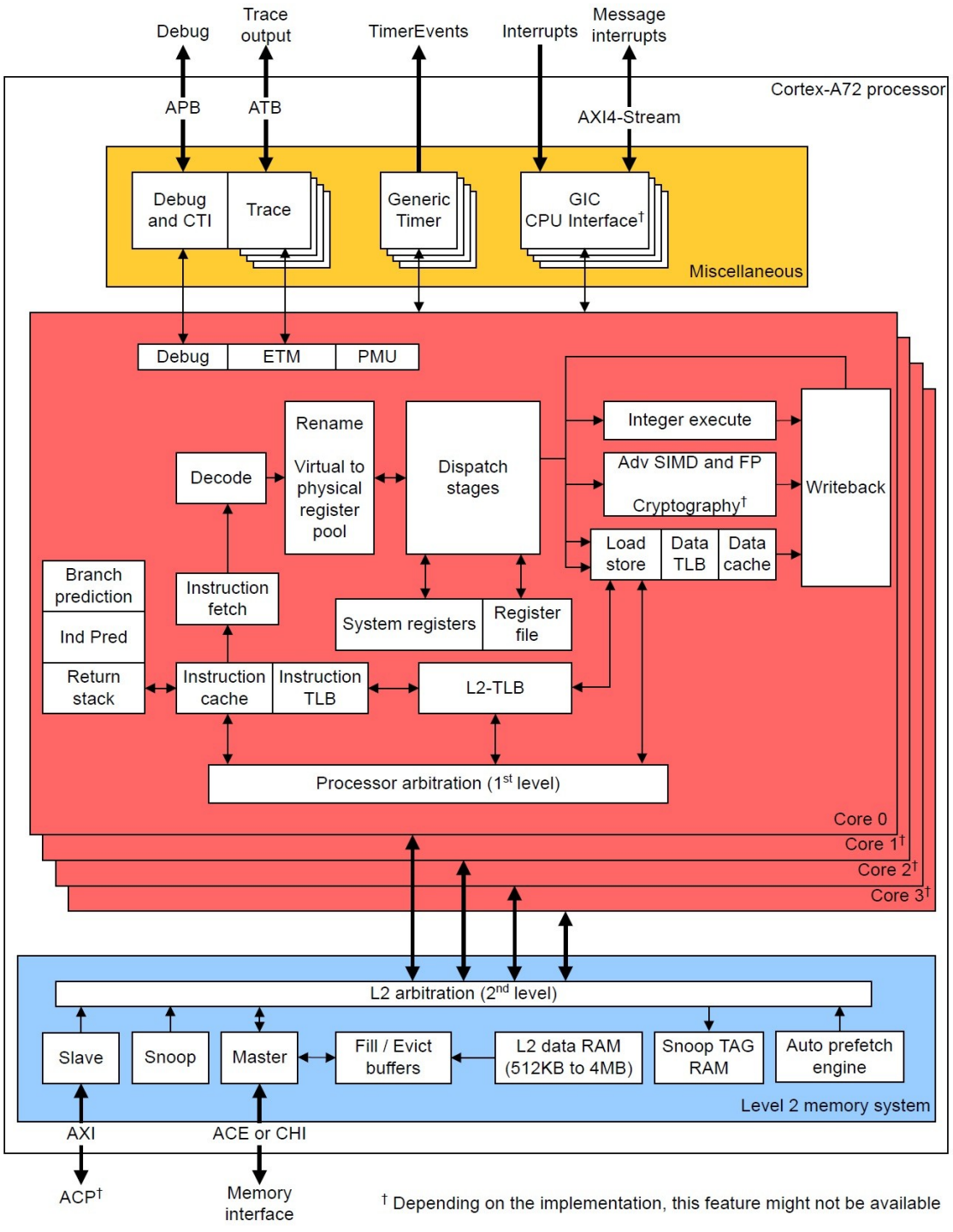
Figure 2 Cortex-A15 Pipeline

Herz des Raspberry Pi 4: Broadcom BCM2711

Das System-on-Chip (SoC) BCM2711 vereint nicht nur vier CPU-Kerne mit einer GPU, sondern enthält auch Controller für viele Schnittstellen.



Cortex A72


Debug
Trace output
TimerEvents
Interrupts
Message interrupts
Cortex-A72 processor
APB
ATB
AXI4-Stream
Debug and CTI
Trace
Generic Timer
GIC CPU Interface†
Miscellaneous
Debug
ETM
PMU
Rename
Virtual to physical register pool
Dispatch stages
Integer execute
Adv SIMD and FP
Cryptography†
Writeback
Decode
Load store
Data TLB
Data cache
Branch prediction
Ind Pred
Return stack
Instruction fetch
System registers
Register file
Instruction cache
Instruction TLB
L2-TLB
Processor arbitration (1st level)
Core 0
Core 1†
Core 2†
Core 3†
L2 arbitration (2nd level)
Slave
Snoop
Master
Fill / Evict buffers
L2 data RAM (512KB to 4MB)
Snoop TAG RAM
Auto prefetch engine
Level 2 memory system
AXI
ACE or CHI
ACP†
Memory interface
† Depending on the implementation, this feature might not be available