HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Lecture 9

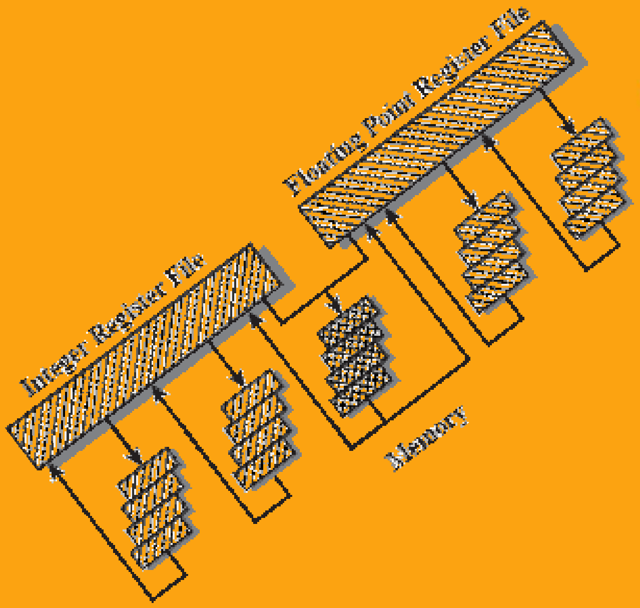
Superscalar- processing

Stallings: Ch 14

Instruction dependences

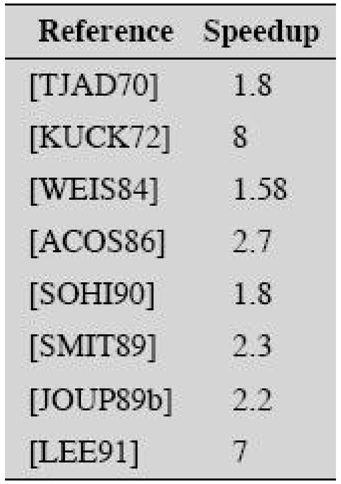
Register renaming

Pentium / PowerPC

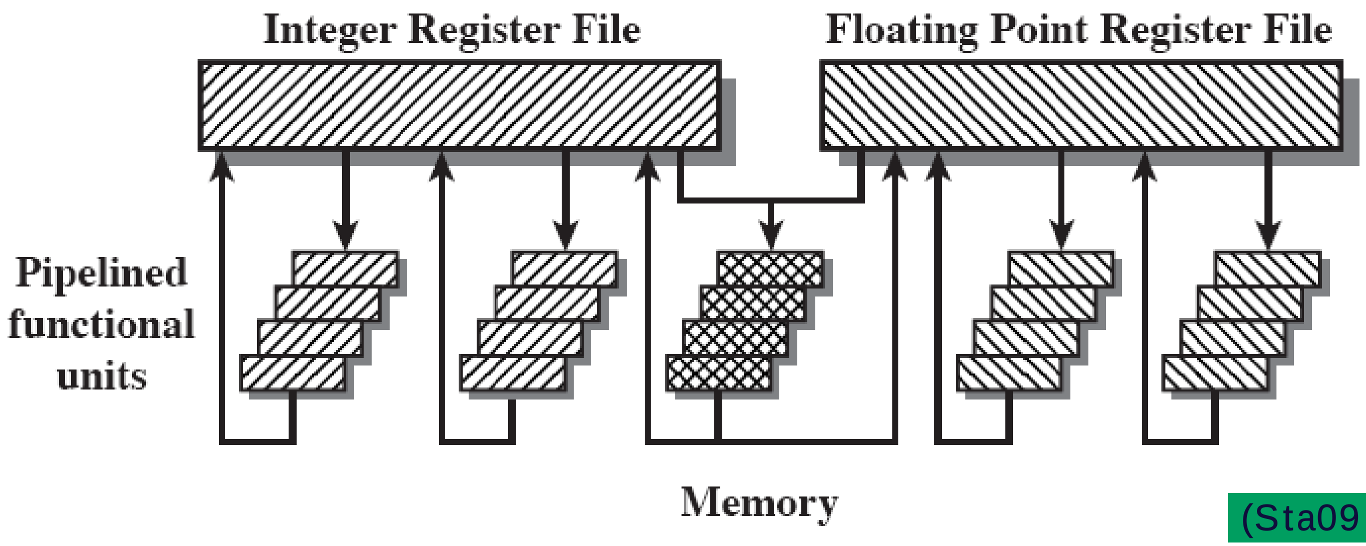


Superscalar processors

- Goal
  - Concurrent execution of scalar instructions
- Several independent pipelines
  - Not just more stages in one pipeline
  - Own functional units in each pipeline



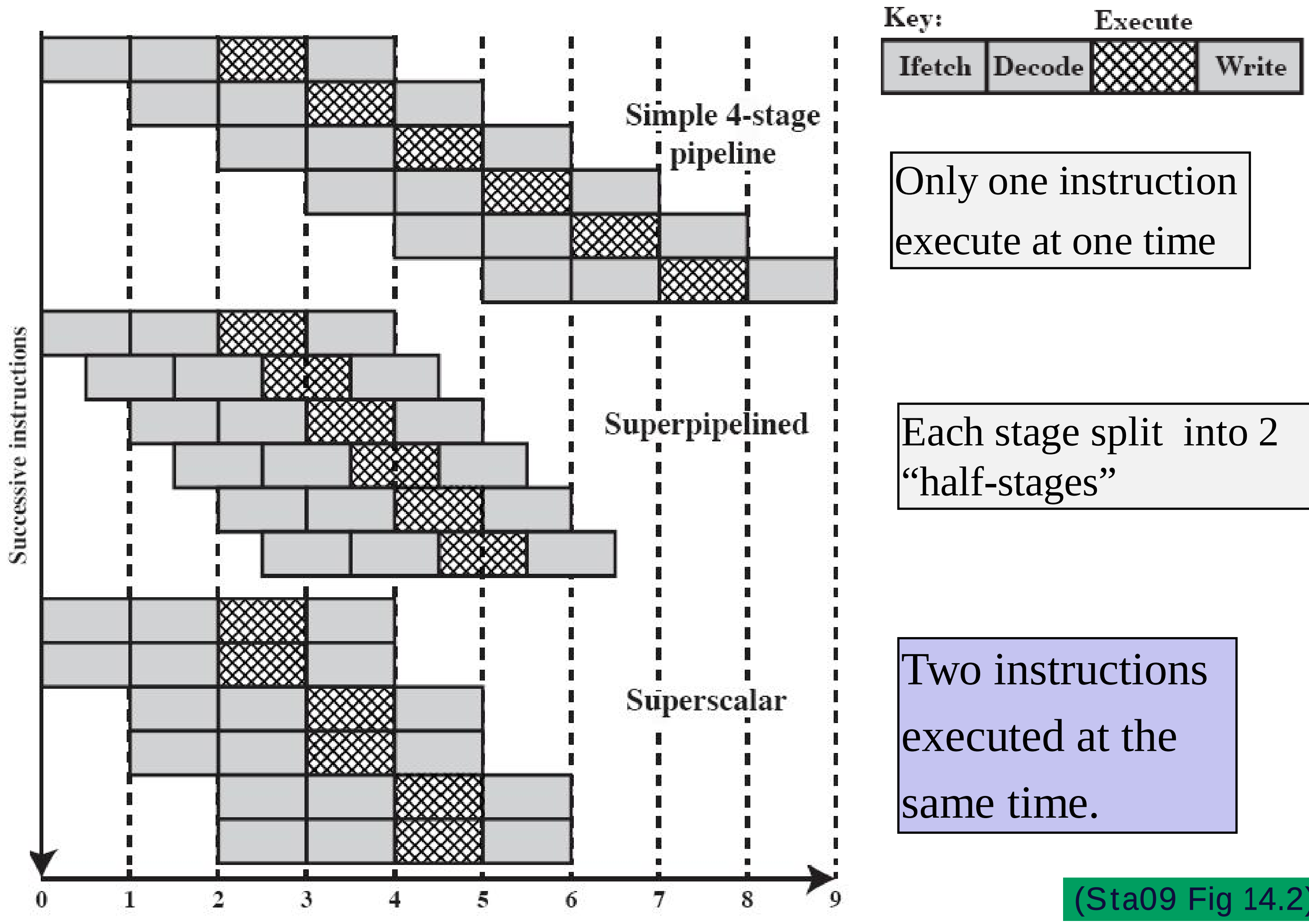
Reference	Speedup
[TJAD70]	1.8
[KUCK72]	8
[WEIS84]	1.58
[ACOS86]	2.7
[SOHI90]	1.8
[SMIT89]	2.3
[JOUP89b]	2.2
[LEE91]	7



Instruction-level parallelism

(Sta09 Fig 14.1, Tbl 14.1)

Superscalar



Only one instruction execute at one time

Each stage split into 2 "half-stages"

Two instructions executed at the same time.

(Sta09 Fig 14.2)

Superscalar processor

- Efficient memory usage
  - Fetch several instructions at once, prefetching (*ennaltanouto*)
  - Data fetch and store (read and write)
  - Concurrency
- Several instructions of the <u>same</u> process executed concurrently on different pipelines
  - Select executable instruction from the prefetched one following a policy (in-order issue/out-of-order issue)
- Finish more than one instruction during each cycle
  - Instructions may complete in different order than started (out-of-order completion)
- When can an instruction finish before the preceeding ones?

Dependencies (*riippuvuus*)

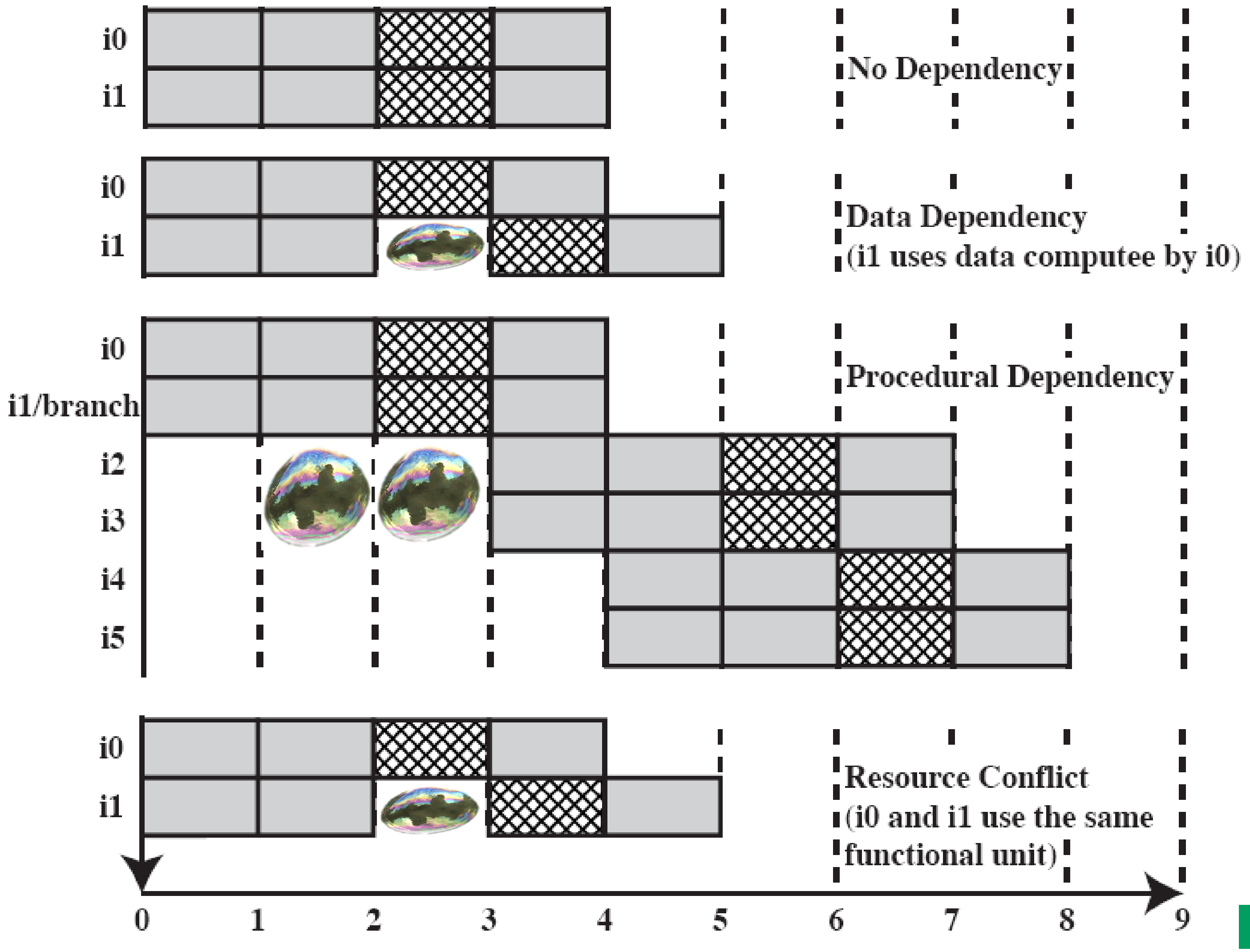
```
add r1,r2
move r3,r1
```

- True Data/Flow Dependency (*datariippuvuus*)
  - Read after Write (RAW)
  - The latter instruction needs data from former instruction
- Procedural/Control Dependency (*kontrolliriippuvuus*)
  - Instruction after the jump executed only, when jump does not happen
  - Superscalar pipeline has more instructions to waste
  - Variable-length instructions: some additional parts known only during execution

```
JNZ  R2, 100
ADD  R1, =1
```

- Resource Conflict (*Resurssiriippuvuus*)
  - One or more pipeline stage needs the same resource
  - Memory buffer, ALU, access to register file, ...

Effect of dependencies



(Sta09 Fig 14.3)

Dependencies specific to out-of-order completion

- Output Dependency (Kirjoitusriippuvuus)
  - write-after-write (WAW)
  - Two instructions alter the same register or memory location, the latter in the original code must stay

```
load r1,X
add r2,r1,r3
add r1,r4,r5
```

- Antidependency (Antiriippuvuus)
  - Write-after-read (WAR)
  - The former read instruction must be able to
  - fetch the register content, before the latter
  - write stores new value there

```
move r2,r1
add  r1,r4,r5
```

- Alias?
  - Two registers use indirect references to the same memory location?
  - Different virtual address, same physical address?
  - What is visible on instruction level (before MMU)?

```
store R5, 40(R1)
              ?
load R6, 0(R2)
```

Dependencies

NOTE: Newest 8th ed. only
Ch14 -Incorrect definitions!
SEE ERRATA
Ch 12 - Correct definitions!

- i: ”write” R1

  ………

  j: ”read” R1

data dependency (RAW)

In data dependency instruction j cannot be executed before instr. i!

- i: ”read” R1

  ……..

  j: ”write” R1

antidependency (WAR)

Anti- and output dependency allow change in execution order for instructions i and j, but afterwards must be checked that the right value and result remains

- i: ”write” R1

  …….

  j: ”write” R1

output dependency (WAW)

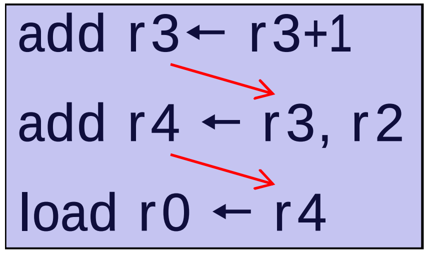
How to handle dependencies?

- Starting point
  - All dependences must be handled one way or other
- Simple solution (as before)
  - **Special hardware** detects dependency and force the pipeline to wait (bubble)
- Alternative solution
  - **Compiler** generates instructions in such a way that there will be NO dependencies
  - No special hardware
    - simpler CPU that need not detect dependencies
  - Compiler must have very detailed and specific information about the target processor’s functionality

Parallelism (rinnakkaisuus)

```
load r1← r2
add r3 ← r3+1
add r4 ← r4, r2
```

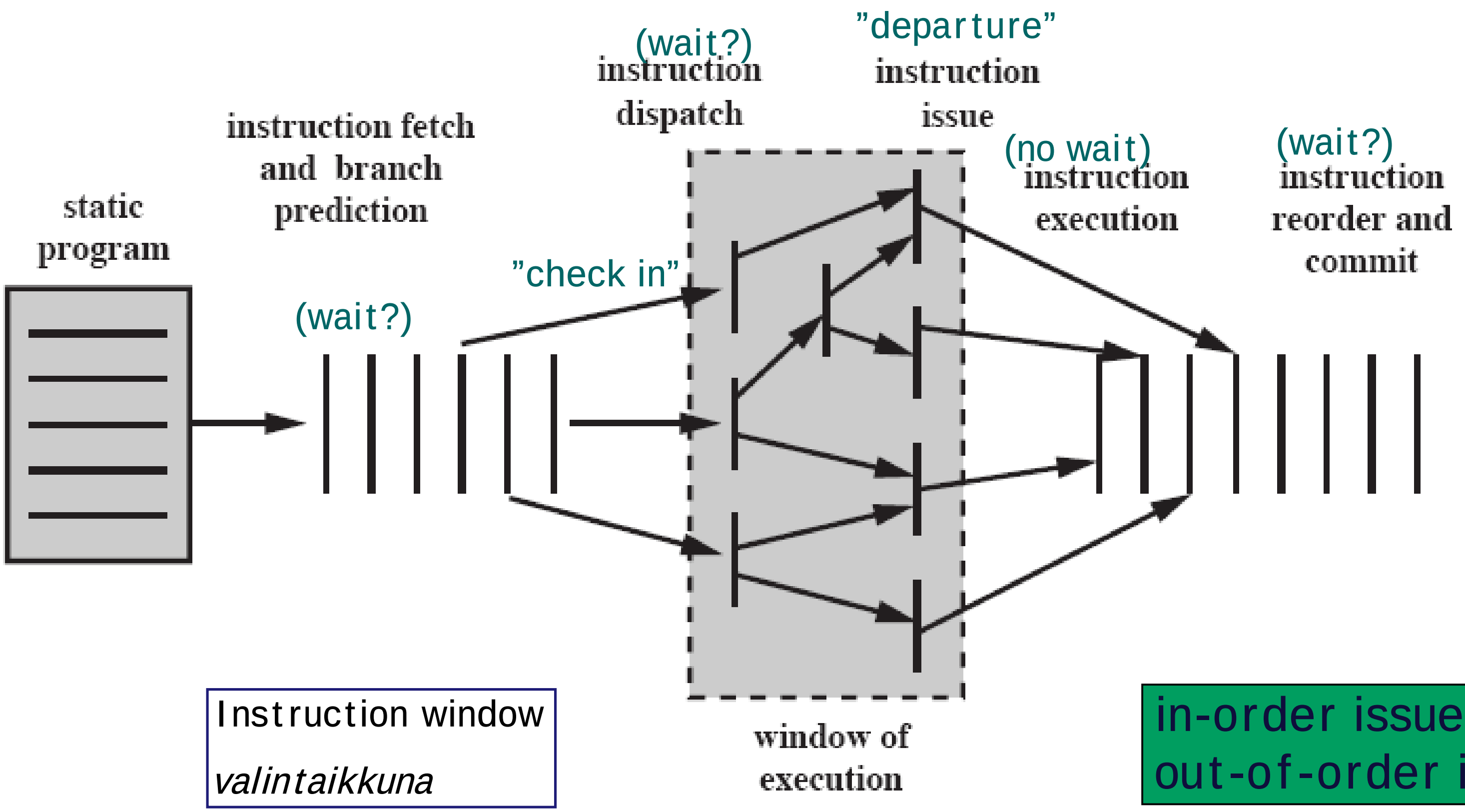
- Instruction-level parallelism (käskytason rinnakkaisuus)
  - Independent instructions of a sequence can be executed in parallel by overlapping
  - Theoretical upper limit for parallel execution of instructions
    - Depends on the code itself



```
add r3← r3+1
add r4 ← r3, r2
load r0 ← r4
```

- Machine parallelism (konetason rinnakkaisuus)
  - Ability of the processor to execute instructions parallel
  - How many instructions can be fetched and executed at the same time?
  - ~ How many pipelines can be used
  - Always smaller than instruction-level parallelism
    - Cannot exceed what instructions allow, but can limit the true parallelism
    - Dependences, bad optimization?

Superscalar execution



Instruction window
valintaikkuna

in-order issue vs.
out-of-order issue

in-order complete vs.
out-of-order complete

issue ~ *laukaisu, liikkeellelaskeminen*

dispatch ~ *vuorottaminen, lähettää suorittamaan*

(Sta09 Fig 14.6)

Superscalar execution

- Instruction fetch (*käskyjen nouto*)
  - Branch prediction (*hyppyjen ennustus*)

    → prefetch (*ennaltanouto*) from memory to CPU
  - Instruction window (*valintaikkuna*)

    ~ set of fetched instructions
- Instruction dispatch/issue (*käskyn päästäminen hihnalle*)
  - Check (and remove) data, control and resource dependencies
  - Reorder; dispatch the suitable instructions to pipelines
  - Pipelines proceed without waits
  - If no suitable instruction, wait here
- Instruction complete, retire (*suoritus valmistuu*)
  - Commit or abort (*hyväksy tai hylkää*)
  - Check and remove write and antidependencies

    → wait / reorder (*järjestä uudelleen*)

In-order issue, in-order complete

- Traditional sequencial execution order
- No need for instruction window
- Instructions dispatched to pipelines in original order
  - Compiler handles most of the dependencies
  - Still need to check dependencies, if needed add bubbles
  - Can allow overlapping on multiple pipelines
- Instructions complete and commit in original oder
  - Cannot pass, overtake (*ohittaa*) on other pipeline
  - Several instructions can complete at same time
  - Commit/Abort

In-order issue, in-order complete

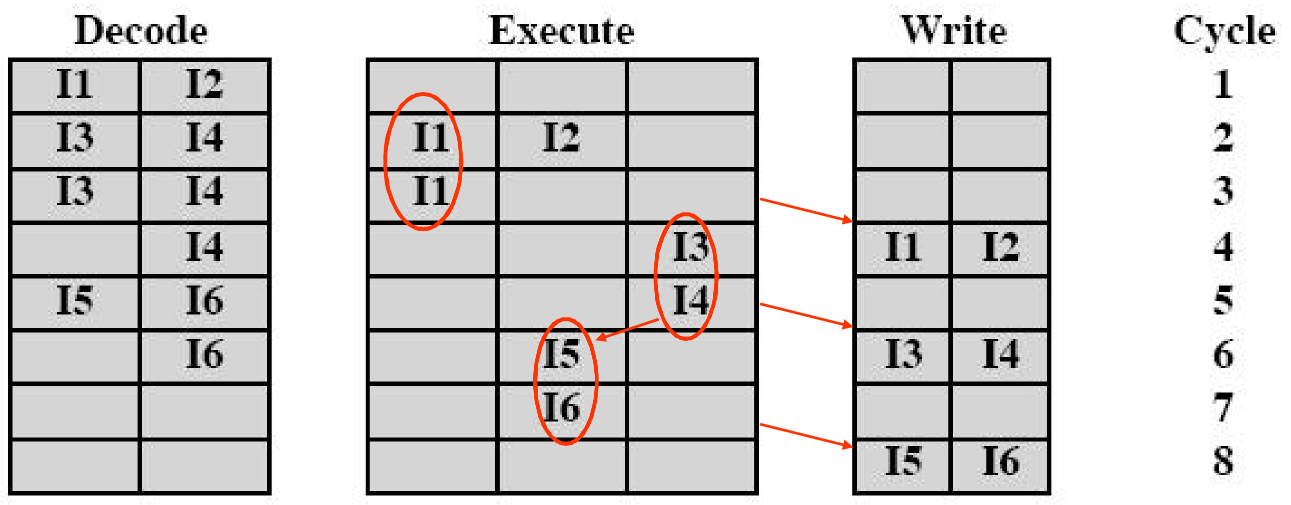
Fetch 2 instructions at the same time

I1 needs 2 cycles for execution

I3 and I4: resource dependency

I5 (consume) and I4 (produce): data dependency

I5 and I6: resource dependency



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

Decode clean before fetching next two instructions
Instructions queue for execution in decode unit
Writes delayed to maintain in-order completion

(Sta09 Fig 14.4a)

In-order issue, out-of-order complete

- Like previous, but
  - Allow commit in different order than issued order (allow passing)
  - Clear write and antidep. before writing the results

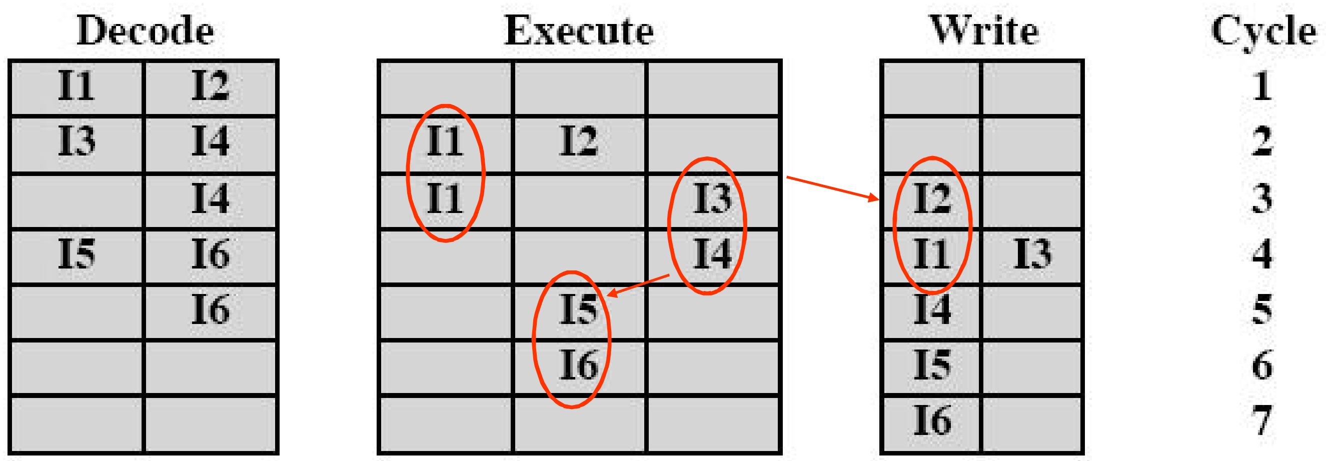
Fetch 2 instructions at the same time

I1 needs 2 cycles for execution

I3 and I4: resource dependency

I5 (consume) and I4 (produce): data dep.

I5 and I6: resource dependency



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

(Sta09 Fig 14.4b)

(WAW)

Output dependency

1: R3 ← R3 op R5

2: R4 ← R3 + 1

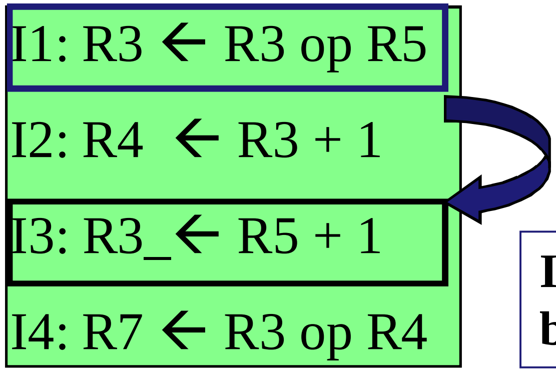
3: R3 ← R5 + 1

4: R7 ← R3 op R4

Out-of-order issue, out-of-order complete

- Dispatch instructions for execution in any suitable order
  - Need instruction window
  - Processor looks ahead (at the future instructions)
  - Must concider the dependencies during dispatch
- Allow instructions to complete and commit in any suitable order
  - Check and clear write and antidependencies

Anti dependency (WAR)



I1: R3 $\leftarrow$ R3 op R5

I2: R4 $\leftarrow$ R3 + 1

I3: R3_$\leftarrow$ R5 + 1

I4: R7 $\leftarrow$ R3 op R4

I3 must not write to R3, before I1 has read the content

True superscalar processor

Out-of-order issue, Out-of-order complete

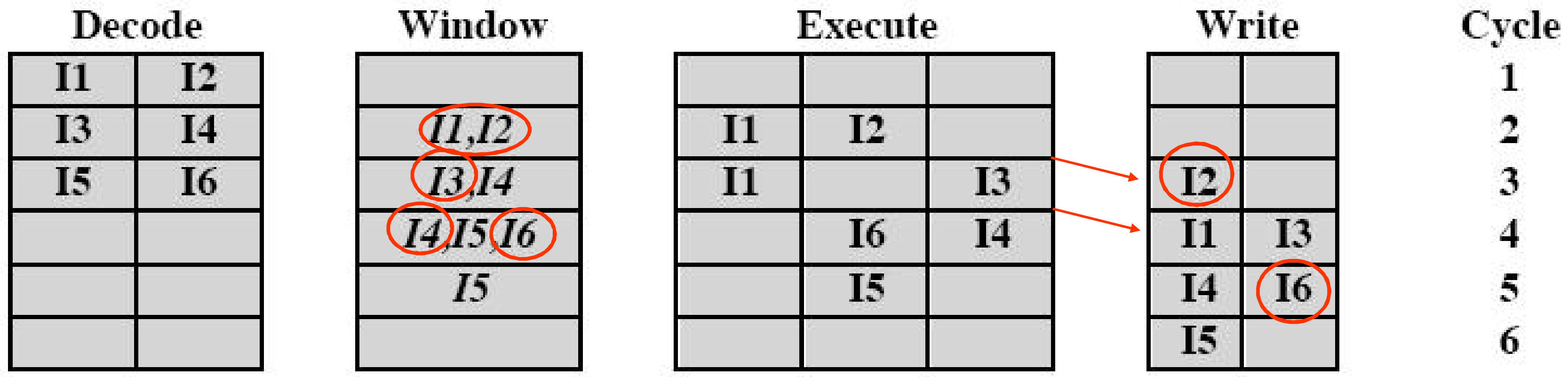
Fetch 2 instructions at the same time

I1 needs 2 cycles for execution

I3 and I4: resource dependency

I5 (consume) and I4 (produce): data dependency

I5 and I6: resource dependency



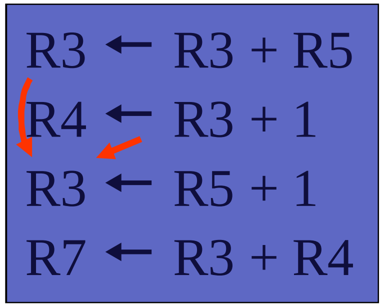
Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1,I2*	I1	I2				2
I5	I6	*I3,I4*	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

Instruction window,
(just a buffer,
not a pipeline stage)

(Sta09 Fig 14.4c)

Register renaming
(*rekistereiden uudelleennimeäminen*)

- One cause for some of the dependencies is the usage of names
  - The same name could be used for several independent elements
  - Thus, instructions have unneeded write and antidependencies
  - Causing unnecessary waits



$R3 \leftarrow R3 + R5$
$R4 \leftarrow R3 + 1$
$R3 \leftarrow R5 + 1$
$R7 \leftarrow R3 + R4$

- Solution: Register renaming
  - Hardware must have more registers (than visible to the programmer and compiler)
  - Hardware allocates new real registers during execution in order to avoid name-based dependencies (nimiriippuvuus)
- Need
  - More internal registers (register files, register set), e.g. Pentium II has 40 working registers
  - Hardware that is cabable of allocating and managing registers and performing the needed mapping

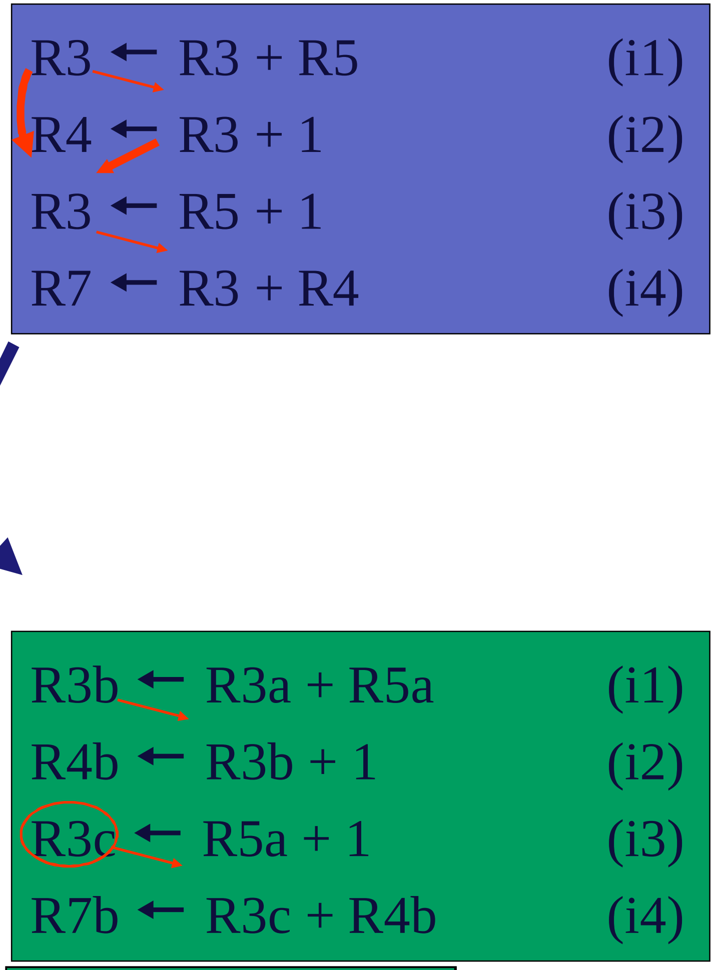
Register renaming

Output dependency(WAW):
(*Kirjoitusriippuvuus*)
i3 must not finish before i1

Anti dependency (RAW):
(*antiriippuvuus*)
i3 must not finish before i2
has read the value from R3

Rename R3 use work registers
R3a, R3b, R3c
Other registers similarly:
R4b, R5a, R7b

No more dependencies
based on names!



```
R3  ← R3 + R5        (i1)
R4  ← R3 + 1         (i2)
R3  ← R5 + 1         (i3)
R7  ← R3 + R4        (i4)
```

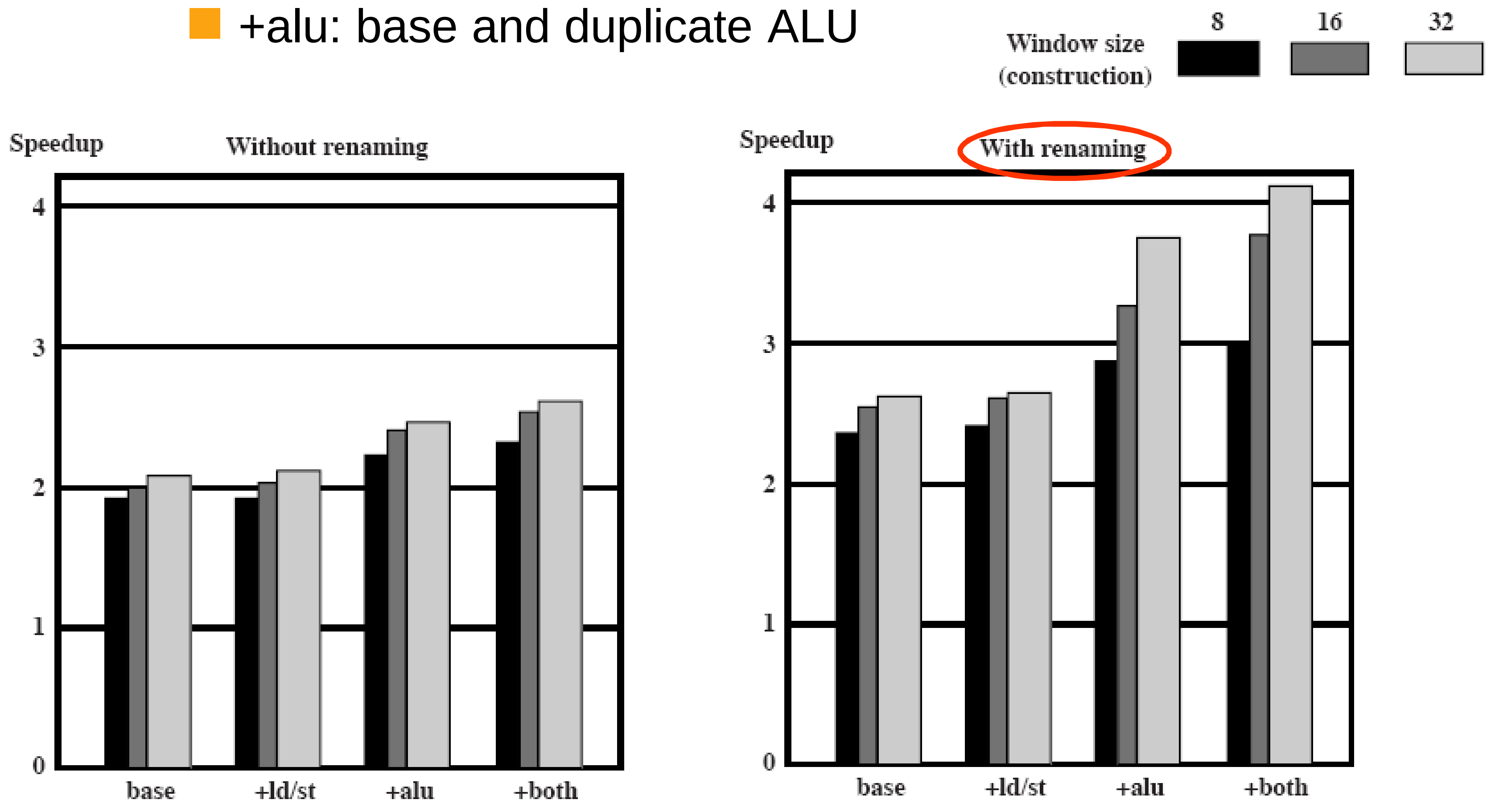
```
R3b ← R3a + R5a      (i1)
R4b ← R3b + 1        (i2)
R3c ← R5a + 1        (i3)
R7b ← R3c + R4b      (i4)
```

Why R3a and R3b?

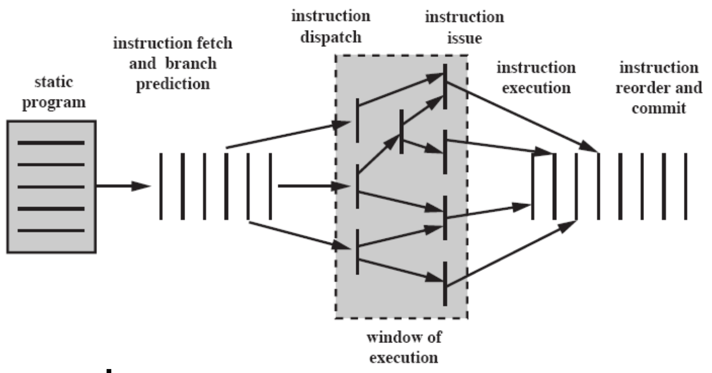
Impact of additional hardware

(Sta09 Fig 14.5)

- base: out-of-order issue
- +ld/st: base and duplicate load/store unit for data cache
- +alu: base and duplicate ALU



Superscalar – conclusion



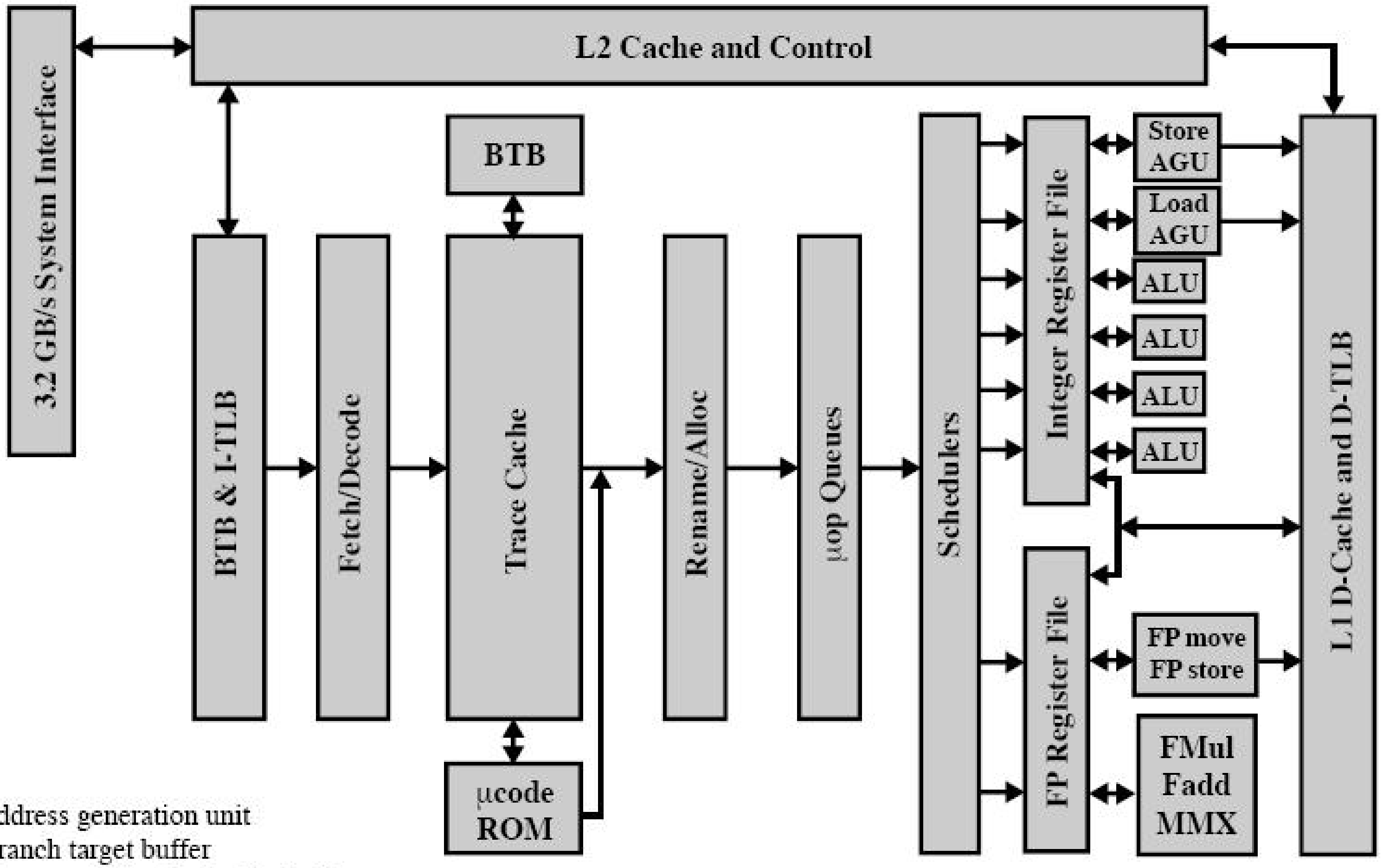
Sta09Fig 14.6

- Several functionally independent units
- Efficient use of memory hierarchy
  - Allows parallel memory fetch and store
- Instruction prefetch (*käskyjen ennaltanouto)*
  - Branch prediction (*hyppyjen ennustaminen*)
- Hardware-level logic for dependency detections
  - Circuits to pass information for other functional unit at the same time as storing to register or memory
- Hardware-level logic to dispatch several independent instructions
  - Dependencies → dispatching order
- Hardware-level logic to maintain correct completion order (*valmistumisjärjestys*)
  - Dependencies → commit-order

Computer Organization II

Pentium 4

Pentium 4

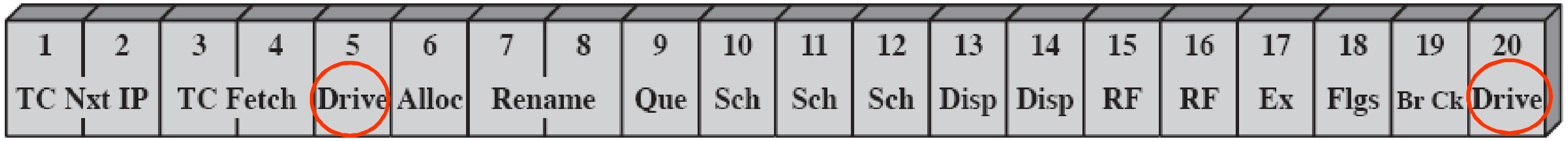


AGU	=	address generation unit
BTB	=	branch target buffer
D-TLB	=	data translation lookaside buffer
I-TLB	=	instruction translation lookaside buffer

(Sta09 Fig 14.7)

Pipeline

- Outside CISC (IA-32)
- Execution in micro-operations as RISC
  - Fetch CISC instruction and translate it to one or more micro-operations (μops) to L1-level cache (<u>trace cache</u>)
  - Rest of the superscalar pipeline operates with these fixed-length micro-operations (118b)
- Long pipeline
  - Extra stages (5 and 20) for propagation delayes



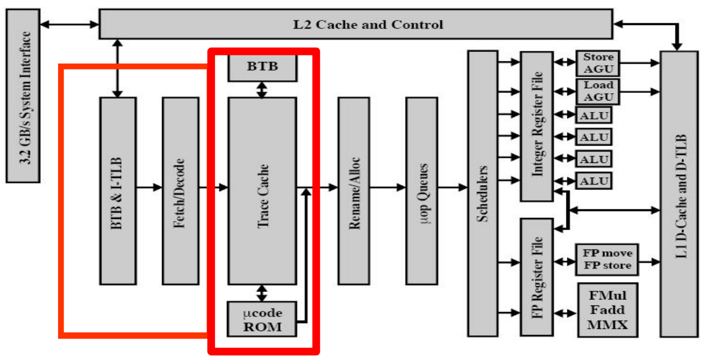
TC Next IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate

Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch

RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check

(Sta09 Fig 14.8)

Sta09 Fig 14.9a-d



Generation of µops

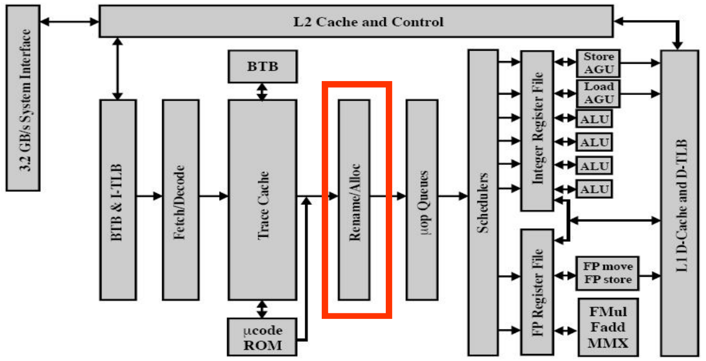
a) Fetch IA-32 instruction from L2 cache and generate µops to L1
- Uses Instruction Lookaside Buffer (I-TLB)
- and Branch Target Buffer (BTB)
  - four-way set-associative cache, 512 lines
- 1-4 µops (=118 bit RISC) per instruction (most cases), if more then stored to microcode ROM

b) Trace Cache Next Instruction Pointer - instruction selection
- Dynamic branch prediction based on history (4-bit)
- If no history available, Static branch prediction
  - backward, predict ”taken”
  - forward, predict ”not taken”

c) Fetch instruction from L1-level trace cache

d) Drive – wait (instruction from trace cache to rename/allocator)

Sta09 Fig 14.9e



Resource allocation

e) Allocate resources

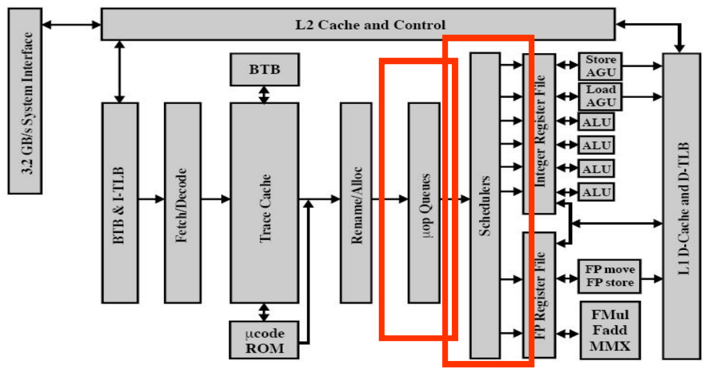
- 3 micro-operations per cycle
- Allocate an entry from Reorder Buffer (ROB) for the µops (126 entries available)
- Allocate one of the 128 internal work registers for the result
- And, possibly, one load (of 48) OR store (of 24) buffer

f) Register renaming

- Clear 'name dependencies' by remapping registers
  (16 architectural regs to 128 physical registers)
- If no free resource, wait (→ out-of-order)

- ROB-entry contains bookkeeping of the instruction progress
  - Micro-operation and the address of the original IA-32 instr.
  - State: scheduled, dispatched, completed, ready
  - Register Alias Table (RAT):
    which IA-32 register → which physical register

Window of Execution



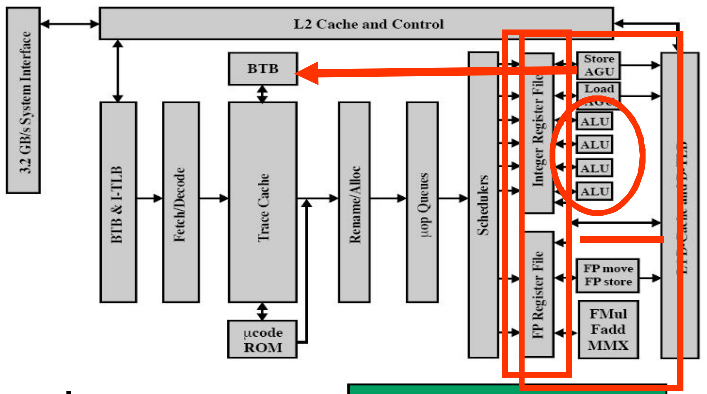
Sta069Fig 14.9f-h

g) Micro-Op Queueing
- 2 FIFO queues for µops
  - One for memory operations (load, store)
  - One for everything else
- No dependencies, proceed when room in scheduling

h) Micro-Op Scheduling
- Retrieve µops from queue and dispatch for execution
- Only when operands ready (check from ROB-entry)

i) Dispatching
- Check the first instructions of FIFO-queues (their ROB-entries)
- If execution unit needed is free, dispatch to that unit
- Two queues → out-of-order issue
- max 6 micro-ops dispatched in one cycle
  - ALU and FPU can handle 2 per cycle
  - Load and store each can handle 1 per cycle

Integer and FP Units



Sta09 Fig 14.9i-l

- j) Get data from register or L1 cache
- k) Execute instruction, set flags (lipuke)
  - Several pipelined execution units
    - 4 * Alu, 2 * FPU, 2 * load/store
    - E.g. fast ALU for simple ops, own ALU for multiplications
  - Result storing: in-order complete
  - Update ROB, allow next instruction to the unit
- l) Branch check
  - What happend in the jump /branch instruction
  - Was the prediction correct?
  - Abort incorrect instruction from the pipeline (no result storing)
- m) Drive – update BTB with the branch result

Pentium 4 Hyperthreading

- One physical IA-32 CPU, but 2 logical CPUs
- OS sees as 2 CPU SMP (symmetric multiprocessing)
  - Processors execute different processes or threads
  - No code-level issues
  - OS must be cabable to handle more processors (like scheduling, locks)
- Uses CPU wait cycles
  - Cache miss, dependences, wrong branch prediction
- If one logical CPU uses FP unit the other one can use INT unit
  - Benefits depend on the applications

Pentium 4 Hyperthreading

- Duplicated (kahdennettu)
  - IP, EFLAGS and other control registers
  - Instruction TLB
  - Register renaming logic
- Split (puolitettu)
  - No monopoly, non-even split allowed
  - Reordering buffers (ROB)
  - Micro-op queues
  - Load/store buffers
- Shared (jaettu)
  - Register files (128 GPRs, 128 FPRs)
  - Caches: trace cache, L1, L2, L3
  - Registers needed during µops execution
  - Functional units: 2 ALU, 2 FPU, 2 ld/st-units

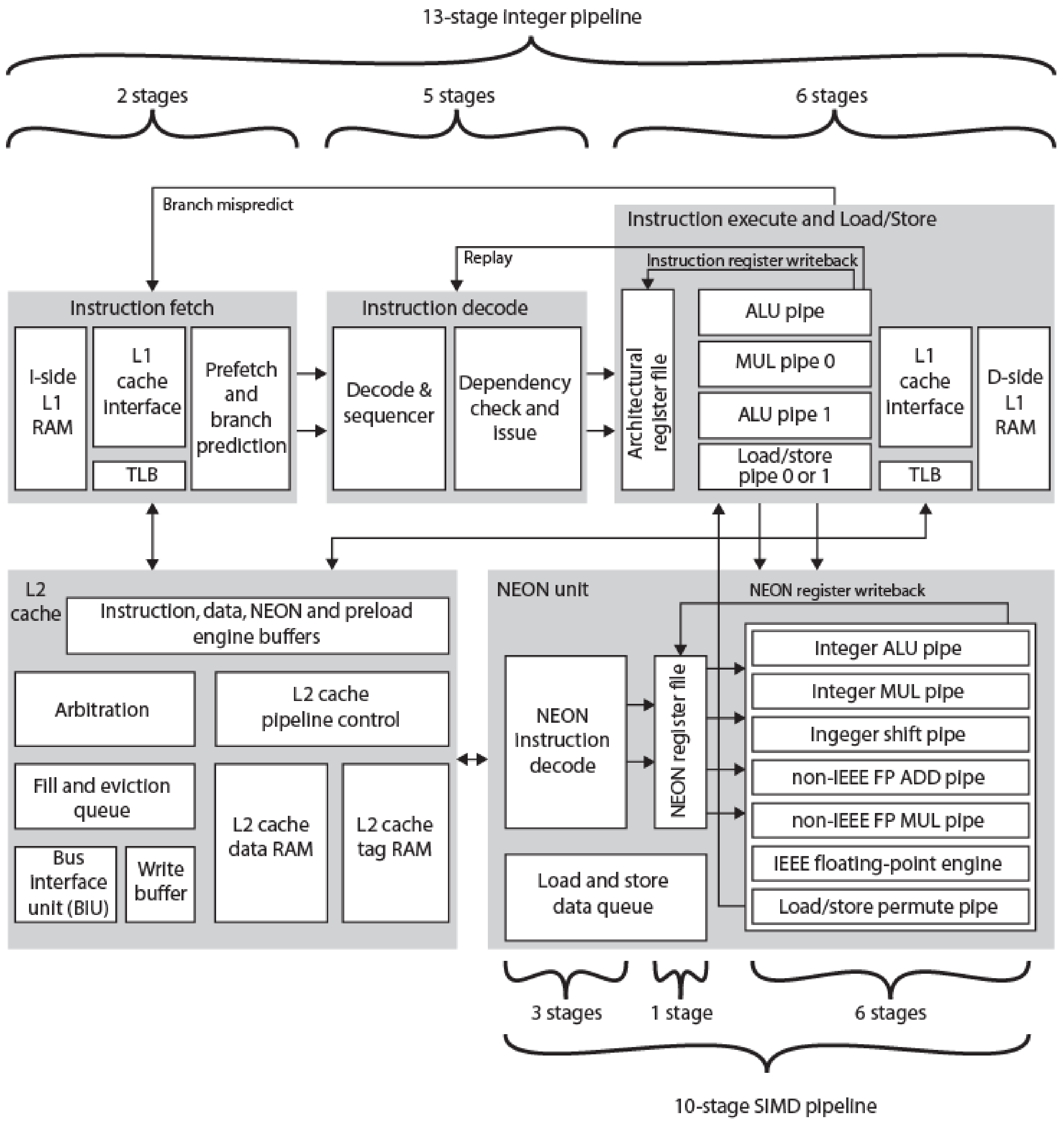
Intel Nehalem arch.:
8 cores on one chip,
1-16 threads (820
million transistors)
First lauched processor
Core i7 (Nov 2008)

ARM Cortex-A8

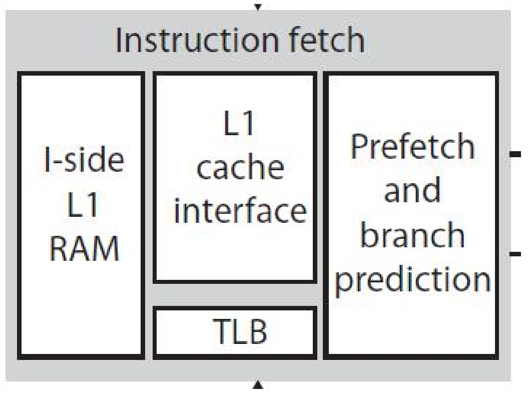
ARM CORTEX-A8

- 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
- Separate SIMD (single-instruction-multiple-data) unit called NEON
  - 10-stage pipeline

ARM Cortex-A8 Block Diagram



Instruction Fetch Unit



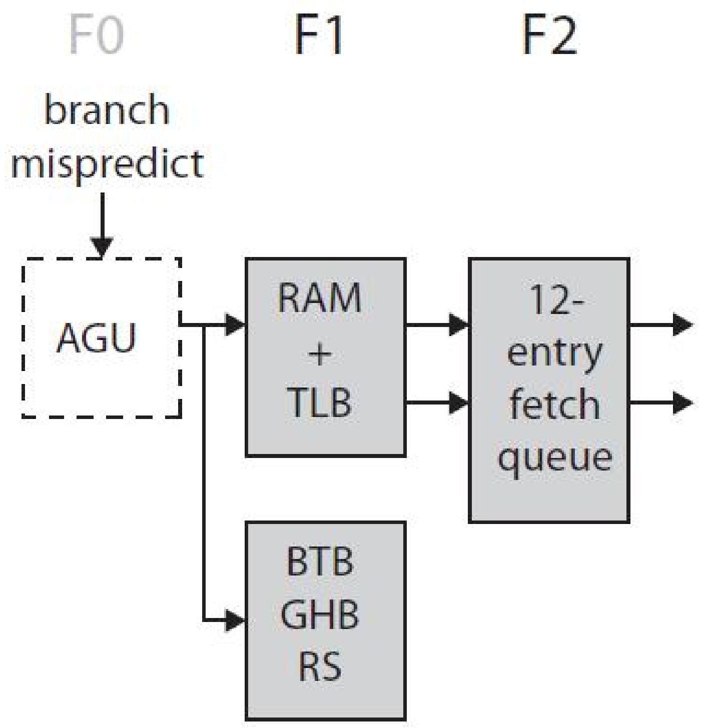
- Predicts instruction stream
- Fetches instructions from the (included) L1 instruction cache
  - Into buffer for decode pipeline
  - Up to four instructions per cycle
- Speculative instruction fetches

- Branch or exceptional instruction cause pipeline flush
- 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

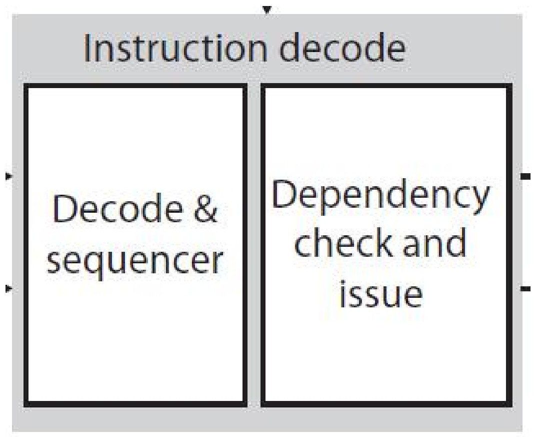
Instruction Fetch Unit: Processing Stages

- F0 address generation unit (AGU)
  - Next address sequentially
  - Or branch target address (from branch prediction of previous address)
- F1 fetch instructions from L1
  - In parallel, check the branch prediction for the next address
- F2 Place instruction to instr. queue
  - If branch prediction, new target address sent to AGU



- Issues instructions to decode two at a time

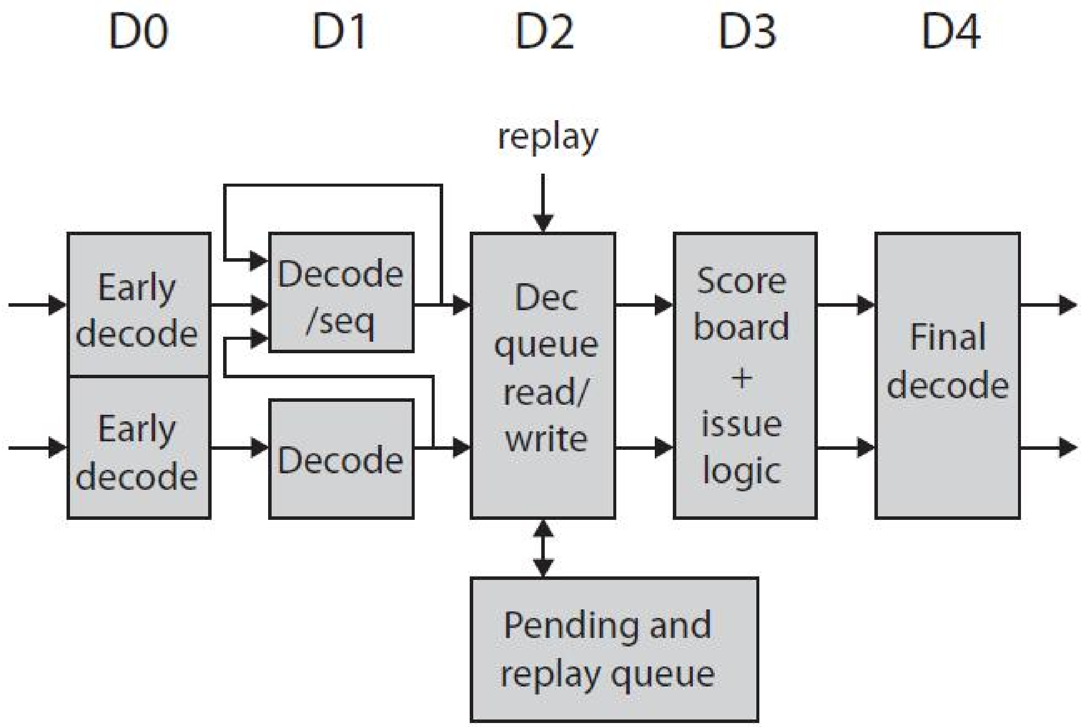
Instruction Decode Unit



- Dual pipeline structure, *pipe0* and *pipe1*
  - Two instructions at a time
  - Pipe0 contains older instruction in program order
  - If instruction in pipe0 cannot issue, pipe1 will not issue
- In-order instruction issue and retire
  - Results written back to register file at end of execution pipeline
  - no WAR hazards
  - tracks WAW hazards and straightforward recovery from flush
  - Decode pipeline to prevent RAW hazards

Instruction Decode Unit: Processing Stages

- D0 Decompress Thumbs and do preliminary decode
- D1 Instruction decode completed
- D2 Write instruction to and read instructions from pending/replay queue
- D3 instruction scheduling logic
  - Scoreboard predicts register availability using static scheduling
  - Hazard checking
- D4 Final decode - control signals for integer execute load/store units

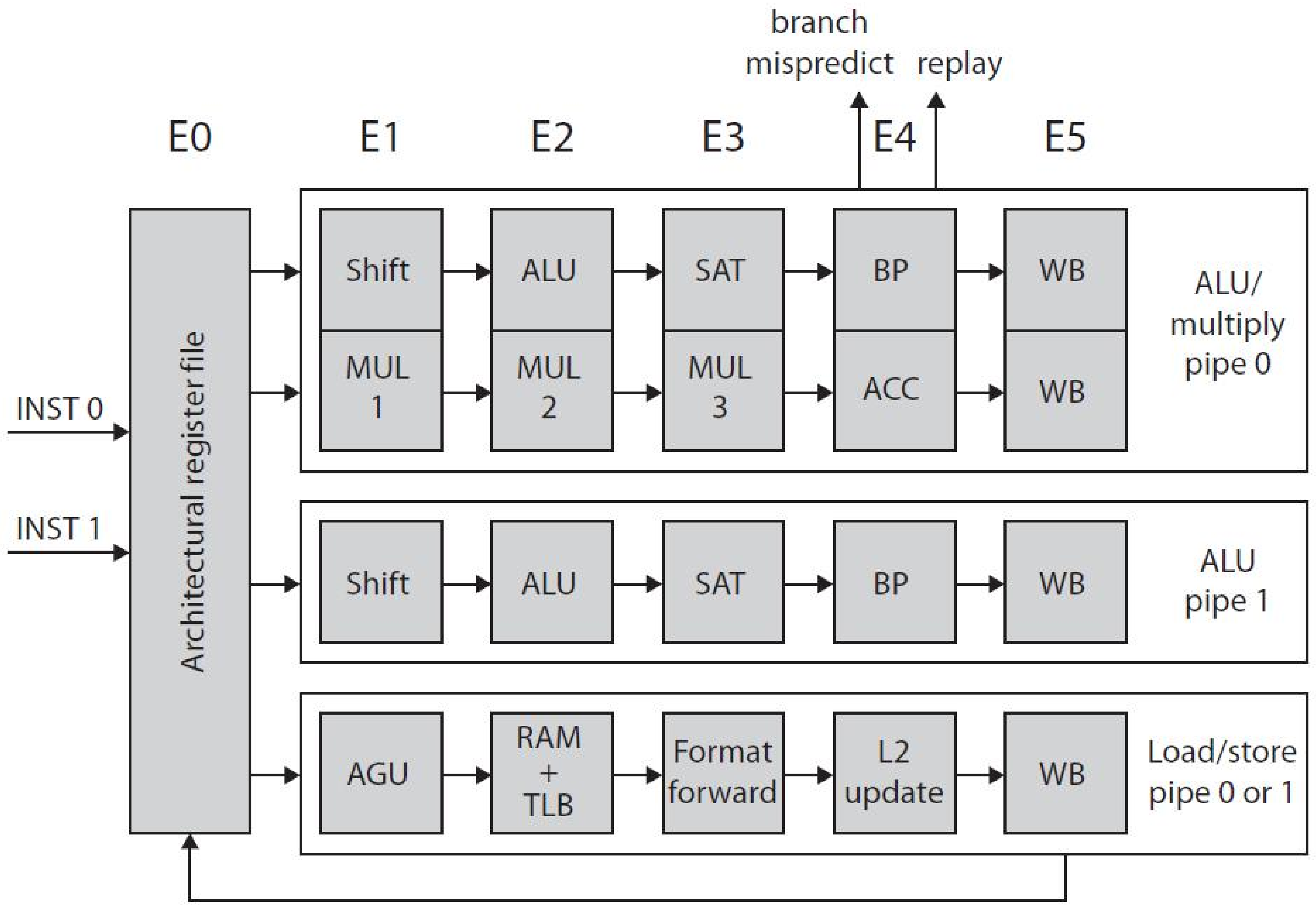


Integer Execution Unit

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

- 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

Integer Execution Unit



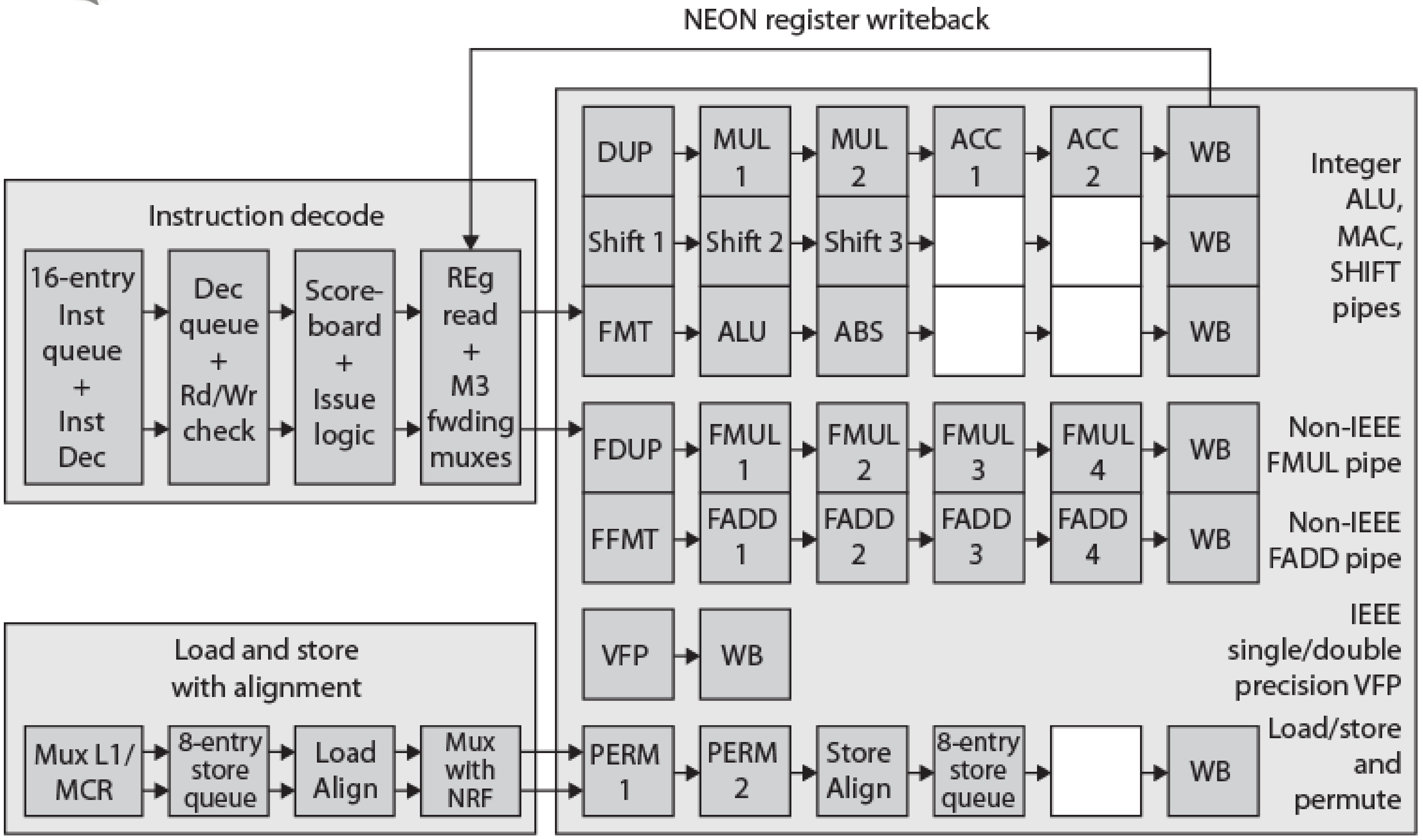
Load/store pipeline

- 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

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



Review Questions / Kertauskysymyksiä

- Differences / similarities of superscalar and trad. pipeline?
- What new problems must be solved?
- How to solve those?
- What is register renaming and why it is used?

- Miten superskalaaritoteutus eroaa tavallisesta liukuhihnoitetusta toteutuksesta?
- Mitä uusia rakenteesta johtuvia ongelmia tulee ratkottavaksi?
- Miten niitä ongelmia ratkotaan?
- Mitä tarkoittaa rekistereiden uudelleennimeäminen ja mitä hyötyä siitä on?