

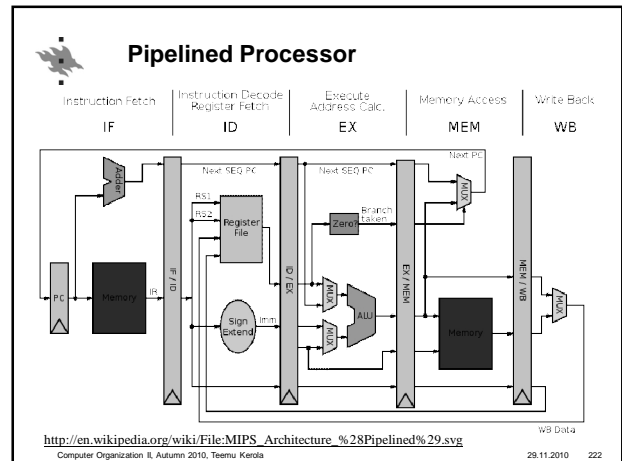
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Lecture 9

Superscalar- processing

Stallings 2010: 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
[TIAD70]	1.8
[KUCK72]	8
[WEISS4]	1.58
[ACOSS6]	2.7
[SOHB90]	1.8
[SMT89]	2.3
[JOUTP86b]	2.2
[LEE91]	7
[Sta10 Tbl 14.1]	

Ancient!

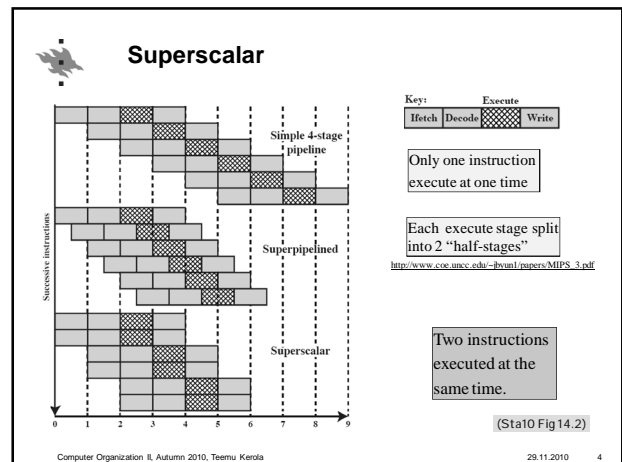
Integer Register File | Floating Point Register File

Pipelined functional units | Memory

Instruction-level parallelism

Sta10 Fig 14.1

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 3



Superscalar processor

- Efficient memory usage
 - Fetch several instructions at once, prefetching (*ennaltanouto*)
 - Data fetch and store (read and write)
 - Concurrency
- Several instructions of the same process executed concurrently on different pipelines
 - Select executable instruction (ready for execute stage) from the prefetched instruction 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 is it ok for an instruction finish before the preceding ones?

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 5

Effect of Dependencies

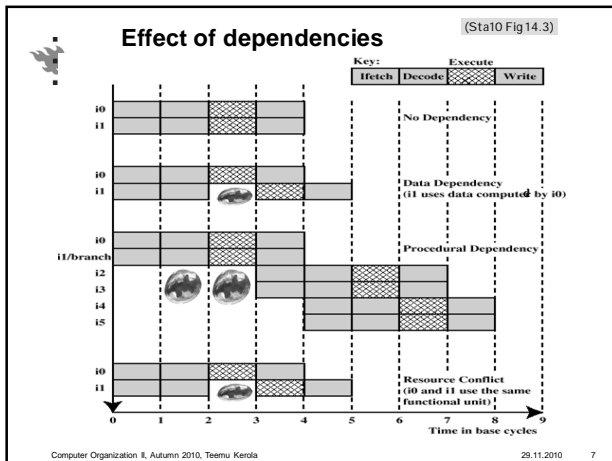
- True Data/Flow Dependency (*datariippuvuus*)
 - Read after Write (RAW) Typo on p. 545, line 2 [Sta10]
 - The latter instruction needs data from former instruction
- Procedural/Control Dependency (*kontrolliriippuvuus*)
 - Instruction after the jump executed only, when jump does not happen
 - Superscalar pipeline has more instructions to waste
 - Variable-length instructions: some additional parts known only during execution
- Resource Conflict (*Resurssiiriippuvuus*)
 - One or more pipeline stage needs the same resource
 - Memory buffer, ALU, access to register file, ...

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

```
jmp r2, 100
add r1, =1
```

```
fadd f1, f2, f4
fadd f4, f5, f6
```

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 6



WAR and WAW Dependencies

- True data dependency (RAW)**
 i: "write" R1

 j: "read" R1
 data dependent instruction j cannot be executed before instruction i
- Antidependency (WAR)**
 Typo on p. 550, line 2 from bottom [Sta10]
 i: "read" R1

 j: "write" R1
 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
- Output dependency (WAW)**
 i: "write" R1

 j: "write" R1

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 8

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)
```

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 9

How to Handle Dependencies?

- Starting point
 - All dependencies must be handled one way or other
- Simple solution (like before)
 - Special hardware detects dependency and force the pipeline to wait (bubble)
- Alternative solution (like before, bit more important now)
 - 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
 - which dependencies must/can be solved by compiler?

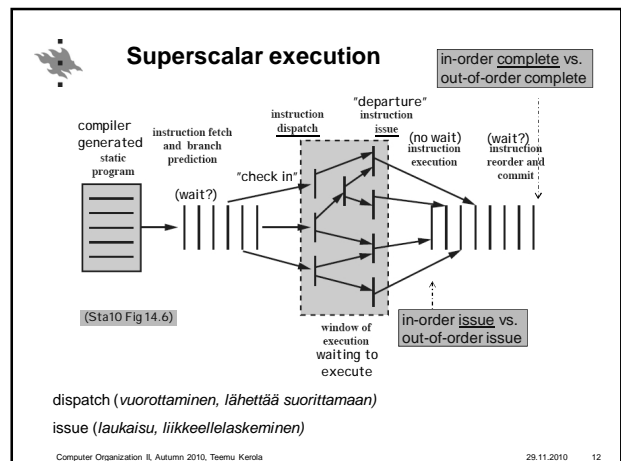
Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 10

ILP vs. Machine Parallelism

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

- Instruction-level parallelism, ILP (käskytason rinnakkaisuus)**
 - Independent instructions that could be executed in parallel by overlapping
 - Theoretical upper limit for parallel execution of instructions
 - Depends on the code
- Machine parallelism (konetason rinnakkaisuus)**
 - Ability of the processor to actually 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
 - Dependencies solved inefficiently, bad optimization?

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 11



Superscalar Execution

- Instruction fetch (*käskeyjen nouto*)
 - Branch prediction (*hyppyjen ennustus*)
 - prefetch (*ennaltanouto*) from memory to CPU
 - Dispatch to instruction window (*valintaikkuna*)
- Instruction issue (*käskeyn päästäminen hihnalle*)
 - Check (and remove) data, control and resource dependencies
 - Reorder; issue suitable instructions to pipelines
 - Pipelines proceed without waits
- Instruction complete, retire (*suoritus valmistuu*)
 - Commit or abort (*hyväksy tai hylkää*)
 - Usually all state changes occur here
 - Check and remove write and antidependencies
 - wait / reorder (*järjestä uudelleen*)

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 13

In-order Issue, In-order Complete

- Traditional sequential execution order
- No need for instruction window
- Instructions dispatched to pipelines in original order (determined by the compiler)
 - 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 order
 - Cannot pass, overtake (*ohittaa*) on other pipeline
 - Several instructions can complete at same time
 - Commit/Abort

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 14

In-order Issue, In-order Complete Pipeline

Fetch 2 instructions at the same time
 I1 needs 2 cycles for execution
 I3 and I4: resource dependency
 I4 → I5: data dependency
 I5 and I6: resource dependency

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

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

(Sta10 Fig 14.4a)

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 15

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
 I4 → I5: data dep.
 I5 and I6: resource dependency

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

(Sta10 Fig 14.4b)

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 16

Out-of-order Issue, Out-of-order Complete

True superscalar processor

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

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 17

Out-of-order issue, Out-of-order Completion

Fetch 2 instructions at the same time
 I1 needs 2 cycles for execution
 I3 and I4: resource dependency
 I4 → I5: data dependency
 I5 and I6: resource dependency

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

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

(Sta10 Fig 14.4c)

Discussion?

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 18

Solve Dependencies by Register Renaming

name dependency vs. data dependency?

- Some dependencies are caused by register names, not data
 - The same name could be used for several independent elements
 - Thus, instructions have unnecessary write and antidependencies
 - Causing unnecessary waits

$$\begin{matrix} R3 \leftarrow R3 + R5 \\ R4 \leftarrow R3 + 1 \\ R3 \leftarrow R5 + 1 \\ R7 \leftarrow R3 + R4 \end{matrix}$$

- 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 can allocate and manage registers, and perform the mapping dynamically at execution time

29.11.2010 19

Register Renaming

Output dependency (WAW):
i3 must not write R3 before i1 writes R3

Anti dependency (WAR):
i3 must not write R3 before i2 has read the value from R3

$$\begin{matrix} R3 \leftarrow R3 + R5 & (i1) \\ R4 \leftarrow R3 + 1 & (i2) \\ R3 \leftarrow R5 + 1 & (i3) \\ R7 \leftarrow R3 + R4 & (i4) \end{matrix}$$

Typo on p. 552, line 15 [Sta110]

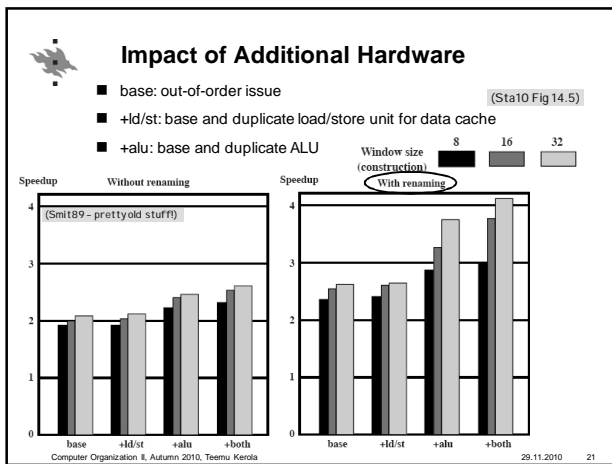
Solution

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

$$\begin{matrix} R3b \leftarrow R3a + R5a & (i1) \\ R4b \leftarrow R3b + 1 & (i2) \\ R3c \leftarrow R5a + 1 & (i3) \\ R7b \leftarrow R3c + R4b & (i4) \end{matrix}$$

No more dependencies based on names!

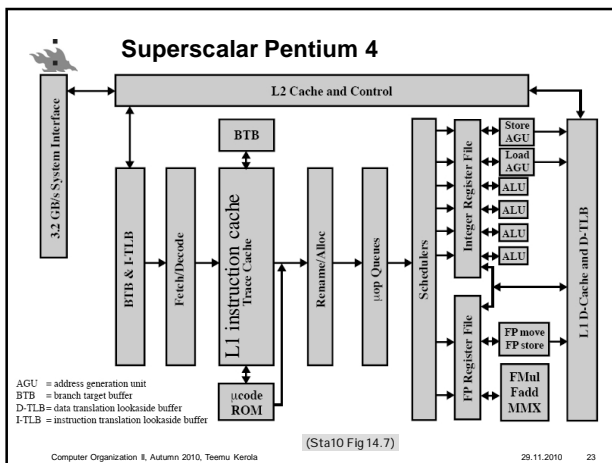
29.11.2010 20



Superscalar Conclusion

- Several functionally independent units
- Efficient use of memory hierarchy
 - Allows parallel memory fetch and store
- Instruction prefetch
 - Branch prediction important
- 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 issue several independent instructions
 - Dependencies → issue order
- Hardware-level logic to maintain correct completion order
 - Dependencies → commit-order

29.11.2010 22



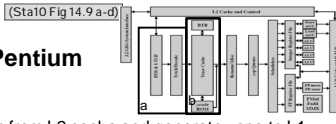
Pentium 4 Pipeline

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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC	Next IP	TC	Fetch	Drive	Alloc	Rename	Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive	

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

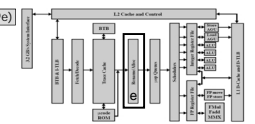
29.11.2010 24



Generation of Pentium Pipeline μ ops

- Fetch IA-32 instruction from L2 cache and generate μ ops to L1
 - Uses Instruction Lookaside Buffer (I-TLB) and Branch Target Buffer (BTB)
 - four-way set-associative cache, 512 lines
 - 1-4 μ ops (=118 bit RISC) per instruction (most cases), if more then stored to microcode ROM
- 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"
- Fetch instruction from L1-level trace cache
- Drive – wait (instruction from trace cache to rename/allocator)

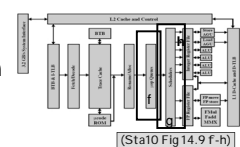
Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 25



Pentium Pipeline Resource allocation

- Allocate resources
 - 3 micro-operations per cycle
 - Allocate an entry from Reorder Buffer (ROB) for the μ ops (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
- Register renaming
 - Clear name dependencies by renaming (16 architectural regs to 128 physical registers)
 - If no free resource, wait (\rightarrow 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 \rightarrow which physical register

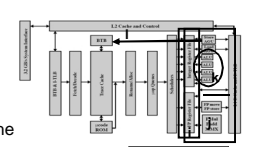
Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 26



Pentium Pipeline Window of Execution

- Micro-Op Queuing
 - 2 FIFO queues for μ ops
 - One for memory operations (load, store)
 - One for everything else
 - No dependencies, proceed when room in scheduling
- Micro-Op Scheduling
 - Retrieve μ ops from queue and dispatch (issue) for execution
 - Only when operands ready (check from ROB-entry)
- Dispatching
 - Check the first instructions of FIFO-queues (their ROB-entries)
 - If execution unit needed is free, dispatch to that unit
 - Two queues \rightarrow 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


Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 27



Pentium Pipeline Integer and FP Units

- Get data from register or L1 cache
- 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
- Branch check
 - What happend in the jump /branch instruction
 - Was the prediction correct?
 - Abort incorrect instruction from the pipeline (no result storing)
- Drive – update BTB with the branch result


Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 28



Pentium 4 Hyperthreading

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

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 29



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 μ ops 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)

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 30

Superscalar ARM CORTEX-A8

- In family of ARM 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 (four?) functional units
 - Fetch pipeline, decode pipeline, execute pipeline
 - SIMD pipeline NEON (10-stages)
- Dual, in-order-issue, 13-stage pipeline
 - Keep power required to a minimum
 - Out-of-order issue would need extra logic consuming extra power

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 31

ARM Cortex-A8 Block Diagram

The diagram illustrates the ARM Cortex-A8 architecture. It features a 13-stage integer pipeline divided into three sections: 2 stages (Instruction fetch), 5 stages (Instruction decode), and 6 stages (Instruction execute and Load/Store). The NEON SIMD pipeline consists of 10 stages: 3 stages for NEON instruction decode, 1 stage for NEON register writeback, and 6 stages for various NEON operations including Integer ALU, Integer MUL, Integer shift, non-IEEE FP ADD, non-IEEE FP MUL, IEEE floating-point engine, and Load/store permute. Other components include L1 and L2 caches, TLBs, and various buffers and queues.

(Sta10 Fig 14.10) Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 32

ARM Cortex-A8 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

(Sta10 Fig 14.10) Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 33

ARM Cortex-A8 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 instruction queue
 - If branch prediction, new target address sent to AGU
- Issues instructions to decode two at a time

(Sta10 Fig 14.11) Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 34

ARM Cortex-A8 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

(Sta10 Fig 14.10) Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 35

ARM Cortex-A8 Instruction Decode Unit Processing Stages

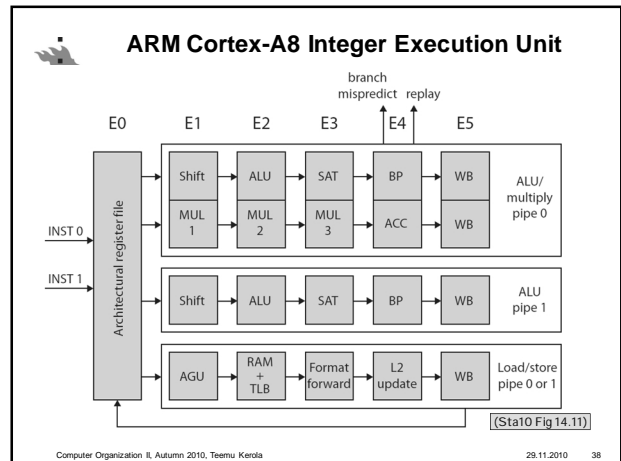
- D0 Decompress Thumbs and do preliminary decode
- D1 Instruction decode completed
- D2 Write/read instructions to/from pending/replay queue
- D3 instruction scheduling logic
 - Scoreboard predicts register availability using static scheduling
 - Dependency checking
- D4 Final decode - control signals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

(Sta10 Fig 14.11) Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 36

ARM Cortex-A8 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

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 37



ARM Cortex-A8 Load/Store Pipeline

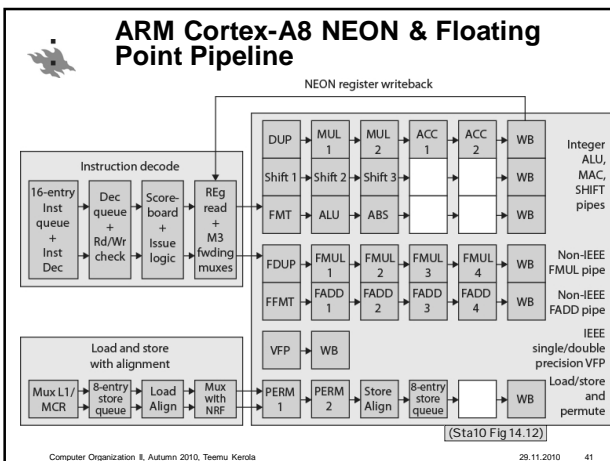
- Parallel to integer pipeline
- E1 Memory address generated from base and index register
- E2 address applied to cache arrays
- E3 load, data returned and formatted
- E3 store, data are formatted and ready to be written to cache
- E4 Updates L2 cache, if required
- E5 Results are written to register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

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 39

ARM Cortex-A8 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 (non-IEEE) floating-point operations

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 40



Summary

- What does superscalar mean?
- ILP vs. machine level parallelism?
- Dispatch, issue, window of execution
- Out-of-order completion
- New dependencies and solutions for them?
- Renaming, solution for name dependencies
- Superscalar Pentium and ARM

Computer Organization II, Autumn 2010, Teemu Kerola 29.11.2010 42



Review Questions

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