Superscalar Processors
Ch 14

Limitations, Hazards

Instruction Issue Policy

Register Renaming

Branch Prediction

PowerPC, Pentium 4

Superscalar Processing (5)

- Basic idea: more than one instruction <u>completion</u> per cycle
- Aimed at speeding up scalar processing
  - use many pipelines and not just more pipeline phases

Fig. 14.2

(Fig. 13.2 [Stal99])

- Many instructions in <u>execution phase</u> simultaneously
  - need parallelism also in earlier & later phases
  - may not execute (completely) in given order
- Multiple pipelines
  - question: when can instruction be executed?

(Fig. 13.1 [Stal99])

Fig. 14.1

- Fetch many instructions at the same time
  - memory access must not be bottleneck

Why couldn't we execute this instruction right now? (4)

Fig. 14.3
(Fig. 13.3 [Stal99])

- (True) Data Dependency
  (datariippuvuus)

```
load    r4, salary(r6)
mul     r2, r4, r10
```

- Procedural or Control Dependency (kontrolli-riippuvuus)
  – even more costlier than with normal pipeline
  – now may waste more than one instruction!
- Resource Conflict (resurssi-konflikti)
  – there is no available circuit right now
  – memory buffer, FP adder, register file port
- Usual solution: circuits to detect problem and stall pipeline when needed

9.10.2002 3

New dependency for superscalar case? (8)

- <u>Name dependency</u> (nimiriippuvuus)
  – two instructions use the same data item
    - register or in memory
  – no value passed from one instruction to another
  – instructions have all their correct data available
  – each individual result is the one intended
  – <u>overall result</u> is <u>not</u> the one intended
  – two cases: Output Dependency & Antidependency
    (kirjoitusriippuvuus?) (antiriippuvuus)
    - examples on next 2 slides
  – what if there are aliases?
    - E.g., two registers point to same physical address

9.10.2002 4

Output Dependency?

- Some earlier instruction has not yet finished writing from the same location (register) that we want to write to
  - execution time semantics determined by the original order of machine instructions
- Need to preserve order

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

Want to have sum of r4 and r5 in r1 after all these three instructions were executed (not value of variable X)

Antidependency

Some earlier instruction has not yet finished reading from the same location that we want to write to
Need to preserve order

```
mv      r2, r1
add     r1, r4, r5
```

Want to have original value of r1 in r2

Machine Parallelism (2)

- Instruction-level parallelism (ILP)
  - How much parallelism is there
  - Theoretical maximum
- Machine parallelism
  - How much parallelism is achieved by any specific machine or architecture?
  - At most as much as instruction-level parallelism
    - dependencies?
    - physical resources?
    - not optimized (I.e., stupid?) design?

Superscalar Processor (4)

Fig. 14.6

(Fig. 13.6 [Stal99])

- Instruction dispatch
  - get next available executable instruction from instruction stream
- Window of execution
  - all instructions that are considered to be issued
- Instruction issue
  - allow instruction to start execution
  - execution and completion phase should continue now with no stalls
  - if any stalls needed, do them before issue
- Instruction reorder and commit (retiring)
  - hopefully all system state changes here!
  - last chance to change order or abandon results

Instruction Dispatch (7)

Fig. 14.6
(Fig. 13.6 [Stal99])

- Whenever there are both
  - available slots in window of execution
  - ready instructions from prefetch or branch prediction buffer
    - instructions that do not need to stall at all during execution
    - all dependencies do not need to be solved yet

9.10.2002 9

Window of Execution

Fig. 14.6
(Fig. 13.6 [Stal99])

- Bigger is better
  - easier to find a good candidate that can be issued right now
  - more work to figure out all dependencies
  - too small value will limit machine parallelism significantly
    - E.g., 6th instruction could be issued, but only 4 next ones are even considered

9.10.2002 10

Instruction Issue (3)

Fig. 14.6

(Fig. 13.6 [Stal99])

- Select next instruction(s) for execution
- Check first everything so that execution can proceed with no stalls (stopping) to the end
  - resource conflicts
  - data dependencies
  - control dependencies
  - output dependencies
  - antidependencies

”data in R4 is not yet there, but it will be there in three cycles when it is needed by this instruction”

- Simpler instruction execution pipelines
  - no need to check for dependencies

9.10.2002 11

Instruction Issue Policies (3)

- Instruction fetch policy
  - constraints on how many instructions are considered to be dispatched at a time
    - E.g., 2 instructions fetched and decoded at a time ⇒ both must be dispatched before next 2 fetched
- Instruction execution policy
  - constraints on which order dispatched instructions may start execution
- Completion policy
  - constraints the order of completions

9.10.2002 12

Example 1 of Issue Policy (7)

- In-order issue with in-order completion
  - same as purely sequential execution
  - no instruction window needed
  - instruction issued only in original order
    - many can be issued at the same time
  - instructions completed only in original order
    - many can be completed at the same time
  - check before issue:
    - resource conflicts, data & control dependencies
    - execution time, so that completions occur in order: wait long enough that earlier instructions will complete first
  - Pentium II: out-of-order middle execution for micro-ops (μops) with in-order completion

Fig. 14.4 (a)

(Fig. 13.4 (a) [Stal99])

Example 2 of Issue Policy (5)

- In-order issue with out-of-order completion
  - issue in original order
    - many can be issued at the same time
  - no instruction window needed
  - allow executions complete before those of earlier instructions
  - check before issue:
    - resource conflicts, data & control dependencies
    - output dependencies: wait long enough to solve them

Fig. 14.4 (b)

(Fig. 13.4 (b) [Stal99])

Example 3 of Issue Policy (5)

The real superscalar processor

- Out-of-order issue with out-of-order completion
  - issue in any order (Fig. 13.4 (c) [Stal99]) Fig. 14.4 (c)
    - many can be issued at the same time
  - instruction window for dynamic instruction scheduling
  - allow executions complete before those of earlier instructions
  - Check before issue:
    - resource conflicts, data & control dependencies
    - output dependencies: wait for earlier instructions to write their results before we overwrite them
    - antidependencies: wait for earlier instructions issued later to pick up arguments before overwriting them

Get Rid of Name Dependencies (3)

- Problem: independent data stored in locations with the same name
  - often a storage conflict: same register used for two different purposes
  - results in wait stages (pipeline stalls, “bubbles”)
- Cure: register renaming
  - actual registers may be different than named registers
  - actual registers allocated dynamically to named registers
  - allocate them so that name dependencies are avoided
- Cost:
  - more registers
  - circuits to allocate and keep track of actual registers

Register Renaming (3)

Output dependency: I3 can not complete before I1 has completed first:

Antidependency: I3 can not complete before I2 has read value from R3:

R3:=R3 + R5;	(I1)
R4:=R3 + 1;	(I2)
R3:=R5 + 1;	(I3)
R7:=R3 + R4;	(I4)

Rename data in register R3 to actual hardware registers R3a, R3b, R3c

Rename also other registers: R4b, R5a, R7b

R3b:=R3a + R5a	(I1)
R4b:=R3b + 1	(I2)
R3c:=R5a + 1	(I3)
R7b:=R3c + R4b	(I4)

No name dependencies now:

- Drawback: need more registers
  - Pentium II: 40 extra regs + 16 normal regs
- Why R3a & R3b?

9.10.2002 17

Superscalar Implementation (7)

Fig. 14.6 (Fig. 13.6 [Stal99])

- Fetch strategy
  - prefetch, branch prediction
- Dependency check logic
- Forwarding circuits (shortcuts) to transfer dependency data directly instead via registers or memory (to get data accessible earlier)
- Multiple functional units (pipelines)
- Effective memory hierarchy to service many memory accesses simultaneously
- Logic to issue multiple instruction simultaneously
- Logic to commit instruction in correct order

9.10.2002 18

Overall Gain from Superscalar Implementation

- ”Base” machine, starting point for comparison
  - out-of-order issue
- See the effect of ...
  - renaming ⇒ right graph
  - issue window size ⇒ color of vertical bar
  - duplicated
    - data cache access ⇒ “+ld/st”
    - ALU ⇒ “ALU”
    - both ⇒ “both”
- Max speed-up about 4

(Fig. 13.5 [Stal99])

Fig. 14.5

Example: PowerPC 601 Architecture (2)

- General RISC organization
  - instruction formats Fig. 11.9 (Fig. 10.9 [Stal99])
  - 3 execution units Fig. 14.10 (Fig. 13.10 [Stal99])
- Logical view Fig. 14.11 (Fig. 13.11 [Stal99])
  - 4 instruction window for issue
  - each execution unit picks up next one for it whenever there is room for new instruction
  - integer instructions issued only when 1st (dispatch buffer 0) in queue

9.10.2002 21

PowerPC 601 Pipelines (4)

(Fig. 13.12 [Stal99]) Fig. 14.12

- Instruction pipelines
  - all state changes in final “Write Back” phase
  - up to 3 instruction can be dispatched at the same time, and issued right after that in each pipeline if no dependencies exist
    - dependencies solved by stalls
  - ALU ops place their result in one of 8 condition code field in condition register
    - up to 8 separate conditions active concurrently

9.10.2002 22

PowerPC 601 Branches (4)

- Zero cycle branches
  - branch target addresses computed already in lower dispatch buffers
    - before dispatch or issue!
  - Easy: unconditional branches (jumps) or branch on already resolved condition code field
  - otherwise
    - conditional branch backward: guess taken
    - conditional branch forward: guess not taken
    - if speculation ends up wrong, cancel conditional instructions in pipeline before write-back
    - speculate only on one branch at a time

PowerPC 601 Example

- Conditional branch example
  - Original C code Fig. 14.13 (a) (Fig. 13.13 [Stal99])
  - Assembly code Fig. 14.13 (b)
    - predict branch <u>not taken</u>
  - Correct branch prediction (Fig. 13.14 [Stal99]) Fig. 14.14 (a)
  - Incorrect branch prediction Fig. 14.14 (b)

PowerPC 620 Architecture

- 6 execution units
- Up to 4 instructions dispatched simultaneously
- Reservation stations to store dispatched instructions and their arguments
  - kind of rename registers also!

[HePa96] Fig. 4.49

PowerPC 620 Rename Registers (7)

- Rename registers to store results not yet committed
  - normal uncompleted and speculative instructions
  - 8 int and 12 FP extra rename registers
    - in same register file as normal registers
  - results copied to normal registers at commit
  - information on what to do at commit is in completion unit in reorder buffers
- Instruction completes (commits) from completion unit reorder buffer once all previous instructions are committed
  - max 4 instructions can commit at a time

[HePa96] Fig. 4.49

PowerPC 620 Speculation

- Speculation on branches
  - 256-entry branch target buffer
    - two-way set-associative
  - 2048-entry branch history table
    - used when branch target buffer misses
  - speculation on max 4 unresolved branches

Intel Pentium II speculation

- 512-entry branch target buffer
  - 4-bit prediction state, 4-way set-associative
- Static prediction
  - used before dynamic will work
  - forward ”not taken”, backward ”taken”
- In-order-completion for 40 μops (micro-operations) limits speculation
- 4-entry Return Stack Buffer (RSB)
  - return addresses are often found quickly without accessing Activation Record Stack

Example: Pentium 4

- Outside: CISC ISA
- Inside: full superscalar RISC core with micro-operations (μops)
- Very long pipeline

Fig. 14.7

  - get next ISA instruction (rarely)
    - map it to μops
  - get μops from Trace Cache (usually)
    - Trace Cache = L1 Instruction Cache
    - additional μops from ROM, if needed
  - finish with μops
  - drive stages just to make up for the time for the signal to traverse the chip

Fig. 14.8

9.10.2002 31

Pipeline Front End

a) Fetch instruction from L2 cache and generate μops when needed
  - store them to Trace Cache
  - static branch prediction
    - backward ”taken”, forward ”not taken”

Fig. 14.9 (a-f)

b) Get new trace cache IP (for μops)
  - dynamic 4-bit branch prediction with 512 entry BTB

c) Trace cache fetch

d) Drive – let data traverse the chip

9.10.2002 32

Pipeline Out-of-Order Execution

Fig. 14.9 (a-f)

e) Allocate resources

- reorder buffer (ROB) entry (one of 126)
  - state: scheduled, dispatched, completed, ready
  - original IA-32 instruction address
  - μop and which operands for it are available
  - alias register for result (one of 128 ROB registers referencing one of 8 IA-32 register, or one of 48 load or 24 store buffers)
    - true data dependencies solved with these
    - false dependencies avoided by these
  - 2 Register Alias Tables (RAT) keep track where current version of each 8 IA-32 register (E.g., EAX) is

9.10.2002 33

Pipeline Out-of-Order Execution

Fig. 14.9 (a-f)

f) 2 FIFO queues for μop scheduling

- memory μops
- non-memory μops
- instruction ”dispatch” to execution window from these queues
- in-order from each queue, out-of-order globally

Fig. 14.9 (g-l)

g) Schedule and (h) dispatch μops (superscalar)

- window of execution = ??? μops
- max 6 instructions ”issued” each cycle
- out-of-order scheduling
  (because of 2 queues)

9.10.2002 34

Pipeline Int and FP Units

Fig. 14.9 (g-l)

i) Register access, data cache access

j) Execute, set flags

- Many different, pipelined execution units
  - E.g., double speed ALU for most common cases
- Update RAT, allow new μops to issue

k) Branch checking

- "kill" bad instructions in pipeline

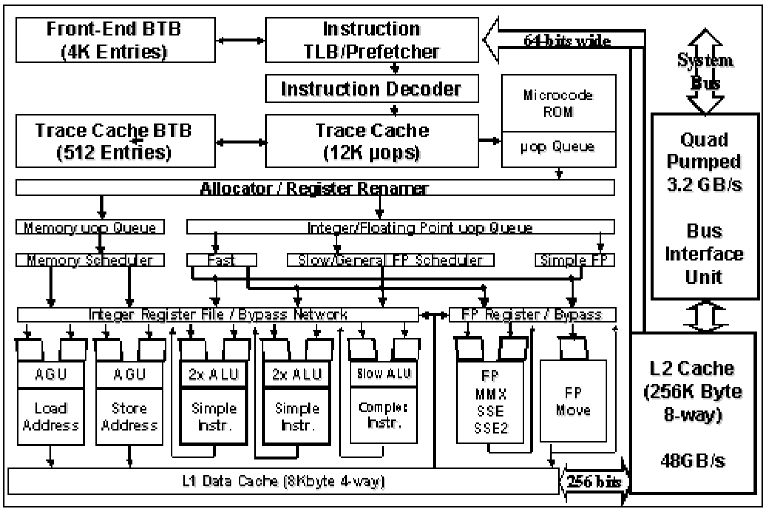
l) Give branch prediction

- let signals propagate

9.10.2002 35

-- End of Chapter 14: Superscalar --

Figure 4: Pentium® **4 processor microarchitecture**



http://www.intel.com/technology/itj/q12001/articles/art_2.htm

9.10.2002 36