Superscalar Processors
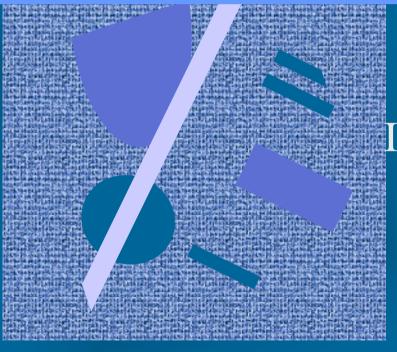# Ch 13

Limitations, Hazards

Instruction Issue Policy

Register Renaming

Branch Prediction

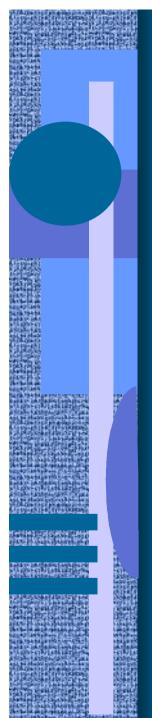# Superscalar Processing (5)

- Basic idea: more than one instruction completion per cycle

- Aimed at speeding up scalar processing
  - use multiple pipelines and not more phases

  Fig. 13.2

- Many instructions in <u>execution phase</u> simultaneously

  - need parallelism also in earlier & later phases

- Multiple pipelines

  Fig. 13.1

  - question: when can instruction be executed?

- Fetch many instructions at the same time

  - memory access must not be bottleneck

Why couldn't we execute this instruction right now? (3)

Fig. 13.3

- (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

Why couldn't we execute this instruction right now?

- Name dependency (nimiriippuvuus)
  - two instruction 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
  - overall result is not the one intended
  - two cases: Output Dependency & Antidependency
    (outputriippuvuus)   (antiriippuvuus)

  - what if there are aliases? (two names, same data)

Output Dependency? (1)

- Some earlier instruction has not yet finished <u>writing</u> from the same location that we want to <u>write</u> to

read	r1, sum
add	r2, r1, r3
add	r1, r4, r5

- Need to preserve order

Want to have sum of r4 and r5 in r1

Antidependency (1)

Some earlier instruction has not yet finished <u>reading</u> from the same location that we want to <u>write</u> 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
  - 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 (stupid) design?

Superscalar Processor

- Instruction dispatch
  - get next available executable instruction from instruction stream

- Window of execution
  - all instructions that are <u>considered</u> to be issued

- Instruction issue
  - allow instruction to start execution
  - execution and completion phase should continue now with <u>no stalls</u>

- Instruction reorder and commit (retiring)
  - hopefully all system state changes here!
  - last chance to change order or abandon results

Instruction Dispatch

- Whenever there are both
  - available slots in window of execution
  - ready instructions from prefetch or branch prediction buffer

Window of Execution

- 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

Instruction Issue

- 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

Instruction Issue Policies (3)

- Instruction fetch policy
  - constraints on how many instruction allowed to be considered to be dispatched at a time
    - 2 instructions fetched and decoded at a time
      $\Rightarrow$ both must be dispatched before next 2 fetched

- Instruction execution policy
  - constraints on which order dispatched instructions may start execution

- Completion policy
  - constraints of which order completions can occur

Copyright Teemu Kerola 1999

Example Issue Policy (6)

- In-order issue with in-order completion
  - same as purely sequential execution
  - no instruction window needed

  Fig. 13.4 (a)

  - 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

Example Issue Policy (5)

- In-order issue with <u>out-of-order completion</u>
  - issue in original order

    Fig. 13.4 (b)

    - 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
    - <u>output</u> dependencies: wait long enough to solve them

Example Issue Policy (5)

- <u>Out-of-order issue</u> with out-of-order completion
  - issue in any order
    - many can be issued at the same time
  - <u>instruction window</u> for dynamic instruction scheduling
  - allow executions complete before those of earlier instructions
  - Check before issue:
    - resource conflicts, data & control dependencies
    - output dependencies
    - antidependencies: must wait for earlier instructions issued later to pick up arguments before overwriting them

The real superscalar processor

Fig. 13.4 (c)

Get Rid of Name Dependencies (2)

- 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

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:

Rename registers to hardware registers R3a, R3b, R3c, R4b, R5a, R7b

No name dependencies now:

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)

- Drawback: need more registers
- Why R3a & R3b?

Superscalar Implementation (6)

Fig. 13.6

- Fetch strategy
  - prefetch, branch prediction
- Dependency check logic
  - forwarding circuits to transfer dependency data directly instead via registers or memory
- 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

Overall Gain from Superscalar Implementation

- Seem the effect of ... Fig. 13.5
    - renaming $\Rightarrow$ right graph
    - window size $\Rightarrow$ color of vertical bar
    - out-of-order issue $\Rightarrow$ "base" machine
    - duplicated
        - data cache access $\Rightarrow$ "+ld/st"
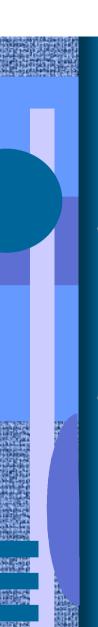   - ALU $\Rightarrow$ "ALU"
        - both $\Rightarrow$ "both"
- Max speed-up about 4

Example: PowerPC 601 Architecture (2)

- General RISC organization
  - instruction formats   Fig. 10.9
  - 3 execution units   Fig. 13.10

- Logical view   Fig. 13.11
  - 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 in queue

PowerPC 601 Pipelines (4)

- Instruction pipelines   Fig. 13.12
  - 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

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. 13.13 (a)
  - Assembly code   Fig. 13.13 (b)
    - predict branch <u>not taken</u>
  - Correct branch prediction   Fig. 13.14 (a)
  - Incorrect branch prediction   Fig. 13.14 (b)

PowerPC <u>620</u> Architecture

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

Fig. 4.25

[HePa96] Fig. 4.49

PowerPC <u>620</u> Rename Registers (7)

- Rename registers to store results not yet committed
  [HePa96] Fig. 4.49

  - 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 <u>completion unit</u> 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

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

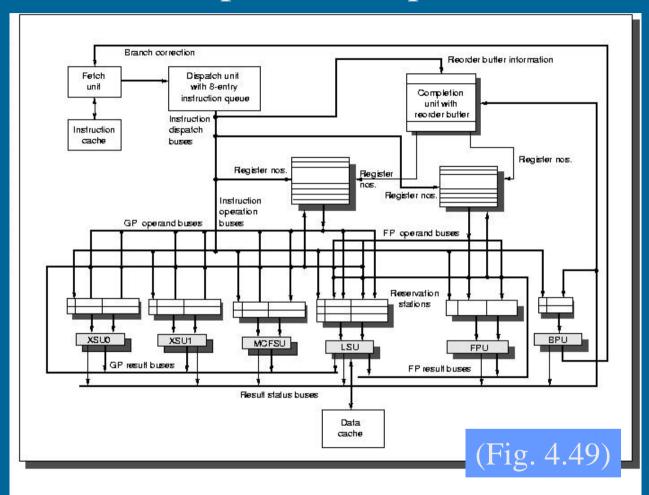# -- End of Chapter 13: Superscalar --

(Fig. 4.49)

FIGURE 4.49 The PowerPC 620 has six different functional units, each with its own reservation stations and a 16-entry reorder buffer, contained in the instruction completion unit.

(Hennessy-Patterson, Computer Architecture, 2nd Ed, 1996)