Multiple Instruction Issue

Multiple instructions issued each cycle

$\Rightarrow$ better performance

increase throughput

decrease in CPI (below 1)

increase in functional unit utilization

$\Rightarrow$ greater hardware complexity, potentially longer wire lengths

$\Rightarrow$ harder code scheduling job for the compiler if statically scheduled

Superscalar processors

- varying number of instructions issued simultaneously
- statically and/or dynamically scheduled

VLIW (very long instruction word) **processors**

- multiple & fixed number of operations formatted as one big instruction
- statically scheduled
- usually LIW (3 operations) today

Superscalars

Features

- issues variable number of instructions
  - in-order execution: dispatch buffer (A)
  - in-order execution: shift register model (S)
  - out-of-order execution (M X)
- restrictions on instruction types that can be issued together
  - to reduce number of FUs
  - to reduce register file ports
  - reduce buses between slots & FUs
- issued instructions are usually independent
  - some exceptions (A S)

Alpha 21164

In-order instruction issue

- **instruction slotting**
    - can issue up to 4 instructions
    - constraints on types of instructions that can issue together
        - 1 shift, integer multiply, branch
        - 2 other integers, load/store, FP
- completely empty the instruction buffer before fill it again
    - compiler can pad with `nops` so the second (conflicting) instruction is issued with the following instructions, not alone
- no data dependences in same issue cycle
    - one exception: compare with conditional move or conditional branch
- a scoreboard
    - detects data hazards
    - controls bypass logic
- independent instructions behind a load miss can issue out-of-order *(later)*

UltraSparc 1

In-order instruction issue

- **shift register model**
- (almost) any instruction in any slot
    - choose from 2 integer, 2 FP/graphics, 1 load/store, 1 branch
- some data dependent instructions can issue in same cycle
    - store address calculation with store
    - branch & instruction that sets its CCs

Pentium Pro

In-order instruction issue

- 3 decoders produce up to 6 μops each cycle
  - 1 for instructions of any complexity (produces up to 4 μops)
    - μop sequencer generates more in subsequent cycles
  - 2 for simple R-type instructions (produce 1 μop each)
    - wait until decoder for complex instructions have determined the length of the first instruction
    - only decode RISC-like instructions
    - if not RISC-like, then shift to complex decoder for decoding the next cycle
  - static code scheduling is important
- **RISClike μops**:
  - follows the load/store model
  - hold operand values (32 bits)
  - 118 bits total
- up to 3 μops renamed and logged into ROB

Out-of-order dispatch to FUs

- 5 μops per cycle
  - restrictions: 2 calculations, 1 load, 1 store address, 1 store data
  - restrictions on which calculations can dispatch together
- older μops go first, if a conflict for functional units

Superscalars

Additional hardware

- wider instruction fetch
- multiple decoders & hazard detection logic
- multiported registers for multiple register access
- buses for data routing
- pipelined functional units
- bypass logic between different pipelines

Static scheduling helps (*some new techniques later*)

VLIW Processors

1 very long instruction contains suboperations

- change in the instruction set architecture

Machines: Multiflow & Cydra 5 (8 or 16 suboperations)
IA-64 today (3 suboperations)

Hardware

- only 1 compiler-generated instruction
  - 1 program counter
  - no multiple issue hardware
- multiple operations must issue in parallel
  - fixed format so could decode operations in parallel
  - separate and multiported registers per suboperation
  - separate FUs per suboperation
  - pipelined FUs
  - destination registers reserved to prevent WAW and RAW hazards

VLIW Processors

Compiler support to increase ILP

- compiler creates each VLIW word
- detects hazards (data, structural, latencies)
  - memory disambiguation wrt memory banks (Multiflow)
  - memory bank conflicts separated by memory latency amount of time (Cydra)
  - structural hazards for register buses
- need for good scheduling greater than with in-order issue superscalars
  - operation doesn't issue if 1 can't
  - schedules taking dependence, functional unit type, operation latencies into account
    - no 2 operations to the same functional unit
    - no 2 operations to the same memory bank
    - no 2 operations that require the same bus to registers
- techniques for increasing ILP
  - loop unrolling
  - global code scheduling
  - software pipelining
  - trace scheduling

IA-64 EPIC

Explicitly Parallel Instruction Computing

3 operations/instruction

Techniques to increase ILP

- predicated execution
    - eliminate branches: better compiler scheduling
    - execute both paths
- speculative loads

Techniques to avoid complex out-of-order execution hardware

- 128 integer & FP registers
    - no hardware register renaming
- unspecified dependency hints between instructions
    - simpler instruction issue logic
- clustered design: partition functional units & register files
    - fewer register ports
    - smaller cross-bar between registers & functional units
- speculative loads

More Techniques to Increase ILP

Compiler

- software pipelining
- trace scheduling

Hardware

- speculative execution

Software Pipelining

Execute instructions from different iterations in parallel

1	L_0								prologue
2	M_0	L_1							
3	A_0	M_1	L_2						
4	S_0	A_1	M_2	L_3					body : S_i A_{i+1} M_{i+2} L_{i+3}
5		S_1	A_2	M_3	L_4				
6			S_2	A_3	M_4	L_5			
7				S_3	A_4	M_5	L_6		
8					S_4	A_5	M_6	L_7	
9						S_5	A_6	M_7	epilogue
10							S_6	A_7	
11								S_7	

ld
mult
add
st

Software Pipelining

Performance compared to the **unoptimized loop**

+ better throughput from executing instructions from different iterations at the same time
- worse register pressure

Performance compared to **loop unrolling**

+ better throughput because filling & draining occurs once per loop, not once per unrolling factor number of iterations
+ better code density
- more loop overhead
- compiler algorithm complexity

Trace Scheduling

Exploits ILP across nonloop branches

Schedules for the most likely path (**trace**)

- **trace selection**
  - create a sequence of basic blocks
  - static heuristics or dynamic profiling
- **trace compaction**
  - schedule traces in order of expected frequency of execution
  - schedule across branch points

Schedules across basic blocks

- **compensation code** to guarantee correct execution if the trace is not executed
- speculative execution by hoisting code above branches
  - disable exceptions or don't move exception-prone code

Advantage

- better throughput because schedules across basic blocks

Disadvantage

- large code size

Speculative Execution

Execution an instruction that is after the branch **before** the branch

+ hide latencies
+ increase issue rate on a wide-issue machine
- extra instruction executed if branch mispredicted
- some schemes require extra registers

Must preserve

- flow of data
- same exception behavior

Speculative Execution

Hardware support for **compiler speculation**

HW & OS cooperation

- handle a resumable exception even if for a speculative instruction
- return undefined value for terminating exception (NaN)

Poison bit associated with each register (Rogers 93?)

- set when speculative instruction that writes its destination register generates a terminating exception
- passed to other speculative instructions
- interrupt taken when real instruction reads it
- resumable instructions handled immediately

Boosting (Smith 92)

- flag instruction as speculative
- provide register renaming HW (~ Tomasulo)
- out-of-order execution/in-order commit
- write real register at commit

Speculative Execution

Hardware speculative execution (M P X)

- out-of-order execution with in-order completion
- reorder buffer or active list
- exceptions handled before commit

Advantages

+ can take advantage of dynamic memory disambiguation
+ can be combined with dynamic branch prediction for a more accurate prediction
+ maintains precise interrupts for speculated instructions
+ only services nonspeculative interrupts
+ no compensation code
+ executes implementation-independent code well (when combined with dynamic scheduling)

Disadvantage

- complex hardware

Limitations on Multiple Instruction Issue

Inherent limits on (ability to extract) ILP

- need more independent instructions
  - data independent instructions
  - control independent instructions
  - pipeline depth * instruction-width number of instructions
- need good instruction mix that matches FU mix & issue restrictions
- need instructions with the same latency

Additional hardware resources to issue & execute multiple instructions

  - more complicated design
  - difficult logic verification
  - challenge to meet clock cycle time
- wider instruction fetch (linear increase)
- additional functional units (linear increase)
- additional register ports (quadratic increase)
- buses between functional units & registers (quadratic increase)
- multiple banked memory
- dynamic (costly) or static (less costly) scheduling hardware (superscalars only)

Limitations on Multiple Instruction Issue

Increase in code size, decrease in code density

(more of an issue for VLIWs)

- inlining to avoid jumps
- unrolled loops
- software pipelining
- trace scheduling with compensation code
- nops if can't fill a VLIW instruction
  - techniques to increase density:
    - 2 formats (Cydra 5)
    - expand compressed instruction when read from memory (Multiflow)

Binary code incompatibility for different implementations (VLIWs)

- implementation-dependent latencies
- different instruction widths, FUs

A need for similar/deterministic operation latencies

(an issue for VLIWs which require lockstep operation;
a stall in 1 FU causes the whole processor to stall)

Multiple Instruction Issue

Maximum ILP

```
a program-size         --> window = 32
instruction window         8-9 instruc/cycles (FP)
                           9-14 instruc/cycles (int)

infinite # FUs         --> 64
                           ?

infinite register      --> 64 registers
renaming                   11-28 instruc/cycle (FP)
                           9-11 instruc/cycle (int)

perfect branch         --> 2 bit
prediction                 13-46 instruc/cycle (FP)
                           6-7 instruc/cycle (int)

aliases known          --> normal compiler
                           technology
                           4-6 instruc/cycle (FP)
                           4-5 instruc/cycle (int)

single cycle           --> ?
instructions
```

```
10s (int)
100s, low 1000s (FP)
```


Latency

Techniques that eliminate or reduce the cost of hazards:

Throughput

Techniques that increase throughput:

Latency

Techniques that eliminate or reduce the cost of hazards:

- **all hazards**
  - static (code) scheduling
- **structural hazards**
  - more hardware
  - register renaming
- **data hazards**
  - code scheduling in the delay slot on delayed loads
  - prefetching
  - forwarding logic
  - several dynamic scheduling (out-of-order execution) schemes
    - scoreboarding
    - Tomasulo's algorithm
    - reorder buffer/active list
  - register renaming
  - software pipelining
  - dynamic memory disambiguation

Latency

- **control hazards**
  - predicated execution
  - pipeline redesign
  - code scheduling in the delay slot on delayed branches
  - static branch prediction
  - several dynamic branch prediction schemes
    - two-bit prediction buffers
    - correlating branch prediction
    - combination of local/global history prediction
  - branch target buffer
  - branch folding
  - return address stack
  - resume cache
  - fetch both instruction streams
  - loop unrolling
  - software pipelining
  - trace scheduling
  - several kinds of hardware support for compiler-based speculative execution schemes
    - HW/OS cooperation
    - poison bits
    - boosting

Throughput

Techniques that increase throughput:

- pipelining
- multiple instruction issue
  - superscalars
  - VLIW processors