Superscalars

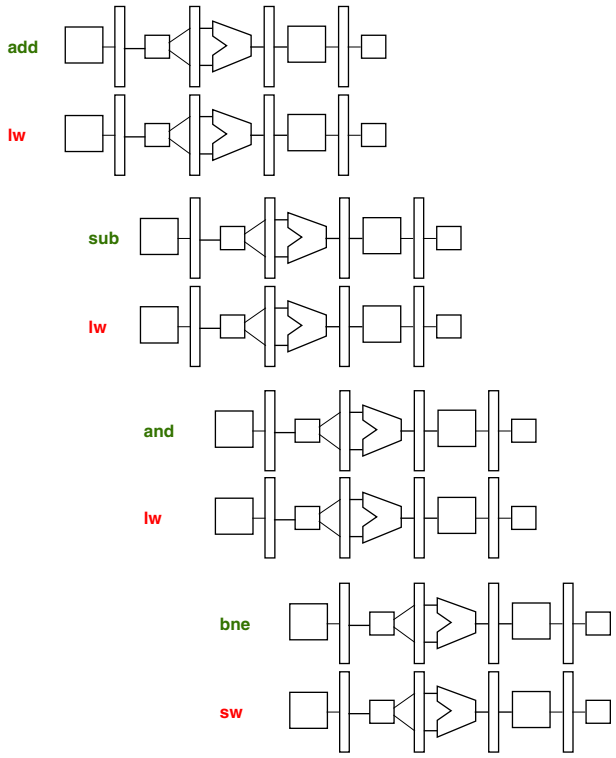
Definition:

- a processor that can issue & execute more than one instruction per cycle
- for example,
  integer computation, floating point computation, data transfer, transfer of control
- **issue width** = the number of **issue slots**, 1 slot/instruction
- the hardware decides which instructions can issue
  - static scheduling processors:
    - how many of the next *n* instructions can be issued, where *n* is the superscalar issue width
  - dynamic scheduling processors *(a little later)*

Sometimes there are restrictions on what type of instructions can issue together, for example:

- R-type or conditional branch
- memory operation

2-way Superscalar



Superscalars

Performance impact:

- increase performance because execute instructions in parallel, not just overlapped
- CPI potentially < 1 (.5 on our R3000 example) or IPC (instructions/cycle) potentially > 1 (2 on our R3000 example)
- get better functional unit utilization

but

- need **independent instructions** to execute instructions in parallel
  i.e., enough **instruction-level parallelism (ILP)**
  - instructions without data dependences
- need a **good mix of instructions** to utilize the different types of functional units
- need more instructions to hide load delays -- why?
- need to make better branch predictions --- why?

Superscalars

Hardware impact:

- multiple functional units
  How many ALUs on our R3000 superscalar?
- additional read/write ports on the register file
  How many read/write ports on our R3000 superscalar?
- more buses from the register file to the added functional units
- multiple decoders
- more hazard detection logic
- more forwarding logic
- wider instruction fetch

or else the processor has structural hazards (due to an unbalanced design) and stalling!

Code Scheduling

Code scheduling creates sequences of independent instructions
i.e., to exploit ILP
i.e., to break dependences

```
Loop:   lw $t0, 0(s1)
        addu $t0, $t0, $s2
        sw $t0, 0($s1)
        addi $s1, $s1, -4
        bne $s1, $0, Loop
```


Code Scheduling on Superscalars

```
Loop:   lw $t0, 0(s1)
        addi $s1, $s1, -4
        addu $t0, $t0, $s2
        sw $t0, 4($s1)
        bne $s1, $0, Loop
```

	ALU/branch instruction	Data transfer instruction	clock cycle
`Loop:`	`addi $s1, $s1, -4`	`lw $t0, 0($s1)`	1
			2
	`addu $t0, $t0, $s2`		3
	`bne $s1, $0, Loop`	`sw $t0, 4($s1)`	4

`lw addu sw` is the **critical path**

Illustrates why CPI is not .5!

Loop Unrolling

	ALU/branch instruction	Data transfer instruction	clock cycle
`Loop:`	`addi $s1, $s1, -16`	`lw $t0, 0($s1)`	1
		`lw $t1, 12($s1)`	2
	`addu $t0, $t0, $s2`	`lw $t2, 8($s1)`	3
	`addu $t1, $t1, $s2`	`lw $t3, 4($s1)`	4
	`addu $t2, $t2, $s2`	`sw $t0, 16($s1)`	5
	`addu $t3, $t3, $s2`	`sw $t1, 12($s1)`	6
		`sw $t2, 8($s1)`	7
	`bne $s1, $0, Loop`	`sw $t3, 4($s1)`	8

Loop unrolling provides

+ fewer instructions that can cause branch hazards
+ more independent instructions (from different iterations)
+ increase in throughput
- uses **more registers**
- must change **offsets**

What is the cycles per iteration?

What is the IPC?

In-order vs. Out-of-Order Execution

In-order instruction execution

- instructions are fetched, executed & retired in compiler-generated order
  - if one instruction stalls, all instructions behind it stall
- instructions are **statically scheduled** by the processor
  - means the hardware schedules them onto functional units in their compiler-generated order
  - hardware algorithm: how many of the next n instructions can be executed, where n is the superscalar issue width
    - superscalars can have data & structural hazards within the n instructions
- advantages of in-order execution
  - simpler implementation $\Longrightarrow$
    - faster clock cycle
    - fewer transistors

In-order vs. Out-of-Order Execution

Out-of-order execution

- instructions are fetched in compiler-generated order
- instructions complete in compiler-generated order
- in between they may be executed out of their compiler-generated order
- instructions are **dynamically scheduled** by the hardware
  - hardware decides in what order instructions can be executed
  - instructions behind a stalled instruction can pass it
- advantages: better performance
  - better at hiding latencies; less processor stalling; higher throughput
  - better utilization of functional units

Dynamically Scheduled Processors

Dynamically scheduled or out-or-order processors:

- do not necessarily issue instructions in program (compiler-generated) order
- issue instructions as soon as their operands are available
  - have been calculated in the ALU
  - have been loaded from memory
- ready instructions can issue before stalled instructions that are waiting for their operands to be computed
- when go around a **load** instruction that is stalled for a cache miss:
  - use **lockup-free caches** that allow instruction issue to continue while a miss is being satisfied
  - the load-use instruction still stalls
- when go around a **branch** instruction:
  - the instructions that are issued from the predicted path are issued speculatively, called **speculative execution**
  - when the branch is resolved, if the prediction was wrong, **wrong path instructions** are flushed from the pipeline
- instructions fetched and **retired** (**committed**) in order

Dynamically Scheduled Processors

Instruction issue does **NOT** necessarily go in program order

- the hardware decides which instructions should issue next

program order *(in-order execution)*

`lw $3, 100($4)`	in execution, cache miss
`add $2, $3, $4`	waits until the miss is satisfied
`sub $5, $6, $7`	waits for the add

execution order *(out-of-order execution)*

`lw $3, 100($4)`	in execution, cache miss
`sub $5, $6, $7`	in execution
`add $2, $3, $4`	waits until the miss is satisfied -- flow dependence still upheld

Superpipelining

Longer pipelines with shorter stages

- more work to do
  example: dynamic instruction scheduling in an out-or-order processor
- less work is done in each stage
  examples:
  data access in a high-performance processor: cache access on one cycle & data returned on the next cycle
  several cycles for decoding a CISC instruction set

Performance impact:

+ the increased instruction overlap can increase instruction throughput
- additional stages can increase hazard penalties, usually the branch misprediction penalty

DEC Alpha 21164 Integer Unit Pipeline

Fetch & issue

S0: instruction fetch
dynamic branch prediction

S1: opcode decode
target address calculation
if predict taken, redirect the fetch

S2: decide which of the next 4 instructions can be issued
(includes an intra-cycle structural & data hazard check)

S3: inter-cycle load-data hazard check
register read
in-order instruction issue

Execute (2 pipelines, one for arithmetic & branches, one for loads & stores)

S4: integer execution
effective address calculation

S5: branch execution
data cache access

S6: register write

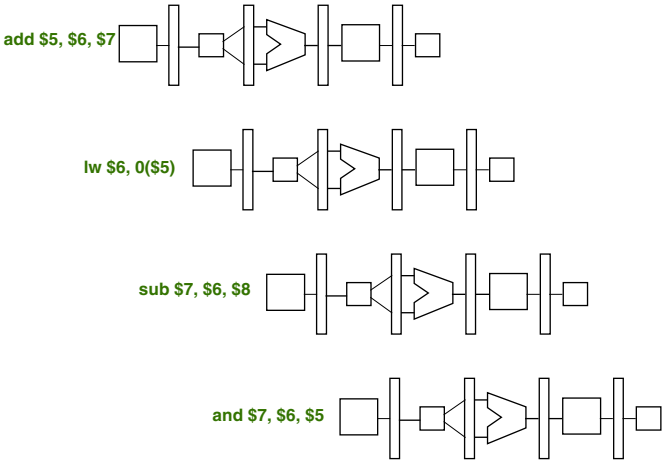
Designing a Pipeline-friendly Architecture

Architectural features that go well with pipelined implementations

- simple instructions
  - all instructions take about the same number of stages
  - work in all stages takes about the same amount of time
- fixed-length instructions
  - can decode all instruction fields in parallel
  - can fetch & decode multiple instructions in parallel (superscalars)
- few instruction formats & fixed fields in most formats
  - can decode all instruction fields in parallel
  - can read operands, decode opcode & generate opcode-specific control signals at the same time
- memory operands only in load/store instructions
  - all instructions take about the same number of stages
  - all stages take about the same amount of time
    - can calculate the effective address in EX stage for a shorter pipeline

What type of architecture does this describe?

A LIttle Practice



Are there any dependences in this code?

What kind & where?

Do they cause any hazards in the pipeline?

If so, where?

Can the hazards be eliminated? How?

How many cycles does it take to execute this code?

Review

Techniques that eliminate hazards.

What types of hazards to they eliminate?

- duplicate hardware
- move the hardware
- flush
- hardware interlock (stall)
- forwarding
- branch prediction
- insert a nop
- code schedule an independent instruction