Superscalars

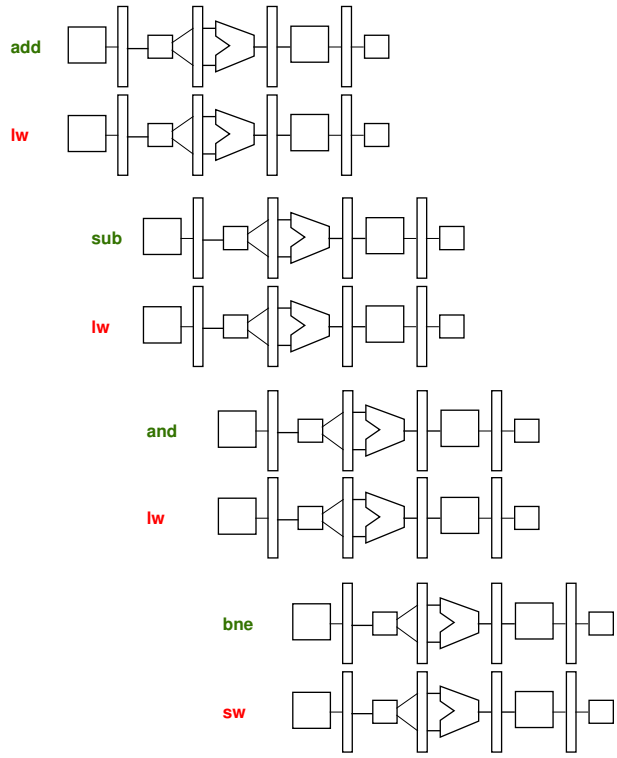
Definition:

- a processor that can 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
    - superscalars can have structural & data hazards within the *n* instructions
  - 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)
- better functional unit utilization

but

- need **independent instructions** to execute instructions in parallel
  i.e., enough **instruction-level parallelism (ILP)**
- 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?
- multiple decoders
- more hazard detection logic
- more forwarding logic, plus buses to the added functional units
- wider instruction fetch

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

Code Scheduling

```
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

Loop unrolling provides:

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

	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

What is the cycles per iteration?

What is the IPC?

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
  - increases throughput
- 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 processors, the fetch order)

`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 processors)

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

Superpipelining

Longer pipelines with shorter stages

- more work to do
  for example: instruction issue in an out-or-order processor
- less work is done in each stage
  for example:
  data access in a high-performance processor: cache access on one cycle & data returned on the next cycle

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
opcode decode
if predict taken, redirect the fetch

S2: instruction slotting: decide which of the next 4 instructions can be issued
- intra-cycle structural & data hazard check

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

Execute (2 pipelines)

S4: integer execution
effective address calculation

S5: branch execution
data cache access

S6: register write

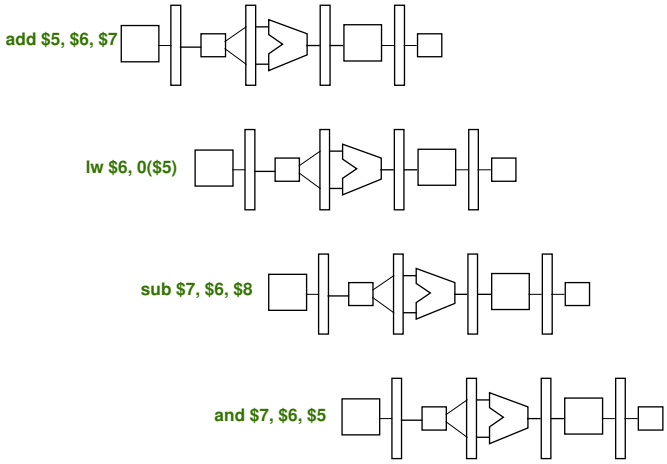
Designing a Pipeline-friendly Architecture

Architectural features that go well with pipelined implementations

- simple instructions
  - ⇒ all instructions take about the same number of stages
  - ⇒ all stages take about the same amount of time
- fixed length instructions
  - ⇒ can decode instruction fields in parallel
  - ⇒ can fetch & decode multiple instructions in parallel (superscalars)
- few instruction formats & fixed fields in most formats
  - ⇒ can decode 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
  - ⇒ can calculate the effective address in EX stage for a shorter pipeline
  - ⇒ all instructions take about the same number of stages
  - ⇒ all stages take about the same amount of time

What type of architecture does this describe?

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 eliminate? How?

How many cycles does it take to execute this code?

Review

Techniques that eliminate hazards:

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

What types of hazards to they eliminate?