In-order vs. Out-of-order Execution

In-order instruction execution

- instructions are fetched, executed & committed in compiler-generated order
  - if one instruction stalls, all instructions behind it stall
- instructions are **statically scheduled** by the hardware
  - scheduled in compiler-generated order
  - 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
- advantage of in-order instruction scheduling: simpler implementation
  - faster clock cycle
  - fewer transistors
  - faster design/development/debug time

In-order vs. Out-of-order Execution

Out-of-order instruction execution

- instructions are fetched in compiler-generated order
- instruction completion may be in-order (today) or out-of-order (older computers)
- in between they may be executed in some other 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 if not dependent upon it
- advantages: higher performance
  - better at hiding latencies, less processor stalling
  - higher utilization of functional units

In-order instruction issue: Alpha 21164

2 styles of static instruction scheduling

- dispatch buffer & instruction slotting (Alpha 21164)
- shift register model (UltraSPARC-1)

In-order instruction issue: Alpha 21164

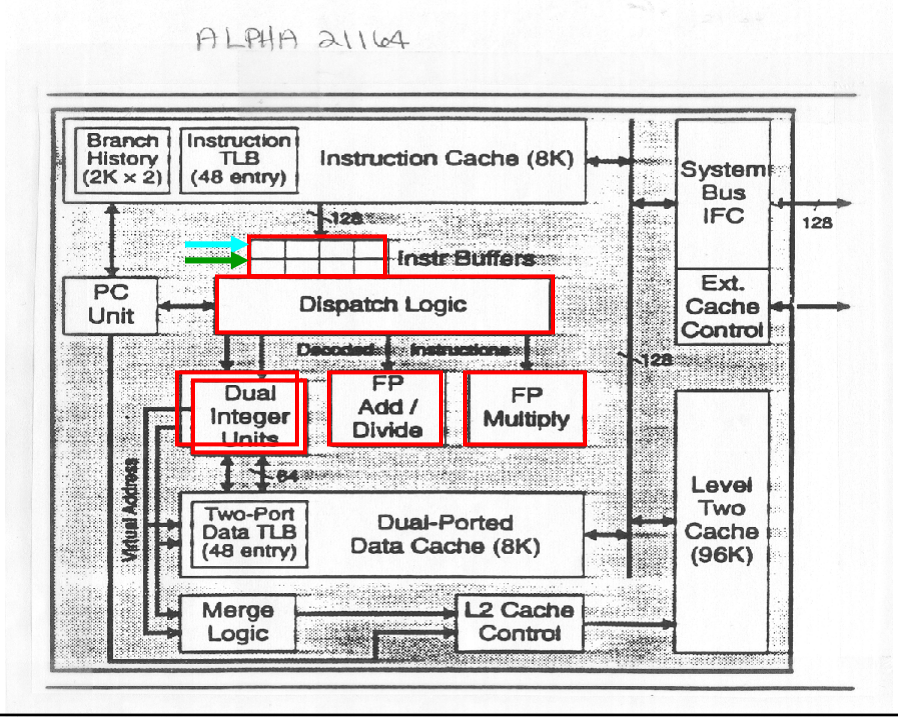
Instruction slotting

- can issue up to 4 instructions
  - completely empty the instruction buffer before filling it again
  - compiler can pad with nops so a conflicting instruction is issued with the following instructions, not alone

21164 Instruction Unit Pipeline

Fetch & issue

- **S0**: instruction fetch
  branch prediction bits read
- **S1**: opcode decode
  target address calculation
  if predict taken, redirect the fetch
- **S2**: **instruction slotting**: decide which of the next 4 instructions can be issued
  - intra-cycle structural hazard check
  - intra-cycle data hazard check
- **S3**: **instruction dispatch**
  - inter-cycle load-use hazard check
  - register read



In-order instruction issue: UltraSparc 1

Shift register model

- can issue up to 4 instructions per cycle
- shift in new instructions after every group of instructions is issued

Code Scheduling on Superscalars

Original code

```
Loop:  lw R1, 0(R5)
       addu R1, R1, R6
       sw R1, 0(R5)
       addi R5, R5, -4
       bne R5, R0, Loop
```

Code Scheduling on Superscalars

Original code

```
Loop:  lw R1, 0(R5)
       addu R1, R1, R6
       sw R1, 0(R5)
       addi R5, R5, -4
       bne R5, R0, Loop
```

With load-latency-hiding code

```
Loop:  lw R1, 0(s1)
       addi R5, R5, -4
       addu R1, R1, R6
       sw R1, 4(R5)
       bne R5, $0, Loop
```

	ALU/branch instructions	memory instructions	clock cycle
Loop:			1
			2
			3
			4

Code Scheduling on Superscalars: Loop Unrolling

	ALU/branch instruction	Data transfer instruction	clock cycle
Loop:	addi R5, R5, -16	lw R1, 0(R5)	1
		lw R2, 12(R5)	2
	addu R1, R1, R6	lw R3, 8(R5)	3
	addu R2, R2, R6	lw R4, 4(R5)	4
	addu R3, R3, R6	sw R1, 16(R5)	5
	addu R4, R4, R6	sw R2, 12(R5)	6
		sw R3, 8(R5)	7
	bne R5, R0, Loop	sw R4, 4(R5)	8

What is the cycles per iteration?
What is the IPC?

Code Scheduling on Superscalars: Loop Unrolling

Advantages:

+

+

-

-

Superscalars

Hardware impact:

- more & pipelined functional units
- multi-ported registers for multiple register access
- more buses from the register file to the additional functional units
- multiple decoders
- more hazard detection logic
- more bypass logic
- wider instruction fetch
- multi-banked L1 data cache

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

There are restrictions on instruction types that can be issued together to reduce the amount of hardware.

Static (compiler) scheduling helps.