Superscalar and VLIW processor design

Multiple instruction issue per clock

Goal: Extracting ILP so that CPI < 1 (or IPC > 1)

Superscalar: • Combine static and dynamic scheduling to issue multiple instructions per clock

- HW-centric and less sensitive to poorly scheduled code
- Predominant: PowerPC, Sparc, Alpha, HP-PA …

Very Long Instruction Words (VLIW):

- Static scheduling used to form packages of independent instructions that can be issued together.
- Relies on compiler to find independent instructions.

Example: A Superscalar MIPS

- Issue 2 instructions simultaneously: 1 FP & 1 integer
- Fetch 64 bits/clock cycle; Integer instr. on left, FP on right
- Can only issue 2nd instruction if 1st instruction issues
- Need more ports to the register file

Type	*Pipe stages*					
Int.	IF	ID	EX	MEM	WB	
FP	**IF**	**ID**	**EX**	**MEM**	**WB**	
Int.		IF	ID	EX	MEM	WB
FP		**IF**	**ID**	**EX**	**MEM**	**WB**
Int.			IF	ID	EX	MEM WB
FP			**IF**	**ID**	**EX**	**MEM WB**

EX stage should be fully pipelined
1 load delay slot corresponds to three instructions!
Example instruction sequence:

	Integer instruction		FP instruction	Clock cycle
Loop:	LD	F0,0(R1)		1
	LD	F6,-8(R1)		2
	LD	F10,-16(R1)	ADDD F4,F0,F2	3
	LD	F14,-24(R1)	ADDD F8,F6,F2	4
	LD	F18,-32(R1)	ADDD F12,F10,F2	5
	SD	0(R1),F4	ADDD F16,F14,F2	6
	SD	-8(R1),F8	ADDD F20,F18,F2	7
	SD	-16(R1),F12		8
	SD	-24(R1),F16		9
	SUBI	R1,R1,#40		10
	BNEZ	R1,LOOP		11
	SD	-32(R1),F20		12

- Difficult to find a sufficient number of instructions to issue.
- Can be scheduled dynamically with Tomasulo's algorithm.

Limits to superscalar execution

Difficulties in scheduling within the constraints on number of functional units and the ILP in the code chunk

- Instruction decode complexity increases with the number of issued instructions.• Data and control dependencies are in general more costly in a superscalar processor than in a single-issue processor.

Techniques to enlarge the instruction window to extract more ILP are important

Example: Very Long Instruction Word (VLIW) MIPS

- Independent funtional units with no hazard detection.
- Compiler is responsible for instruction scheduling.

Mem ref 1	Mem ref 2	FP op 1	FP op 2	Int op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1), F4	SD -8(R1), F8	ADDD F28,F26,F2			6
SD -16(R1), F12	SD -24(R1), F8				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD 0(R1),F28				BNEZ R1,LOOP	9

Limits to VLIW

Difficult to exploit parallelism: *n* functional units and *k* pipeline stages implies $n \times k$ independent instructions are being executed at a time.

Complexity increases with number of functional units

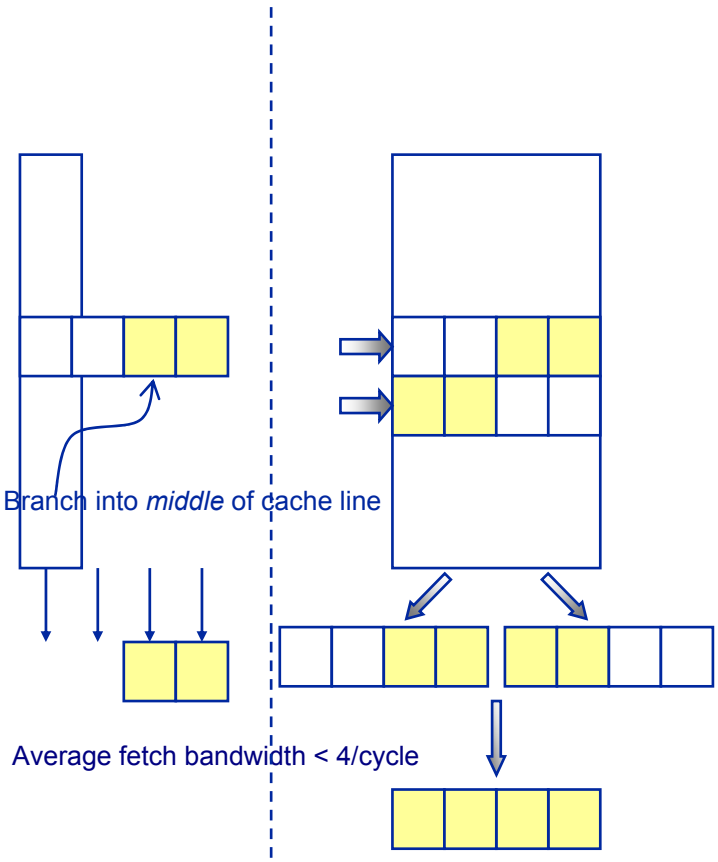
Code size.

No binary code compatibility.

High-bandwidth instruction fetching

If we intend to issue multiple instructions at a time, we must fetch multiple instructions at a time.

At first glance, this requires all of the fetched instructions to reside in the same cache line.



- Dual-port the I$ in order to read out two *consecutive* cache lines.
- Merge needed instructions together to get 4/cycle bandwidth.
- *Expensive*— true dual-porting increases area quadratically.

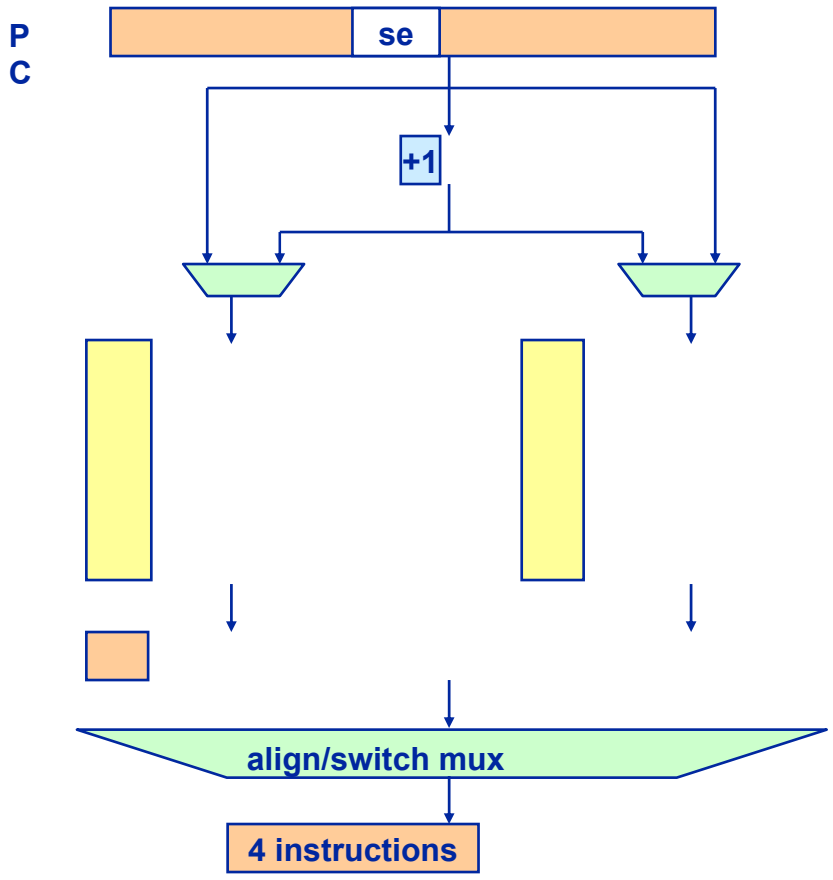
Inexpensive dual-porting: Interleaving

Realization:

- We don't need to read out *any* two cache lines.
- We just need to read out two *consecutive* cache lines.

How can we do this by changing the cache hardware slightly, without adding associativity or full multiporting?

Simplified IBM PowerPC solution:



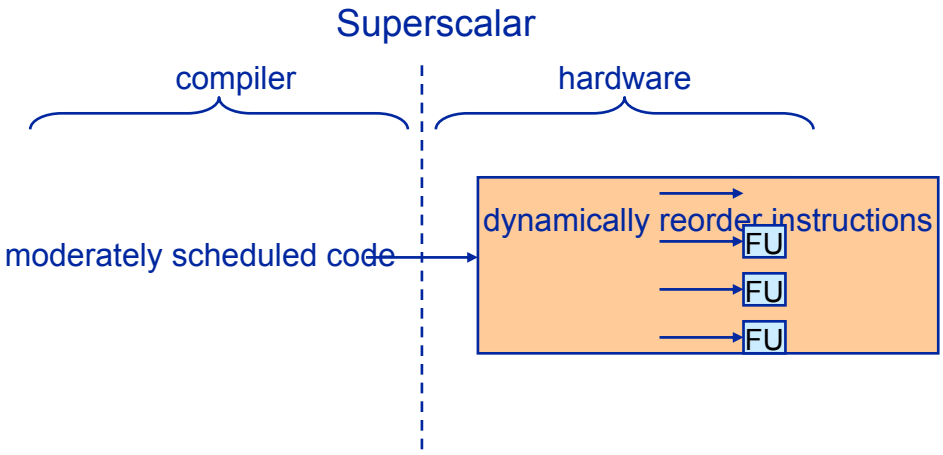
Static Scheduling

So far we have been talking about dynamic scheduling of instructions—the hardware detects instructions that can be issued together.

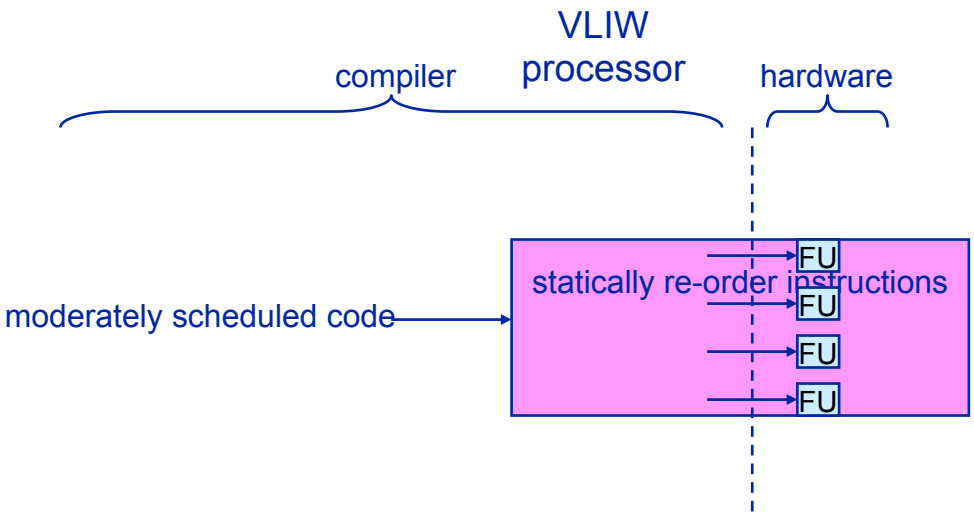
It is also possible to do this analysis at compile time. We can have the compiler reorder code to improve performance.

Static vs. dynamic scheduling

Dynamic



Static



What are some of the advantages of dynamic scheduling?

-

What are some of the advantages of static scheduling?

-

Bottom line: Find a good combination of the two.

Support for high-performance static scheduling

Register pressure → large architectural register file

Branches → compile-time region formation

Ambiguous memory dependences → speculative loads

Static scheduling techniques

Local scheduling (within a basic block)
Loop unrolling
Software pipelining (modulo scheduling)
Trace scheduling
Predication

Local scheduling

(+) Simple: No speculation (no region formation)

(–) Limited parallelism (usually < 2)

Example:

```
Loop: LD     F0, 0(R1)
     (stall)
       ADD.D F4, F0, F2
     (stall)
     (stall)
       SD    F4, 0(R1)
       ADD   R1, R1, 8
       BNE   R1, xxx, Loop
```

⇓

```
Loop: LD    F0, 0(R1)
     (stall)
       ADD.D F4, F0, F2
     (stall)
     (stall)
       SD    F4, 0(R1)
       ADD   R1, R1, 8
       BNE   R1, xxx, Loop
```

How many cycles/iteration?

⇓

```
Loop: LD   F0, 0(R1)
     (stall)
       ADDD F4, F0, F2
       ADD  R1, R1, 8
       BNE  R1, xxx, Loop
       SD   F4, -8(R1)
```