http://inst.eecs.berkeley.edu/~cs152

CS 152/252A Computer Architecture and Engineering

Sophia Shao

Lecture 14: VLIW

The 'Itanic'—Intel's ill-fated Itanium processor—finally sinks

After two decades of failure and endless jokes, the Intel Itanium is officially no more. Intel has finally stopped shipping its doomed-from-the-start 64-bit processor.

Itanium processors were promised to be more efficient because they didn't have the baggage of legacy x86 processors. But it was notoriously difficult to write a good compiler for the platform, and without a developer ecosystem, it went nowhere.

As if that wasn't bad enough, the processor lacked legacy 32-bit x86 support... So with no native software and no backwards compatibility, there was nothing to run on it.

https://www.networkworld.com/article/3628450/the-itanic-finally-sinks.html

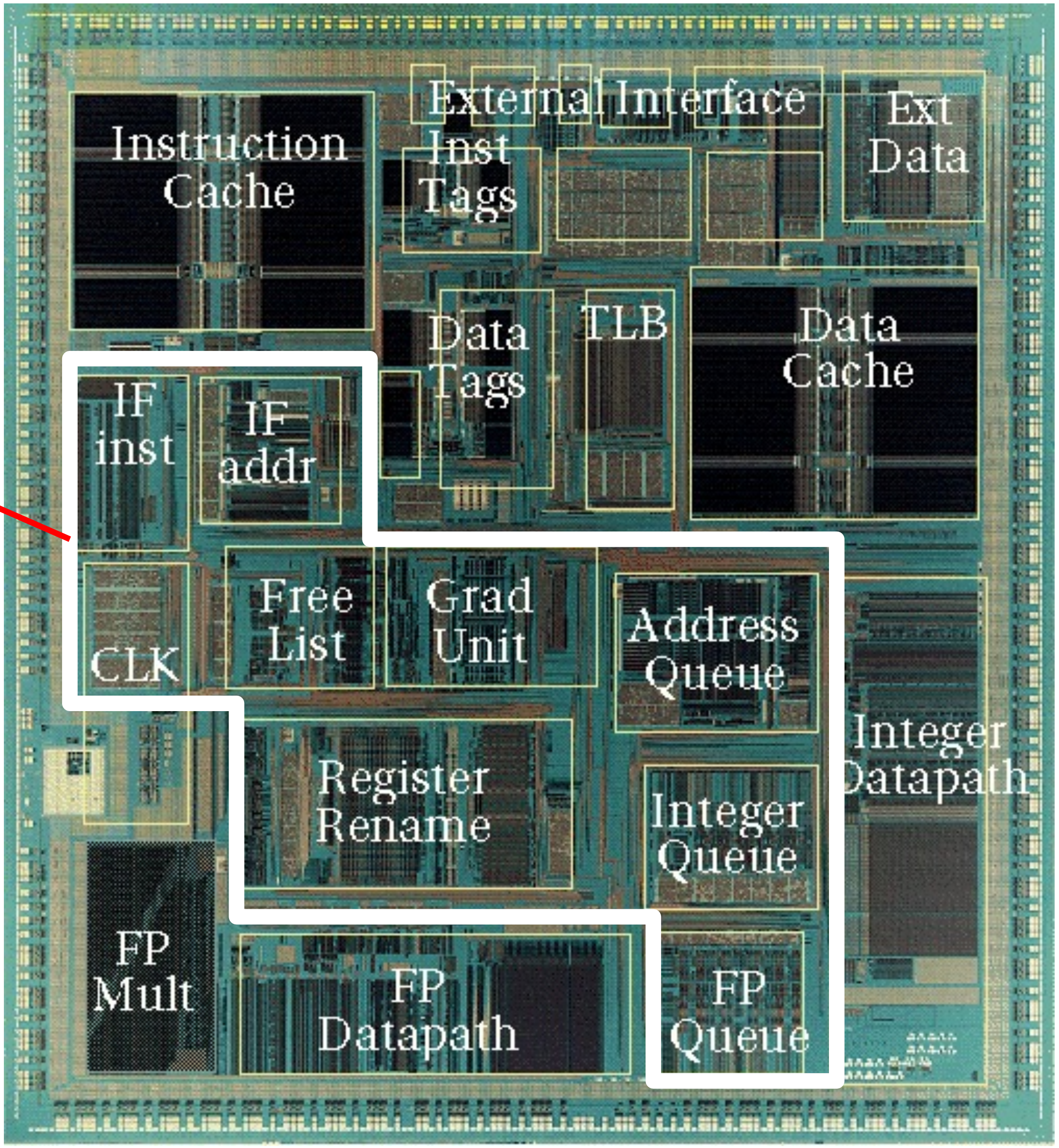
Last Time in Lecture

- Phases of instruction execution:
  - Fetch/decode/rename/dispatch/issue/execute/complete/commit
- Data-in-ROB design versus unified physical register design
- Superscalar register renaming

- Instruction-level parallelism (ILP)
  - Instruction schedule to facilitate parallel processing

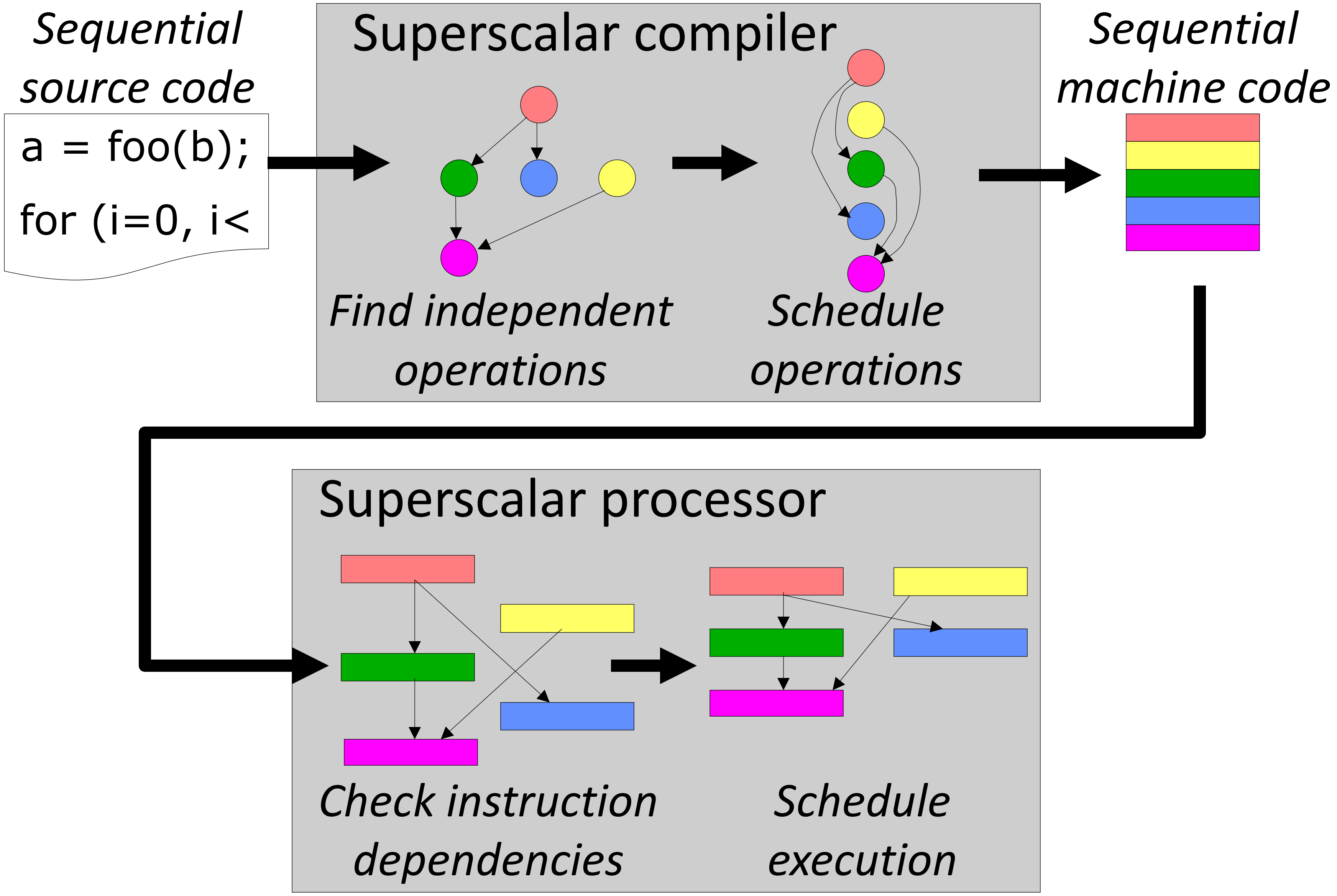
Out-of-Order Control Complexity: MIPS R10000

Control Logic

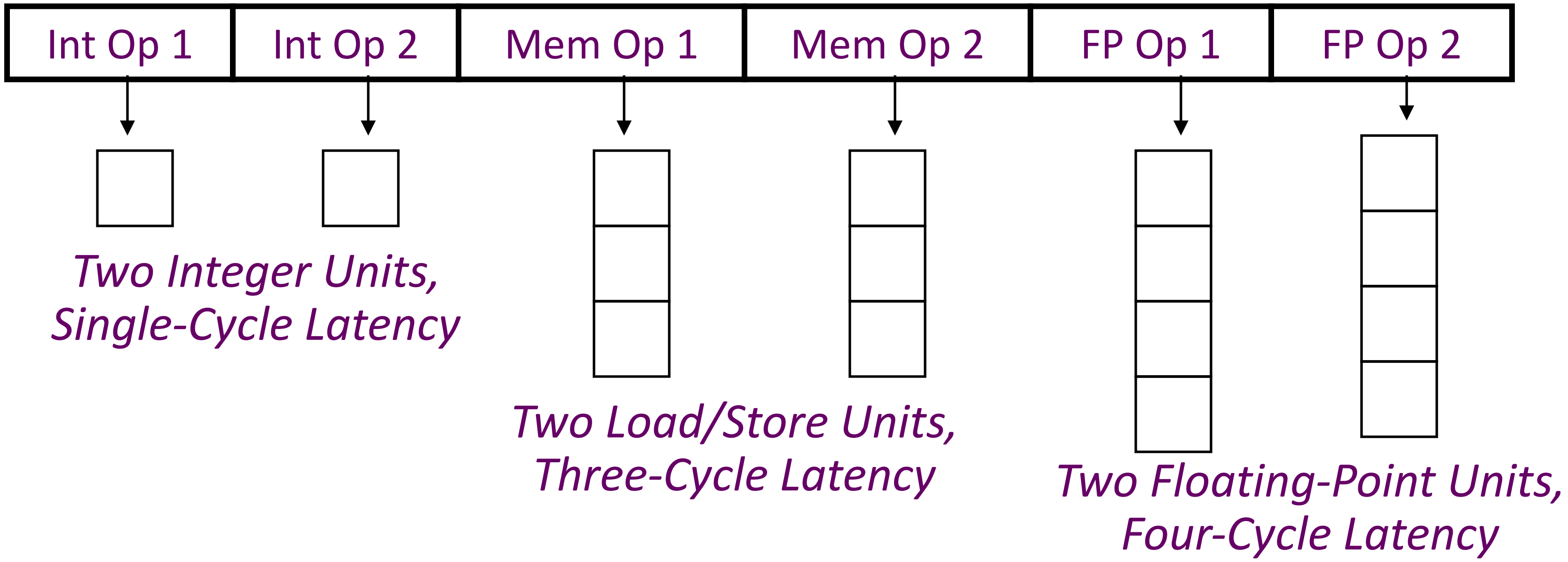


[SGI/MIPS Technologies Inc., 1995]

Sequential ISA Bottleneck

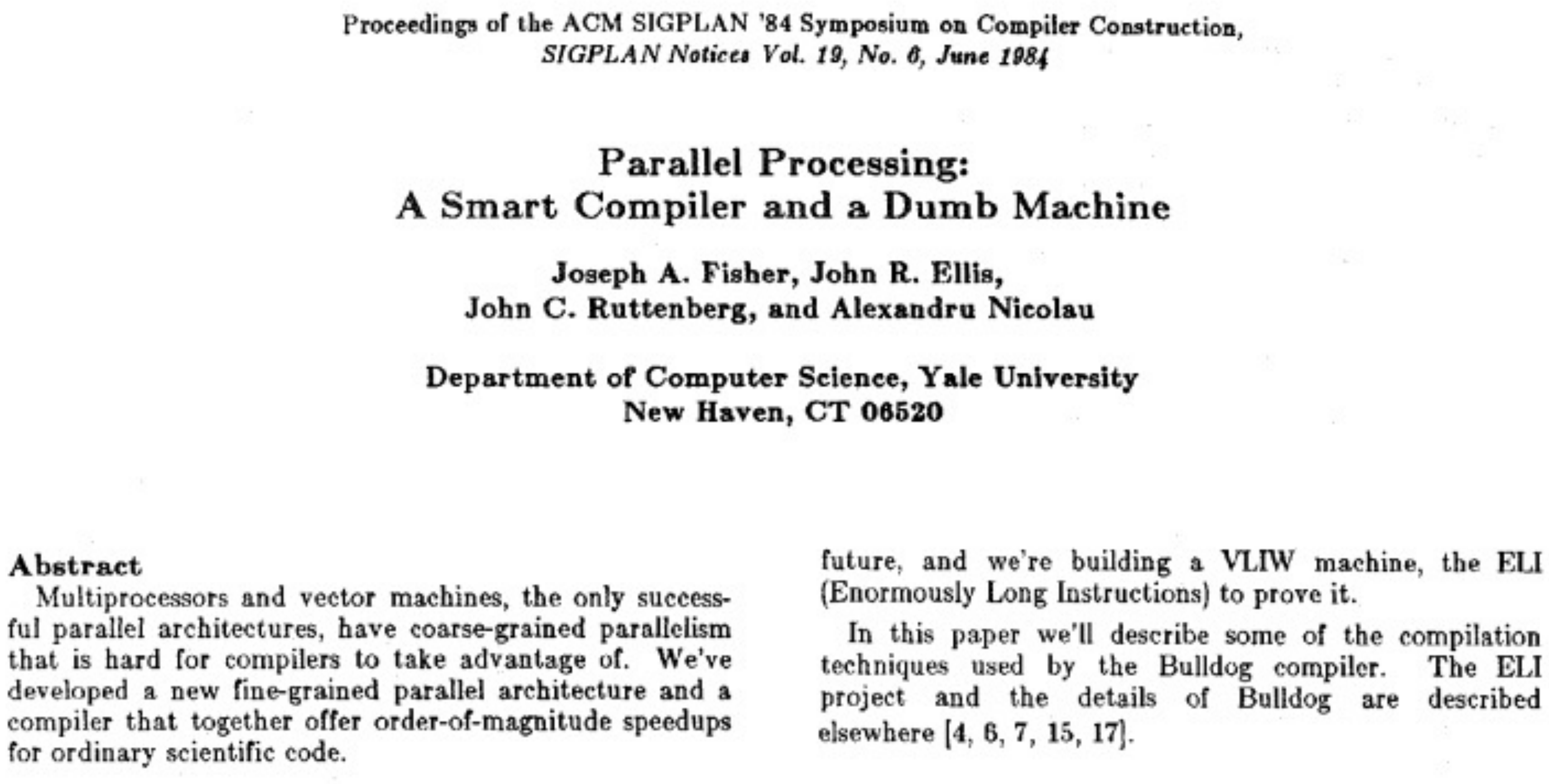


VLIW: Very Long Instruction Word


Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single-Cycle Latency
Two Load/Store Units, Three-Cycle Latency
Two Floating-Point Units, Four-Cycle Latency

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks

VLIW Principles



Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

**Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau**

**Department of Computer Science, Yale University
New Haven, CT 06520**

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

Early VLIW Machines

- FPS AP120B (1976)
  - scientific attached array processor
  - first commercial wide instruction machine
  - hand-coded vector math libraries using software pipelining and loop unrolling
- Multiflow Trace (1987)
  - commercialization of ideas from John Fisher's Yale group including "trace scheduling"
  - available in configurations with 7, 14, or 28 operations/instruction
  - 28 operations packed into a 1024-bit instruction word
- Cydrome Cydra-5 (1987)
  - 7 operations encoded in 256-bit instruction word
  - rotating register file
  - Founded by Bob Rau

VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

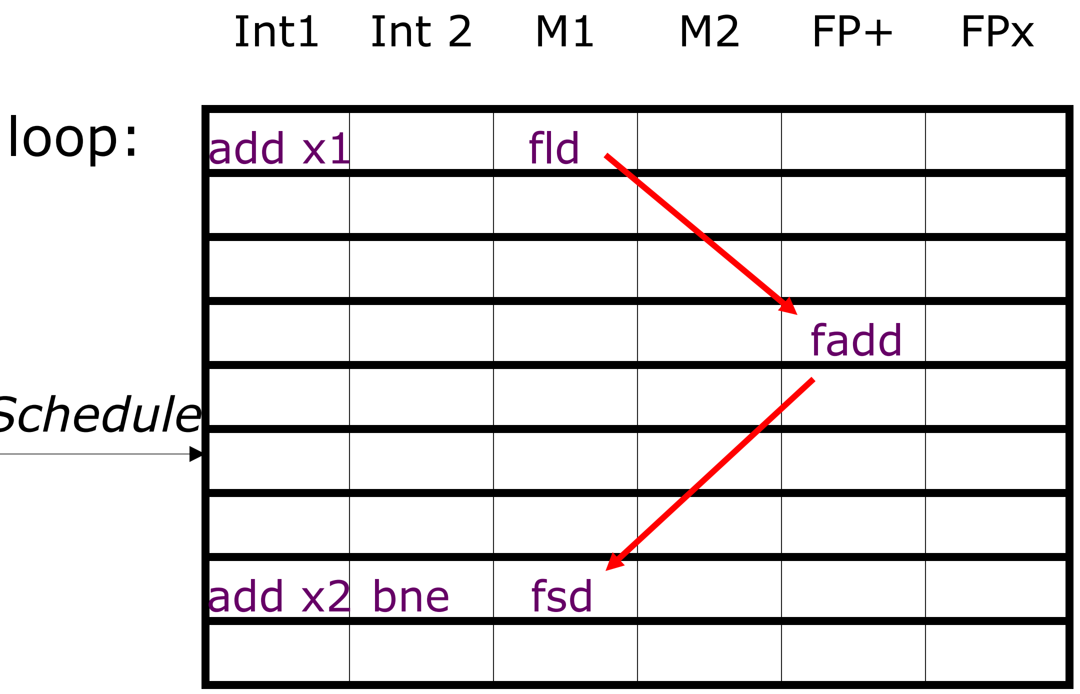
Loop Execution

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Compile

```
loop:   fld f1, 0(x1)
        add x1, 8
        fadd f2, f0, f1
        fsd f2, 0(x2)
        add x2, 8
        bne x1, x3,
loop
```

Schedule



How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

Loop Unrolling

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
      add x2, 32
       bne x1, x3, loop
```

Schedule $\longrightarrow$

	Int1	Int 2	M1	M2	FP+	FPx
loop:			fld f1			
			fld f2			
			fld f3			
	add x1		fld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			fsd f5			
			fsd f6			
			fsd f7			
	add x2	bne	fsd f8			

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

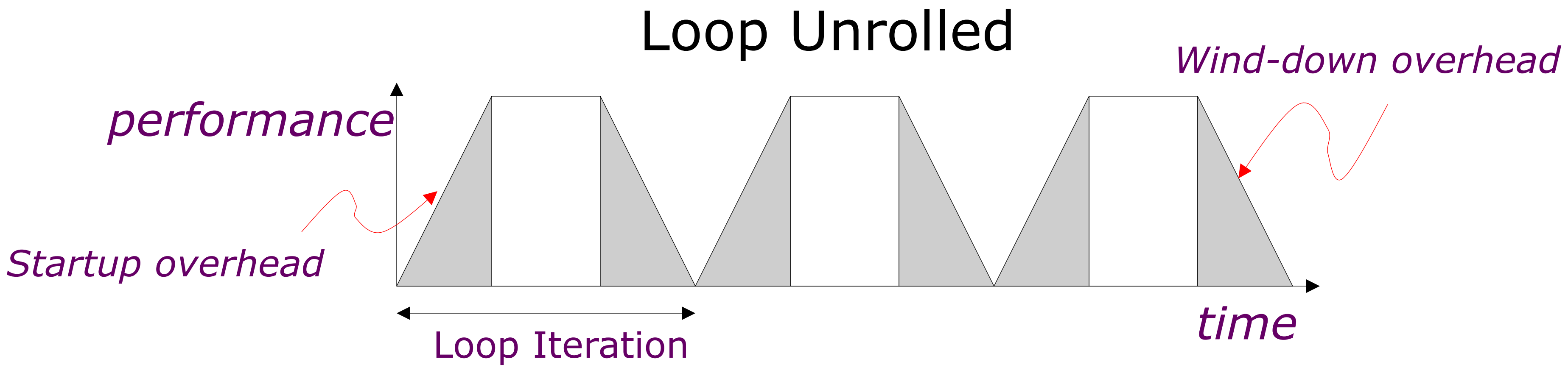
```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```

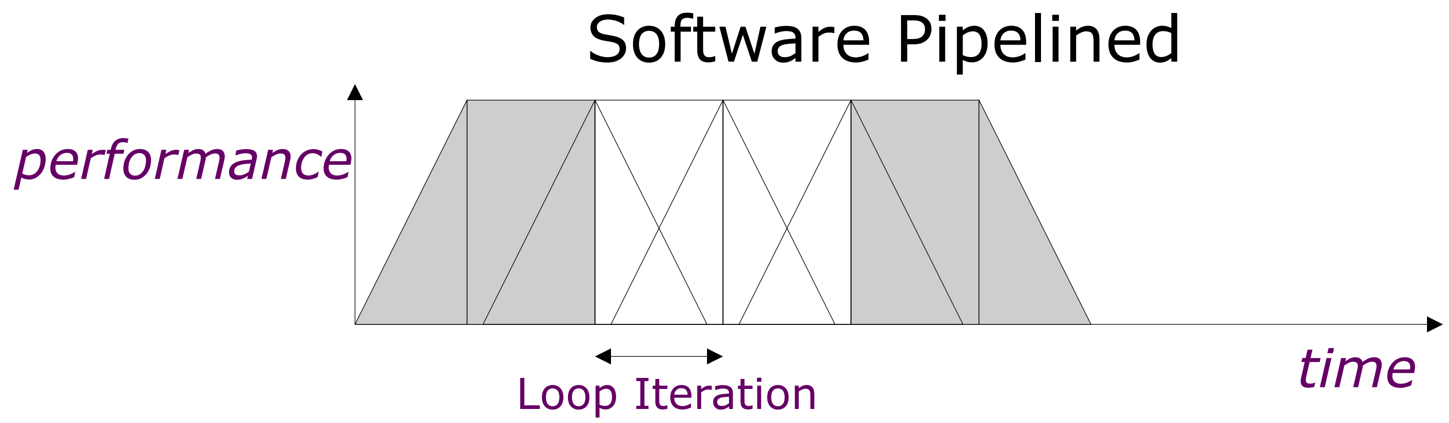
	Int1	Int 2	M1	M2	FP+	FPx
prolog			fld f1			
			fld f2			
			fld f3			
	add x1		fld f4			
			fld f1		fadd f5	
			fld f2		fadd f6	
			fld f3		fadd f7	
	add x1		fld f4		fadd f8	
iterate loop:			fld f1	fsd f5	fadd f5	
			fld f2	fsd f6	fadd f6	
		add x2	fld f3	fsd f7	fadd f7	
	add x1	bne	fld f4	fsd f8	fadd f8	
epilog				fsd f5	fadd f5	
				fsd f6	fadd f6	
		add x2		fsd f7	fadd f7	
		bne		fsd f8	fadd f8	
				fsd f5		

How many FLOPS/cycle?

4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling





Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

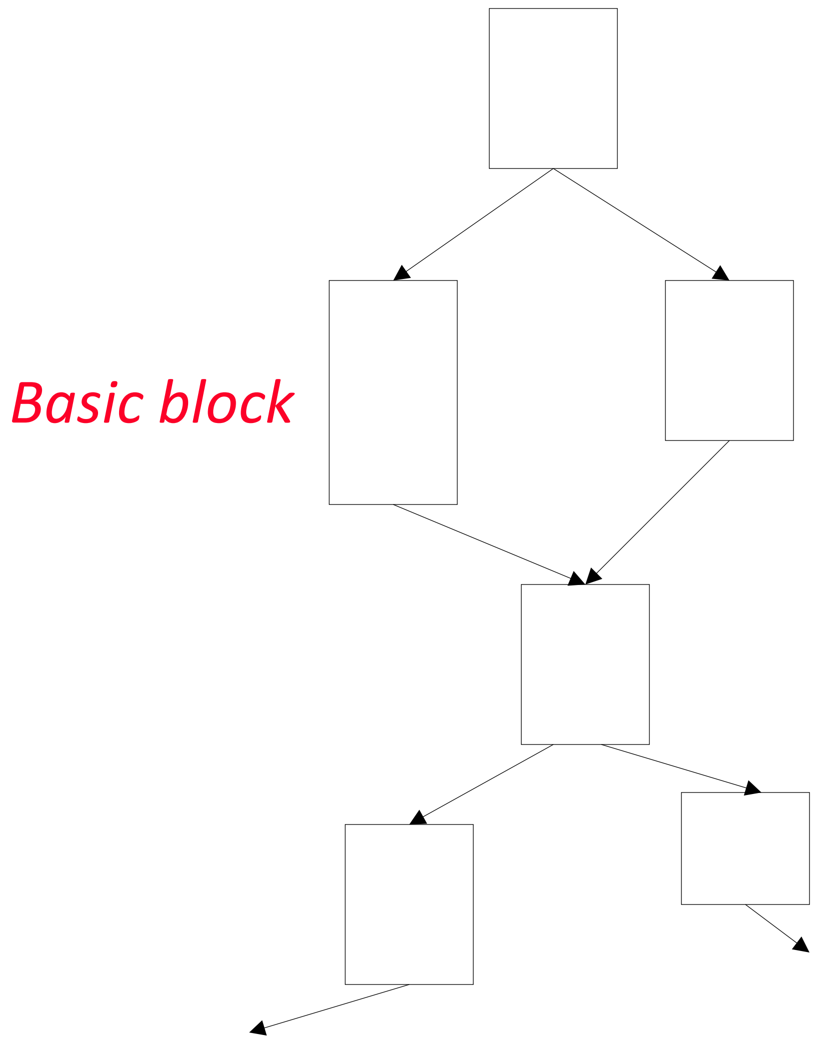
CS152 Administrivia

- Good job on Midterm!
  - Mid-semester survey: https://forms.gle/gD41gCUqf4HuSRKL8
- Lab2 due 3/2
- HW3 due 3/14
- Lab3 will be released this week
- Guest lecture on Branch Prediction next week
- Join our really fun discussions:
  - Prashanth, Wednesday 3-5
  - Abe/Edwin, Thursday 2-4
  - Allison/Divija, Thursday 5-7
  - Animesh/Jamie, Friday 12-2

CS252 Administrivia

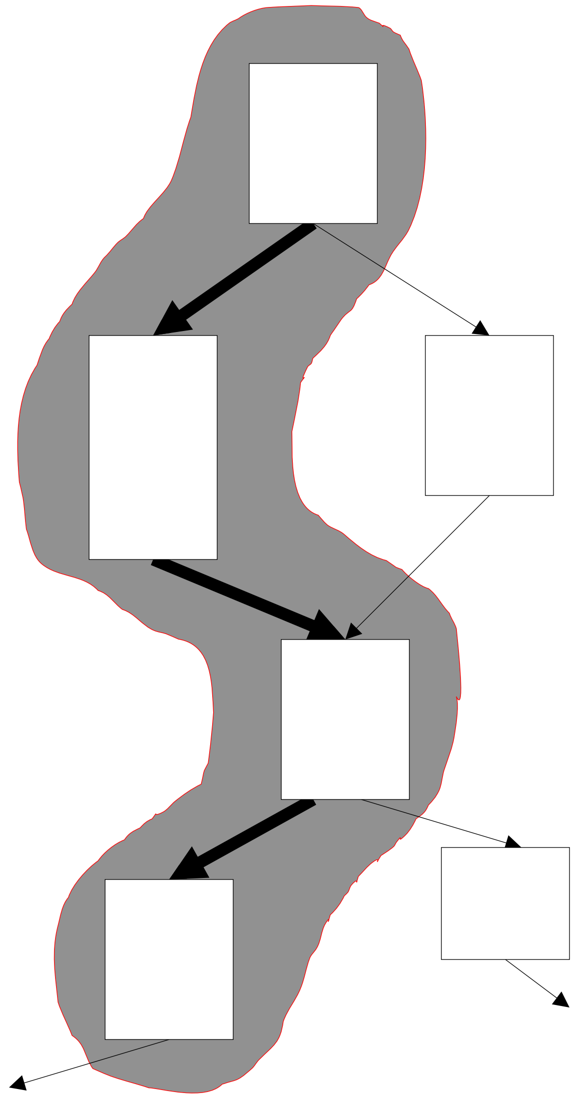
- Second half of project presentations next week

What if there are no loops?



- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

Trace Scheduling *[Fisher,Ellis]*

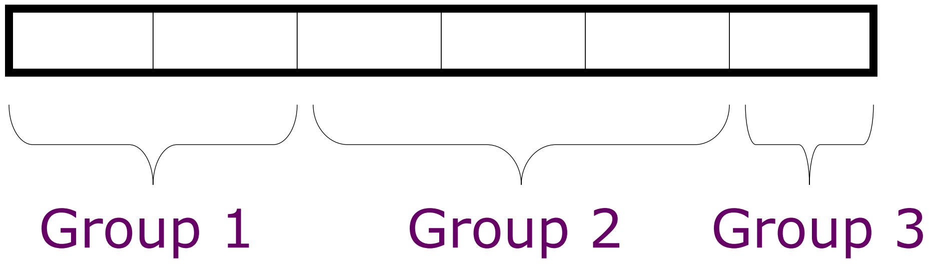


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

Problems with “Classic” VLIW

- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
- Knowing branch probabilities
  - Profiling requires a significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path

VLIW Instruction Encoding

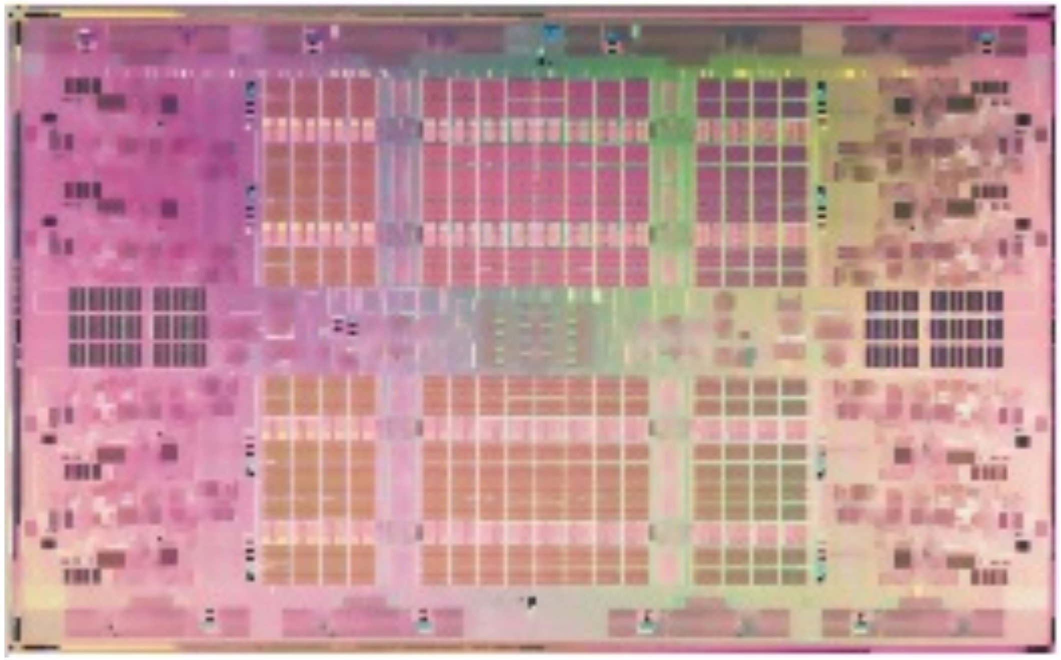


- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions

Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing (really just VLIW)
- IA-64 is Intel’s chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code-compatible VLIW
- Merced was first Itanium implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Recent version, Poulson, eight cores, 32nm, announced 2011

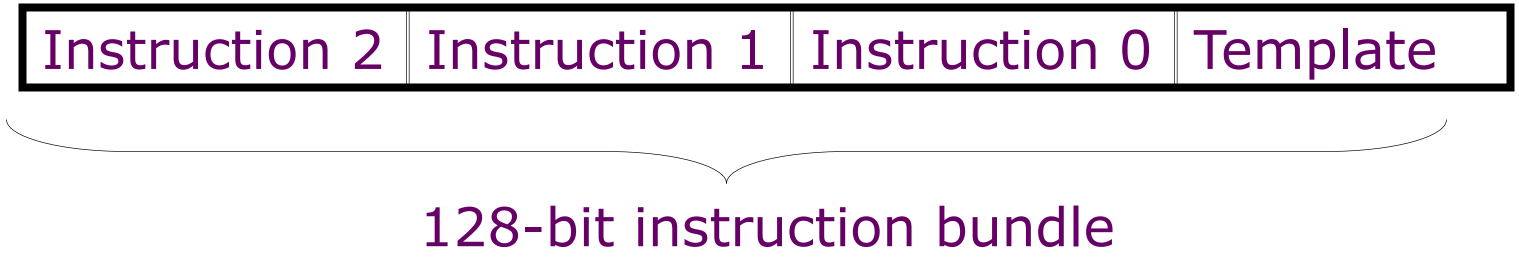
Eight Core Itanium "Poulson" *[Intel 2011]*



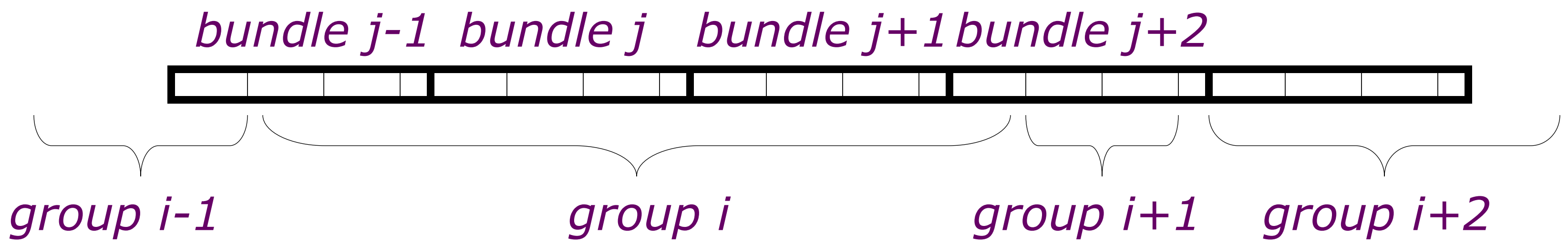
- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm^2 in 32nm CMOS
- Over 3 billion transistors

- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
  - Two 128-bit bundles
- Up to 12 insts/cycle execute

IA-64 Instruction Format



- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel



IA-64 Registers

- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

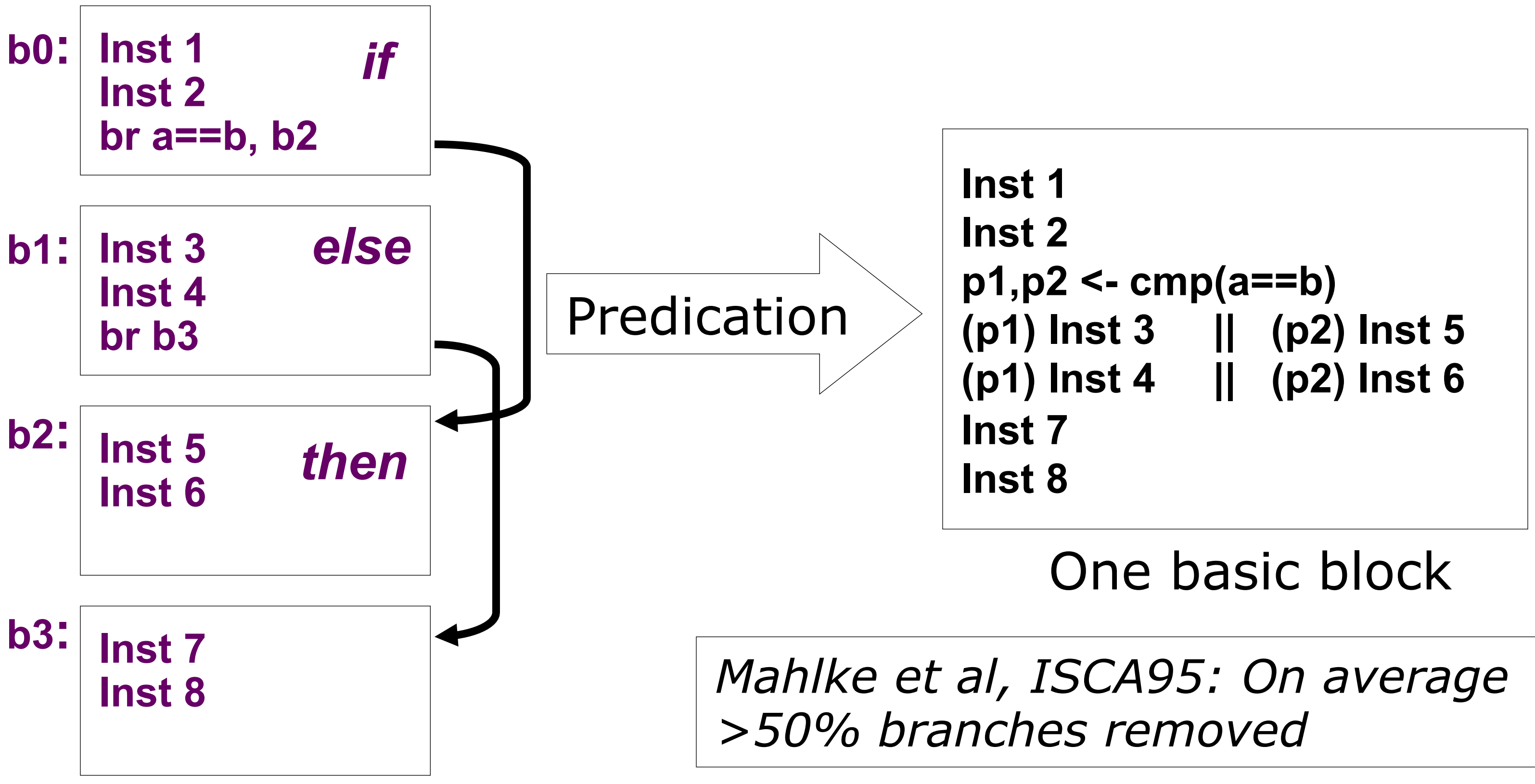
- GPRs “rotate” to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration

IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



One basic block

Mahlke et al, ISCA95: On average >50% branches removed

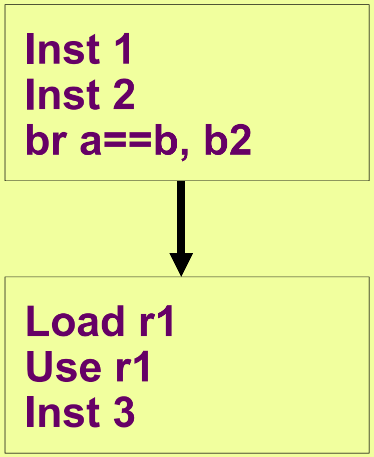
Four basic blocks

Warning: Complicates bypassing!

IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

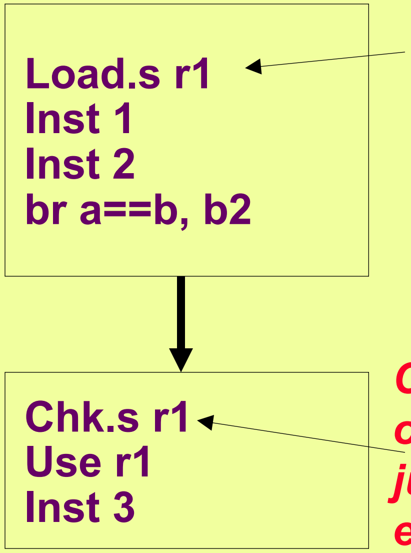
Solution: Speculative operations that don't cause exceptions



```
Inst 1
Inst 2
br a==b, b2
```

```
Load r1
Use r1
Inst 3
```

Can't move load above branch because might cause spurious exception



```
Load.s r1
Inst 1
Inst 2
br a==b, b2
```

```
Chk.s r1
Use r1
Inst 3
```

Speculative load never causes exception, but sets "poison" bit on destination register

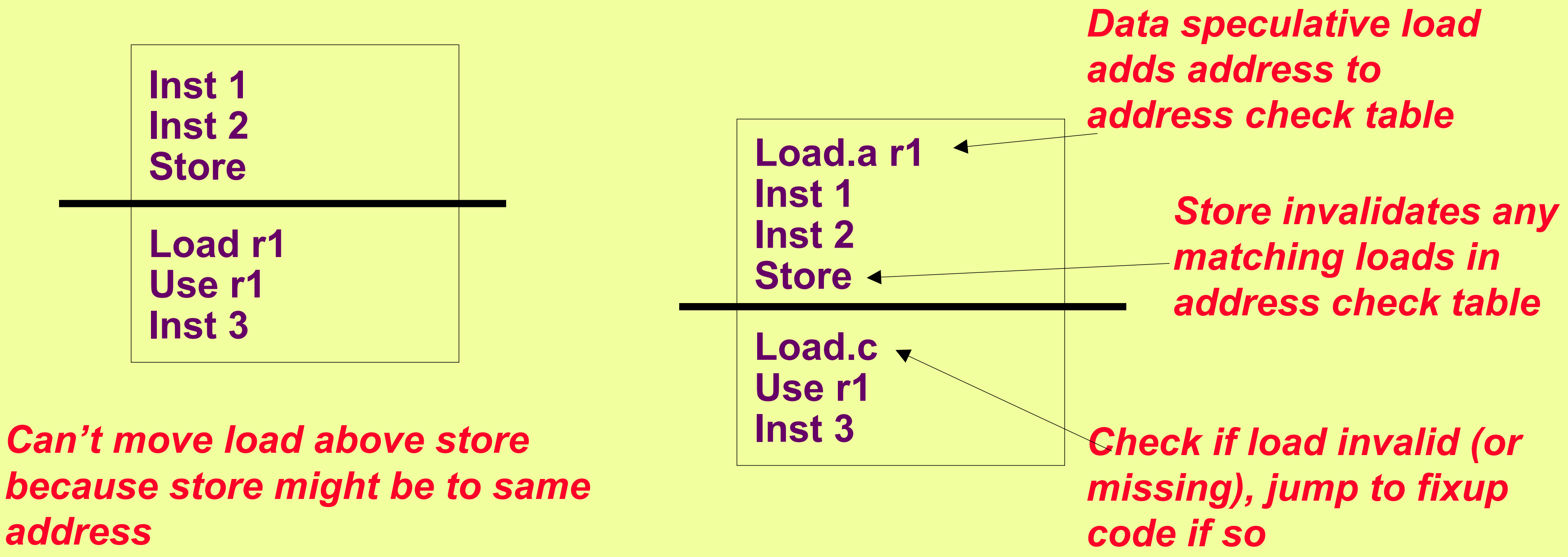
Check for exception in original home block jumps to fixup code if exception detected

Particularly useful for scheduling long latency loads early

IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards



```
Inst 1
Inst 2
Store
--------
Load r1
Use r1
Inst 3
```

Can't move load above store because store might be to same address

```
Load.a r1
Inst 1
Inst 2
Store
--------
Load.c
Use r1
Inst 3
```

Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so

Requires associative hardware in address check table

Limits of Static Scheduling

- Statically unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena.
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.
  - Transmeta: dynamically translate x86 -> VLIW (1st CEO: Dave Ditzel)
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.
  - Google's Tensor Processing Unit is a VLIW machine.

Intel Kills Itanium

- Donald Knuth " … *Itanium approach that was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write*."

- "*Intel officially announced the end of life and product discontinuance of the Itanium CPU family on January 30th, 2019*", Wikipedia

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)