CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 13 –Advanced Out-of-Order Superscalars and Introduction to VLIW

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

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

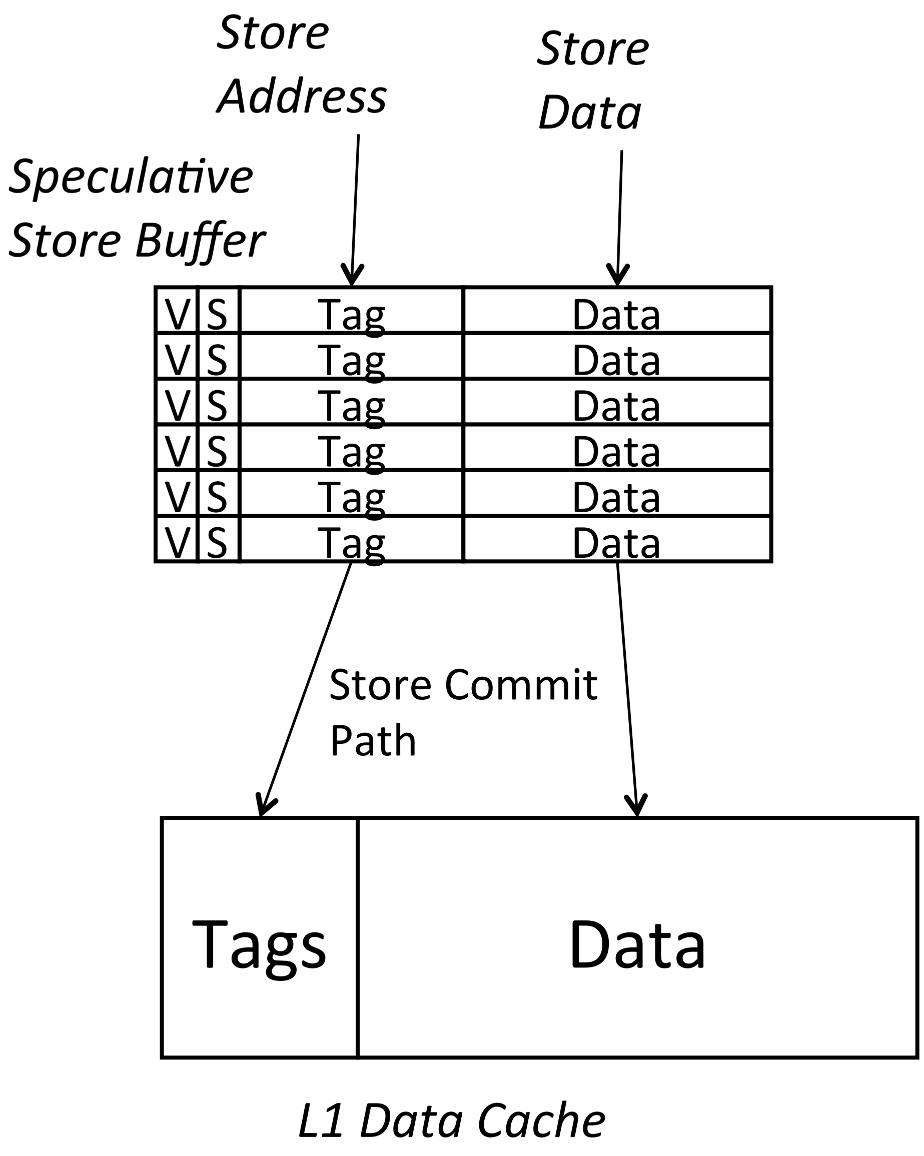
Last Time in Lecture 12

- Branch prediction
  - temporal, history of a single branch
  - spatial, based on path through multiple branches
- Branch History Table (BHT) vs. Branch History Buffer (BTB)
  - tradeoff in capacity versus latency
- Return-Address Stack (RAS)
  - specialized structure to predict subroutine return addresses
- Fetching more than one basic block per cycle
  - predicting multiple branches
  - trace cache

Load-Store Queue Design

- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

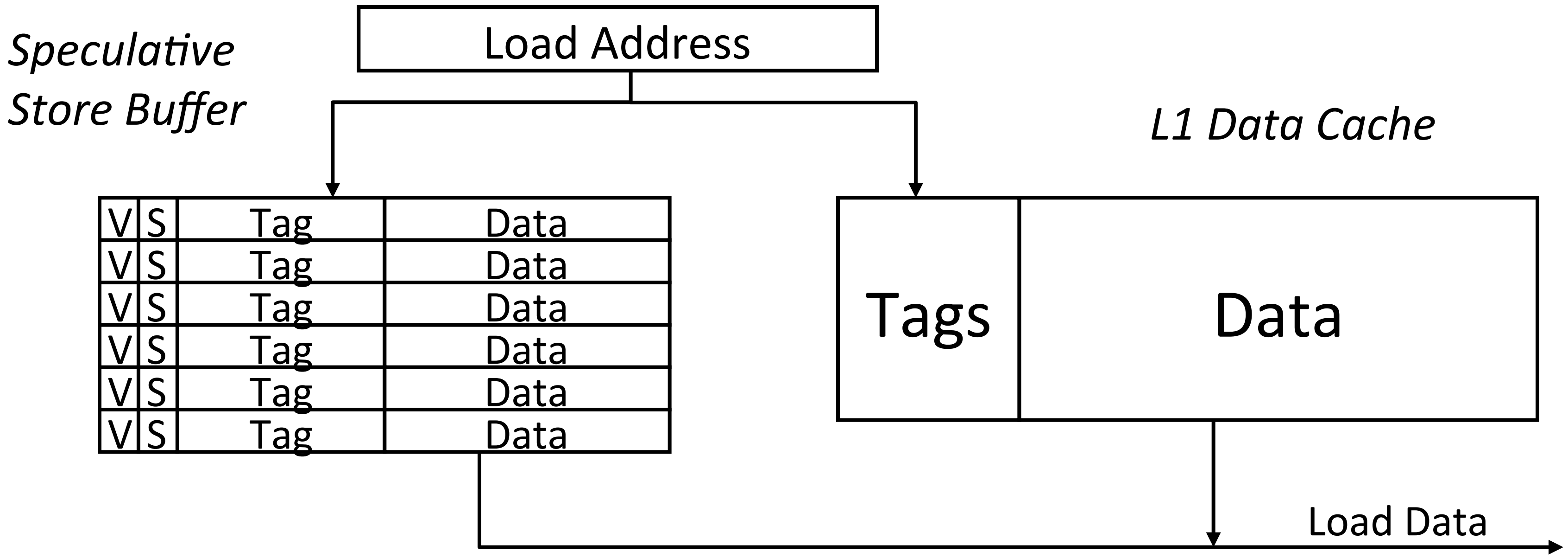
Speculative Store Buffer



V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into “store address” and “store data” micro-operations
- “Store address” execution writes tag
- “Store data” execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from speculative store buffer



Speculative Store Buffer

Load Address

L1 Data Cache

V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

Tags

Data

Load Data

- If data in both store buffer and cache, which should we use?

  Speculative store buffer

- If same address in store buffer twice, which should we use?

  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

- When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and **`x4`** != **`x2`**
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don't execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **`x4`** != **`x2`**
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find **`x4==x2`**, squash load and all following instructions

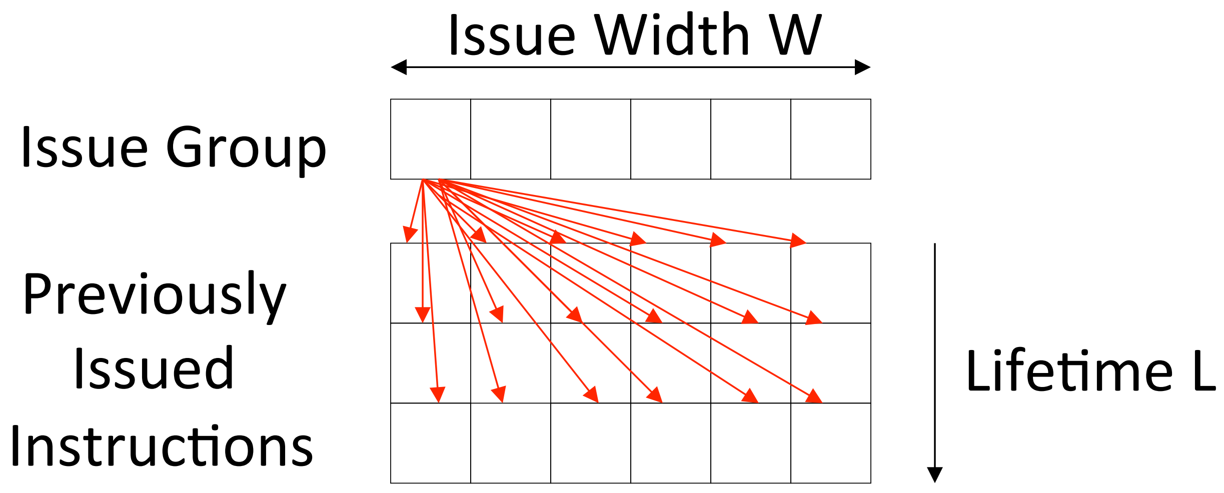
- => Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **`x4`** != **`x2`** and execute load before store
- If later find **`x4==x2`**, squash load and all following instructions, but mark load instruction as store-wait
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear store-wait bits

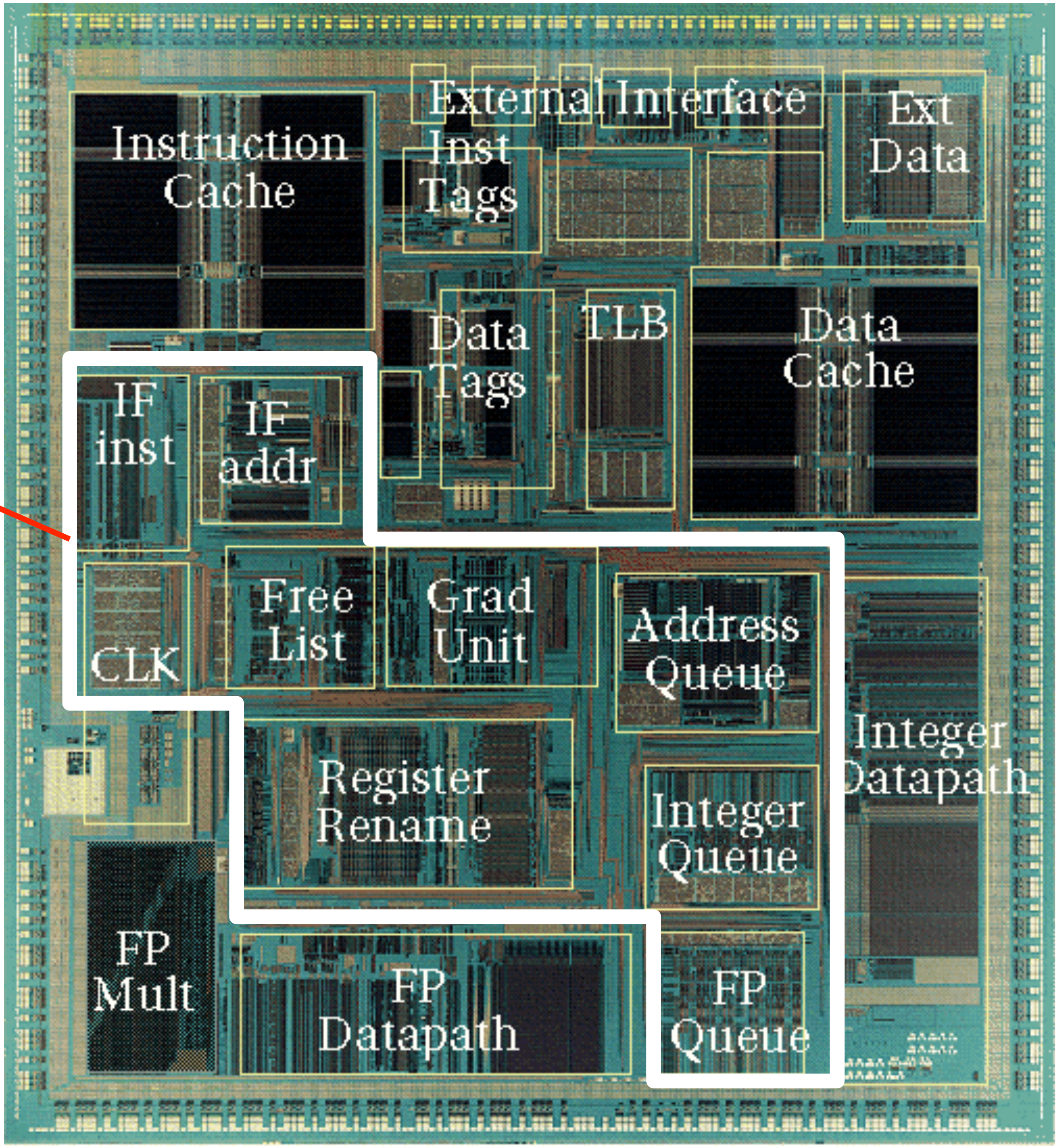
Superscalar Control Logic Scaling



- Each issued instruction must somehow check against W*L instructions, i.e., growth in hardware ∝ W*(W*L)
- For in-order machines, L is related to pipeline latencies and check is done during issue (interlocks or scoreboard)
- For out-of-order machines, L also includes time spent in instruction buffers (instruction window or ROB), and check is done by broadcasting tags to waiting instructions at write back (completion)
- As W increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater L

=> Out-of-order control logic grows faster than W^2 (~W^3)

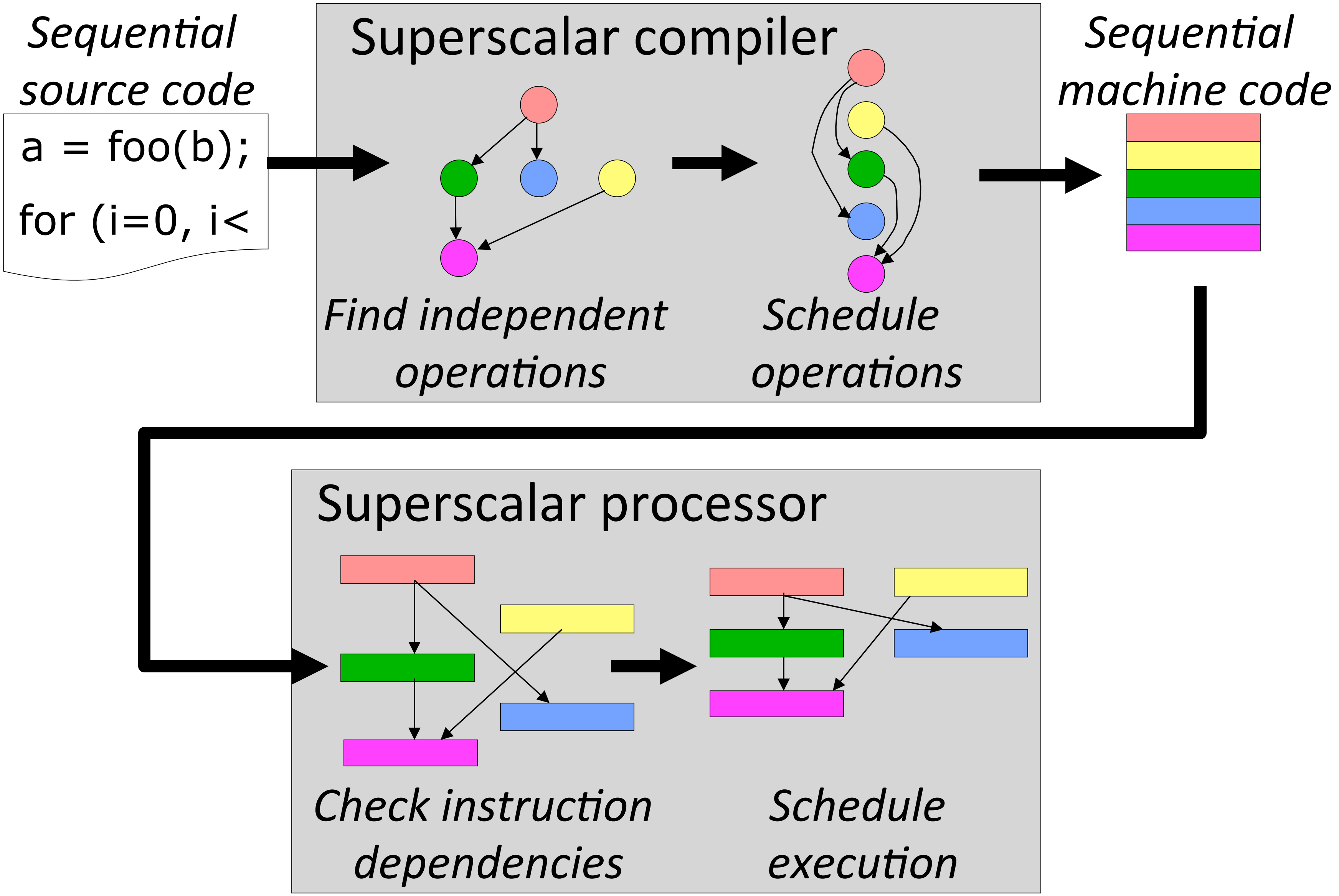
Out-of-Order Control Complexity:



Control Logic

[SGI/MIPS Technologies Inc., 1995]

Sequential ISA Bottleneck



Sequential source code

```
a = foo(b);

for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

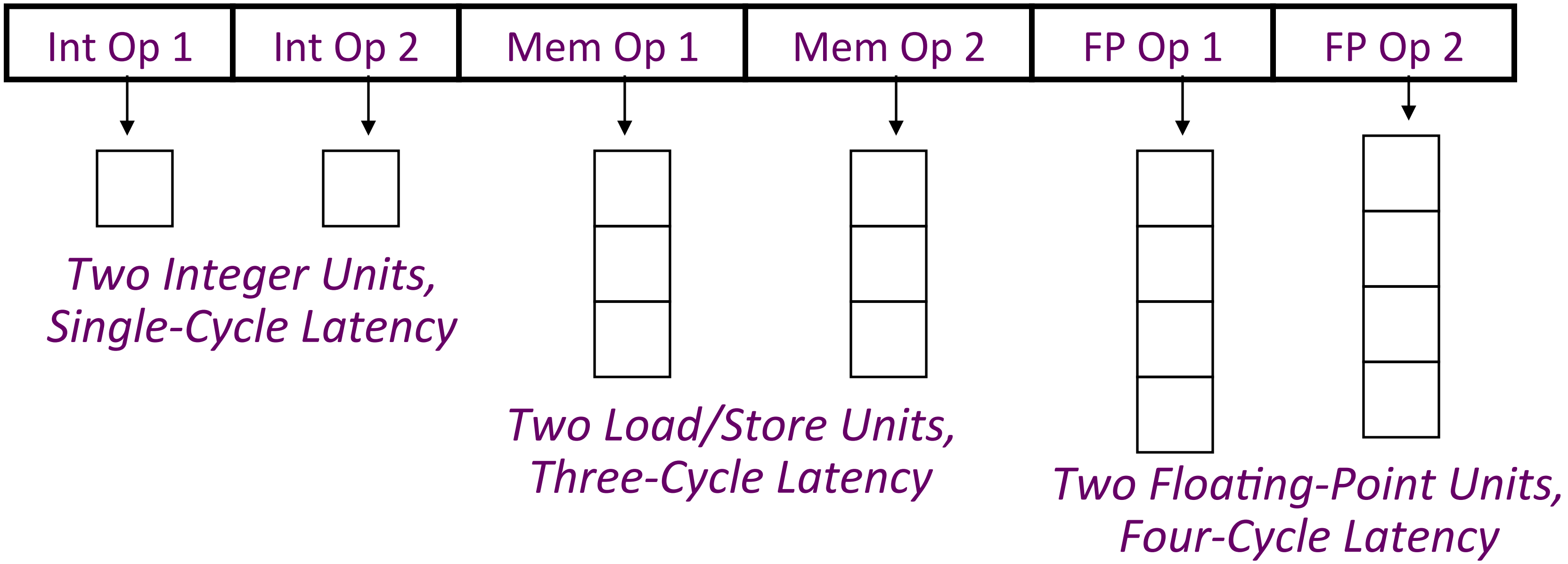
CS152 Administrivia

- PS 3 due Monday March 12
- Lab 3 due Monday March 19
- Submit Midterm-1 regrade requests by Friday
  - score might go up or down with regrade requests

CS252 Administrivia

- Readings for Monday:
  - Tomasulo algorithm
  - Decoupled access/execute
  - MIPS R10K

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

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 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

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

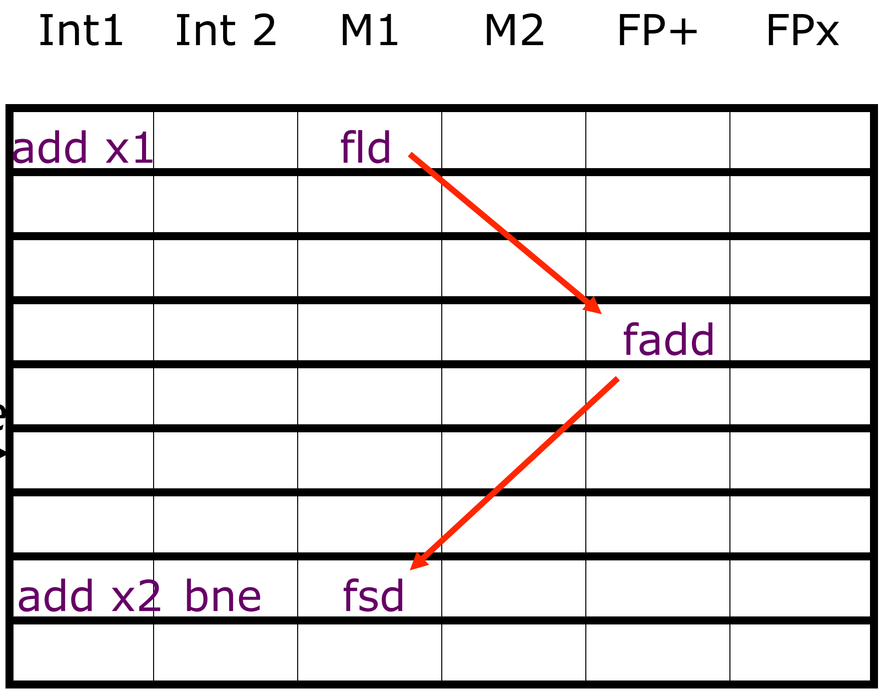
Loop Execution

```
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



	Int1	Int 2	M1	M2	FP+	FPx
loop:	add x1		fld			
					fadd	
	add x2	bne	fsd			

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;
}
```

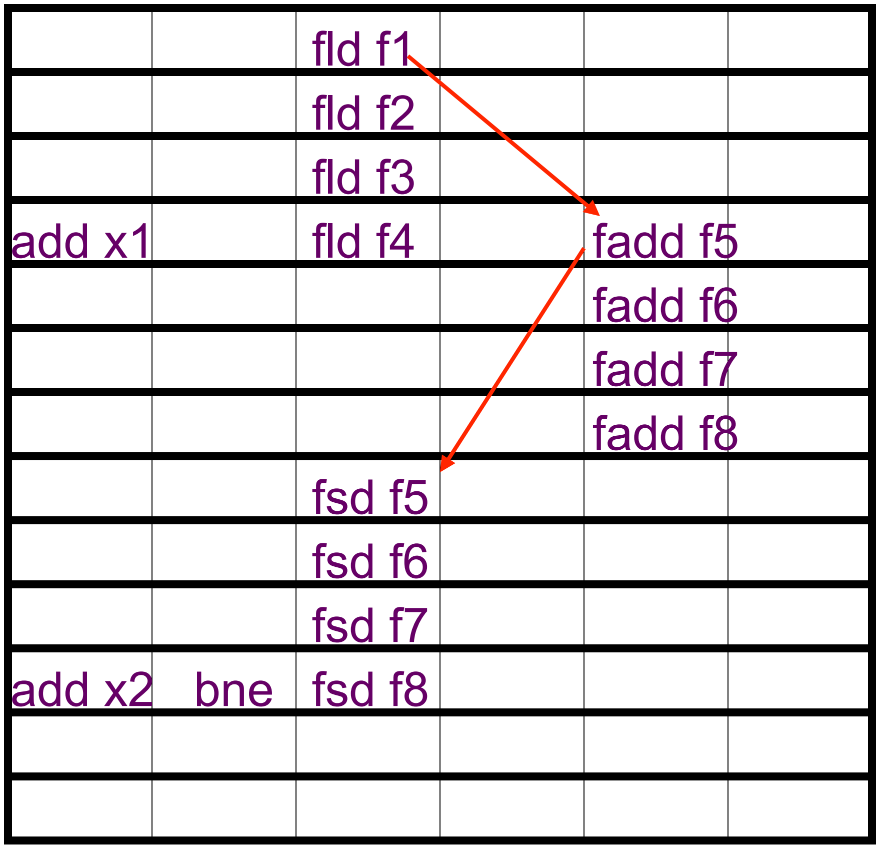
Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

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



	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
```

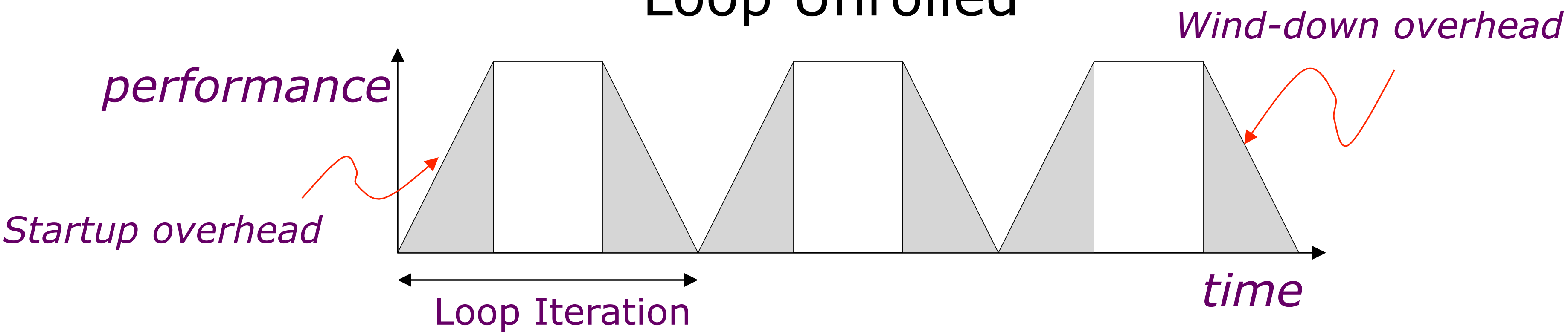
		Int1	Int 2	M1	M2	FP+	FPx
				fld f1			
				fld f2			
				fld f3			
		add x1		fld f4			
prolog				fld f1		fadd f5	
				fld f2		fadd f6	
				fld f3		fadd f7	
		add x1		fld f4		fadd f8	
	loop:			fld f1	fsd f5	fadd f5	
iterate				fld f2	fsd f6	fadd f6	
			add x2	fld f3	fsd f7	fadd f7	
		add x1	bne	fld f4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

How many FLOPS/cycle?

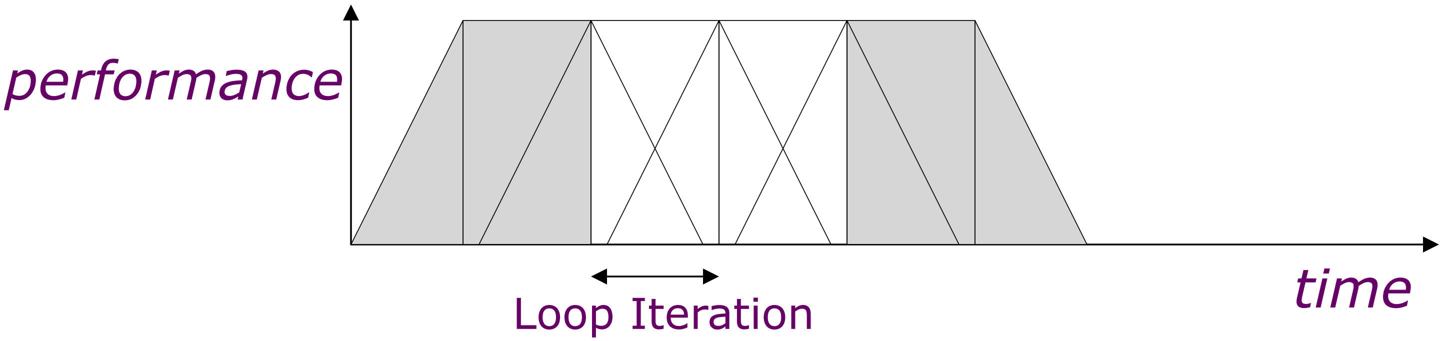
4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling

Loop Unrolled

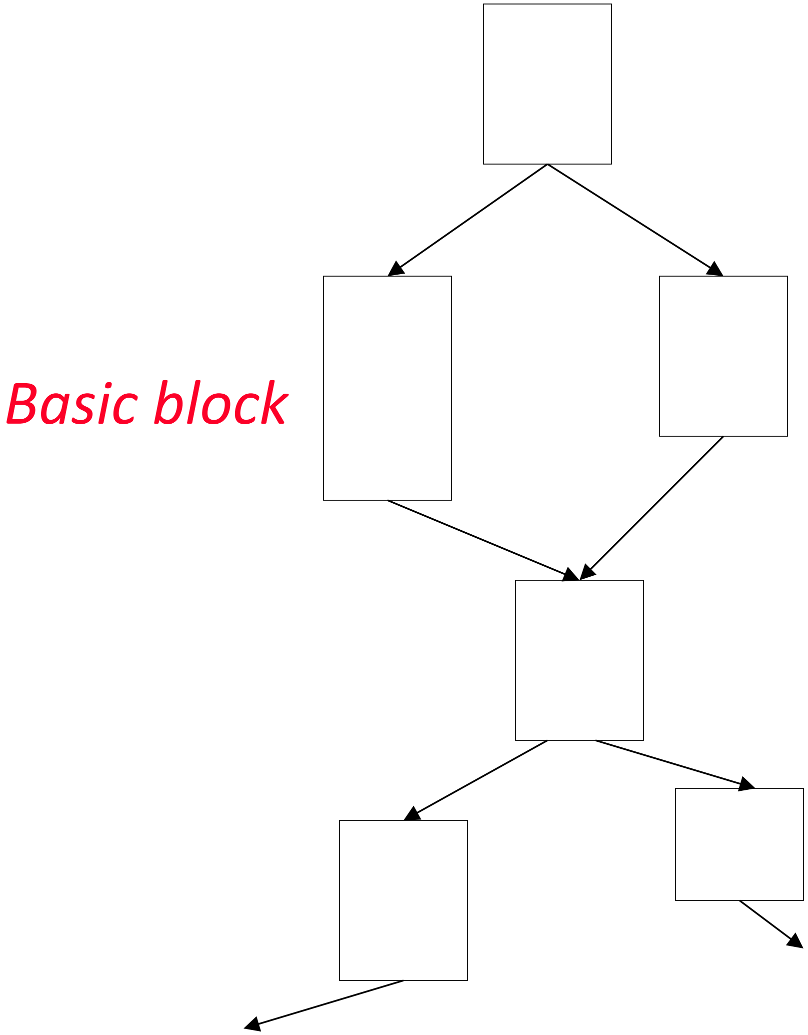


Software Pipelined



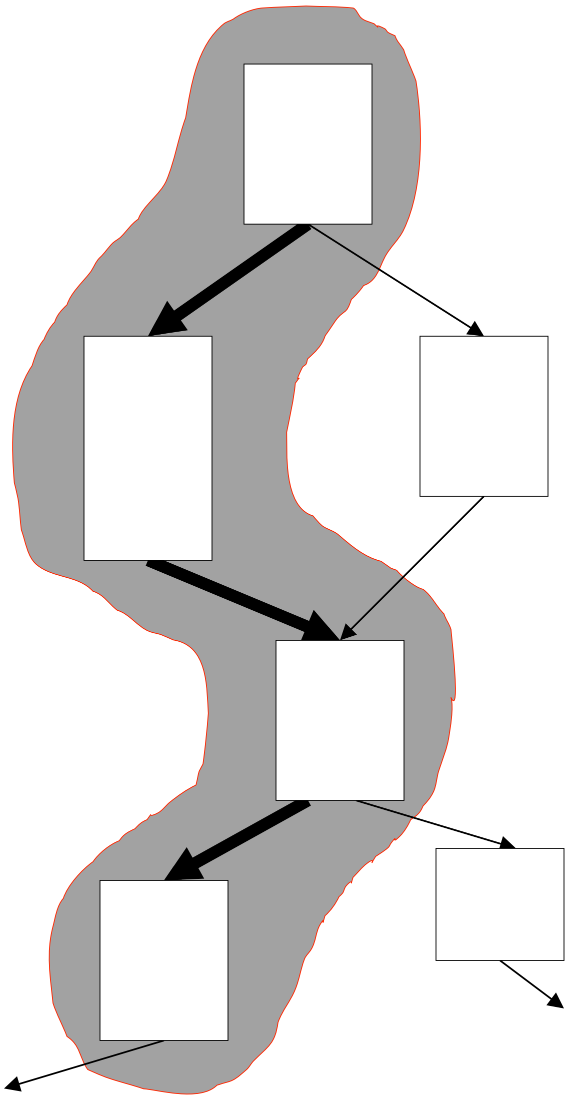
Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

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]*

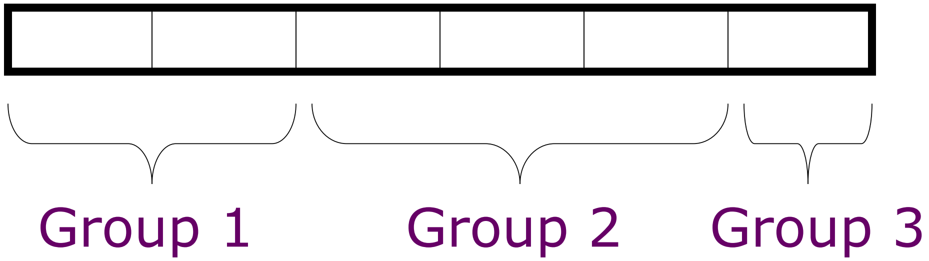


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

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 an significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path

VLIW Instruction Encoding

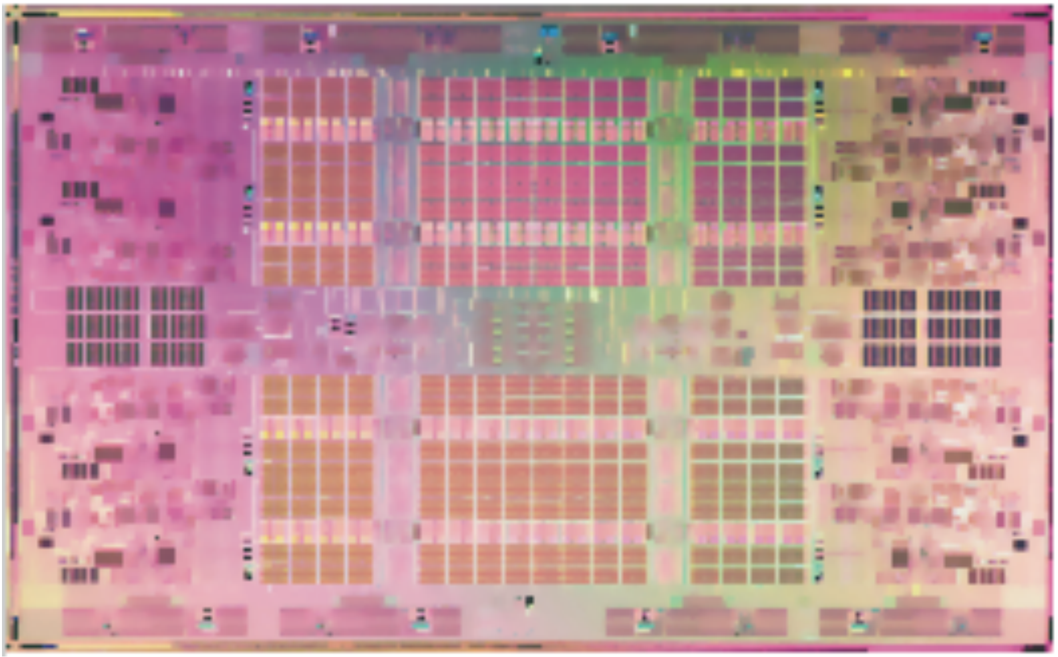


- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5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

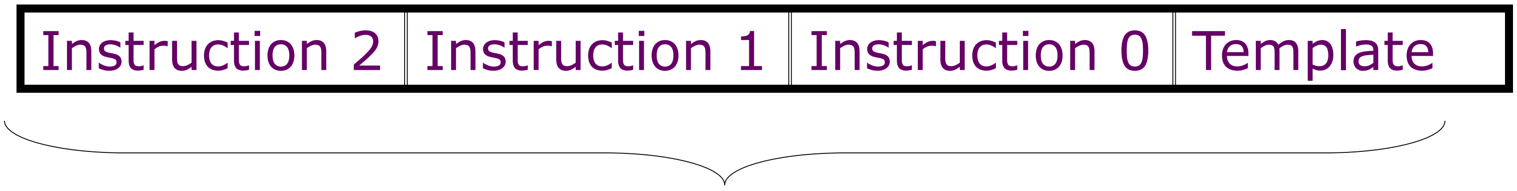
Eight Core Itanium "Poulson" *[Intel 2011]*



- 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

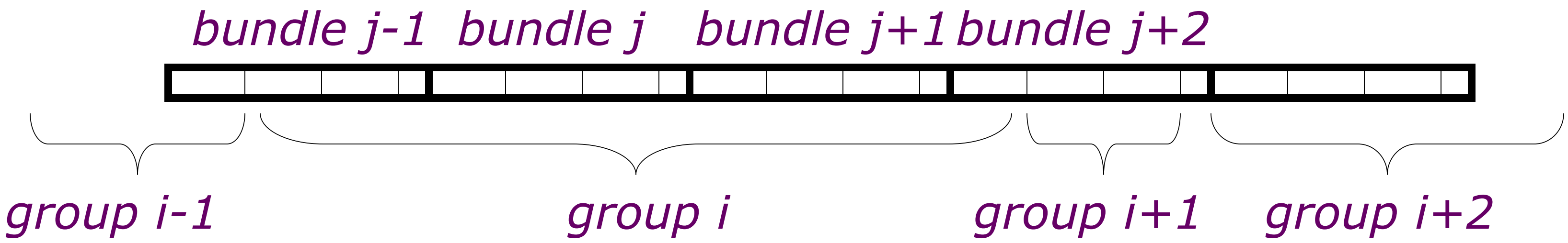
IA-64 Instruction Format



Instruction 2	Instruction 1	Instruction 0	Template

128-bit instruction bundle

- 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

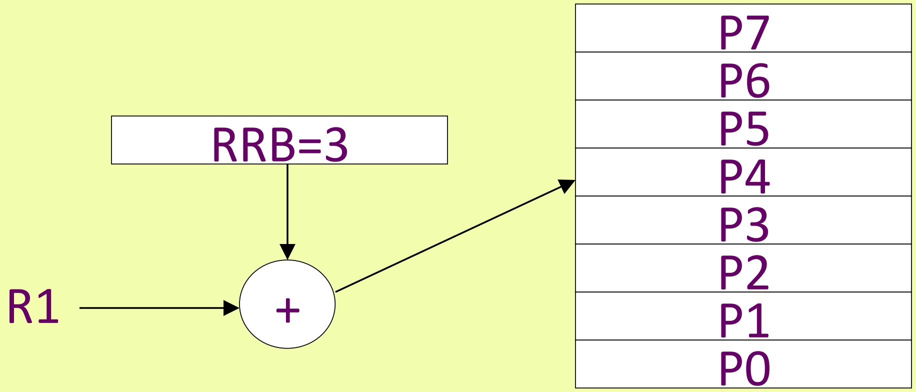
- GPRs “rotate” to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration

Rotating Register Files

Problems: Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog

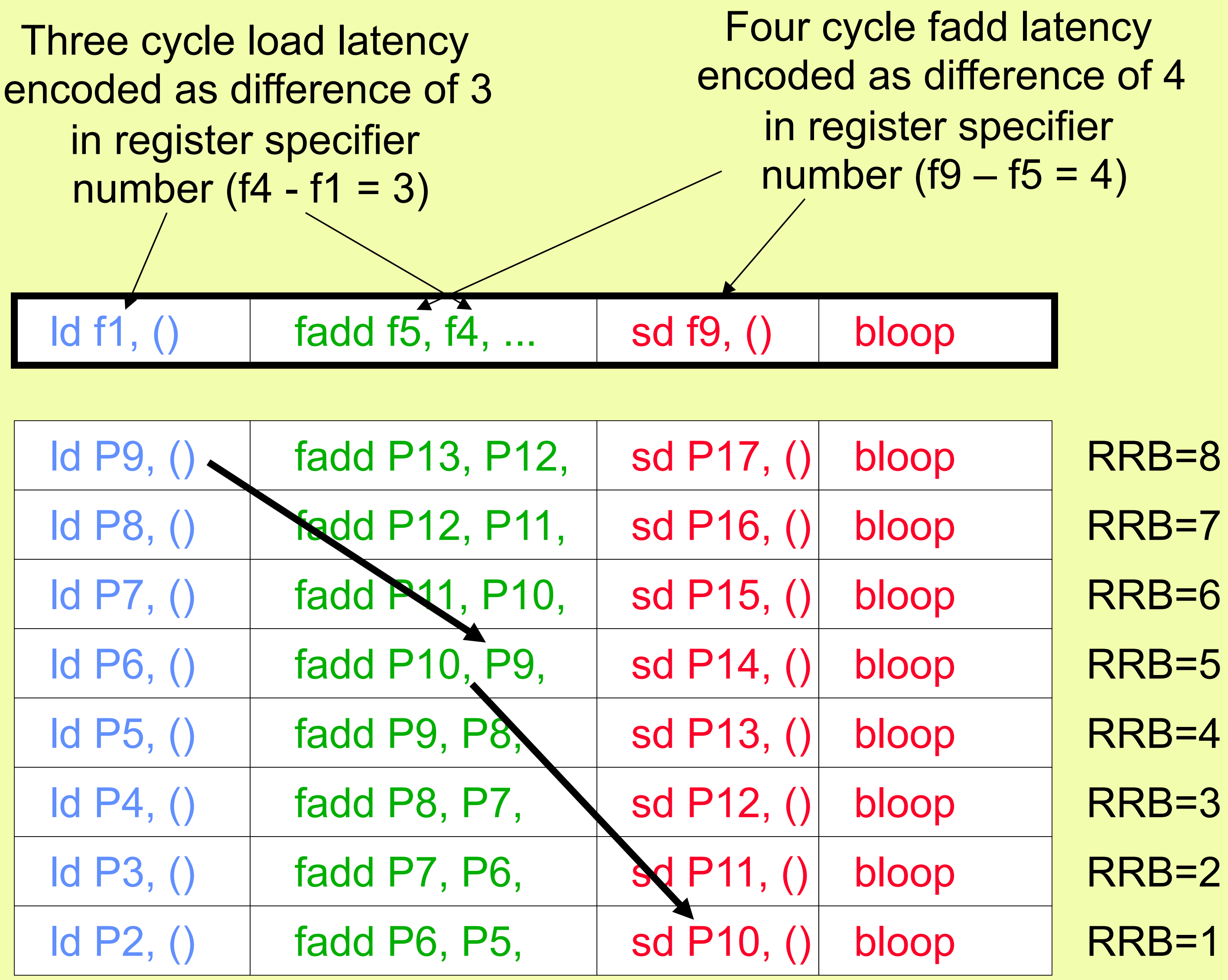
Solution: Allocate new set of registers for each loop iteration

Rotating Register File



Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

Rotating Register File
(Previous Loop Example)



Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop

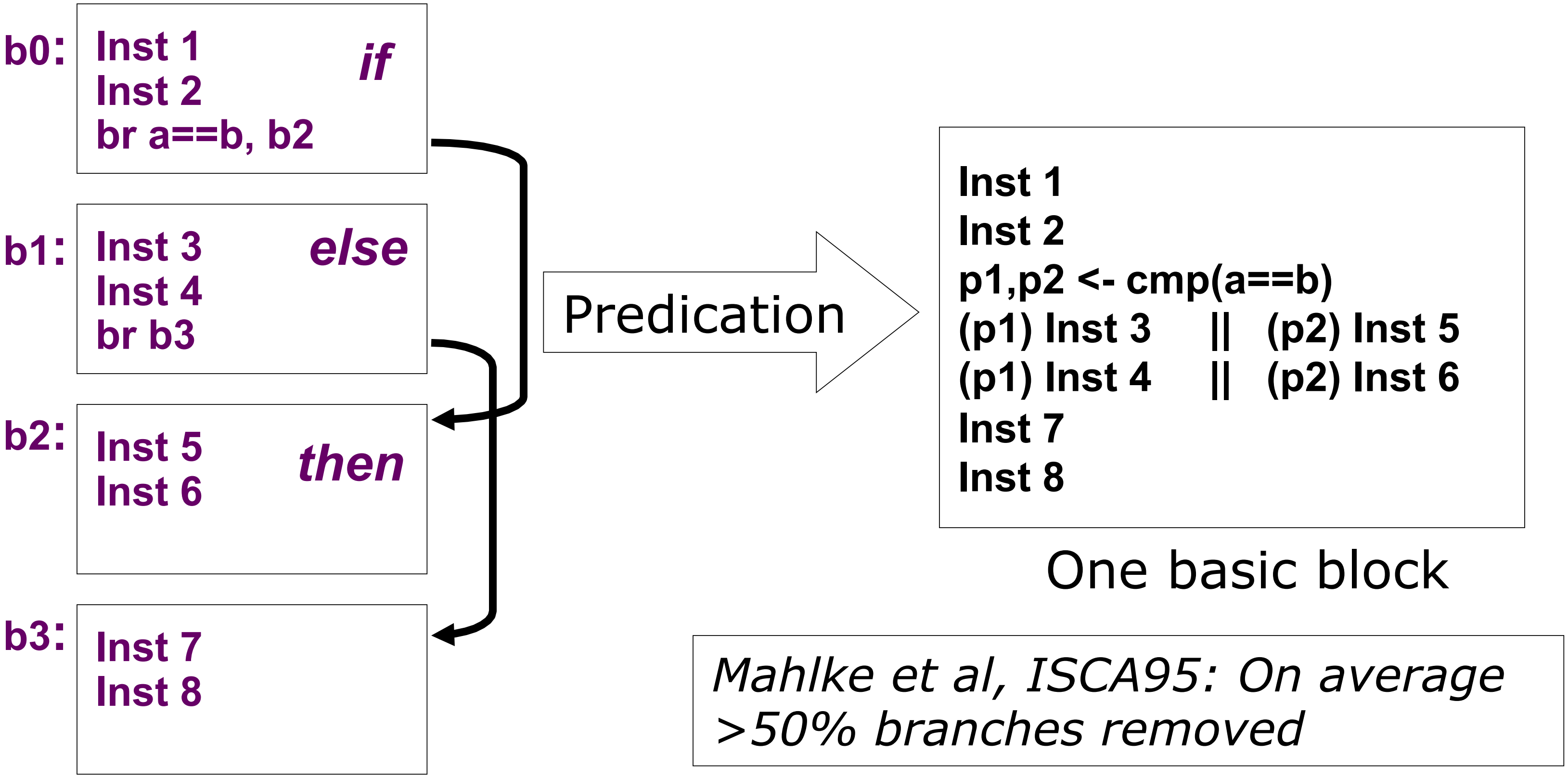
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop	RRB=8
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop	RRB=7
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop	RRB=6
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop	RRB=5
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop	RRB=4
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop	RRB=3
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop	RRB=2
ld P2, ()	fadd P6, P5,	sd P10, ()	bloop	RRB=1

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



```
b0: Inst 1              if
    Inst 2
    br a==b, b2

b1: Inst 3              else
    Inst 4
    br b3

b2: Inst 5              then
    Inst 6

b3: Inst 7
    Inst 8
```

Four basic blocks

```
Inst 1
Inst 2
p1,p2 <- cmp(a==b)
(p1) Inst 3    ||  (p2) Inst 5
(p1) Inst 4    ||  (p2) Inst 6
Inst 7
Inst 8
```

One basic block

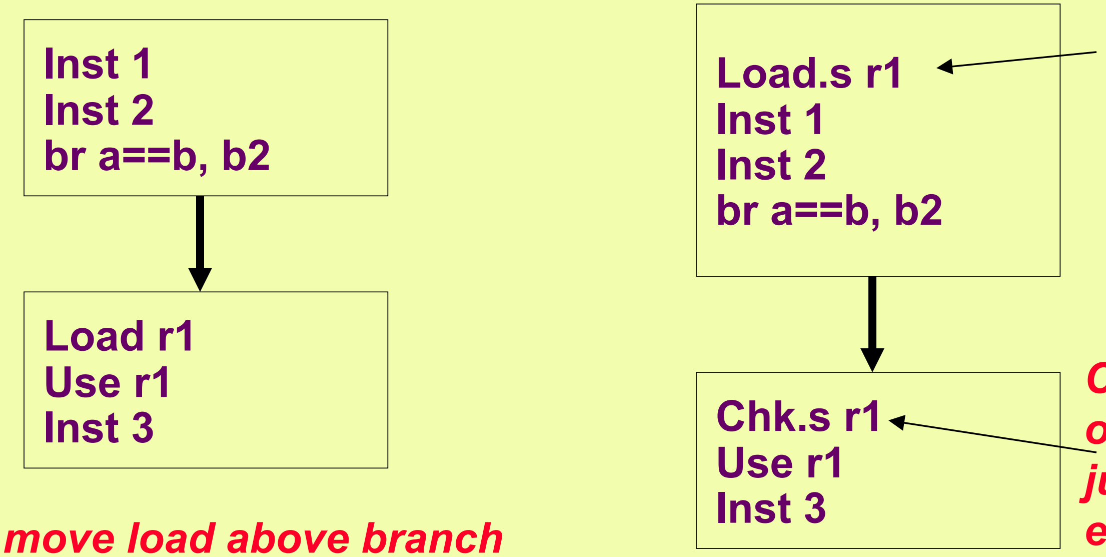
Mahlke et al, ISCA95: On average >50% branches removed

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

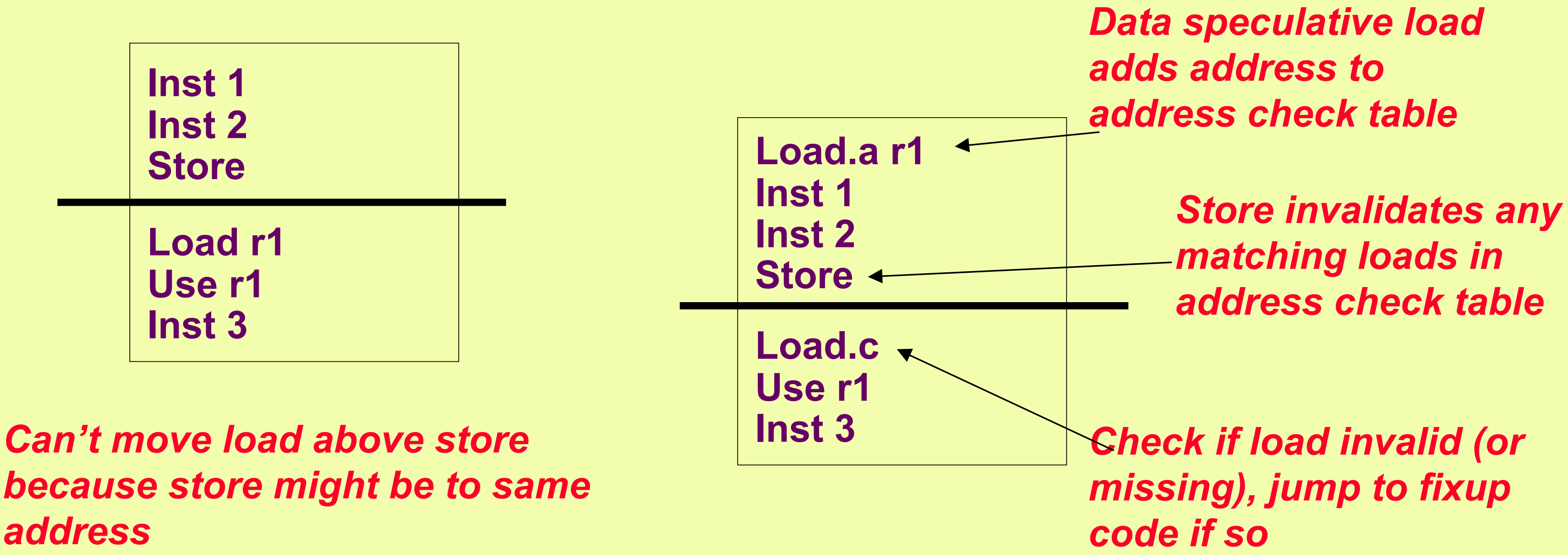
Check for exception in original home block jumps to fixup code if exception detected

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

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena (so far).
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.

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)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)