CS 152 Computer Architecture and Engineering

Lecture 13 - VLIW Machines and Statically Scheduled ILP

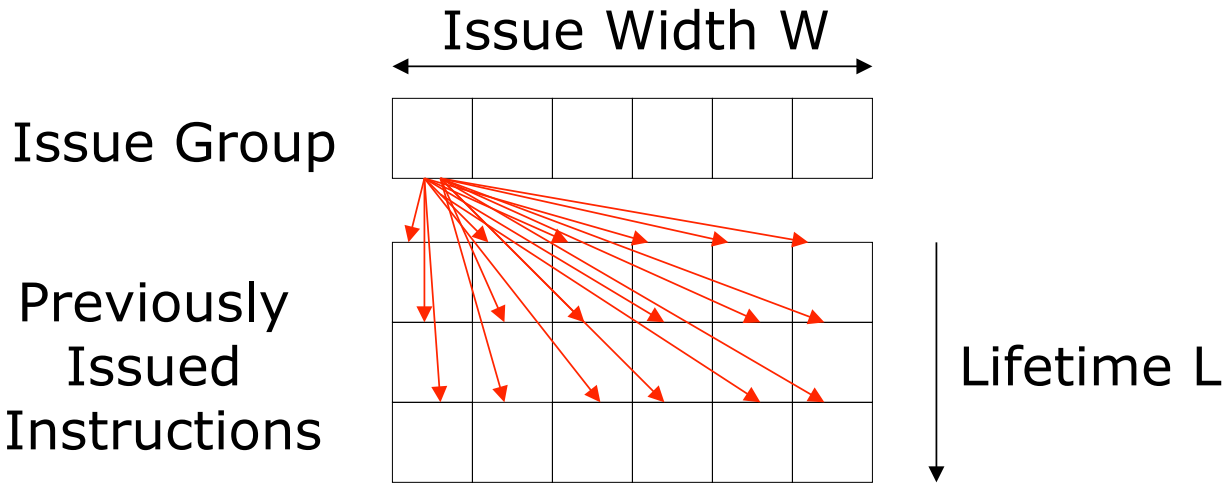
Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

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

Last time in Lecture 12

- Unified physical register file machines remove data values from ROB
  - All values only read and written during execution
  - Only register tags held in ROB
  - Allocate resources (ROB slot, destination physical register, memory reorder queue location) during decode
  - Issue window can be separated from ROB and made smaller than ROB (allocate in decode, free after instruction completes)
  - Free resources on commit
- Speculative store buffer holds store values before commit to allow load-store forwarding
- Can execute later loads past earlier stores when addresses known, or predicted no dependence

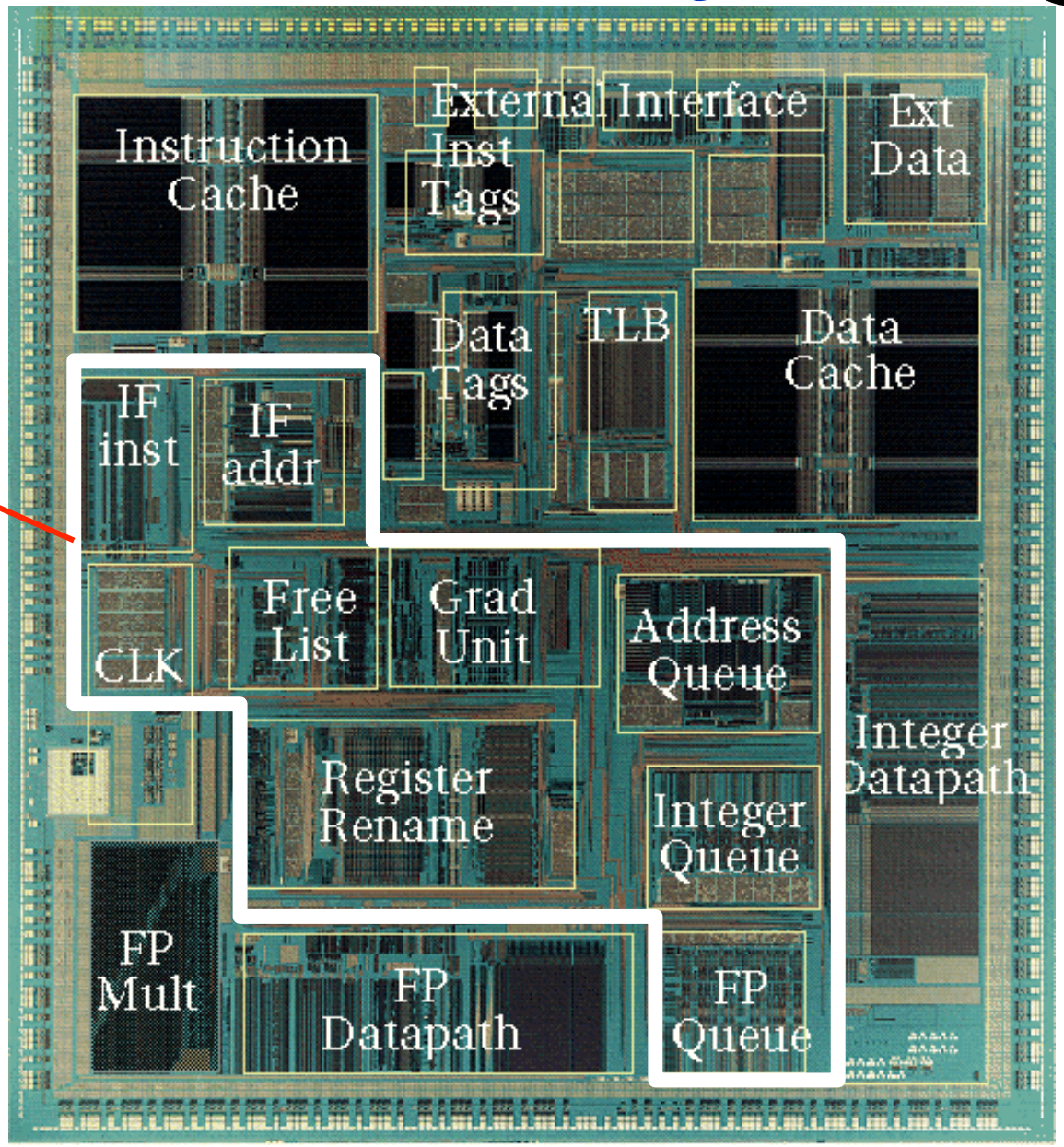
Superscalar Control Logic Scaling



- 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

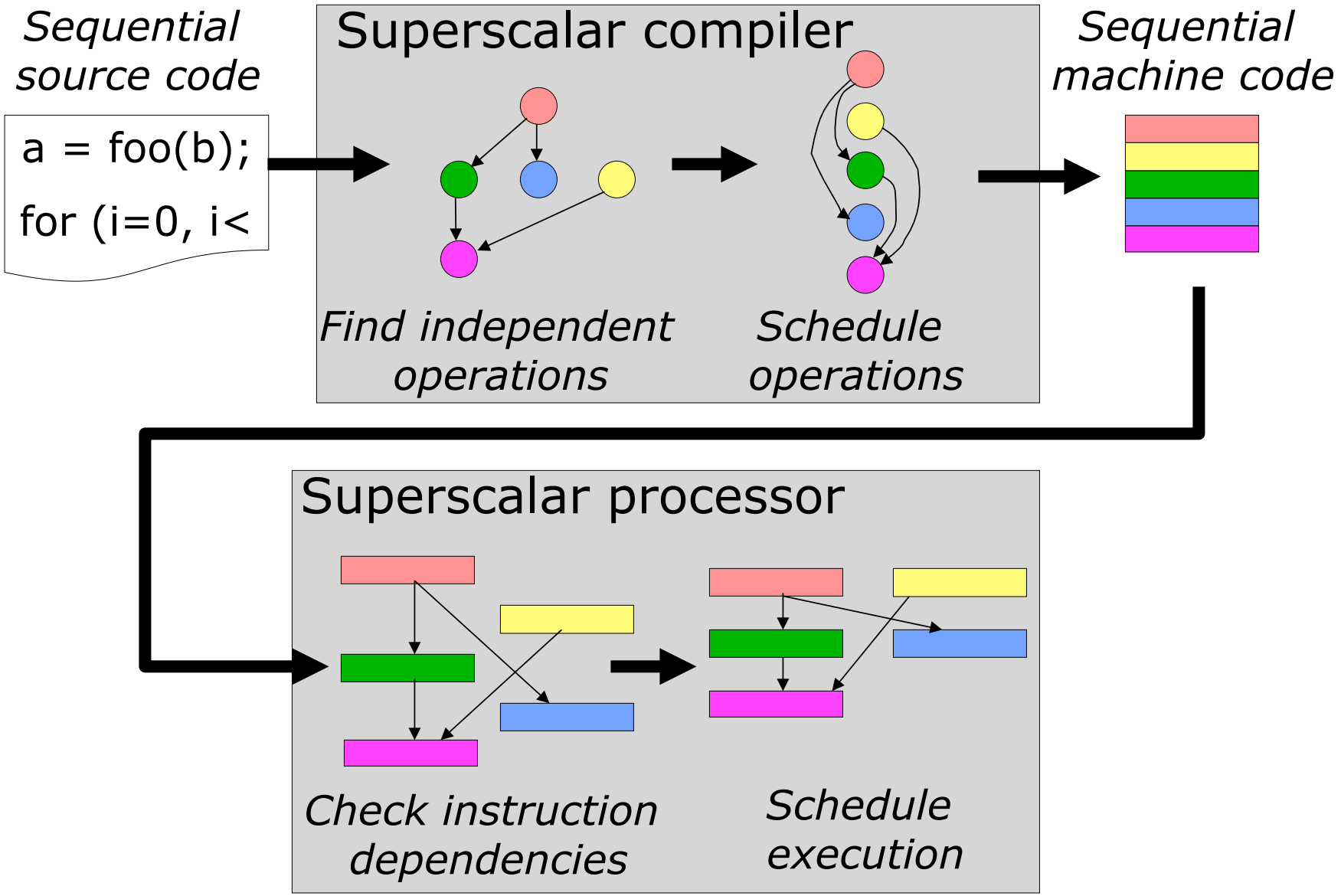
=> Out-of-order control logic grows faster than W^2 (~W^3)

Out-of-Order Control Complexity: MIPS R10000

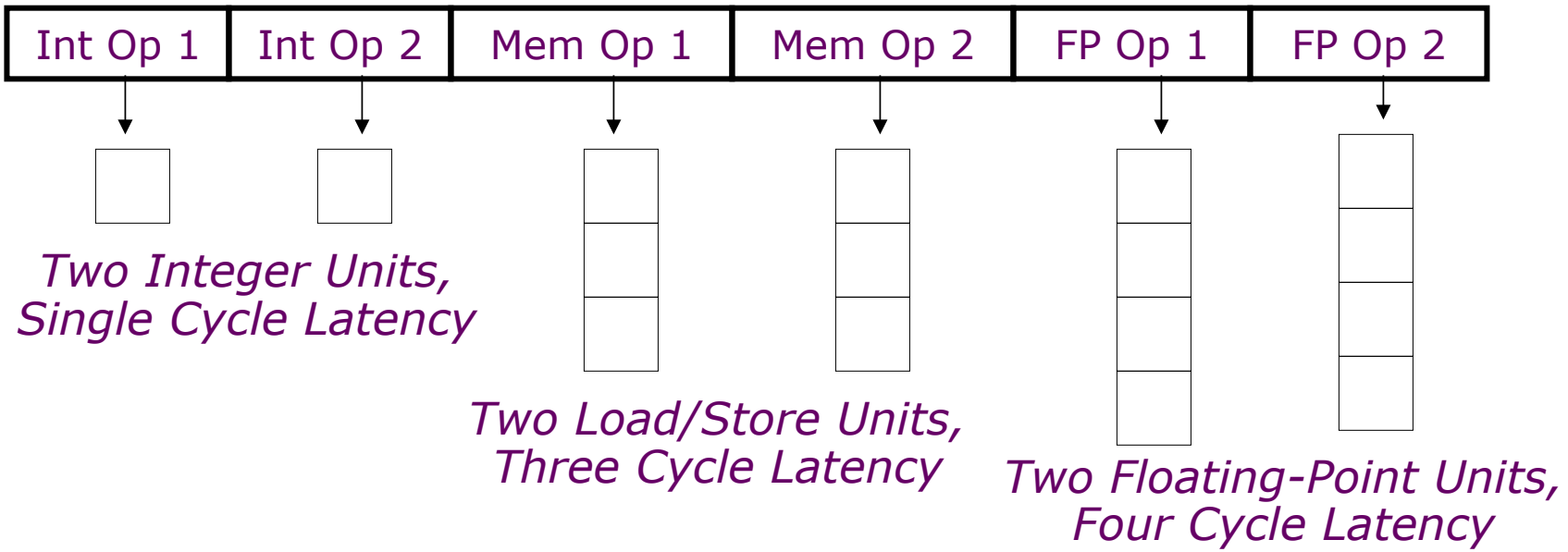


[SGI/MIPS Technologies Inc., 1995]

Sequential ISA Bottleneck



VLIW: Very Long Instruction Word



- 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

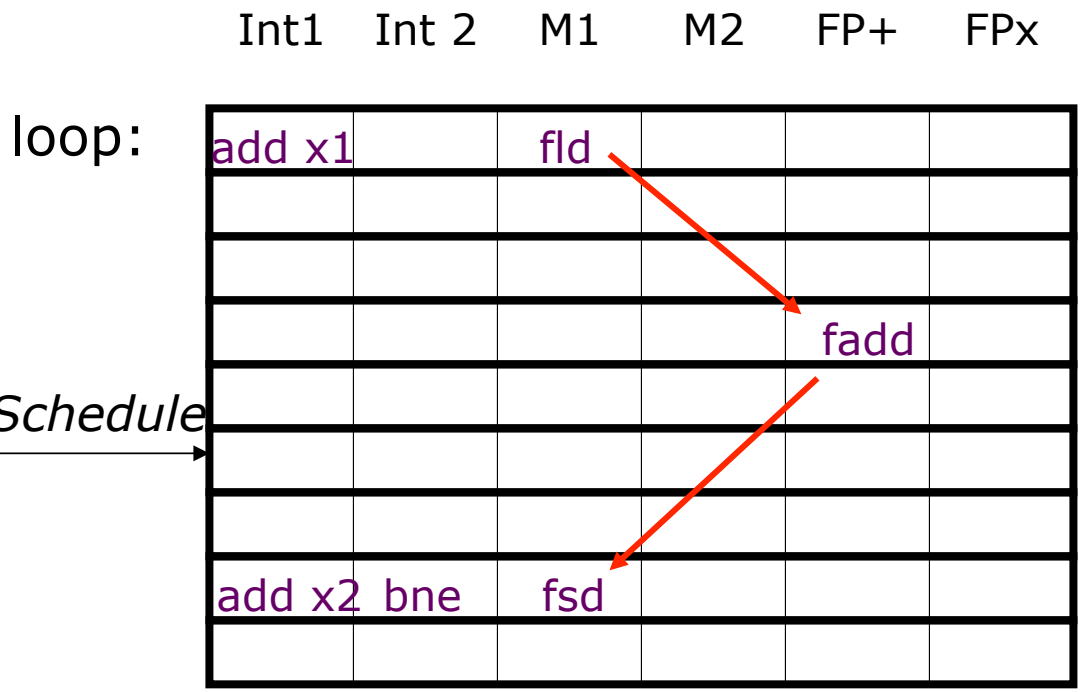
Loop Execution

```
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



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

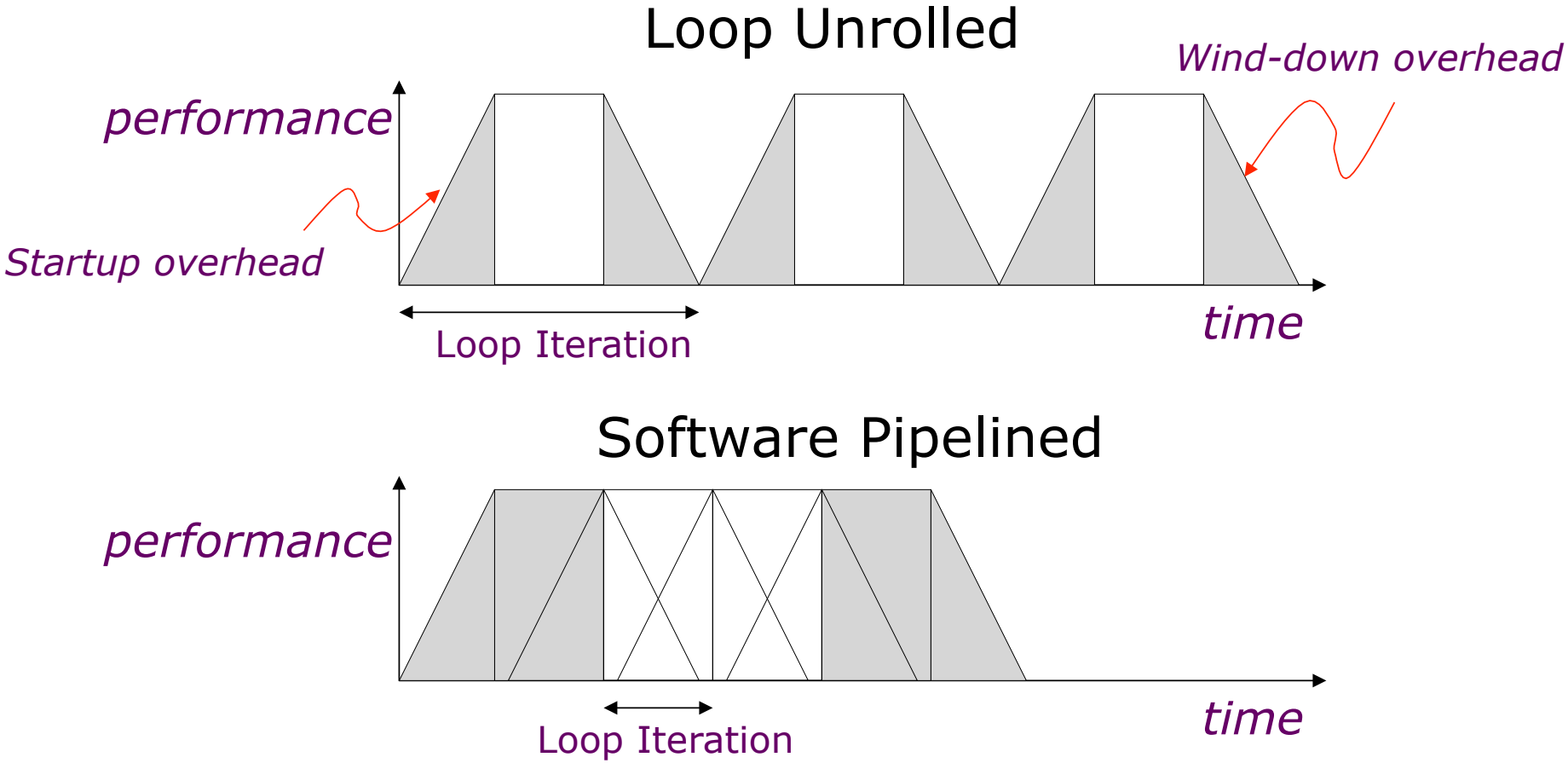
```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)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

	Int1	Int 2	M1	M2	FP+	FPx
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	
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	
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

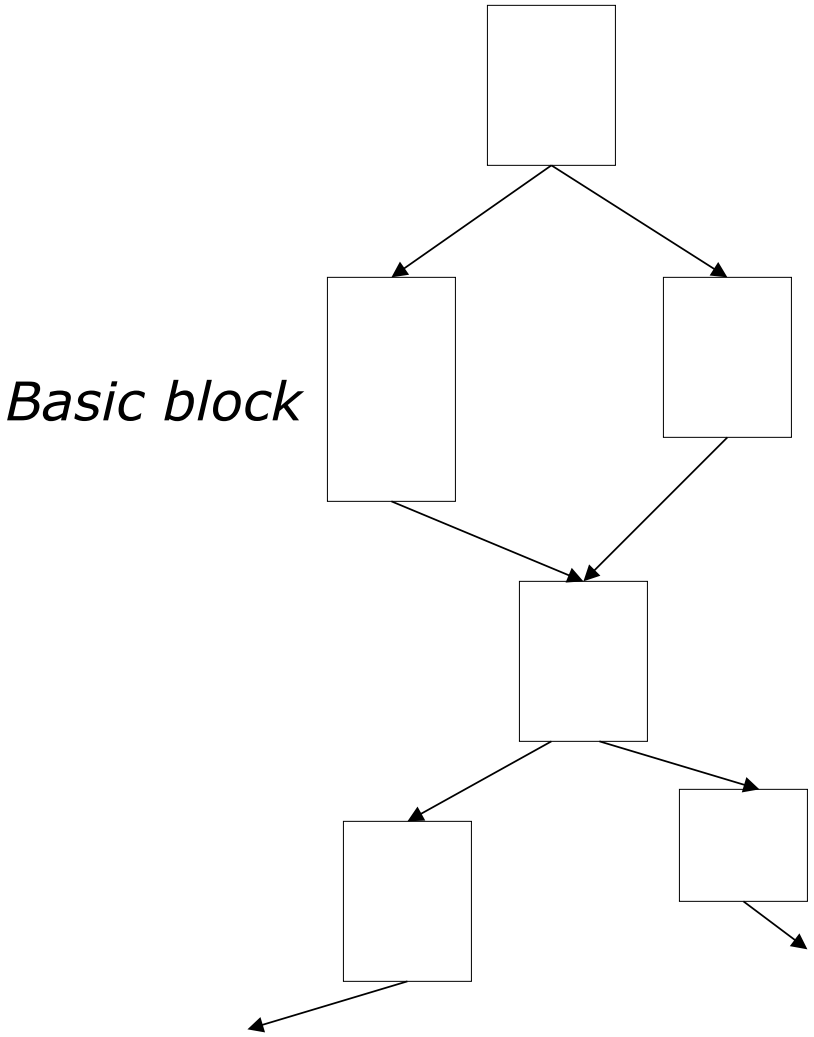


Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

CS152 Administrivia

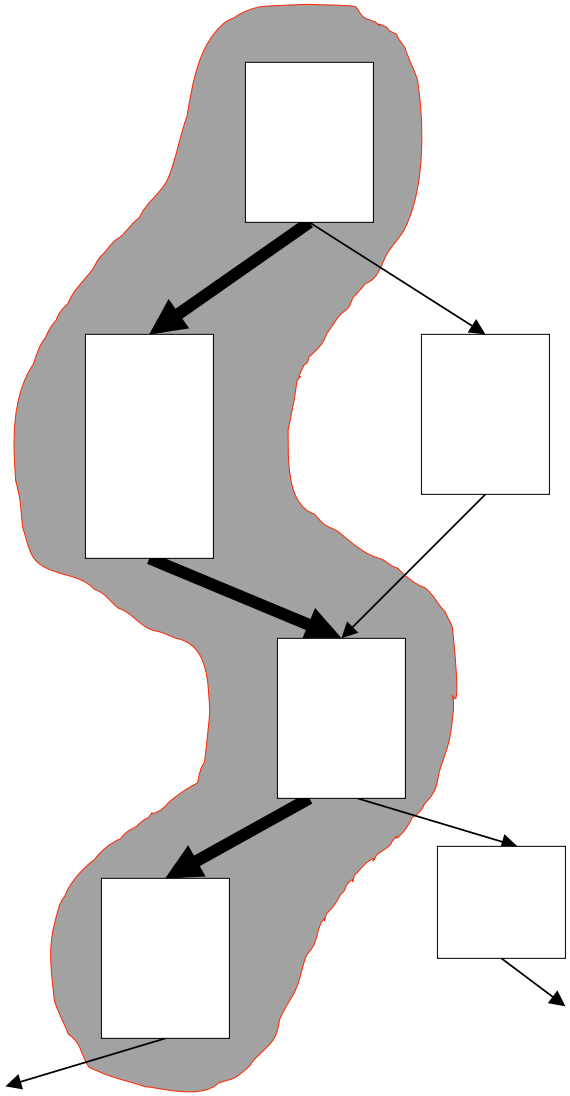
- Lab 3 due date pushed back two days
- Now due on same day as Quiz 3, Thursday Mar 22

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]*

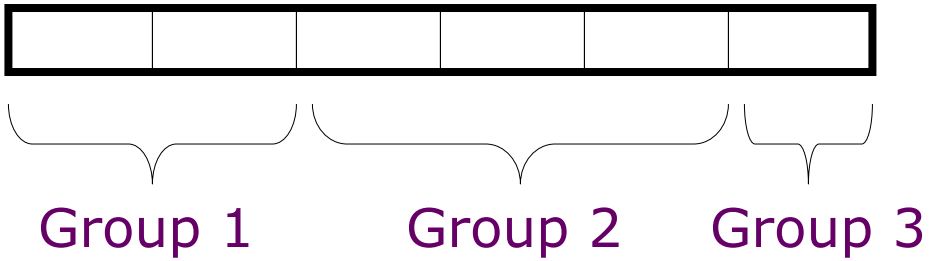


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

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

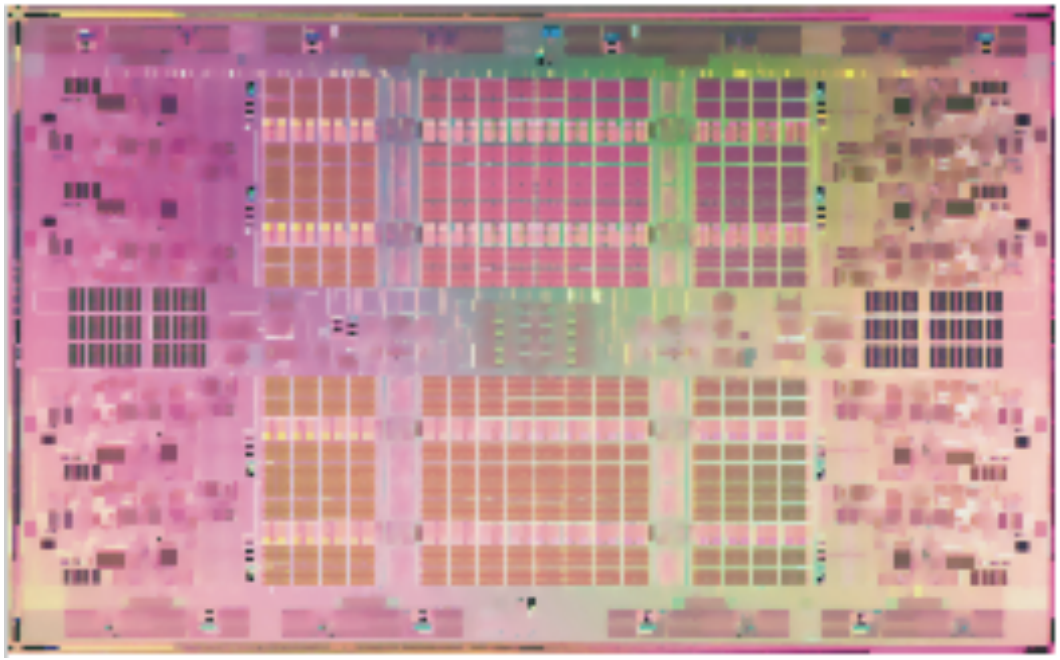


- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - » used in Multiflow Trace
    - » introduces instruction addressing challenge
  - Mark parallel groups
    - » used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    -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

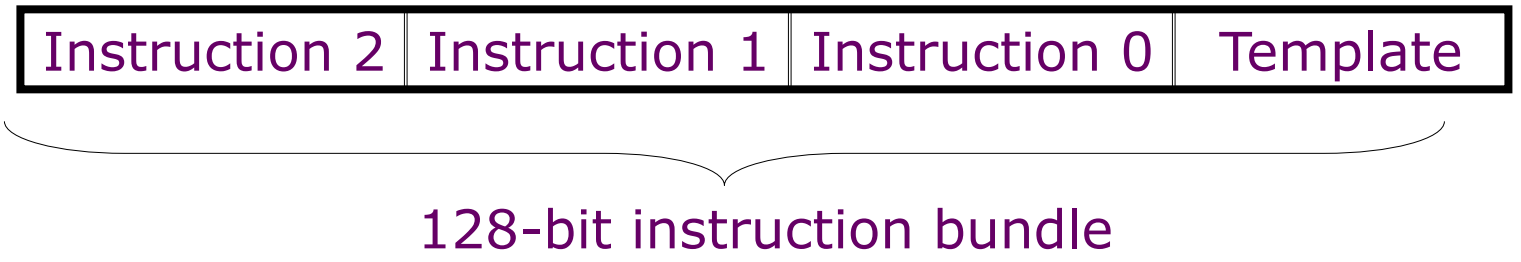
Eight Core Itanium "Poulson" *[Intel 2011]*



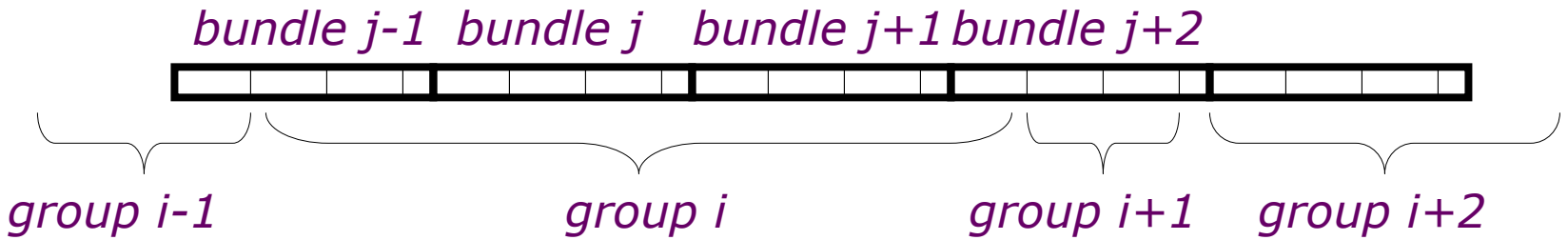
- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm^2 in 32nm CMOS
- Over 3 billion transistors

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

IA-64 Instruction Format



- 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

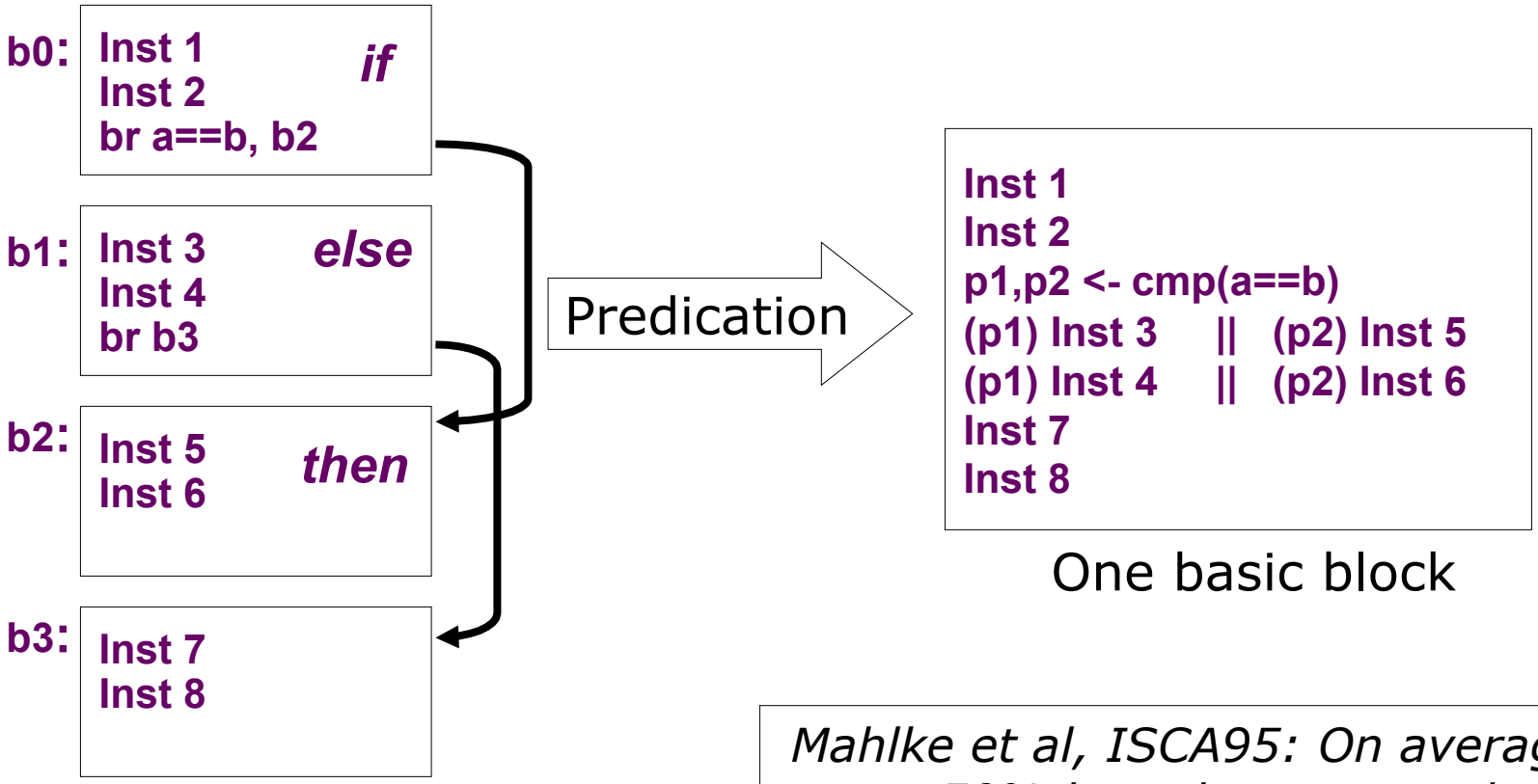
- GPRs “rotate” to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration

IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

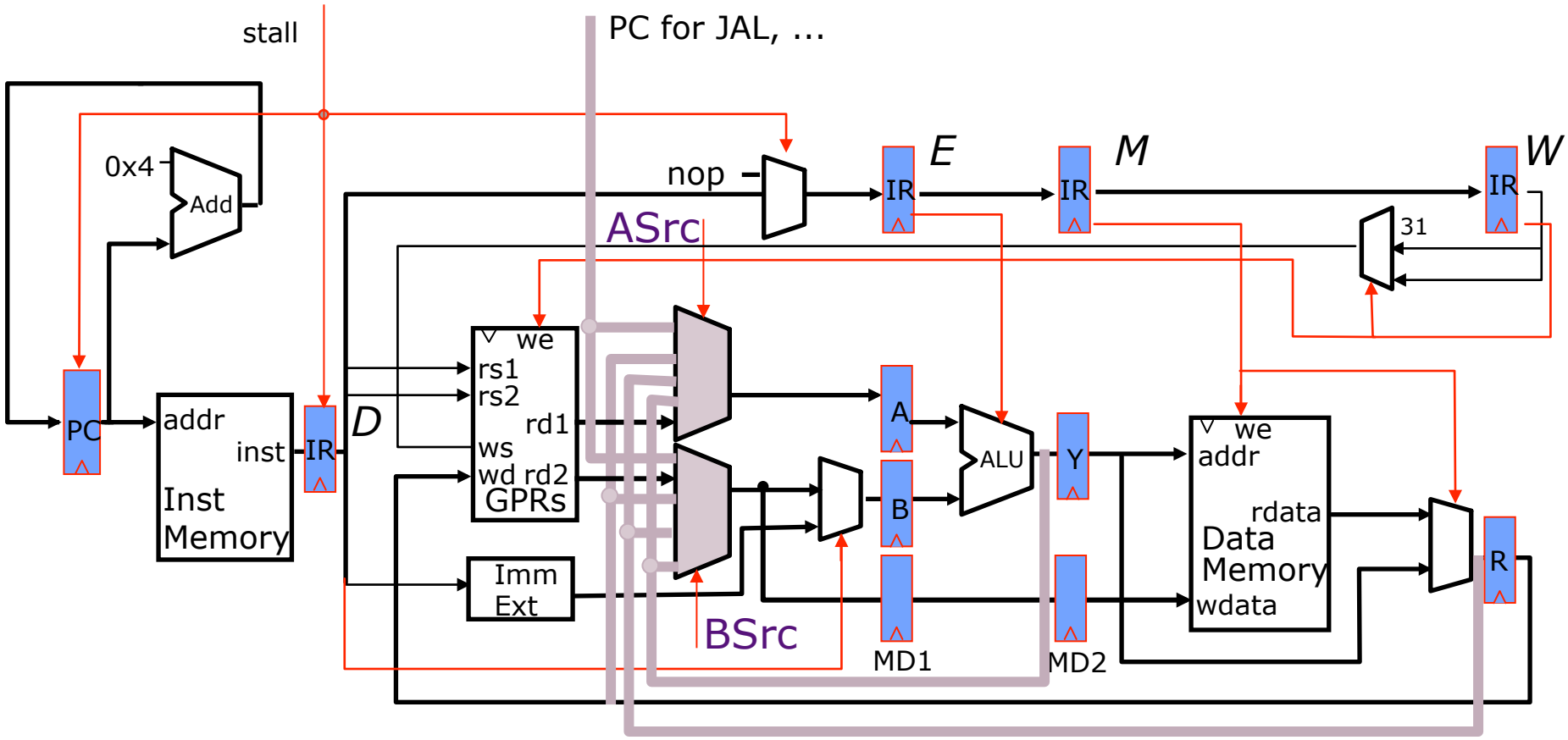
Solution: Eliminate hard to predict branches with predicated execution

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



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

Fully Bypassed Datapath

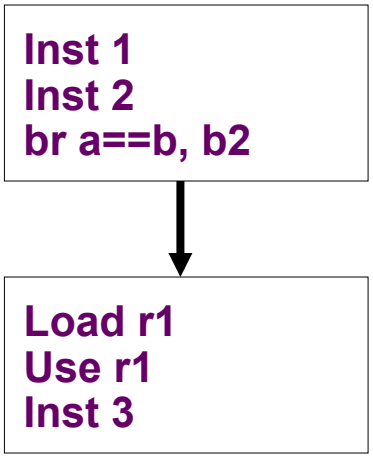


Where does predication fit in?

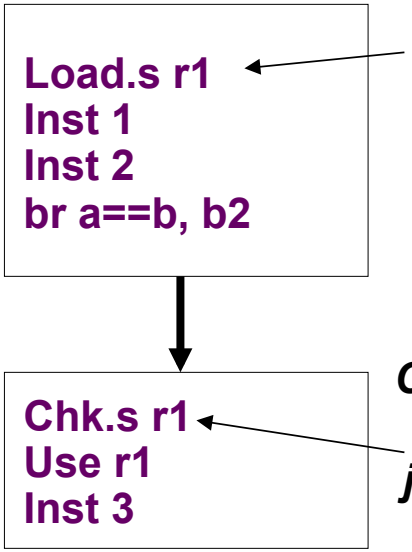
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



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



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

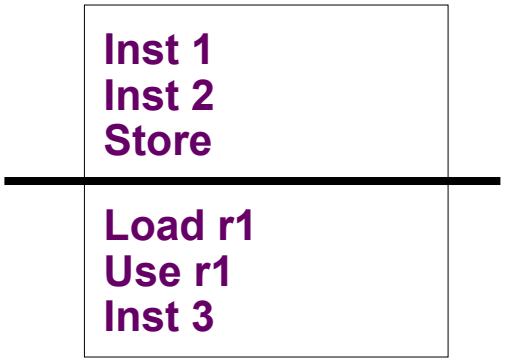
Check for exception in original home block jumps to fixup code if exception detected

Particularly useful for scheduling long latency loads early

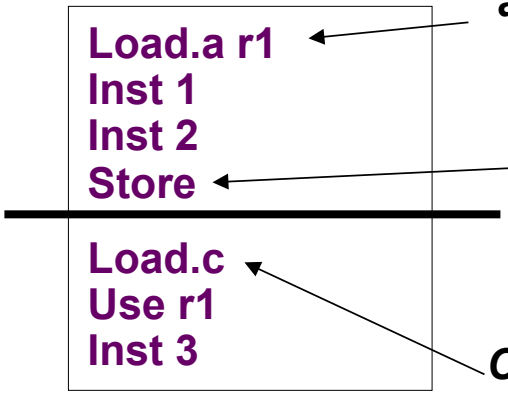
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards



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



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 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

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252