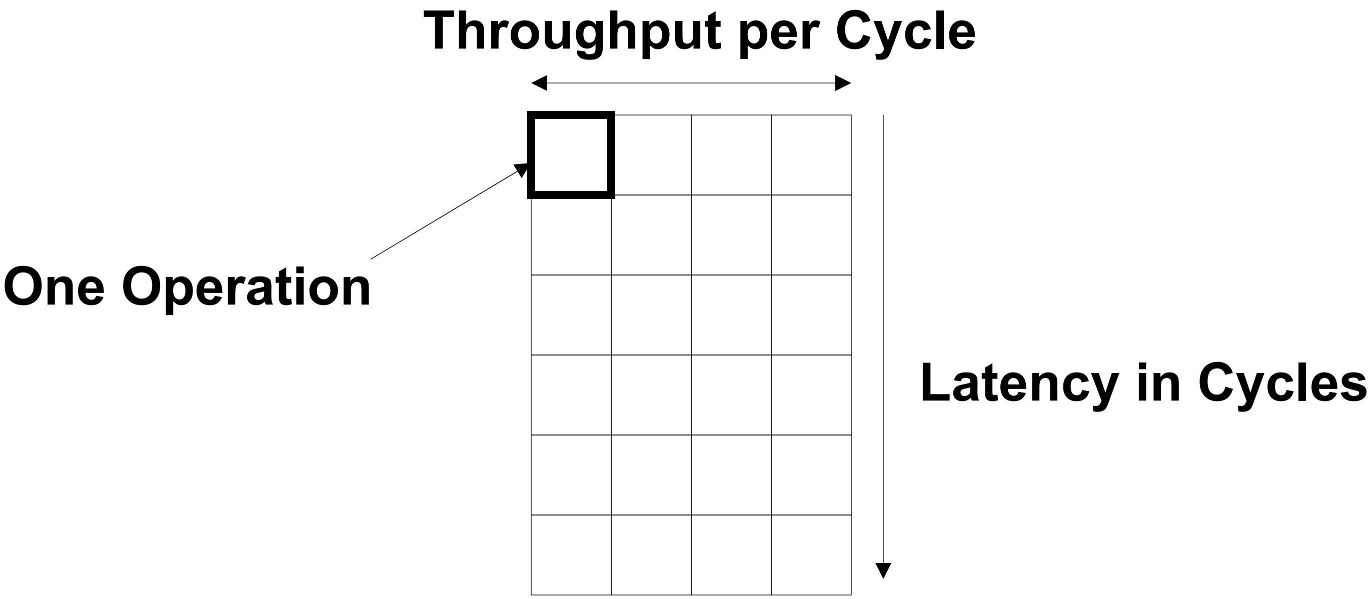
VLIW/EPIC: Statically Scheduled ILP

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology

Little’s Law

Parallelism = Throughput * Latency

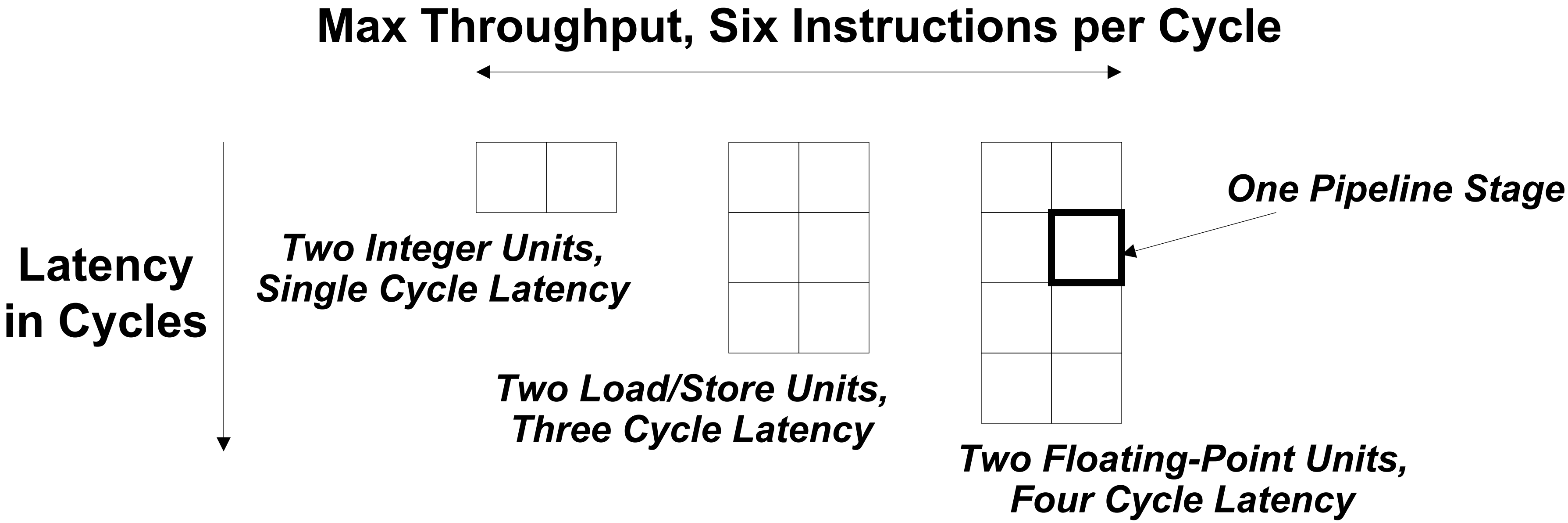


Throughput per Cycle

One Operation

Latency in Cycles

Example Pipelined ILP Machine



Max Throughput, Six Instructions per Cycle

Latency in Cycles

Two Integer Units, Single Cycle Latency

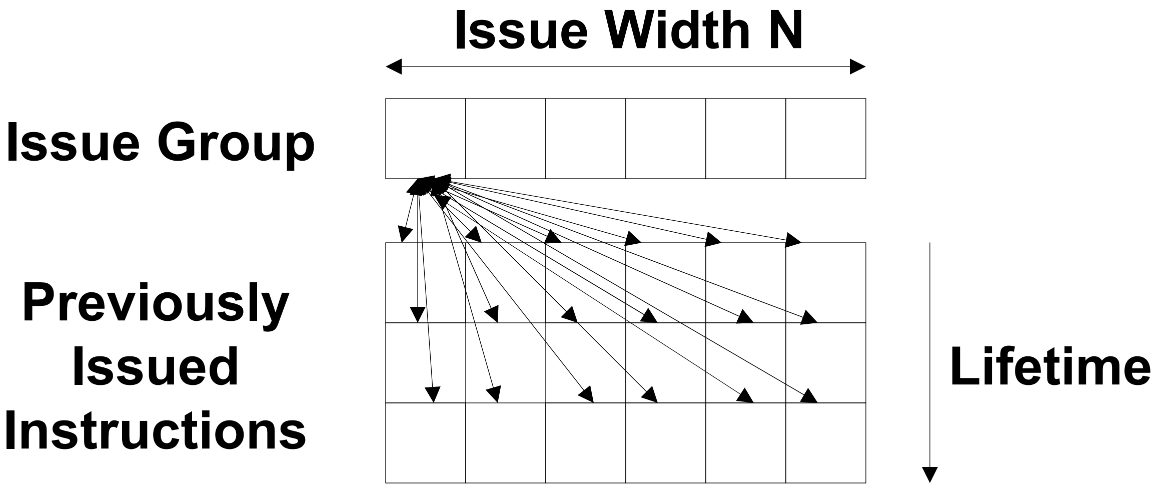
Two Load/Store Units, Three Cycle Latency

Two Floating-Point Units, Four Cycle Latency

One Pipeline Stage

- How much instruction-level parallelism (ILP) required to keep machine pipelines busy?

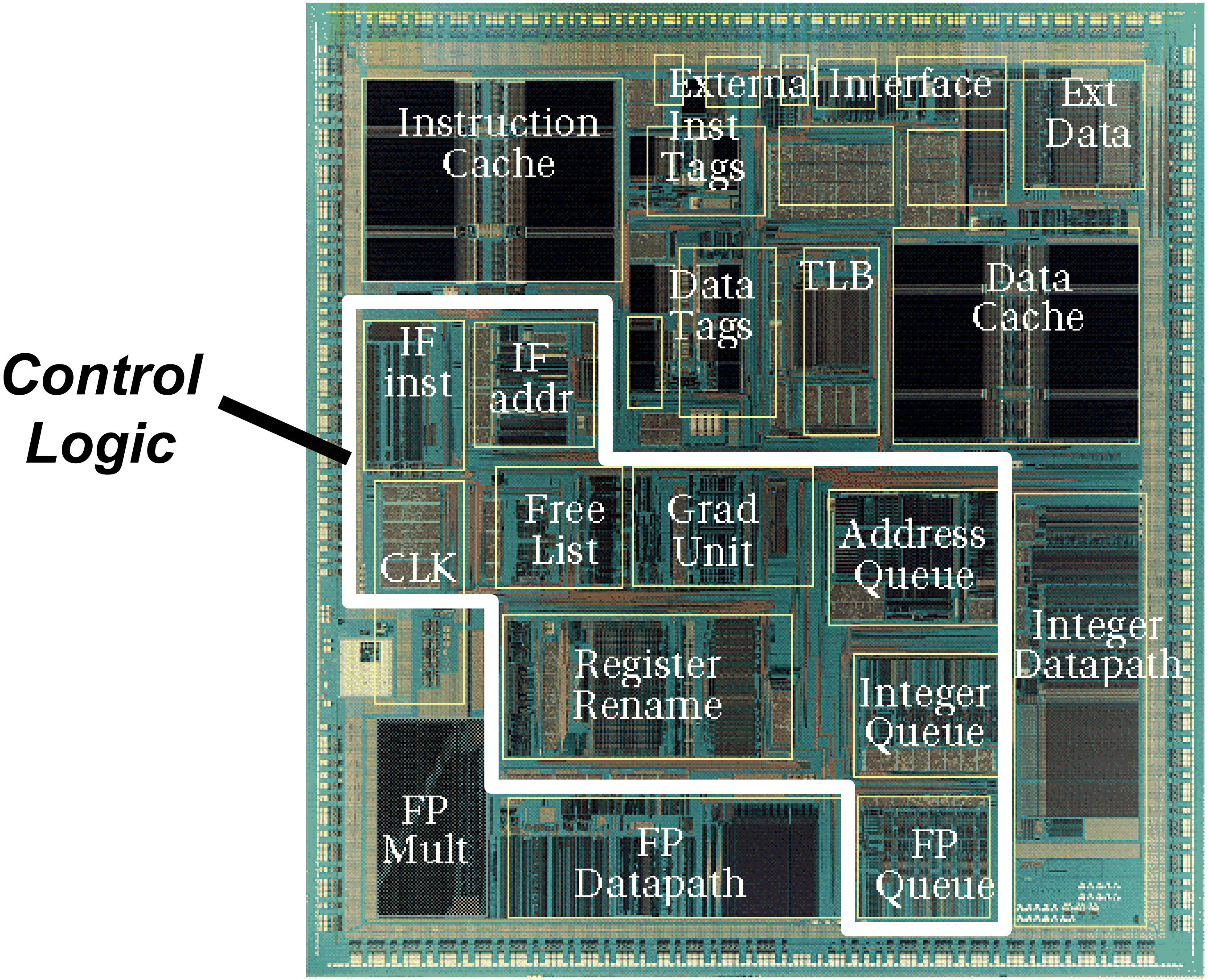
Superscalar Control Logic Scaling



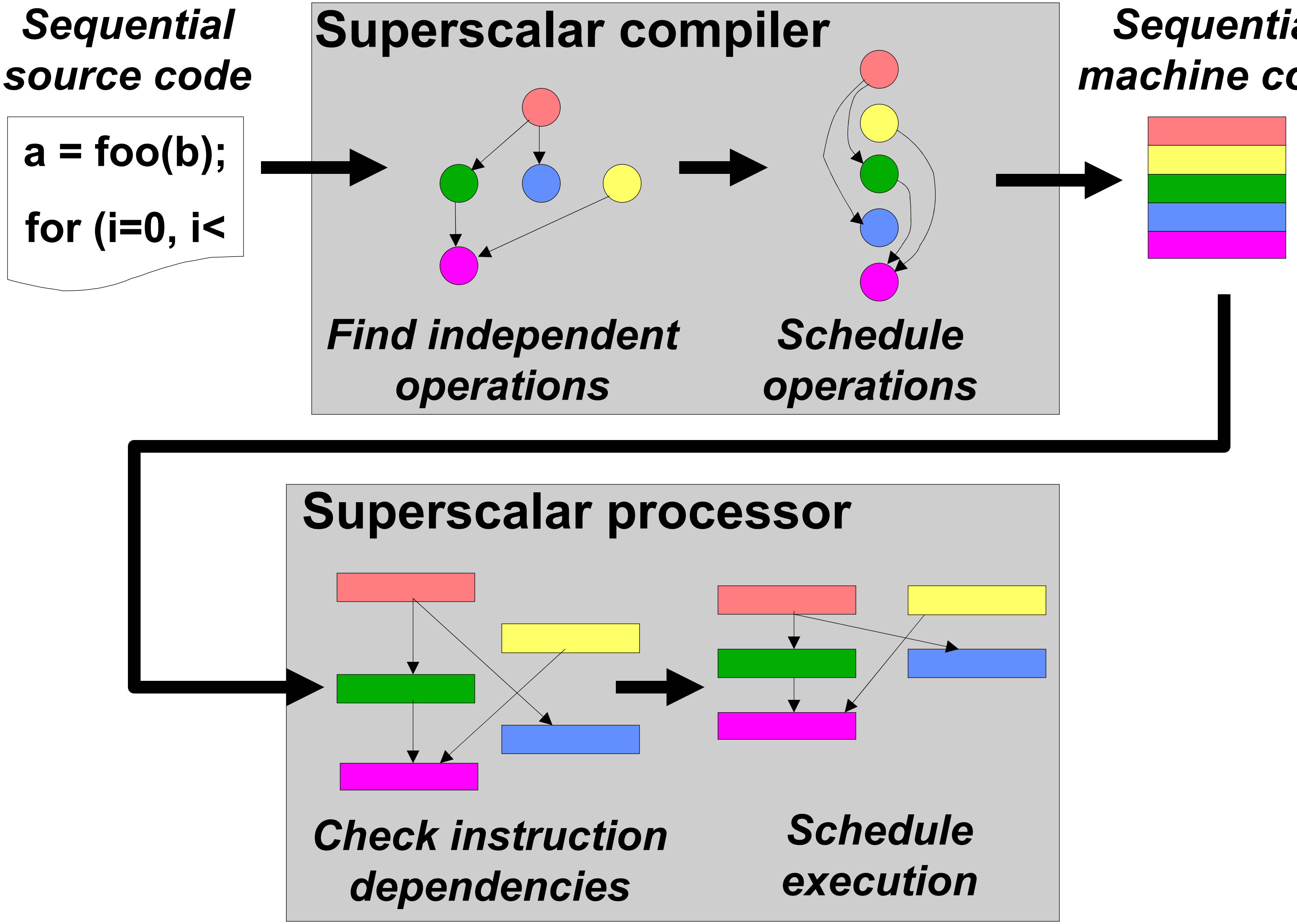
- Number of interlock checks and/or tag compares grows as N*(N*L) where L is lifetime of instructions in machine
  - Each of N instructions issued or completed must check against N*L instructions in flight
- For in-order machines, lifetime L is related to pipeline latencies
- For out-of-order machines, L also includes time spent in instruction buffers (instruction window or ROB)
- As N increases, need larger instruction window to find enough parallelism to keep machine busy => greater lifetime L

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

Out-of-Order Control Complexity: MIPS R10000



Sequential ISA Bottleneck



Sequential source code

```
a = foo(b);

for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations

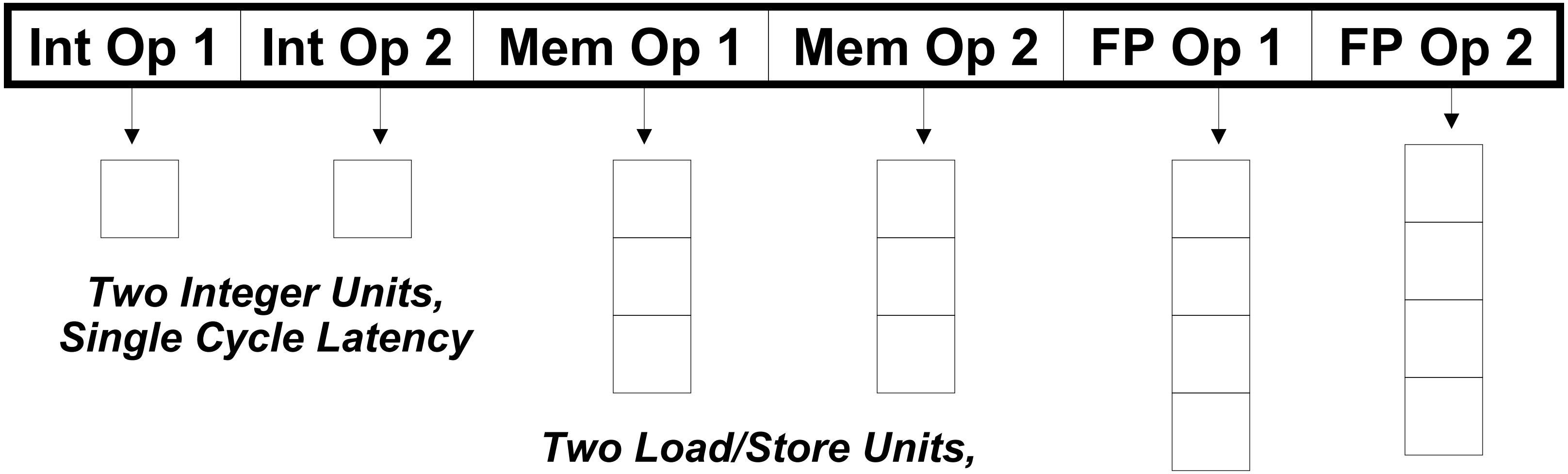
Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

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

- Compiler schedules parallel execution
- Multiple parallel operations packed into one long instruction word
- Compiler must avoid data hazards (no 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

Loop Execution

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

Compile

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

Schedule

	Int1	Int 2	M1	M2	FP+	FPx
loop:						

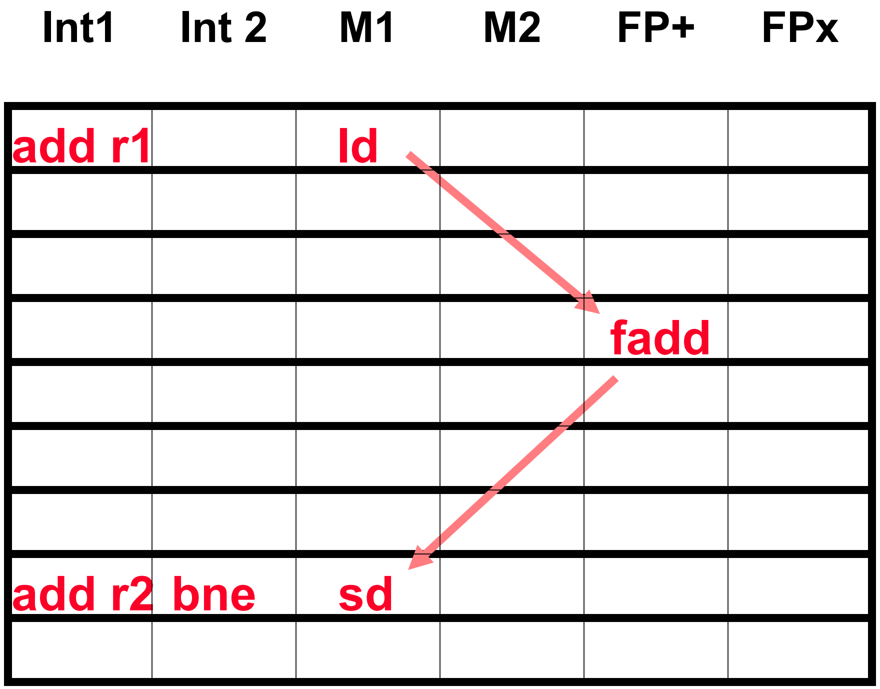
Loop Execution

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

Compile

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

Schedule



	Int1	Int 2	M1	M2	FP+	FPx
loop:	add r1		ld			
					fadd	
	add r2	bne	sd			

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:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Schedule

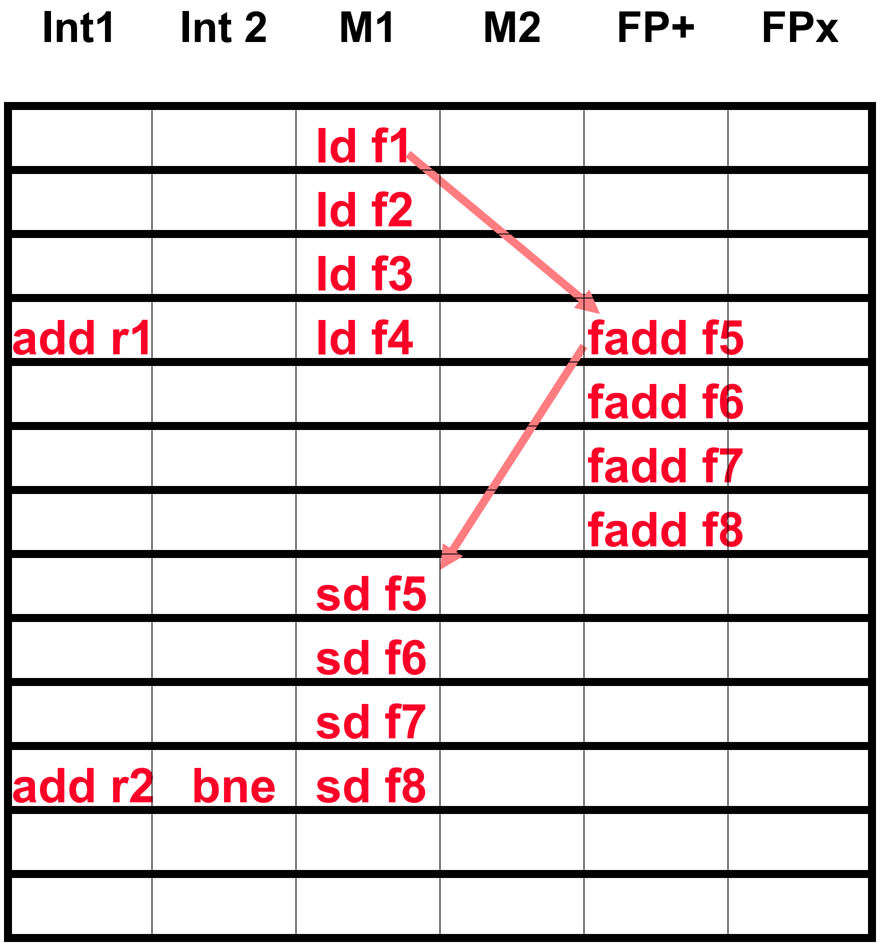
	Int1	Int 2	M1	M2	FP+	FPx
loop:						

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Schedule



	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

How many FLOPS/cycle?

Software Pipelining

Unroll 4 ways first

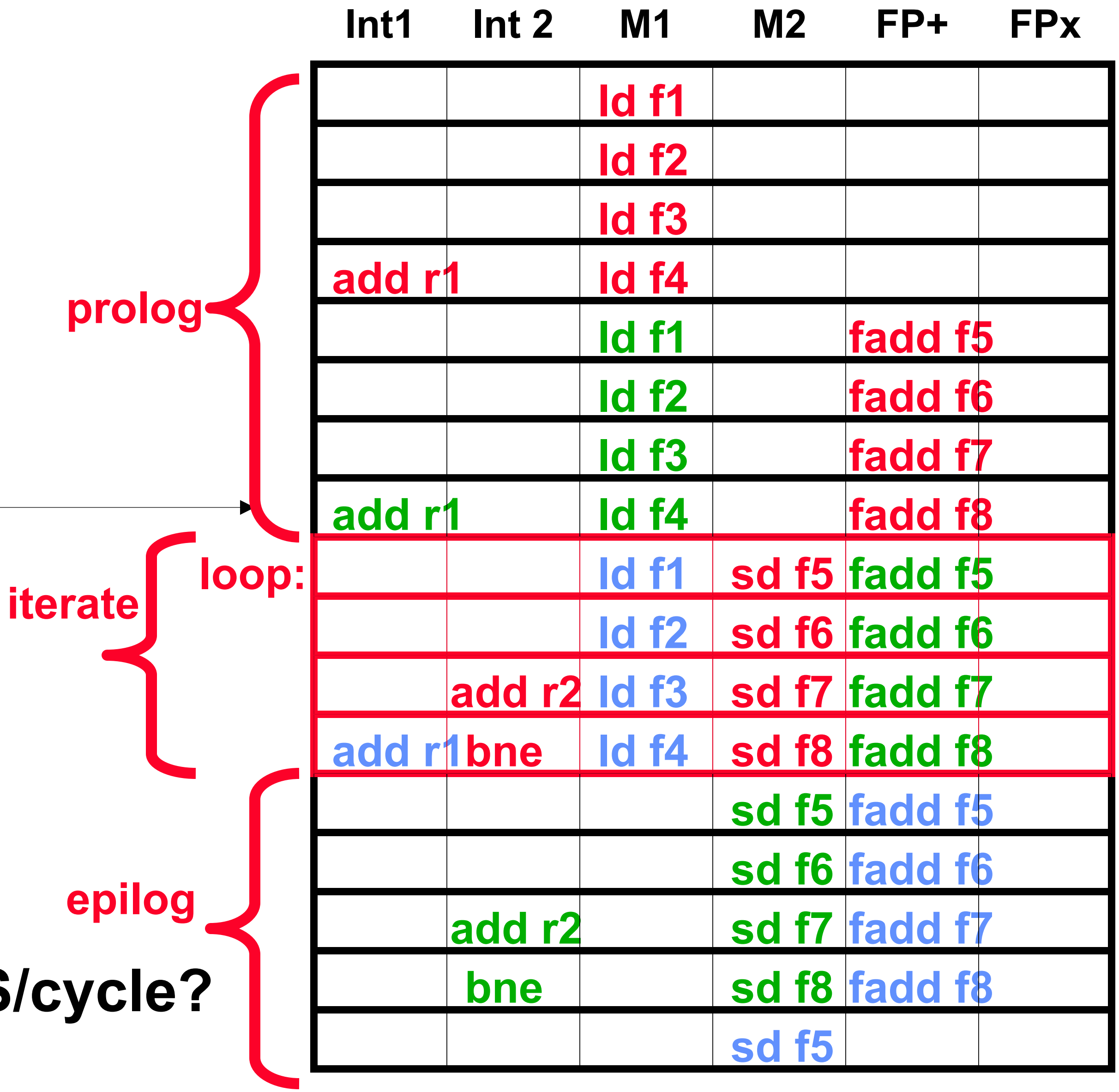
```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

Int1	Int 2	M1	M2	FP+	FPx

Software Pipelining

Unroll 4 ways first

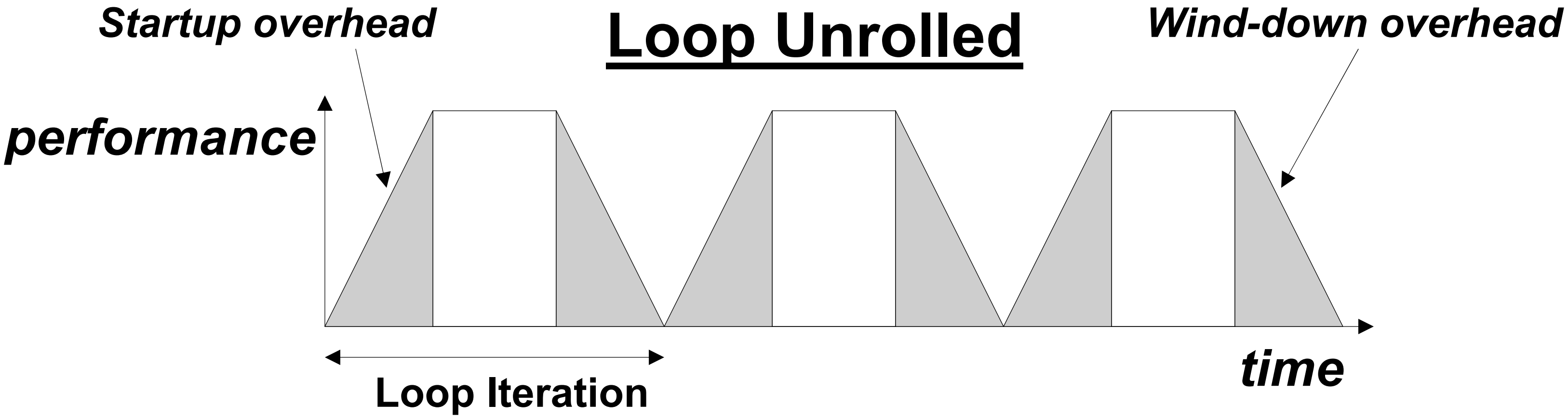
```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

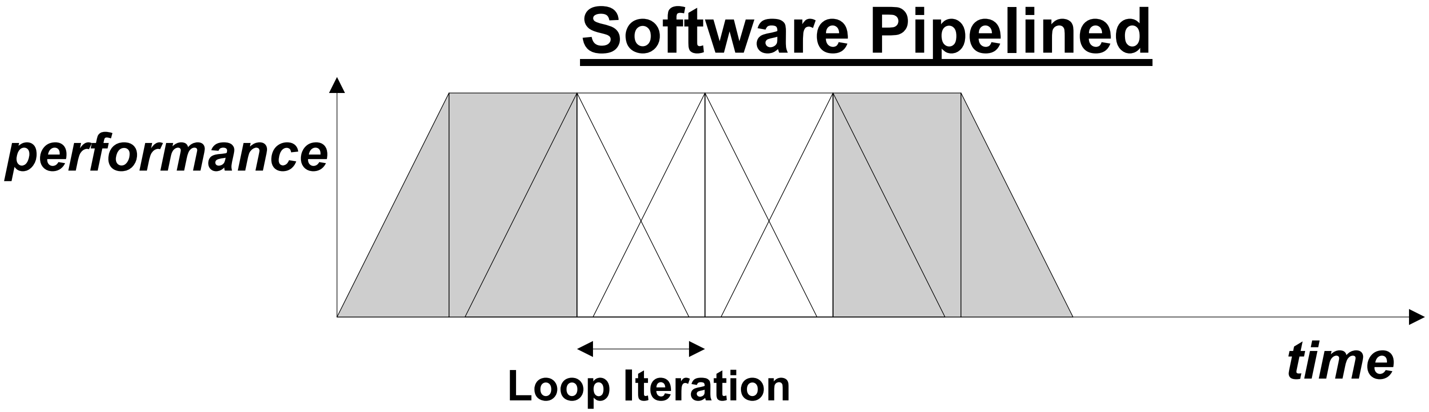


	Int1	Int 2	M1	M2	FP+	FPx
prolog			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
			ld f1		fadd f5	
			ld f2		fadd f6	
			ld f3		fadd f7	
	add r1		ld f4		fadd f8	
iterate loop:			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
	add r1	bne	ld f4	sd f8	fadd f8	
epilog				sd f5	fadd f5	
				sd f6	fadd f6	
		add r2		sd f7	fadd f7	
		bne		sd f8	fadd f8	
				sd f5		

How many FLOPS/cycle?

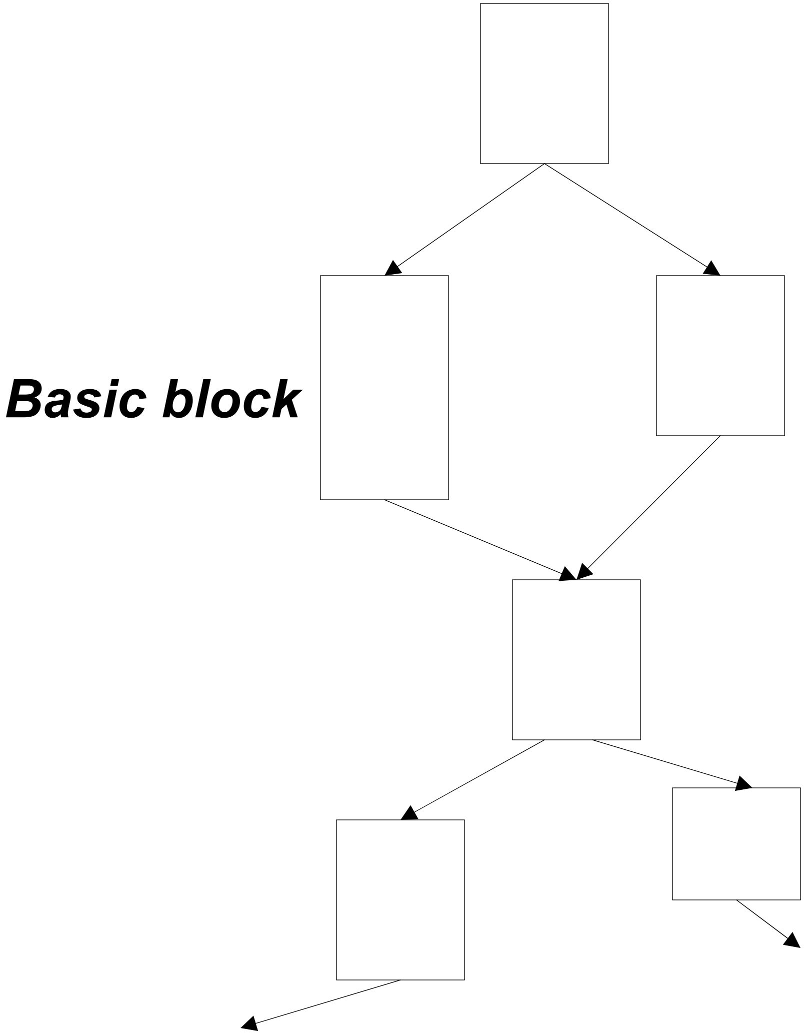
Software Pipelining vs. Loop Unrolling





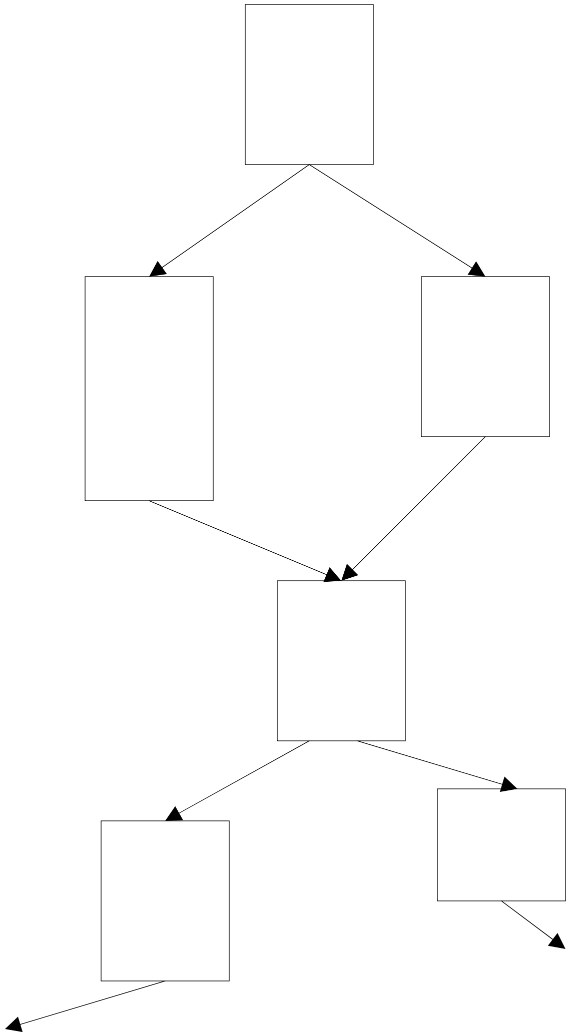
- ***Software pipelining pays startup/wind-down costs only once per loop, not once per iteration***

What if there are no loops?

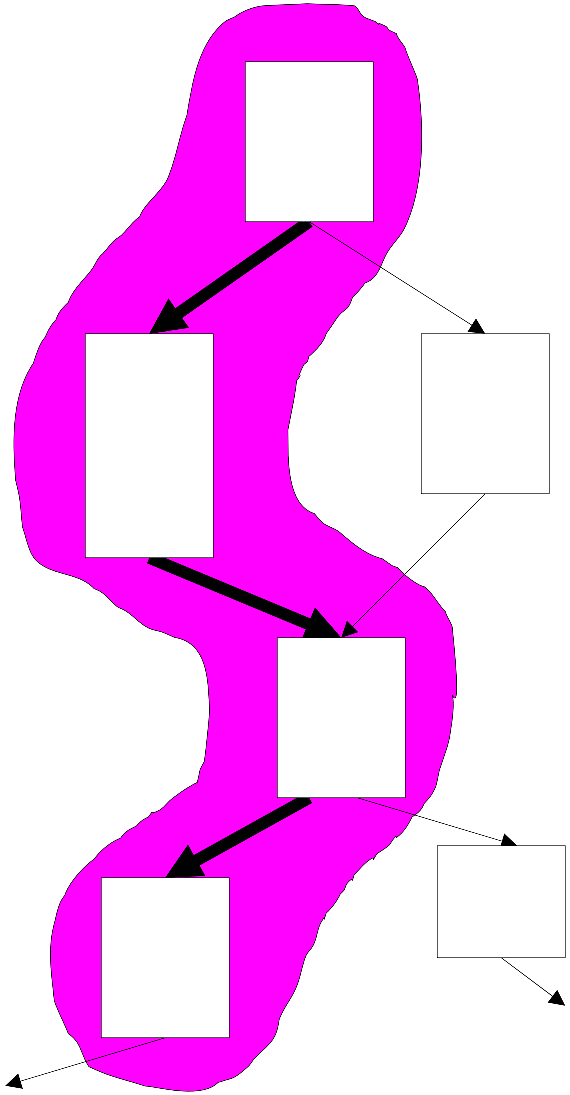


- **Branches limit basic block size in control-flow intensive irregular code**
- **Difficult to find ILP in individual basic blocks**

Trace Scheduling *[Fisher,Ellis]*



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

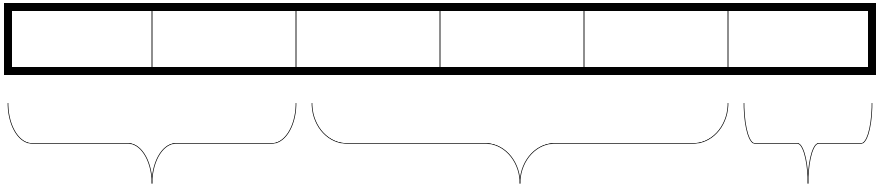
Problems with “Classic” VLIW

- 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
- Scheduling around statically unpredictable branches
  - optimal schedule varies with branch path

VLIW Instruction Encoding

Various schemes to reduce effect of unused fields

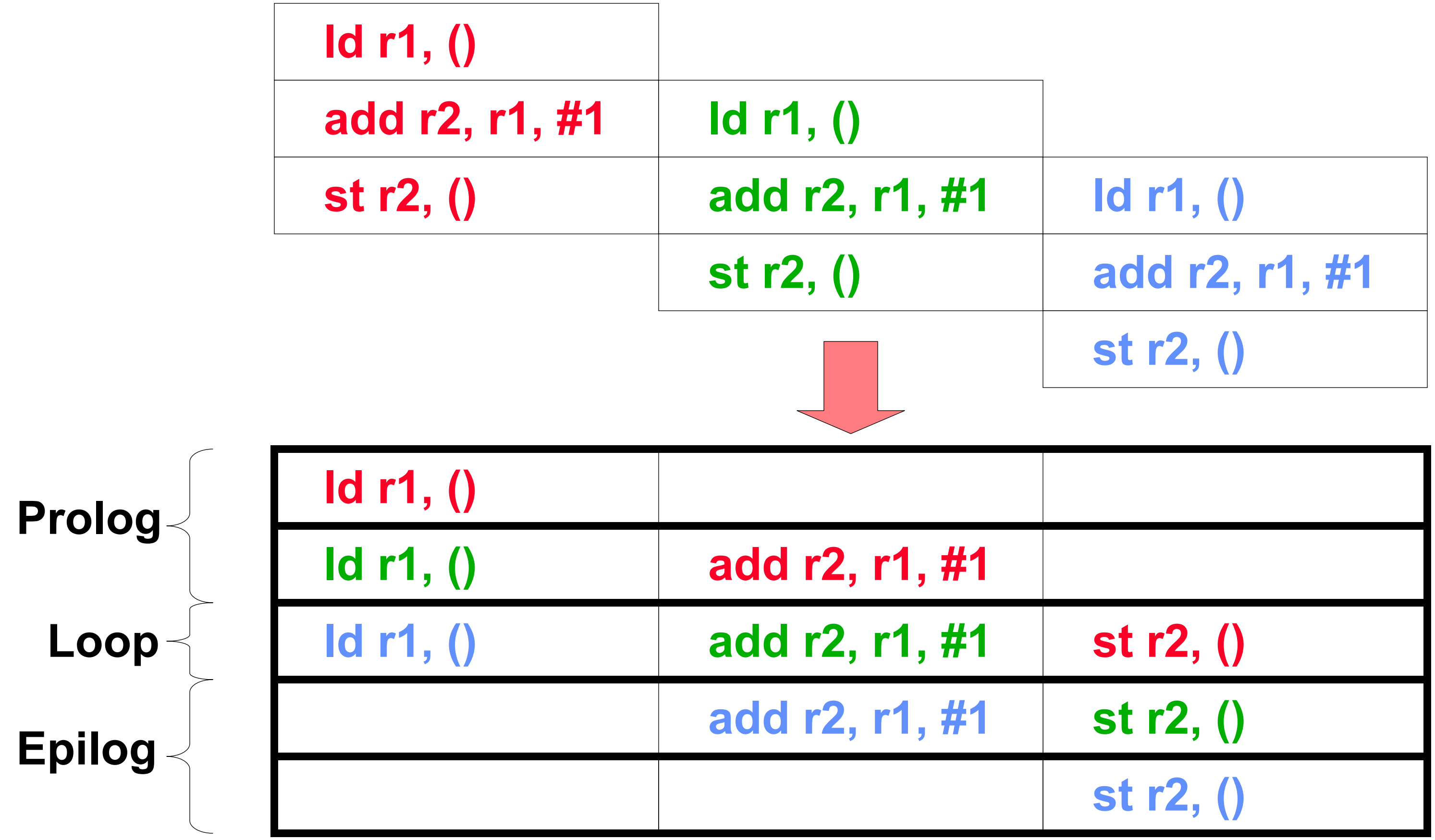
- Compressed format in memory, expand on I-cache refill
- Cydra-5 MultiOp instructions: execute VLIW as sequential operations
- Mark parallel groups (used in TMS320C6x DSPs, Intel IA-64)



Group 1 Group 2 Group 3

Rotating Register Files

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

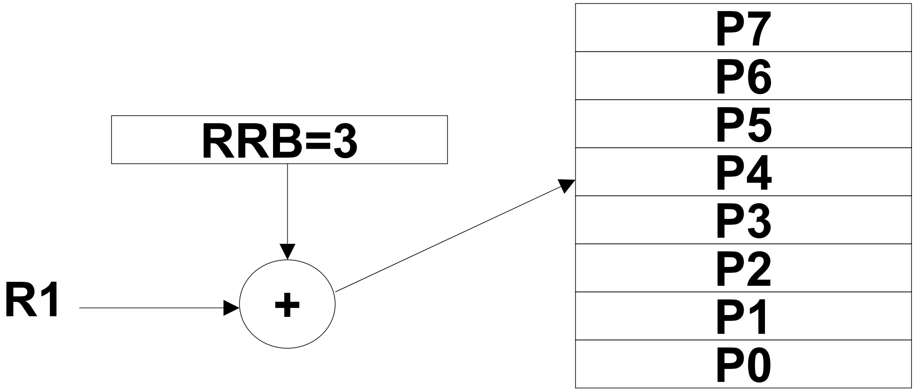


ld r1, ()		
add r2, r1, #1	ld r1, ()	
st r2, ()	add r2, r1, #1	ld r1, ()
	st r2, ()	add r2, r1, #1
		st r2, ()

Prolog	ld r1, ()		
	ld r1, ()	add r2, r1, #1	
Loop	ld r1, ()	add r2, r1, #1	st r2, ()
Epilog		add r2, r1, #1	st r2, ()
			st r2, ()

Solution: Allocate new set of registers for each loop iteration

Rotating Register File

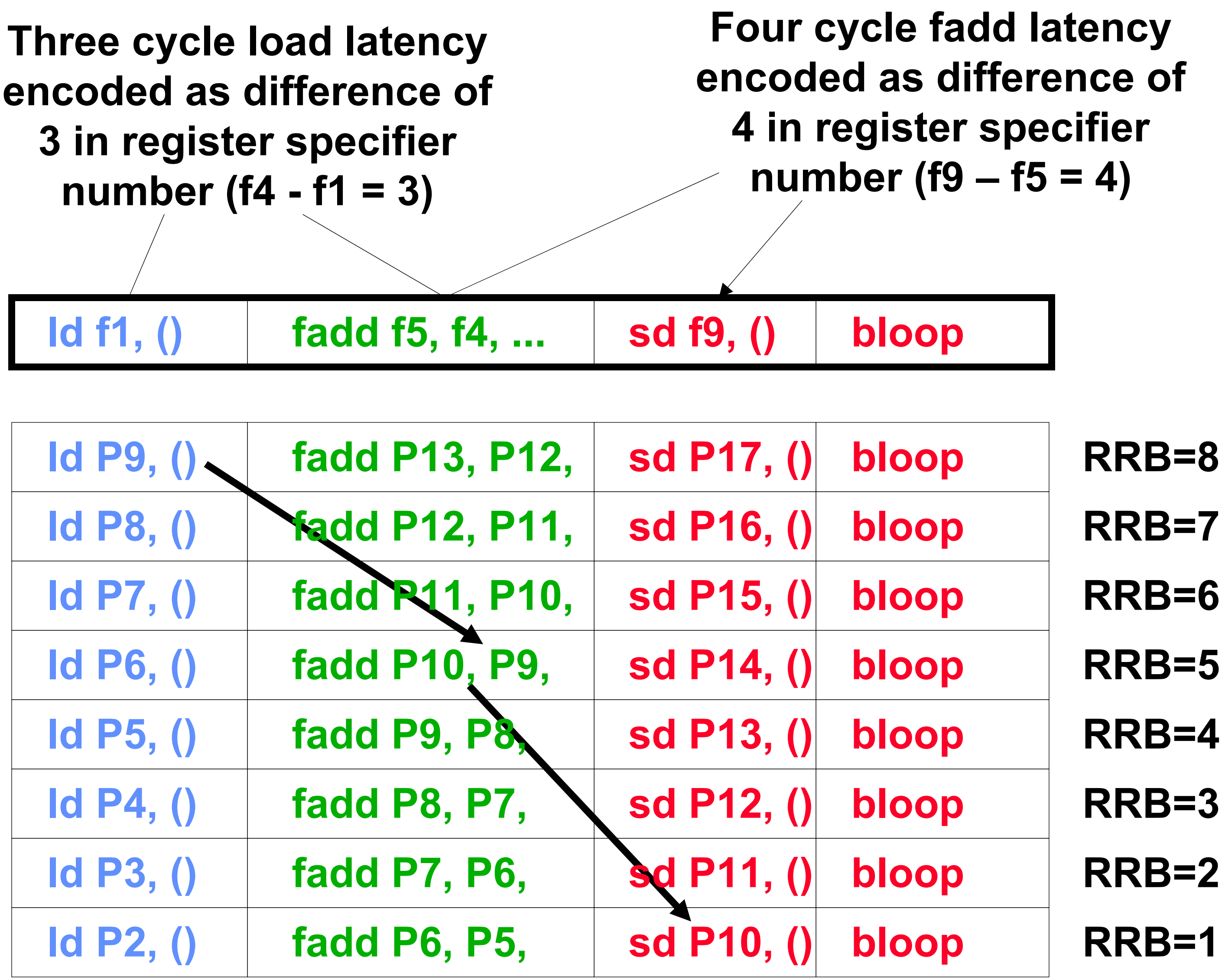


Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

Prolog	ld r1, ()			dec RRB
	ld r1, ()	add r3, r2, #1		dec RRB
Loop	ld r1, ()	add r3, r2, #1	st r4, ()	bloop
Epilog		add r2, r1, #1	st r4, ()	dec RRB
			st r4, ()	dec RRB

Loop closing branch decrements RRB

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

Predicate Software Pipeline Stages

Single VLIW Instruction

(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop

Dynamic Execution

(p1) ld r1			(p1) bloop
(p1) ld r1	(p2) add r3		(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
	(p2) add r3	(p3) st r4	(p1) bloop
		(p3) st r4	(p1) bloop

Software pipeline stages turned on by rotating predicate registers
→ Much denser encoding of loops

Cydra-5: Memory Latency Register (MLR)

Problem: Loads have variable latency

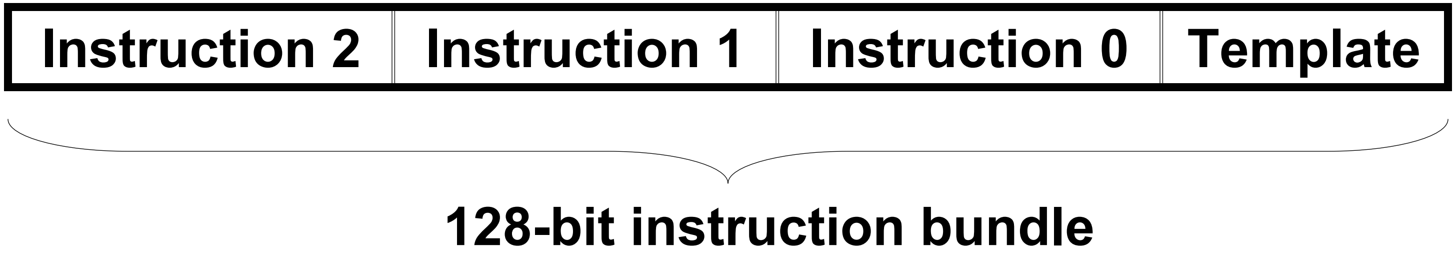
Solution: Let software choose desired memory latency

- Compiler tries to schedule code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late

Intel EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing
- IA-64 is Intel’s chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code compatible VLIW
- Itanium (aka Merced) is first implementation (cf. 8086)
  - First customer shipment should be in ~~1997~~ , ~~1998~~, ~~1999~~, ~~2000~~, 2001
  - McKinley will be second implementation due 2002

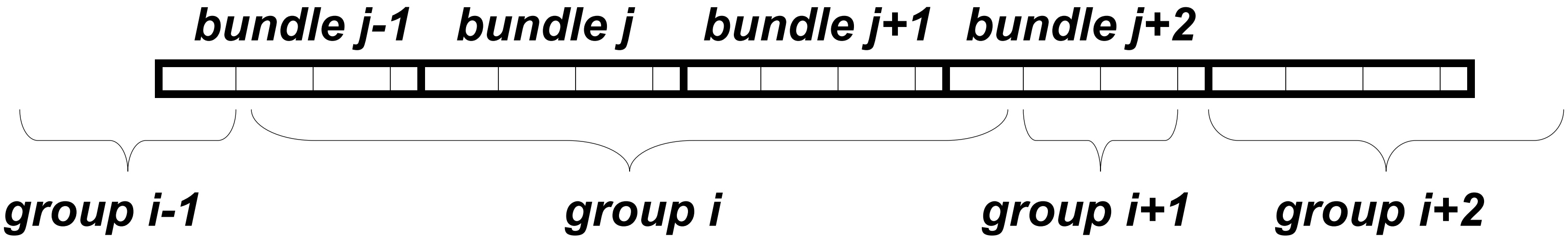
IA-64 Instruction Format



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



bundle j-1 *bundle j* *bundle j+1* *bundle j+2*

group i-1 *group i* *group i+1* *group i+2*

IA-64 Registers

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

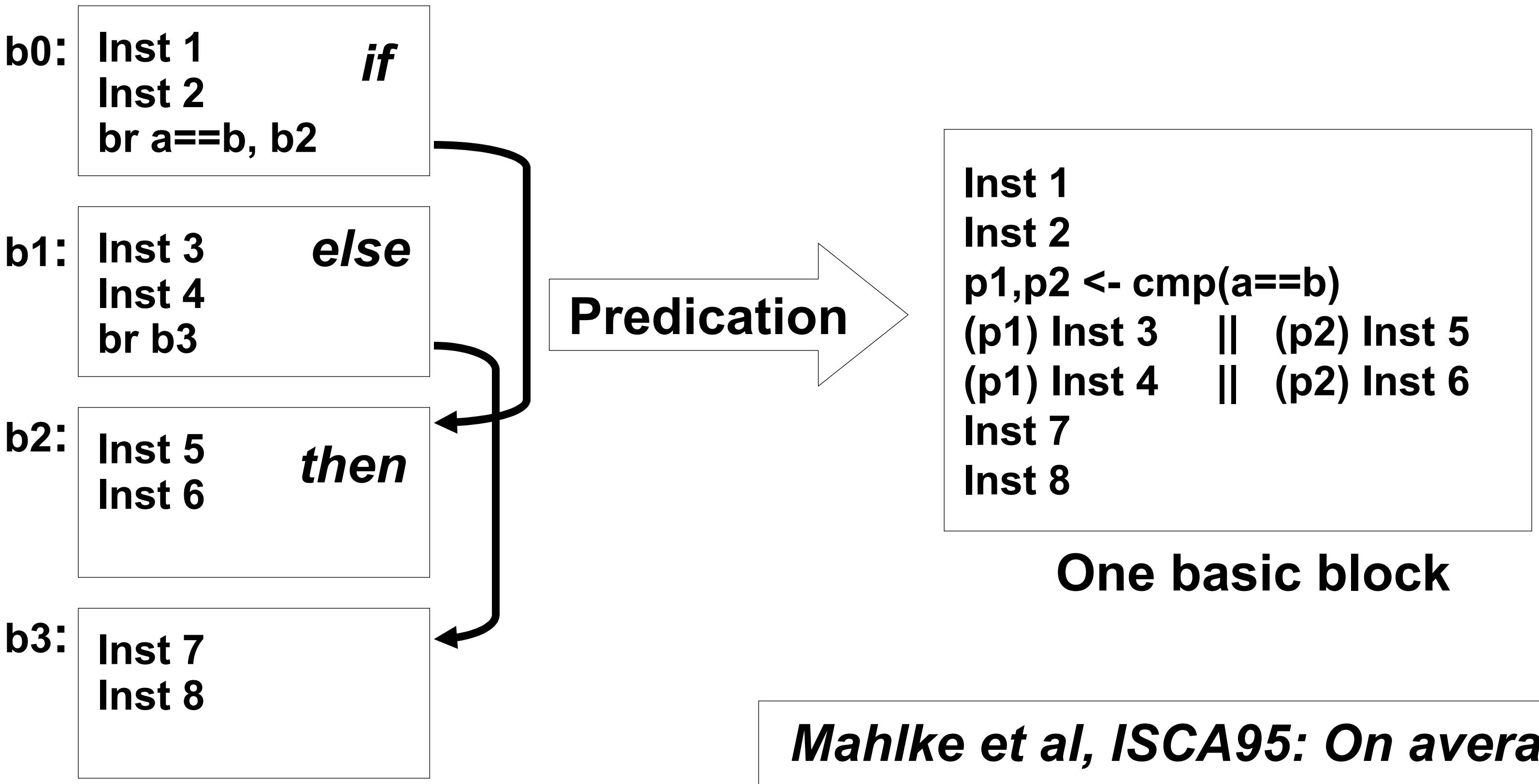
- GPRs rotate to reduce code size for software pipelined loops

IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

Solution: Eliminate some 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
```

```
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
```

Four basic blocks

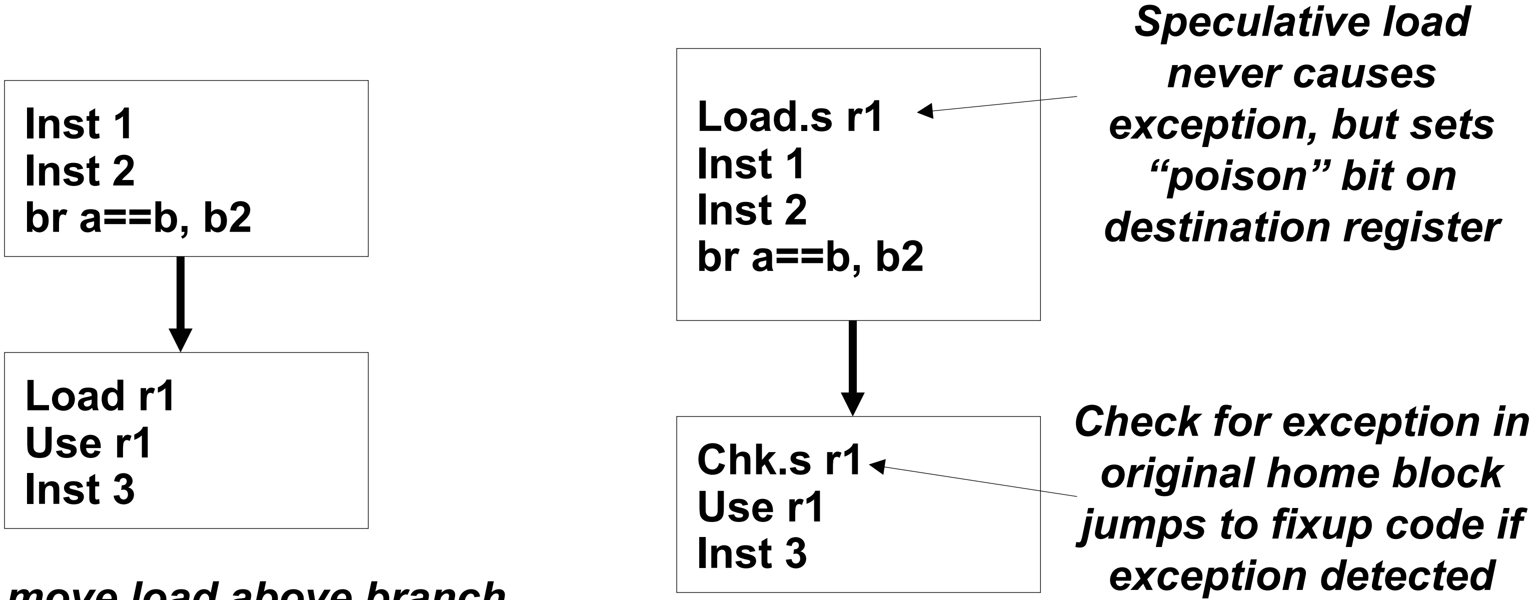
One basic block

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

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

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

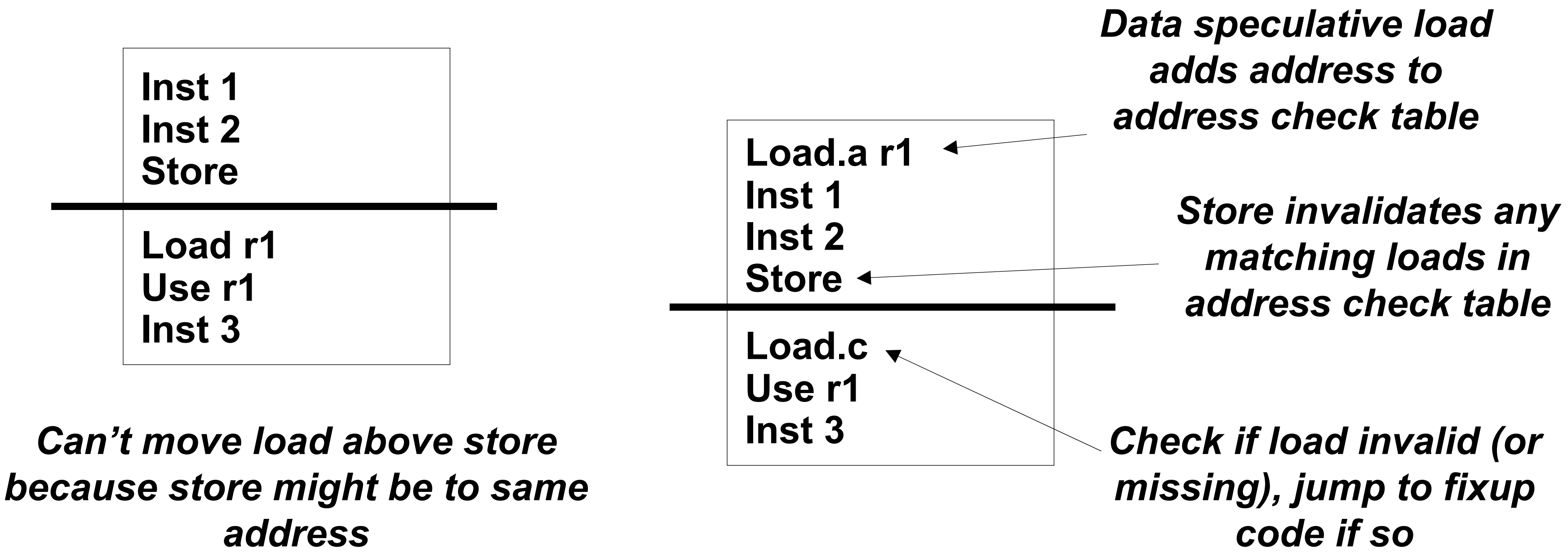
Can't move load above branch because might cause spurious exception

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

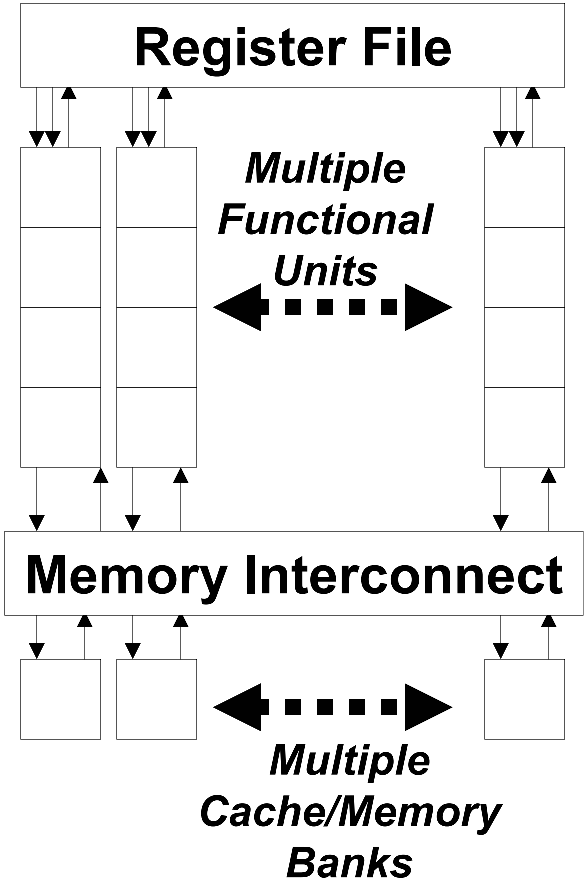
Data speculative load adds address to address check table

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

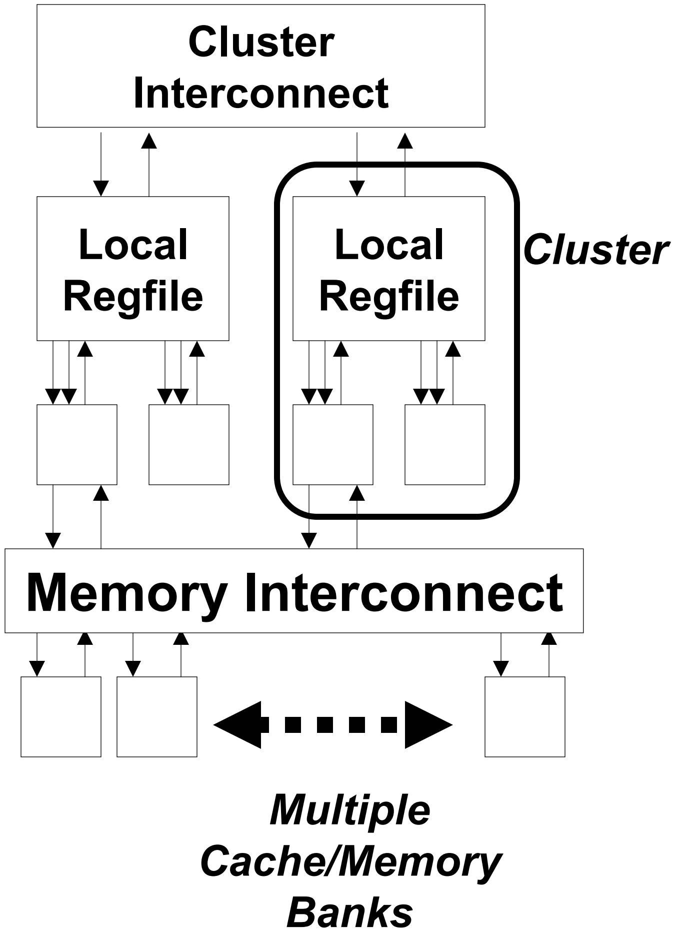
ILP Datapath Hardware Scaling



- Replicating functional units and cache/memory banks is straightforward and scales linearly
- Register file ports and bypass logic for N functional units scale quadratically (N*N)
- Memory interconnection among N functional units and memory banks also scales quadratically
- *(For large N, could try O(N logN) interconnect schemes)*
- Technology scaling: Wires are getting even slower relative to gate delays
- Complex interconnect adds latency as well as area

=> *Need greater parallelism to hide latencies*

Clustered VLIW



- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead

Limits of Static Scheduling

Four major weaknesses of static scheduling:

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity