VLIW/EPIC: Statically Scheduled ILP

Joel Emer
Computer Science & Artificial Intelligence Lab
M.I.T.

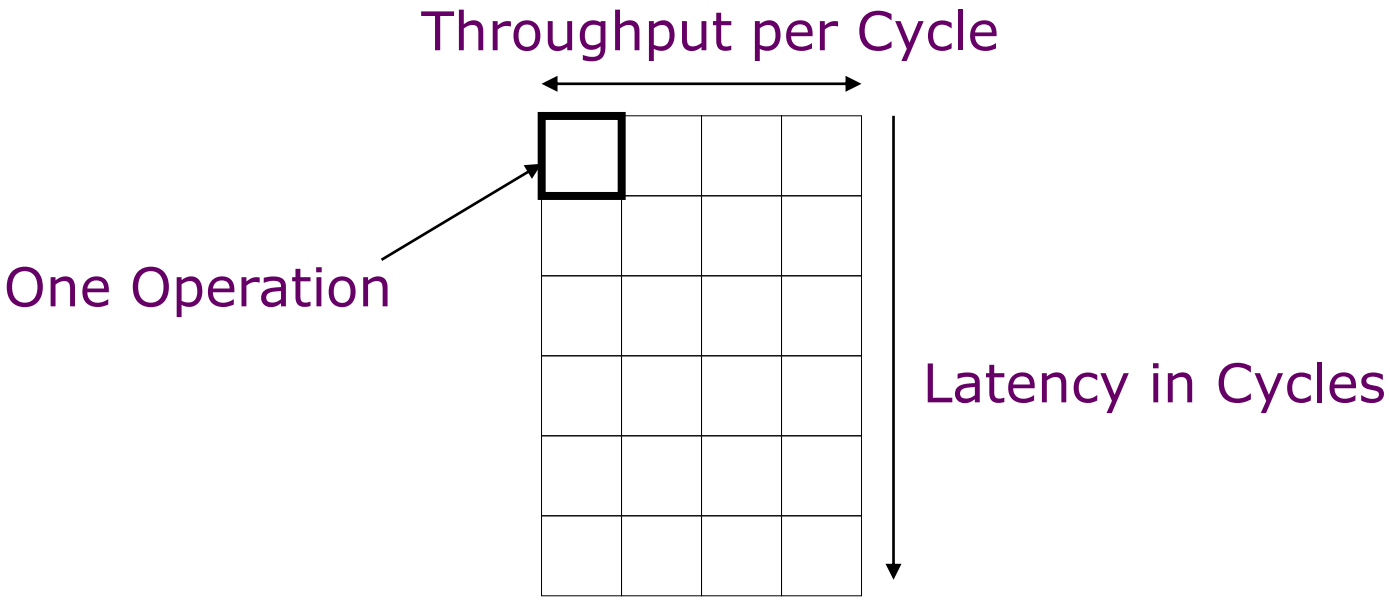
http://www.csg.csail.mit.edu/6.823

Little's Law

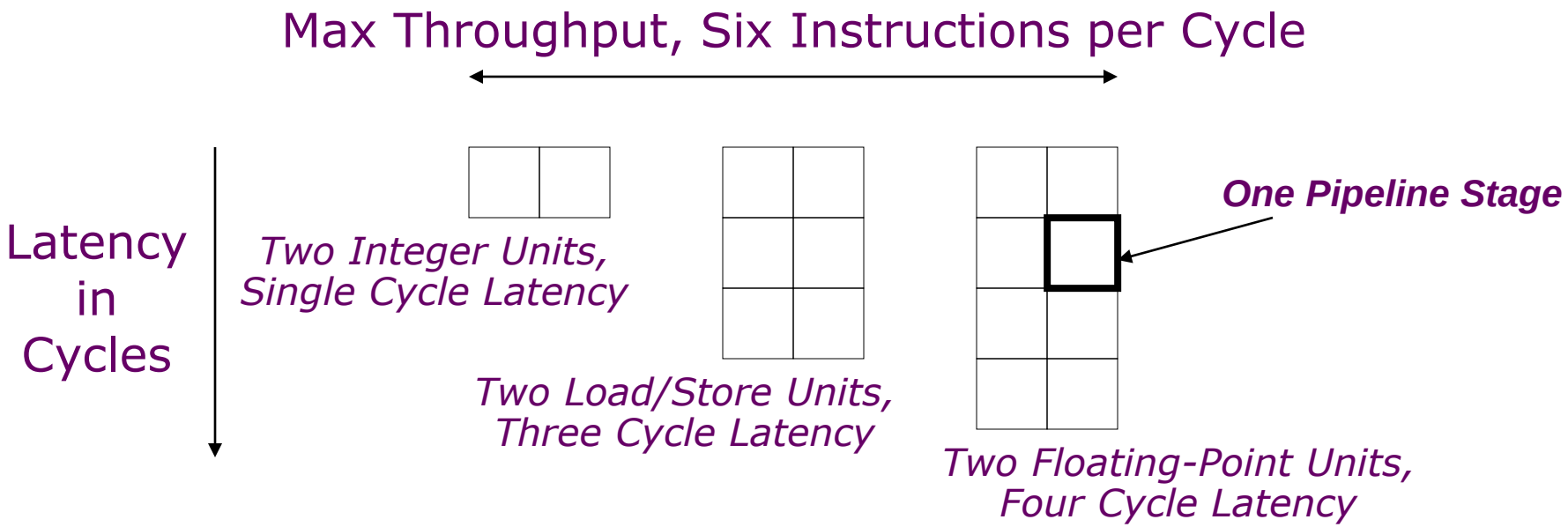
*Parallelism = Throughput * Latency*

or

$$\overline{N} = \overline{T} \times \overline{L}$$



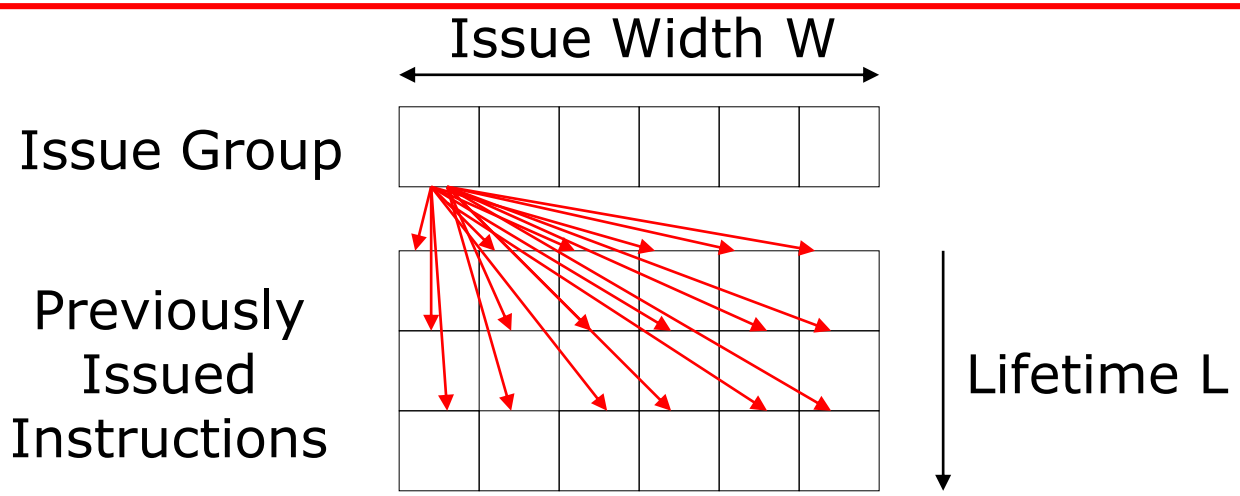
Example Pipelined ILP Machine



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

$$\overline{T} = 6 \qquad \overline{L} = \frac{(2x1 + 2x3 + 2x4)}{6} = 2\frac{2}{3} \qquad \overline{N} = 6 \times 2\frac{2}{3} = 16$$

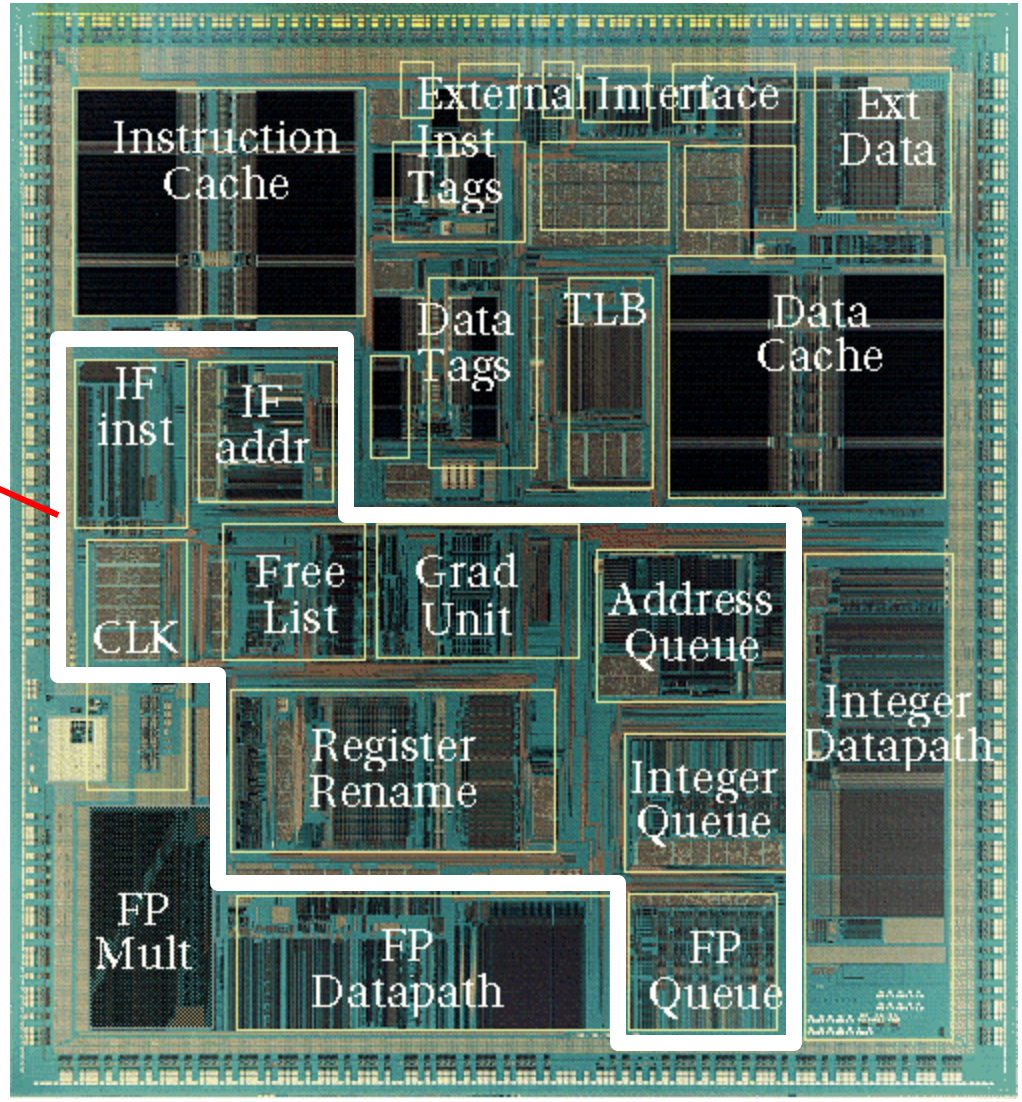
Superscalar Control Logic Scaling



- Each issued instructions must make interlock checks against W*L instructions, i.e., growth in interlocks $\propto$ W*(W*L)
- For in-order machines, L is related to pipeline latencies
- For out-of-order machines, L also includes time spent in instruction buffers (instruction window or ROB)
- As W increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater L

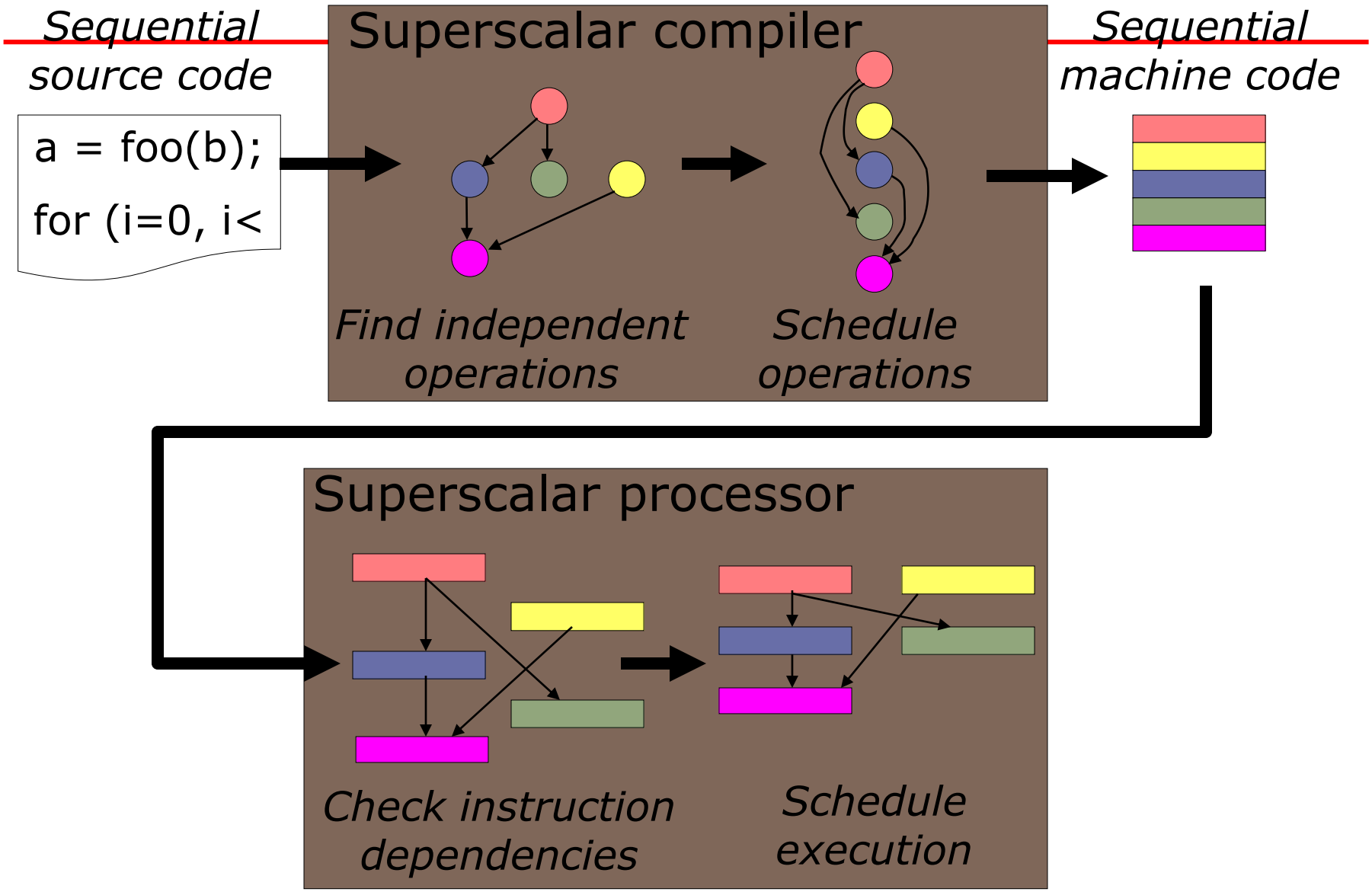
  => Out-of-order control logic grows faster than W^2 ($\sim W^3$)

Out-of-Order Control Complexity: MIPS R10000

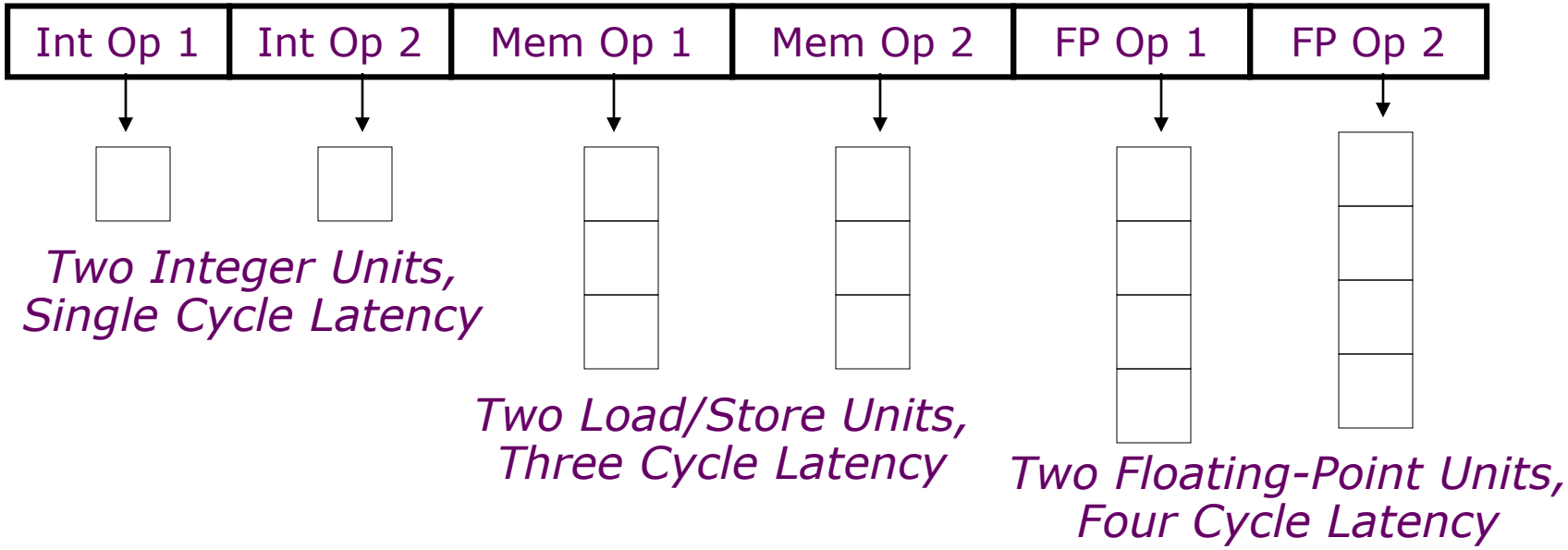


[SGI/MIPS Technologies Inc., 1995]

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 x-operation RAW check
  - No data use before data ready => no data interlocks

VLIW Compiler Responsibilities

The compiler:

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

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

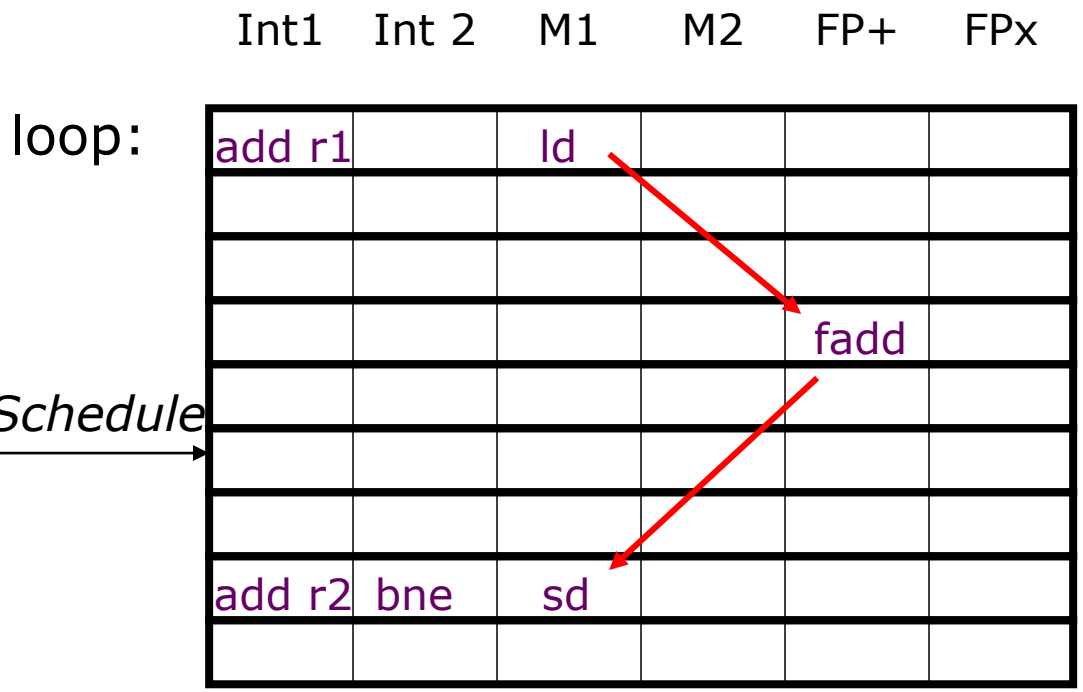
Loop Execution

```
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



How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

Loop Unrolling

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



```
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;
}
```

Unroll inner loop to perform 4 iterations at once

Is this code correct?

No, 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 →

loop:

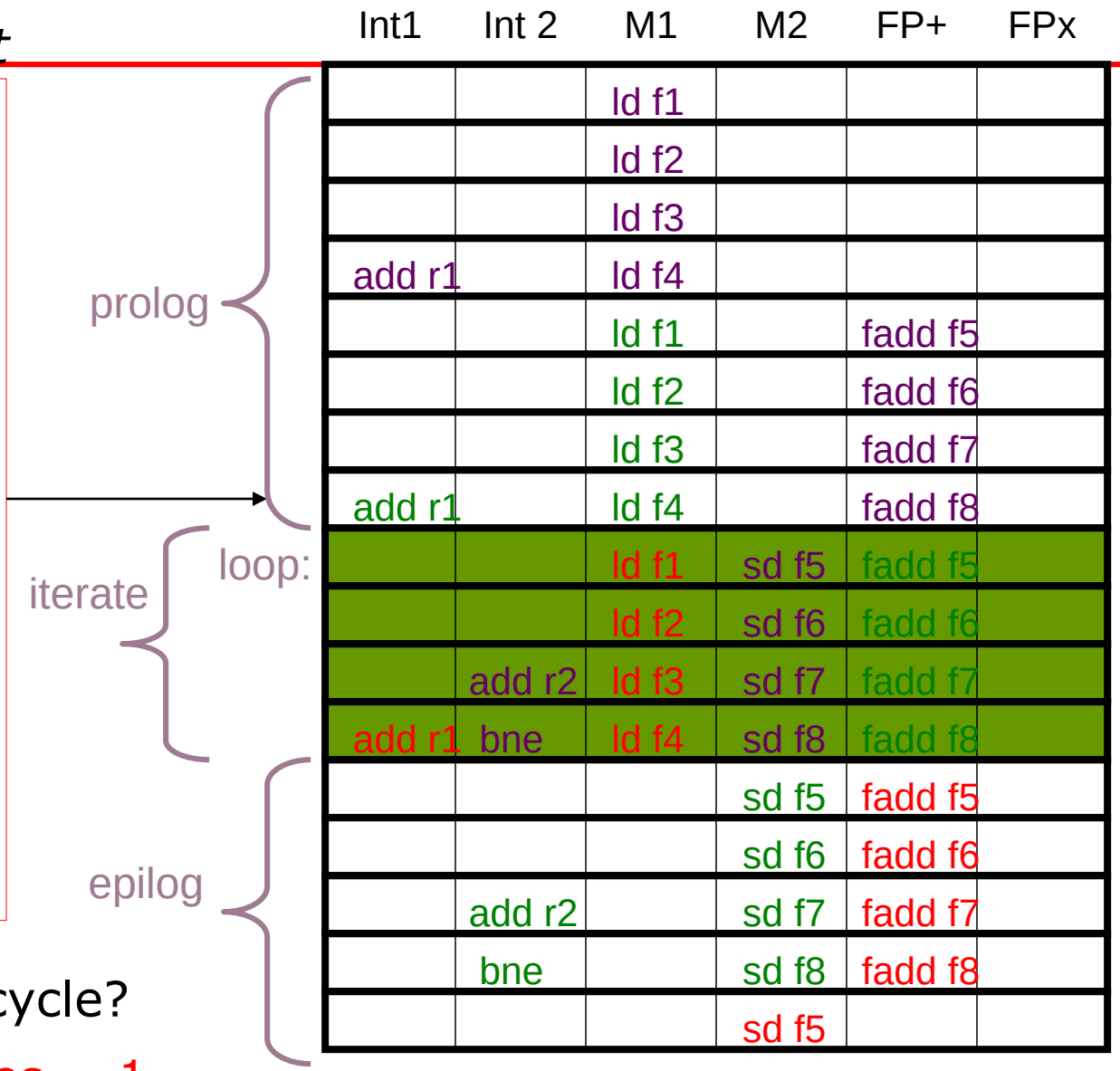
Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

How many FLOPS/cycle? 4 fadds / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

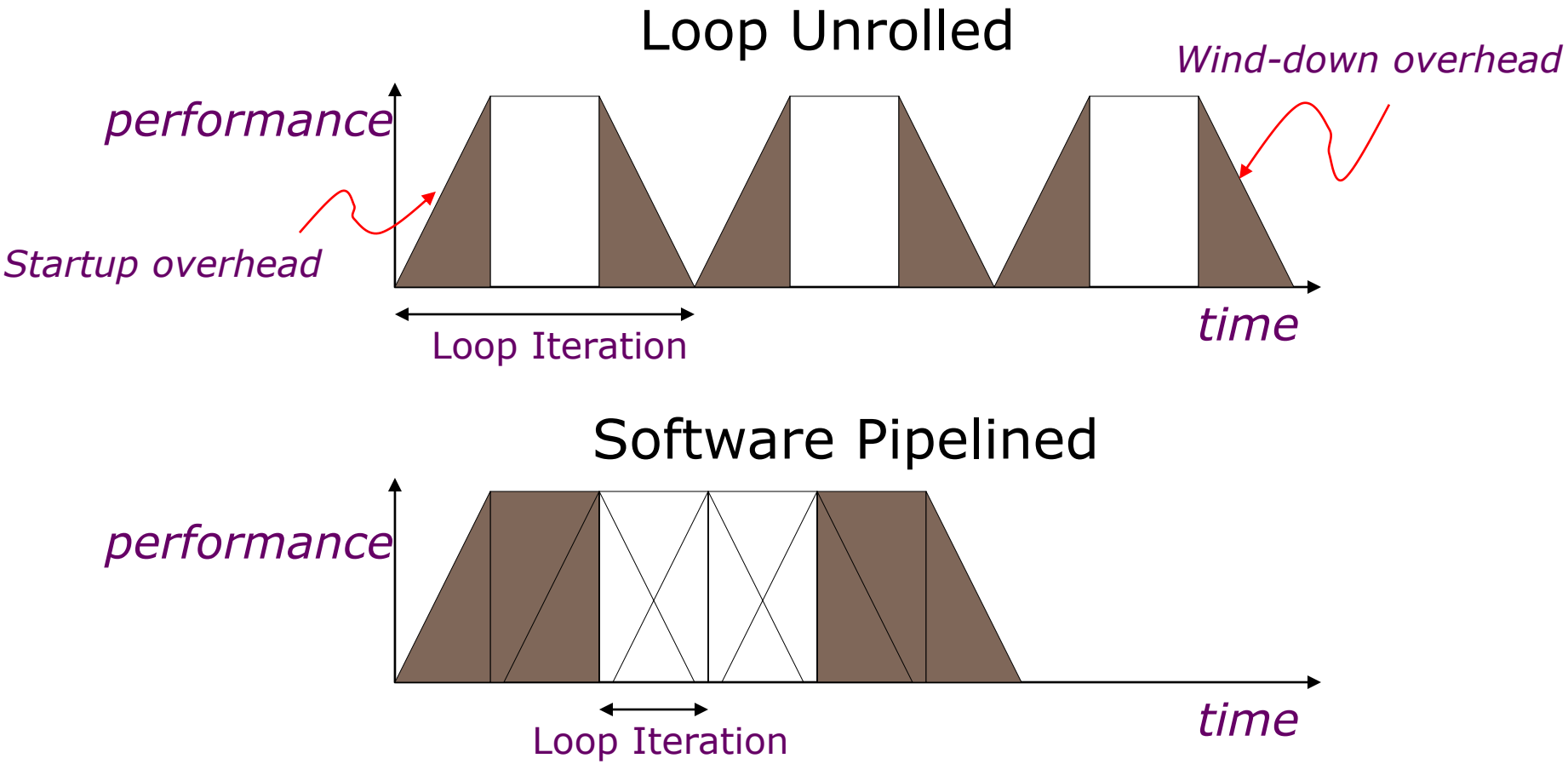
```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop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



How many FLOPS/cycle?

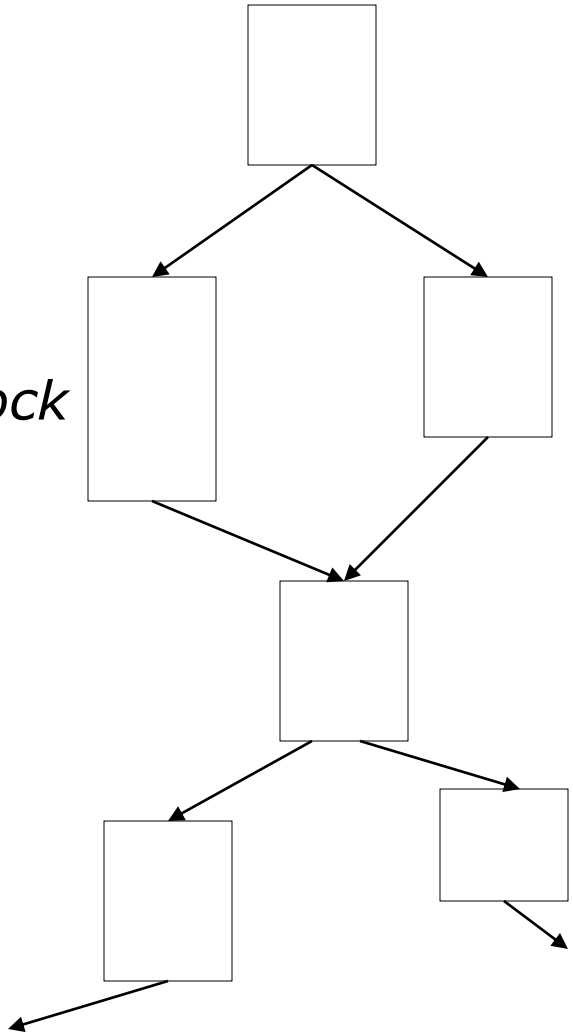
4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling



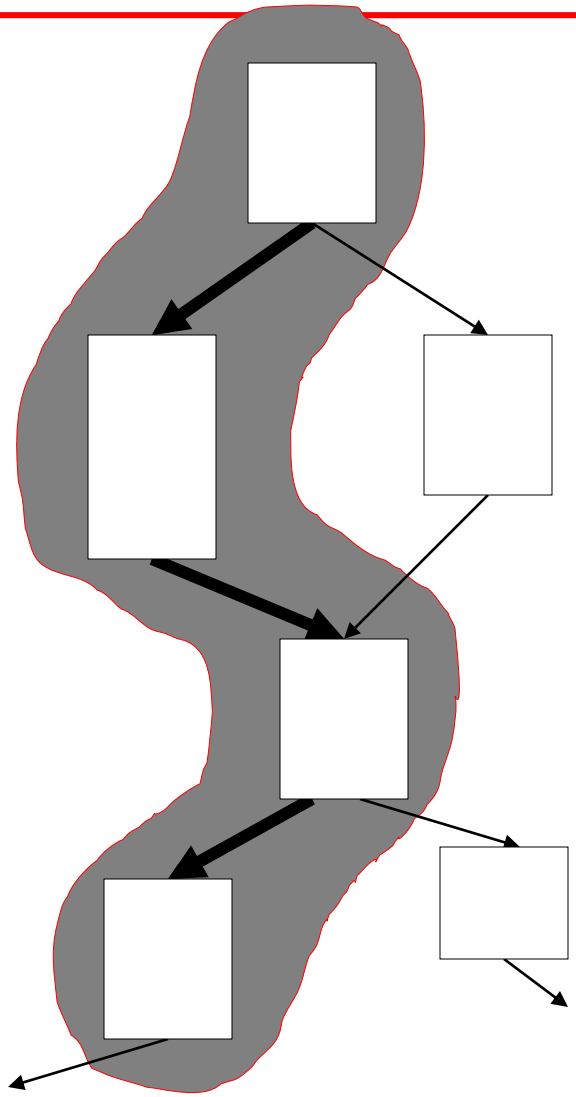
Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

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
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

How do we know which trace to pick?

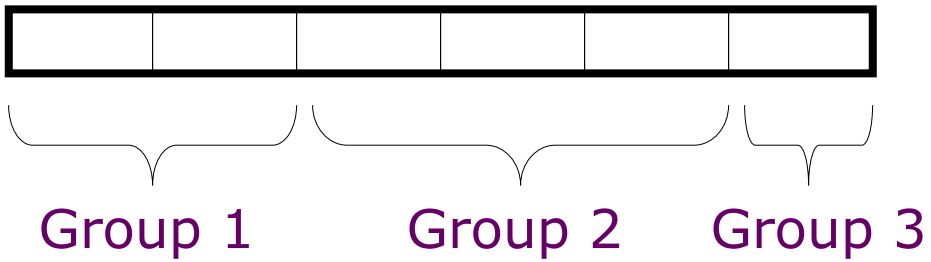
Use profiling feedback or compiler heuristics to find common branch paths

Problems with "Classic" VLIW

- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
- 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 for statically unpredictable branches
  - optimal schedule varies with branch path
- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation

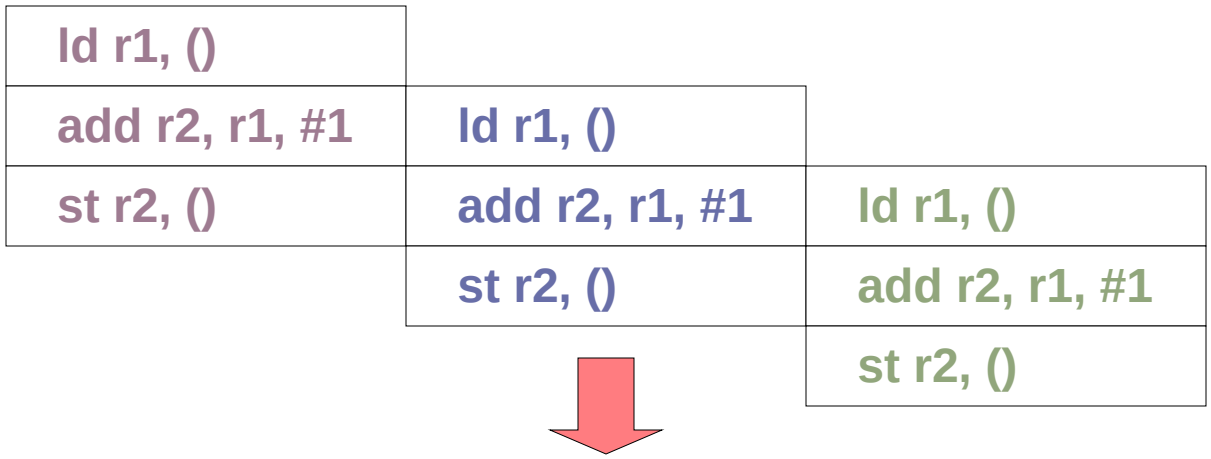
VLIW Instruction Encoding

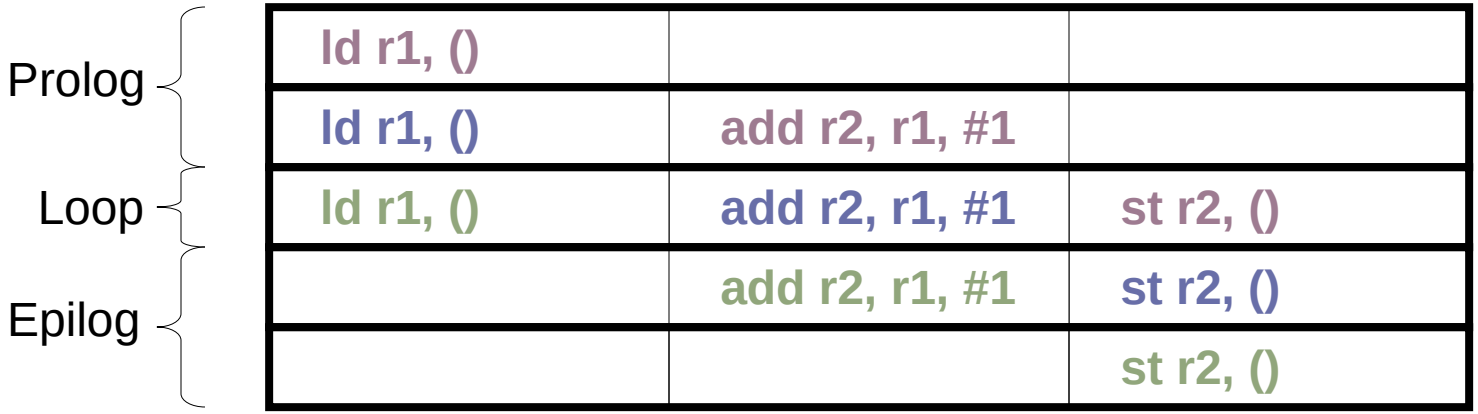
- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions
  - Mark parallel groups
    - used in TMS320C6x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



Rotating Register Files

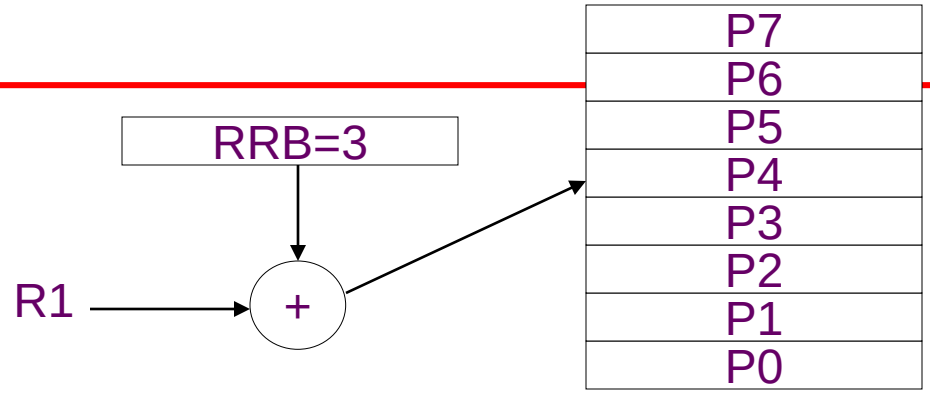
Problems: Scheduled loops require lots of register names,
Lots of duplicated code in prolog, epilog





Solution: Allocate new set of registers for each loop iteration

Rotating Register File

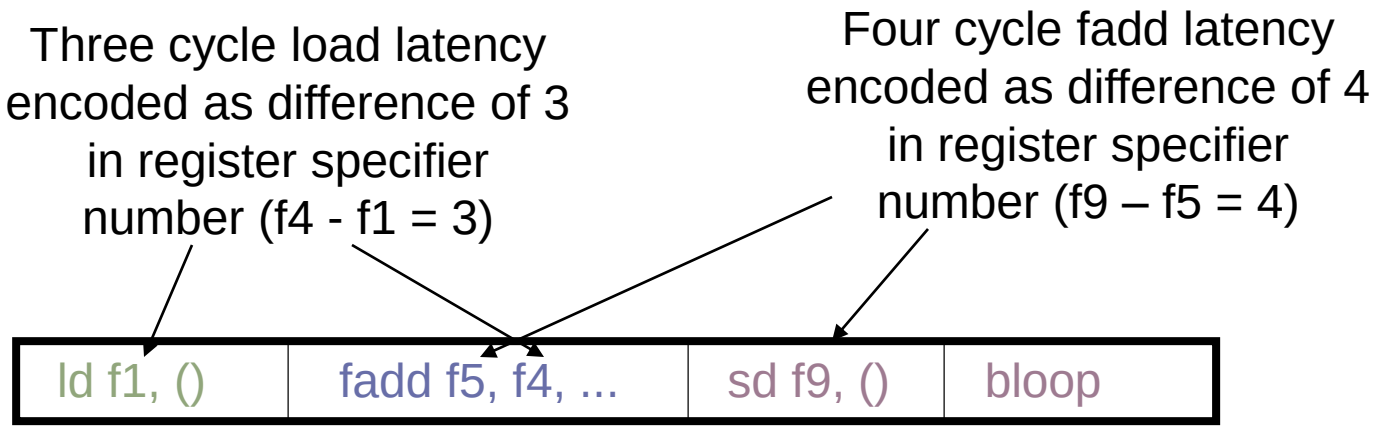


Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

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

Loop closing branch decrements RRB

Rotating Register File
(Previous Loop Example)



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

Cydra-5: Memory Latency Register (MLR)

Problem: Loads have variable latency

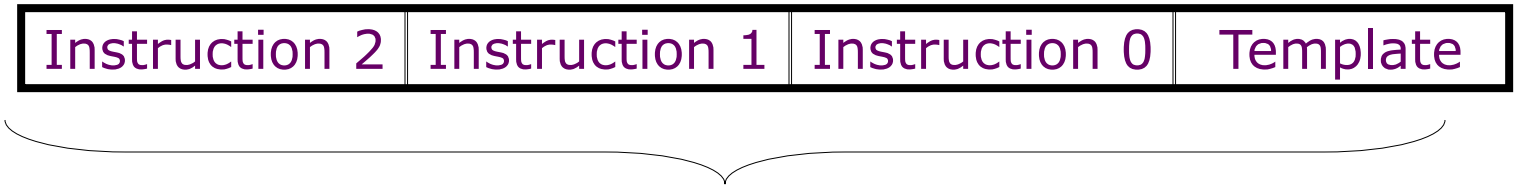
Solution: Let software choose desired memory latency

- Compiler schedules 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

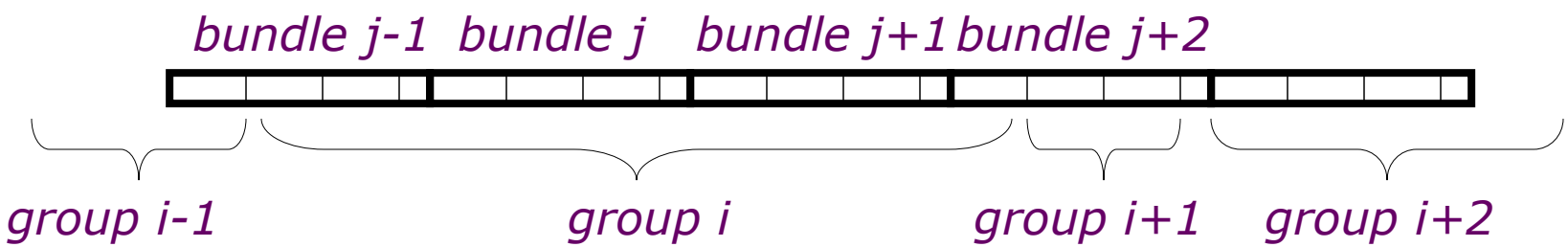
Intel EPIC IA-64

- 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 expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002

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

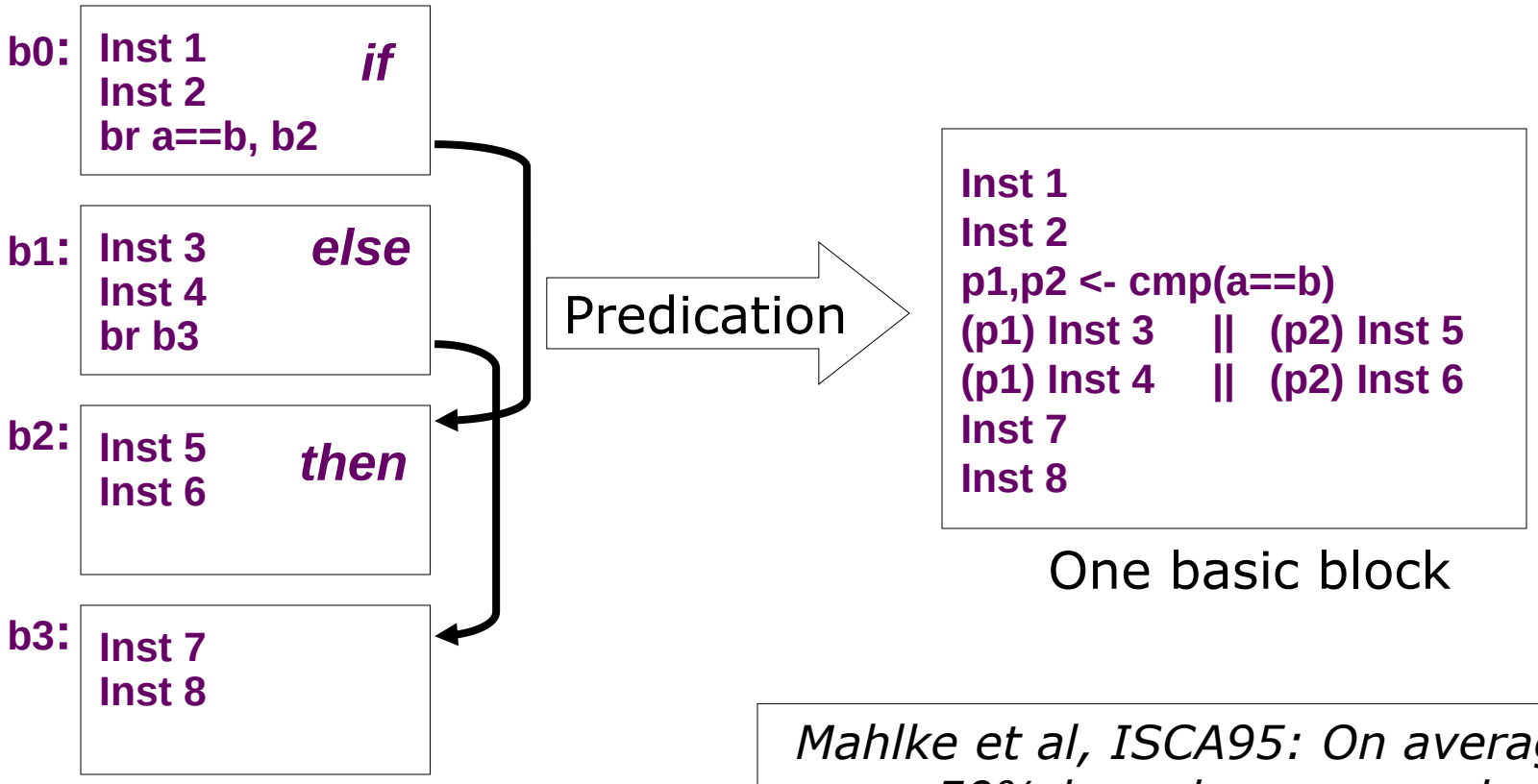
- GPRs rotate to reduce code size for software pipelined loops

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

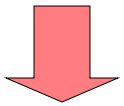


Mahlke et al, ISCA95: On average >50% branches removed

Predicate Software Pipeline Stages

Single VLIW Instruction

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

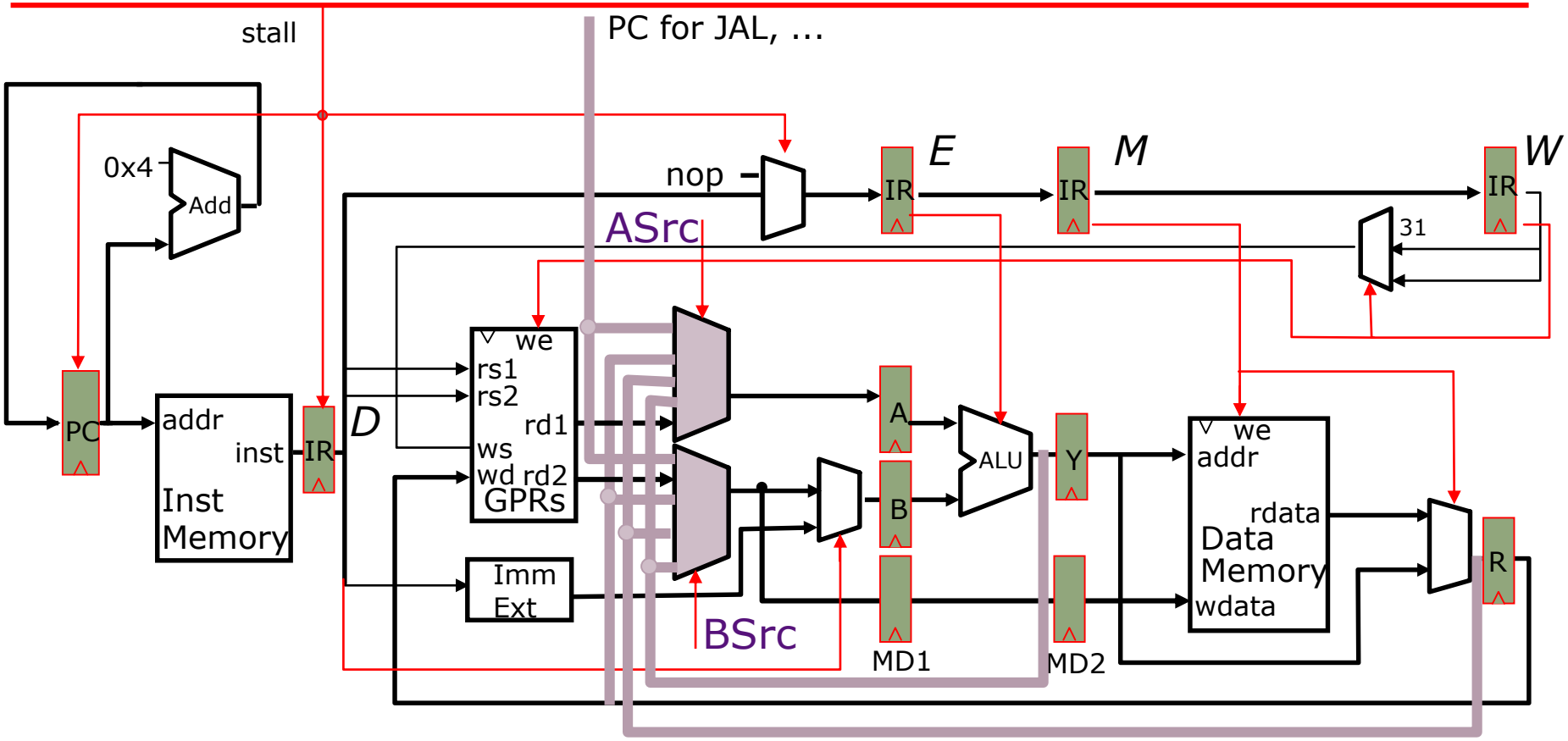


Dynamic Execution

(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

Fully Bypassed Datapath

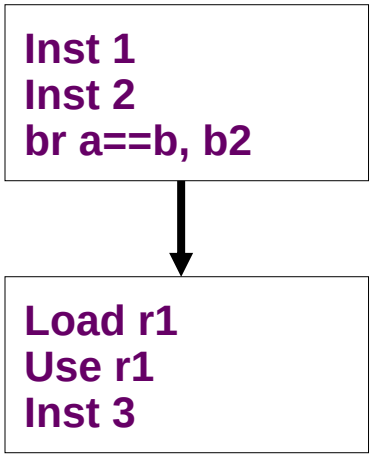


Where does predication fit in?

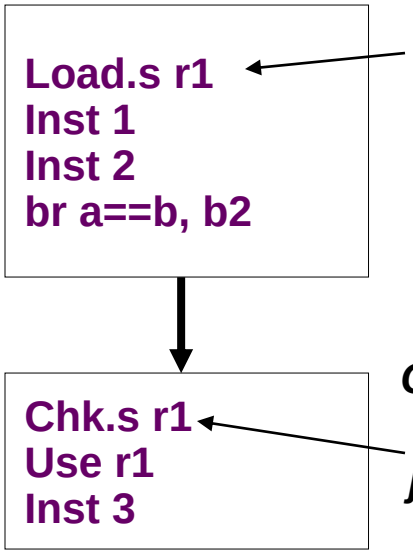
IA-64 Speculative Execution

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

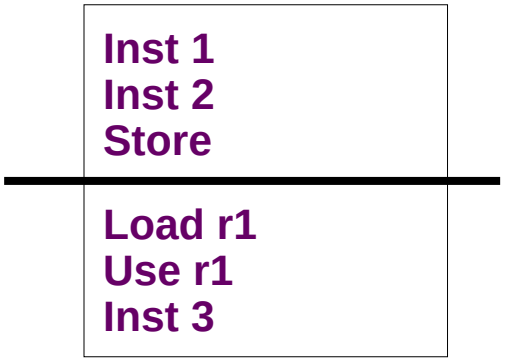
Check for exception in original home block jumps to fixup code if exception detected

Particularly useful for scheduling long latency loads early

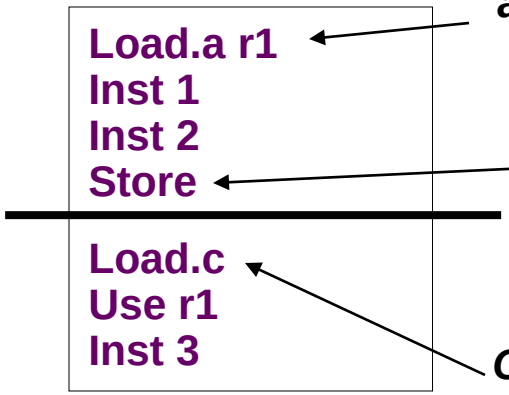
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Instruction-based speculation with hardware monitor to check for pointer hazards



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



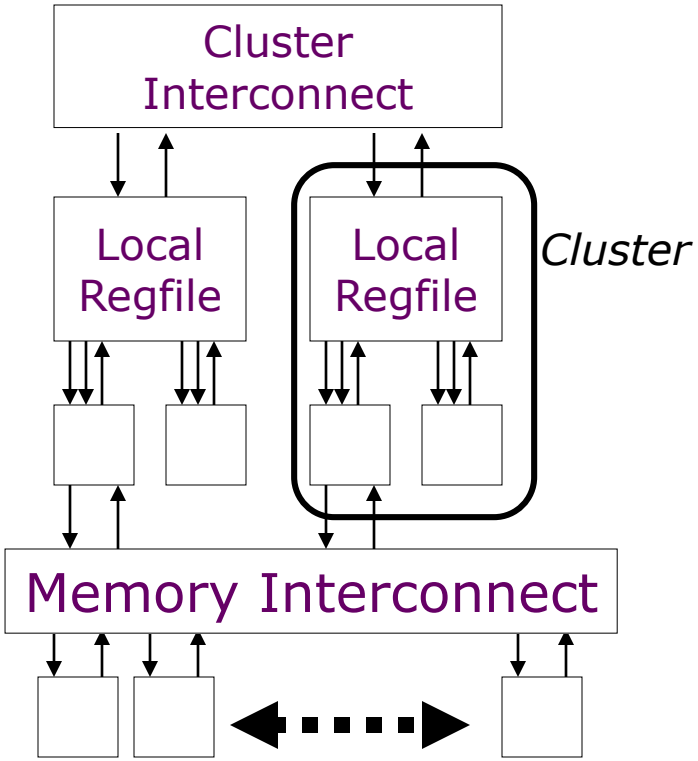
Data speculative load adds address to address check table

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

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
- Exists in some superscalar processors, .e.g., Alpha 21264
- Common in commercial embedded processors, examples include TI C6x series DSPs, and HP Lx processor

Limits of Static Scheduling

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

Question:

How applicable are the VLIW-inspired techniques to traditional RISC/CISC processor architectures?

Thank you !

http://www.csg.csail.mit.edu/6.823