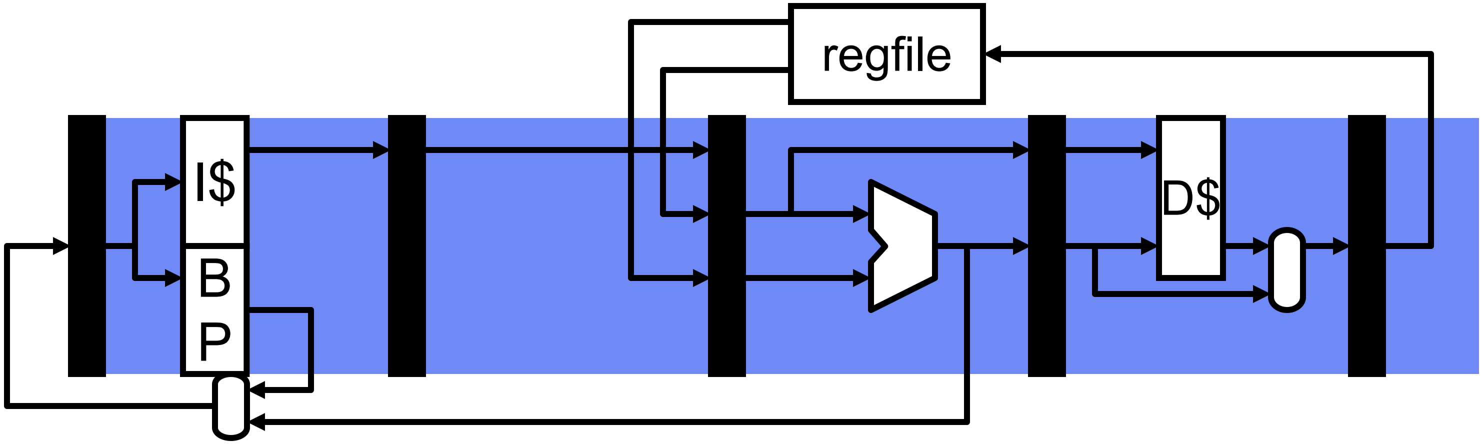
Computer Architecture

Lec 9: Superscalar VS. VLIW

Part I: Superscalar

“Scalar” Pipeline & the Flynn Bottleneck



- So far we have looked at **scalar pipelines**
  - One instruction per stage
    - With control speculation, bypassing, etc.
  - – Performance limit (aka “Flynn Bottleneck”) is CPI = IPC = 1
  - – Limit is not achievable (due to hazards)
  - – Diminishing returns from “super-pipelining” (hazards + overhead)

An Opportunity…

- But consider:

```
ADD r1, r2 -> r3
ADD r4, r5 -> r6
```

  - Why not execute them ***at the same time***? (We can!)

- What about:

```
SUB r1, r2 -> r3
SUB r4, r3 -> r6
```

  - In this case, ***dependences*** prevent parallel execution

- What about three (or more!) instructions at a time?

What Checking Is Required?

- For two instructions: 2 checks

```
ADD src1$_1$, src2$_1$ -> dest$_1$
ADD src1$_2$, src2$_2$ -> dest$_2$   (2 checks)
```

- For three instructions: 6 checks

```
ADD src1$_1$, src2$_1$ -> dest$_1$
ADD src1$_2$, src2$_2$ -> dest$_2$   (2 checks)
ADD src1$_3$, src2$_3$ -> dest$_3$   (4 checks)
```

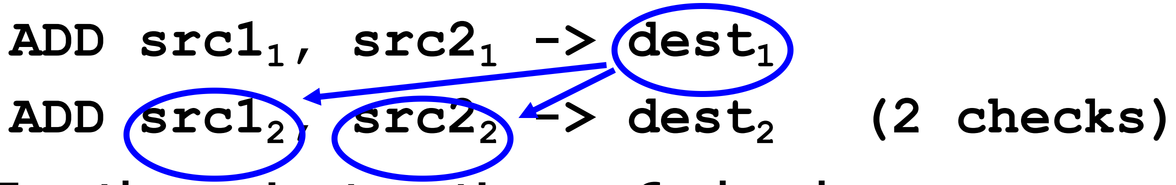
- For four instructions: 12 checks

```
ADD src1$_1$, src2$_1$ -> dest$_1$
ADD src1$_2$, src2$_2$ -> dest$_2$   (2 checks)
ADD src1$_3$, src2$_3$ -> dest$_3$   (4 checks)
ADD src1$_4$, src2$_4$ -> dest$_4$   (6 checks)
```

- Plus checking for load-to-use stalls from prior *n* loads

What Checking Is Required?

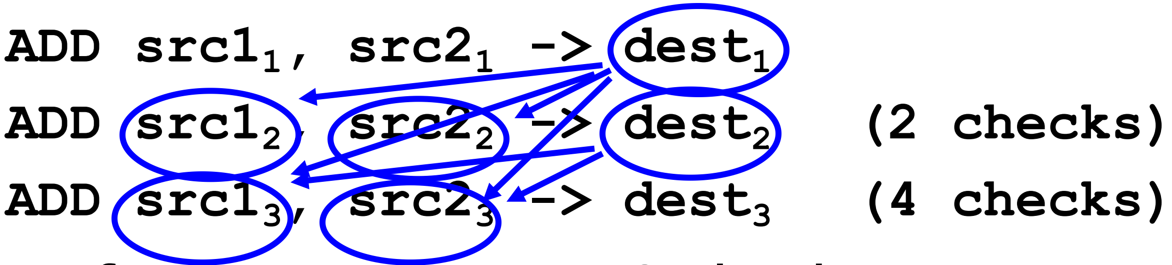
- For two instructions: 2 checks



ADD src1$_1$, src2$_1$ -> dest$_1$

ADD src1$_2$, src2$_2$ -> dest$_2$ (2 checks)

- For three instructions: 6 checks

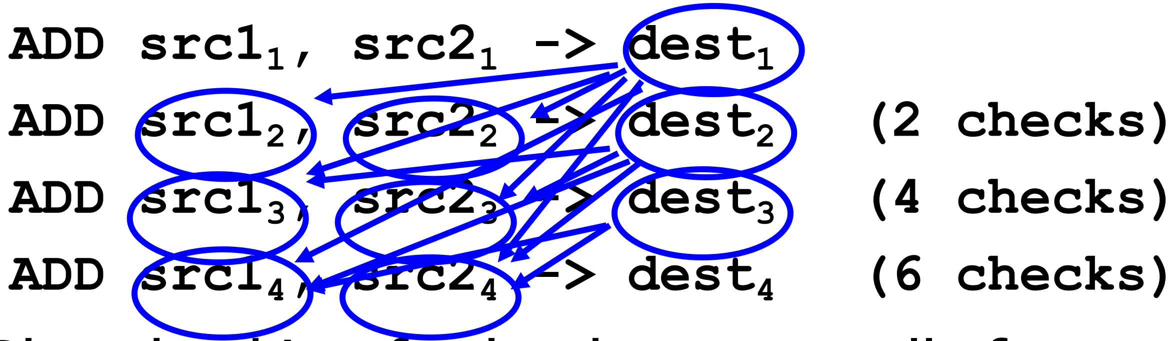


ADD src1$_1$, src2$_1$ -> dest$_1$

ADD src1$_2$, src2$_2$ -> dest$_2$ (2 checks)

ADD src1$_3$, src2$_3$ -> dest$_3$ (4 checks)

- For four instructions: 12 checks



ADD src1$_1$, src2$_1$ -> dest$_1$

ADD src1$_2$, src2$_2$ -> dest$_2$ (2 checks)

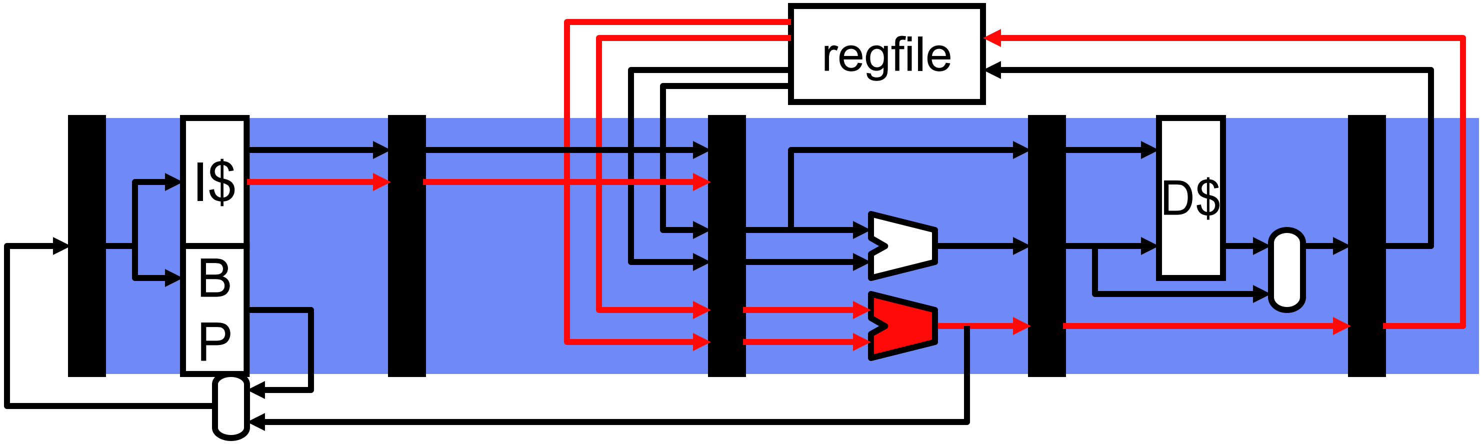
ADD src1$_3$, src2$_3$ -> dest$_3$ (4 checks)

ADD src1$_4$, src2$_4$ -> dest$_4$ (6 checks)

- Plus checking for load-to-use stalls from prior *n* loads

How do we build such “superscalar” hardware?

Multiple-Issue or “Superscalar” Pipeline



- Overcome this limit using **multiple issue**
  - Also called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight…
  - **“Instruction-Level Parallelism (ILP)”** [Fisher, IEEE TC’81]
- Today, typically “4-ish-wide” (Intel Broadwell, AMD Ryzen)
  - Broadwell issues up to 8 in the right circumstances, Ryzen up to 6
  - ARM cores usually issue less

How Much ILP is There?

- The compiler tries to “schedule” code to avoid stalls
    - Even for scalar machines (to fill load-use delay slot)
    - Even harder to schedule multiple-issue (superscalar)
- How much ILP is common?
    - Greatly depends on the application
        - Consider memory copy
        - Unroll loop, lots of independent operations
    - Other programs, less so
- Even given unbounded ILP,
  superscalar has implementation limits
    - IPC (or CPI) vs clock frequency trade-off
    - Given these challenges, what is reasonable today?
        - ~4 instruction per cycle maximum

Superscalar Pipeline Diagrams - Ideal

scalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`		F	D	X	M	W						
`lw 8(r1)→r4`			F	D	X	M	W					
`add r14,r15→r6`				F	D	X	M	W				
`add r12,r13→r7`					F	D	X	M	W			
`add r17,r16→r8`						F	D	X	M	W		
`lw 0(r18)→r9`							F	D	X	M	W	

2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`	F	D	X	M	W							
`lw 8(r1)→r4`		F	D	X	M	W						
`add r14,r15→r6`		F	D	X	M	W						
`add r12,r13→r7`			F	D	X	M	W					
`add r17,r16→r8`			F	D	X	M	W					
`lw 0(r18)→r9`				F	D	X	M	W				

Superscalar Pipeline Diagrams - Realistic

scalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`		F	D	X	M	W						
`lw 8(r1)→r4`			F	D	X	M	W					
`add r4,r5→r6`				F	D	d*	X	M	W			
`add r2,r3→r7`					F	d*	D	X	M	W		
`add r7,r6→r8`							F	D	X	M	W	
`lw 4(r8)→r9`								F	D	X	M	W

2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`	F	D	X	M	W							
`lw 8(r1)→r4`		F	D	X	M	W						
`add r4,r5→r6`		F	D	d*	d*	X	M	W				
`add r2,r3→r7`			F	D	d*	X	M	W				
`add r7,r6→r8`			F	d*	d*	D	X	M	W			
`lw 4(r8)→r9`				F	d*	D	d*	X	M	W		

Superscalar Implementation Challenges

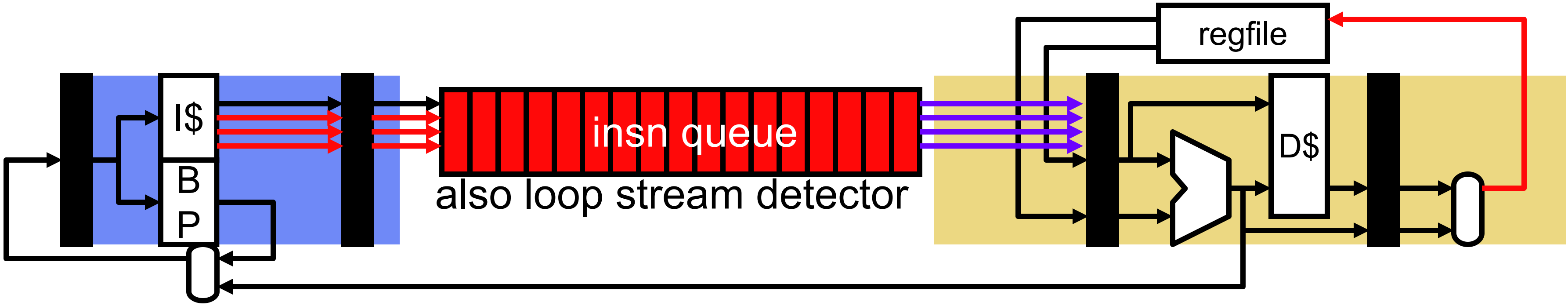
Superscalar Challenges - Front End

- **Superscalar instruction fetch**
  - Modest: fetch multiple instructions per cycle
  - Aggressive: buffer instructions and/or predict multiple branches
- **Superscalar instruction decode**
  - Replicate decoders
- **Superscalar instruction issue**
  - Determine when instructions can proceed in parallel
  - More complex stall logic - order **N^2** for *N*-wide machine
  - Not all combinations of types of instructions possible
- **Superscalar register read**
  - Port for each register read (4-wide superscalar → 8 read "ports")
  - Each port needs its own set of address and data wires
    - Latency & area $\propto$ #ports^2

Challenges of Superscalar Fetch

- What is involved in fetching multiple instructions per cycle?
- In same cache block? no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
  - Fetch only one instruction that cycle
  - Or, some processors may allow fetching from 2 consecutive blocks
- What about taken branches?
  - How many instructions can be fetched on average?
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken → ~10 instructions
- Consider a 5-instruction loop with an 4-issue processor
  - Without smarter fetch, ILP is limited to 2.5 (not 4, which is bad)

Increasing Superscalar Fetch Rate

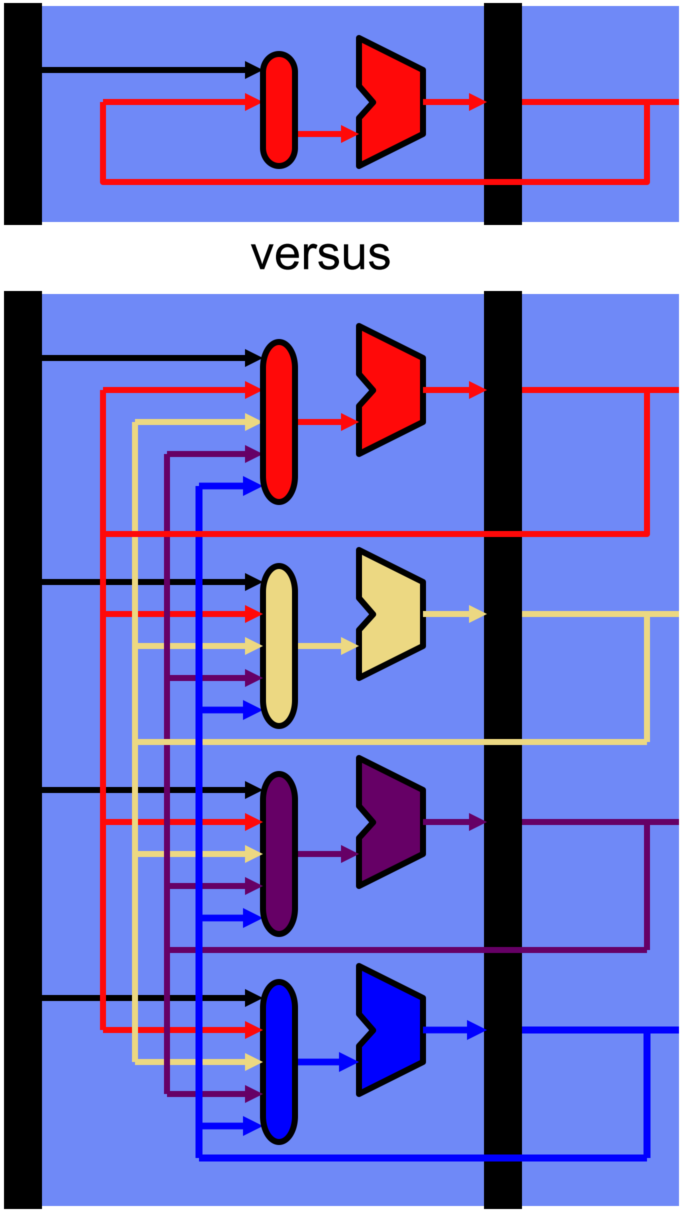


- Option #1: over-fetch and buffer
  - Add a queue between fetch and decode (18 entries in Intel Core2)
  - Compensates for cycles that fetch less than maximum instructions
  - "decouples" the "front end" (fetch) from the "back end" (execute)
- Option #2: "loop stream detector" (Core 2, Core i7)
  - Put entire loop body into a small cache
    - Core2: 18 macro-ops, up to four taken branches
    - Core i7: 28 micro-ops (avoids re-decoding macro-ops!)
  - Any branch mis-prediction requires normal re-fetch
- Other options: next-*next*-block prediction, "trace cache"

Superscalar Challenges - Back End

- **Superscalar instruction execution**
  - Replicate arithmetic units (but not all, for example, integer divider)
  - Perhaps multiple cache ports (slower access, higher energy)
    - Only for 4-wide or larger (why? only ~35% are load/store insn)
- **Superscalar bypass paths**
  - More possible sources for data values
  - Order (N^2 * P) for *N*-wide machine with execute pipeline depth *P*
- **Superscalar instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar ➔ 4 write ports
- **Fundamental challenge:**
  - Amount of ILP (instruction-level parallelism) in the program
  - Compiler must schedule code and extract parallelism

Superscalar Bypass

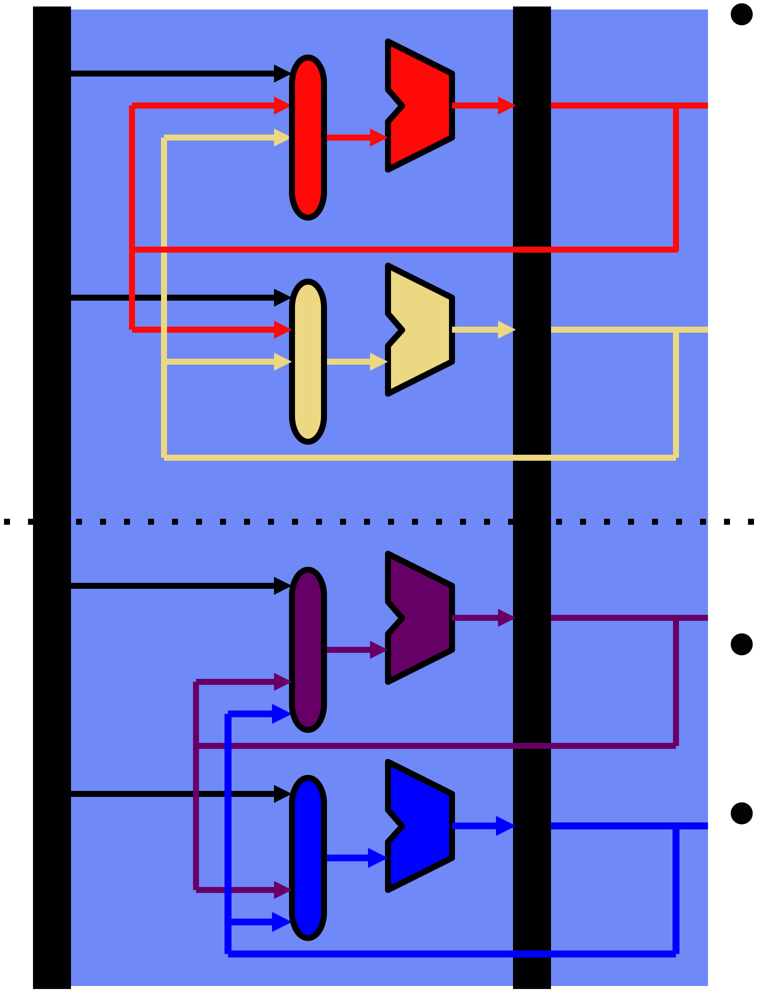


- **N^2 bypass network**
  - N+1 input muxes at each ALU input
  - N^2 point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load

- And this is just one bypass stage (MX)!
  - There is also WX bypassing
  - Even more for deeper pipelines

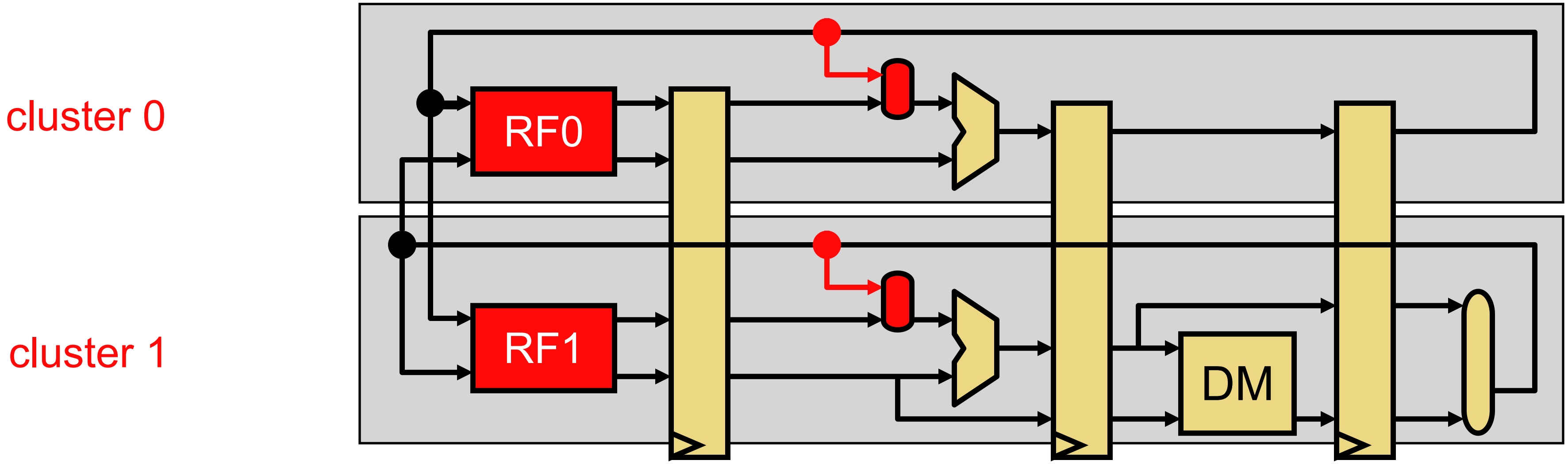
- One of the big problems of superscalar
  - Why? On the critical path of single-cycle “bypass & execute” loop

Mitigating N² Bypass & Register File



- **Clustering**: mitigates N² bypass
  - Group ALUs into **K** clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - **With 1 or 2 cycle delay**
    - Can hurt IPC, but faster clock
  - (N/K) + 1 inputs at each mux
  - $(N/K)^2$ bypass paths in each cluster
- **Steering**: key to performance
  - Steer dependent insns to same cluster
- **Cluster register file**, too
  - Replicate a register file per cluster
  - All register writes update all replicas
  - Fewer read ports; only for cluster

Mitigating N^2 RegFile with Clustering

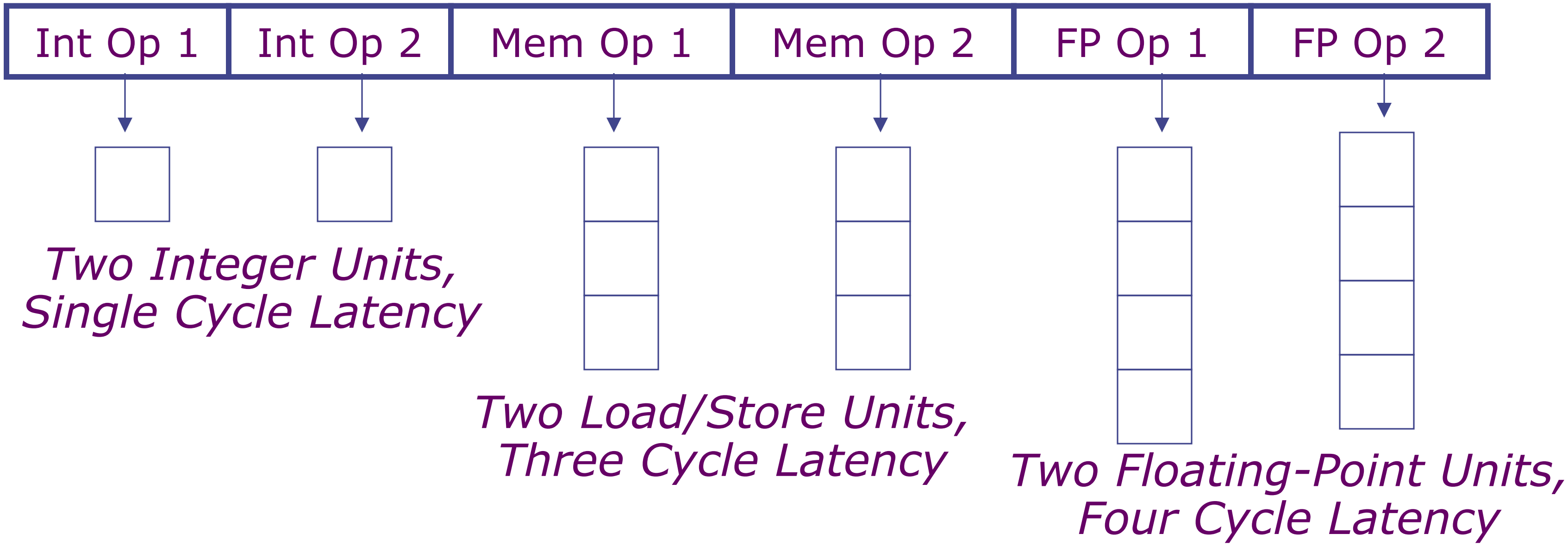


- **Clustering**: split **N**-wide execution pipeline into **K** clusters
  - With centralized register file, 2N read ports and N write ports

- **Clustered register file**: extend clustering to register file
  - Replicate the register file (one replica per cluster)
  - Register file supplies register operands to just its cluster
  - All register writes go to all register files (keep them in sync)
  - Advantage: fewer read ports per register!
    - K register files, each with 2N/K read ports and N write ports

Part II: VLIW

VLIW: Very Long Instruction Word



Int Op 1 | Int Op 2 | Mem Op 1 | Mem Op 2 | FP Op 1 | FP Op 2

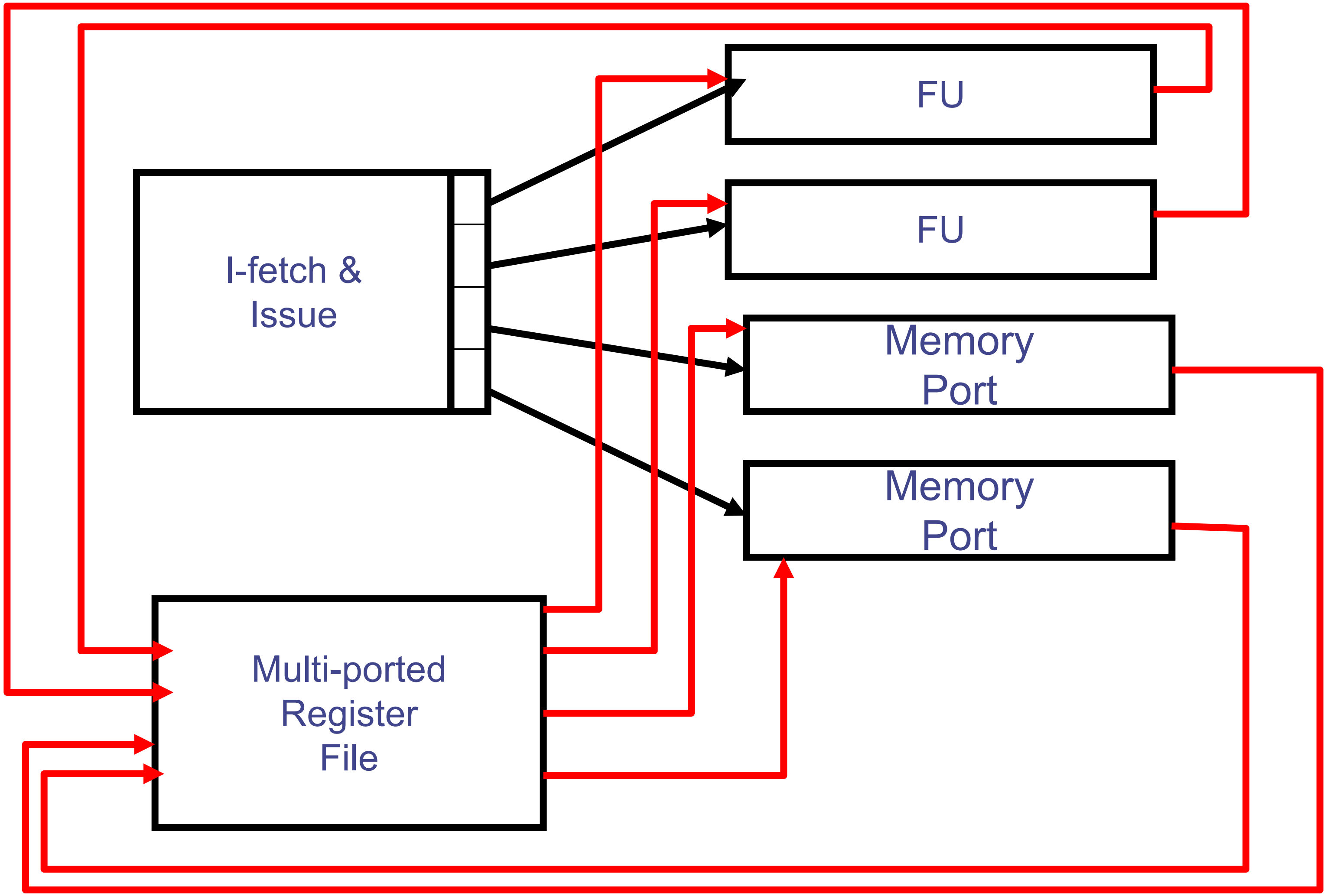
Two Integer Units, Single Cycle Latency

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 Example



VLIW Example

Instruction format

ALU1	ALU2	MEM1	control

Program order and execution order

ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control
ALU1	ALU2	MEM1	control

- Instructions in a VLIW are **independent**
- Latencies are fixed in the architecture spec.
- Hardware does not check anything
- Software has to schedule so that all works

VLIW Compiler Responsibilities

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

Loop Execution

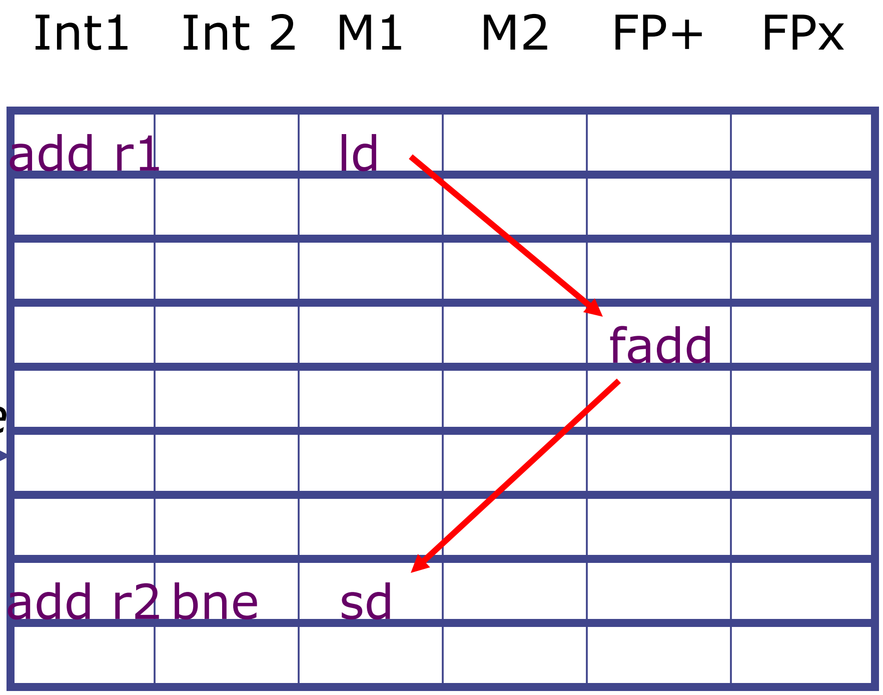
```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Compile

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

Schedule

loop:

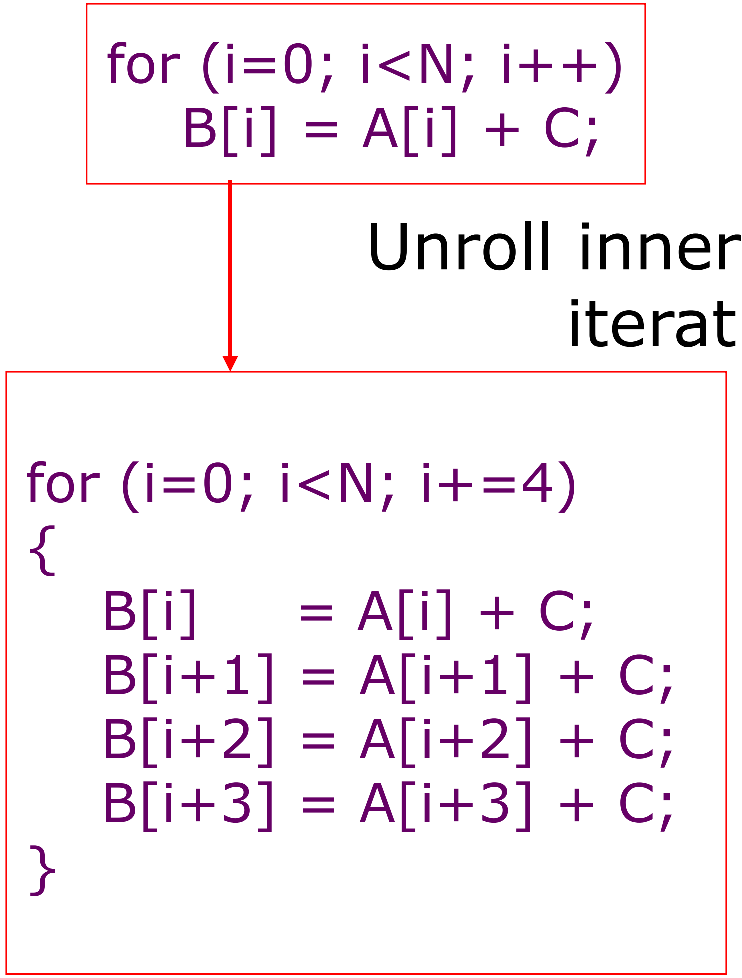


Int1	Int 2	M1	M2	FP+	FPx
add r1		ld			
				fadd	
add r2	bne	sd			

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:			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
```

	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	
		add r2		sd f6	fadd f6	
		bne		sd f7	fadd f7	
				sd f8	fadd f8	
				sd f5		

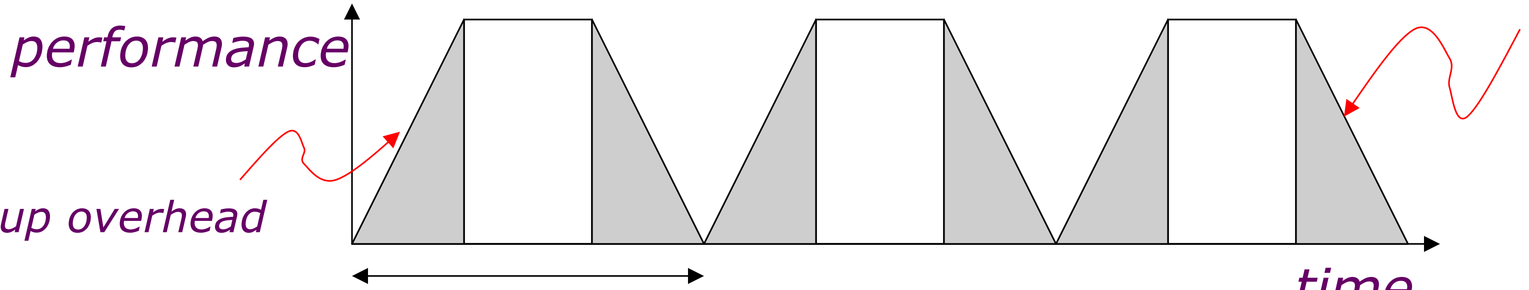
How many FLOPS/cycle?

4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling

Loop Unrolled

Wind-down overhead



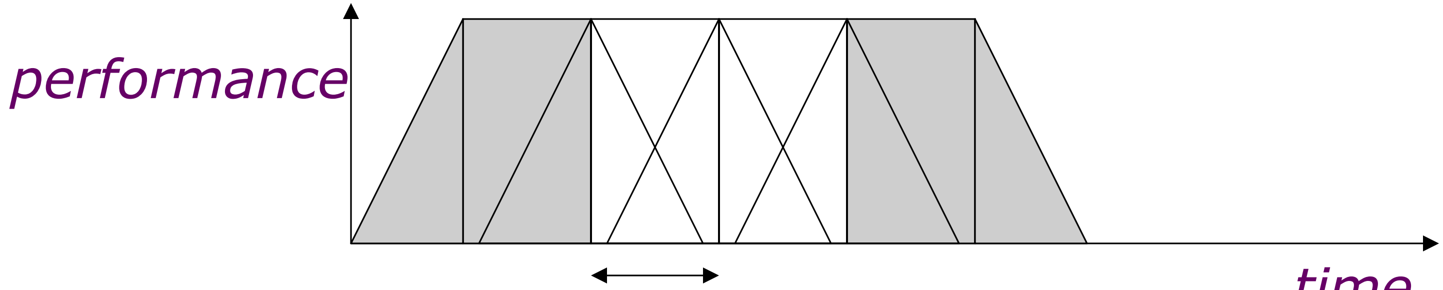
performance

Startup overhead

Loop Iteration

time

Software Pipelined



performance

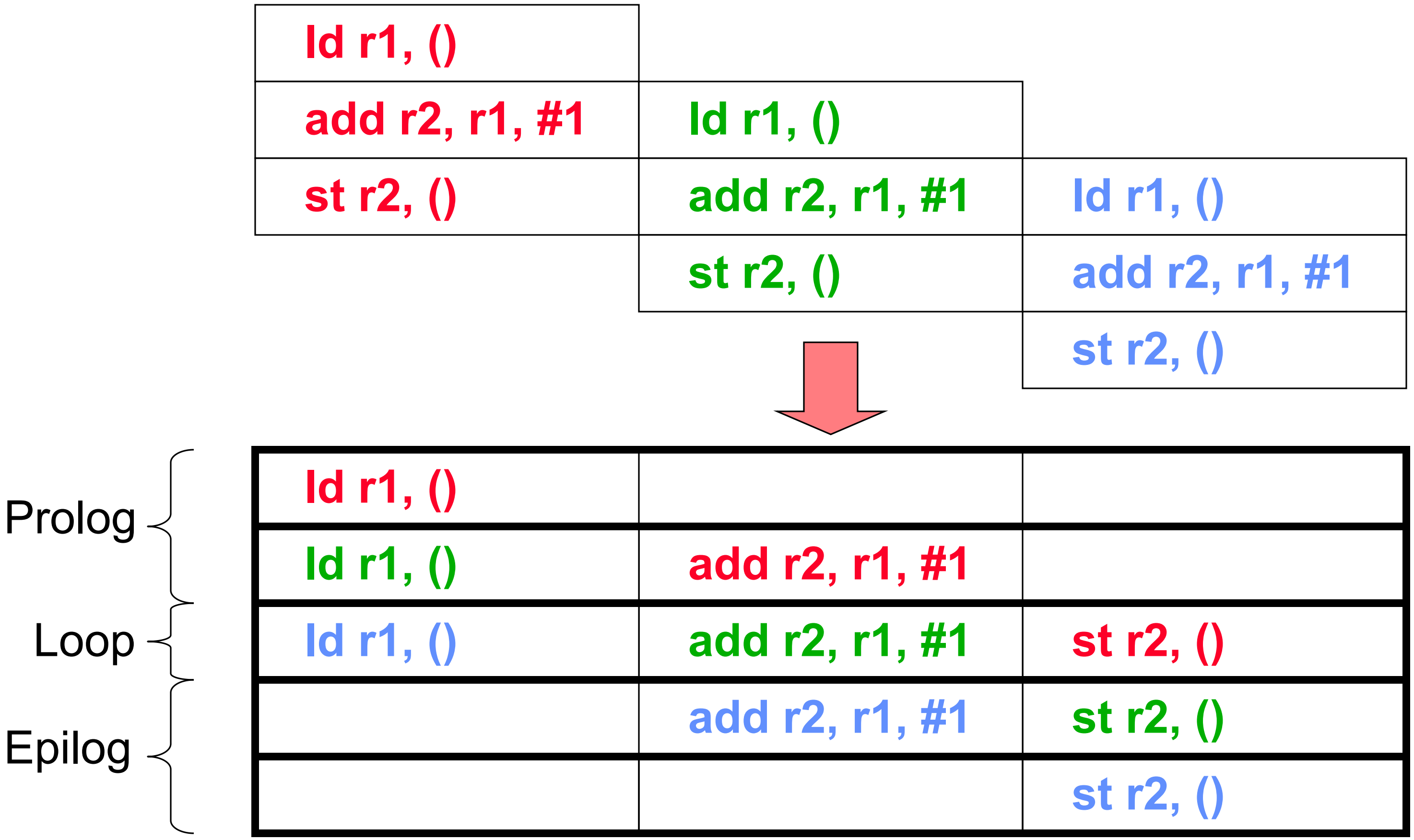
Loop Iteration

time

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

Rotating Register Files

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

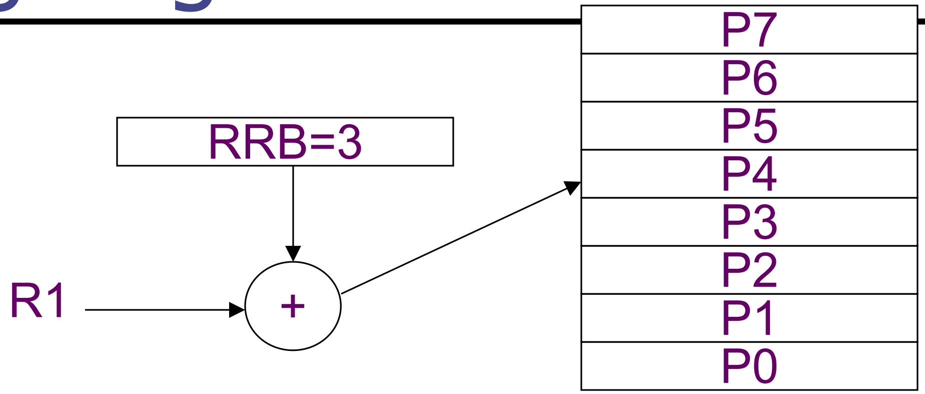


ld r1, ()		
add r2, r1, #1	ld r1, ()	
st r2, ()	add r2, r1, #1	ld r1, ()
	st r2, ()	add r2, r1, #1
		st r2, ()

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

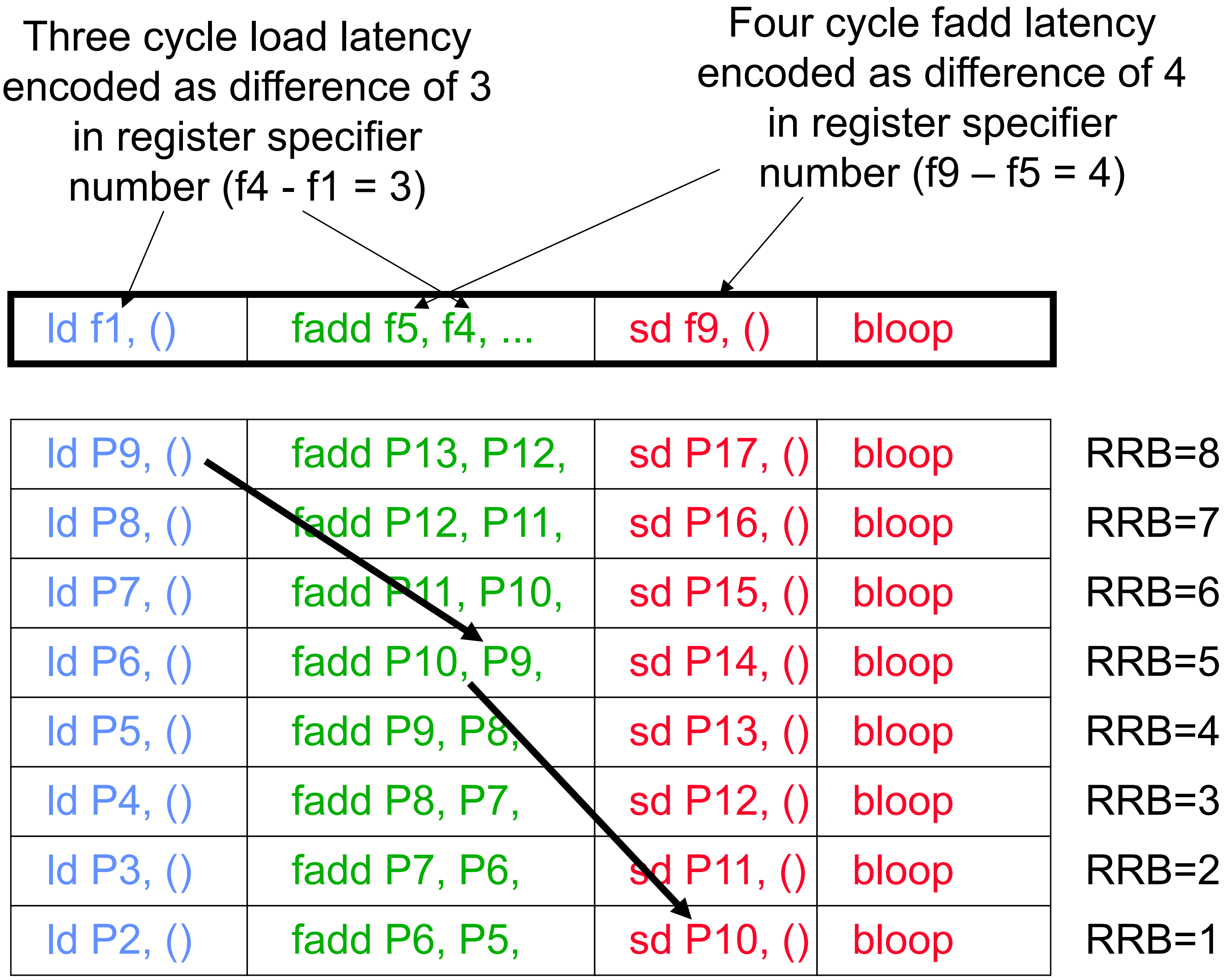


Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

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



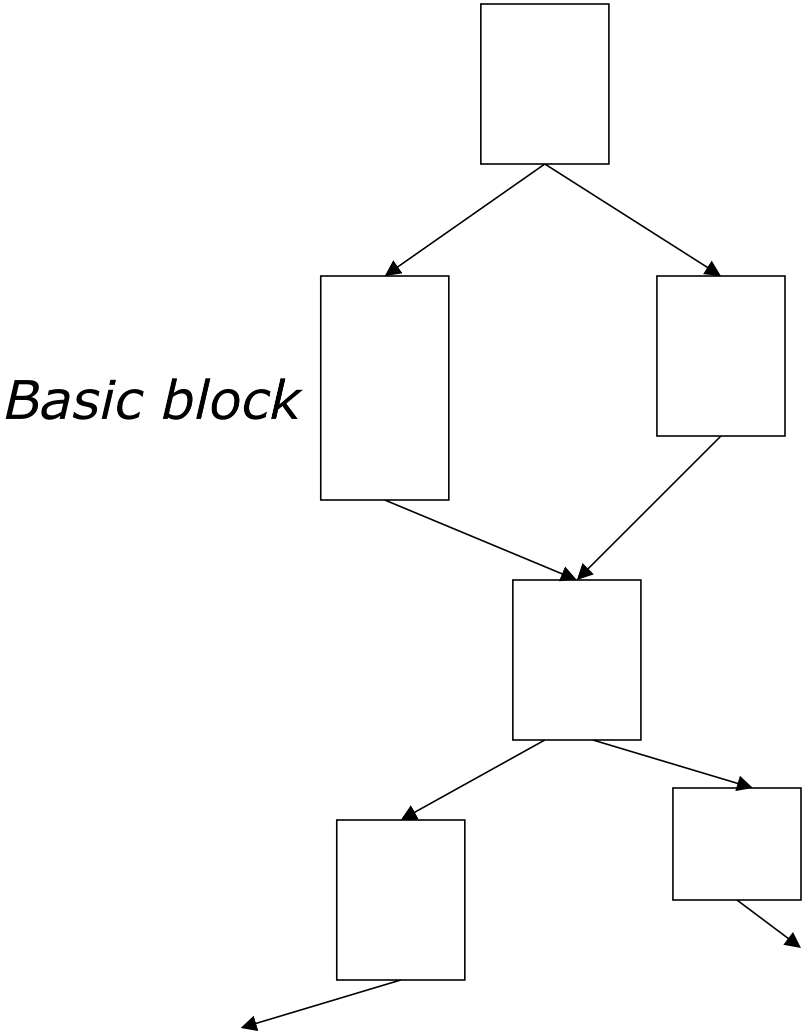
Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

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

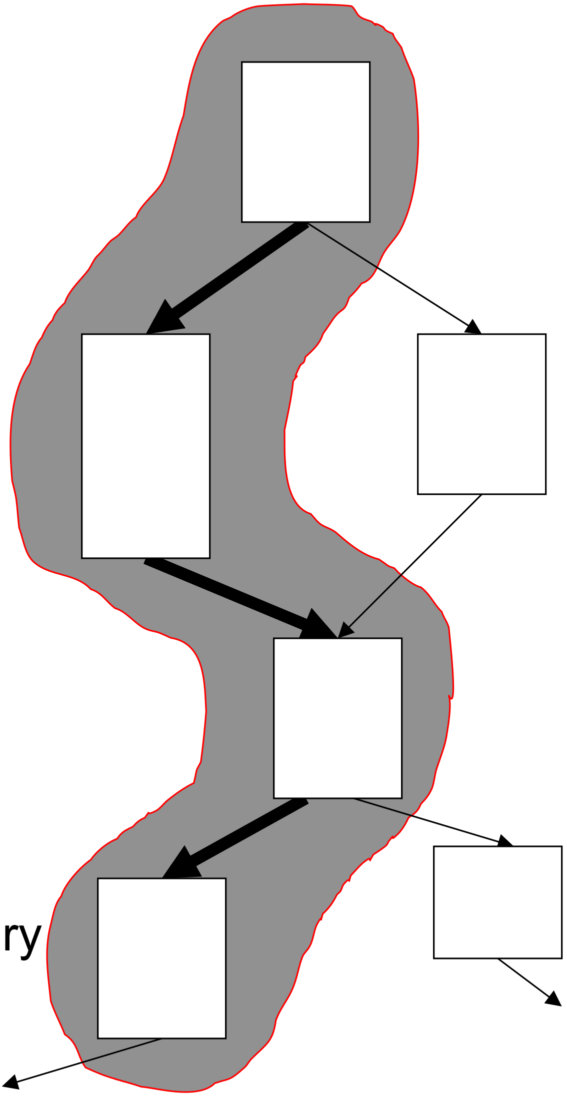
What if there are no loops?



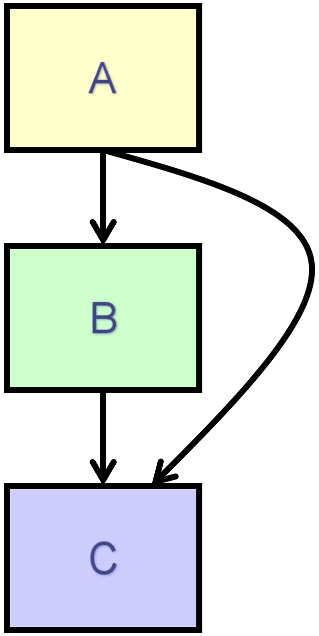
- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

Trace Scheduling

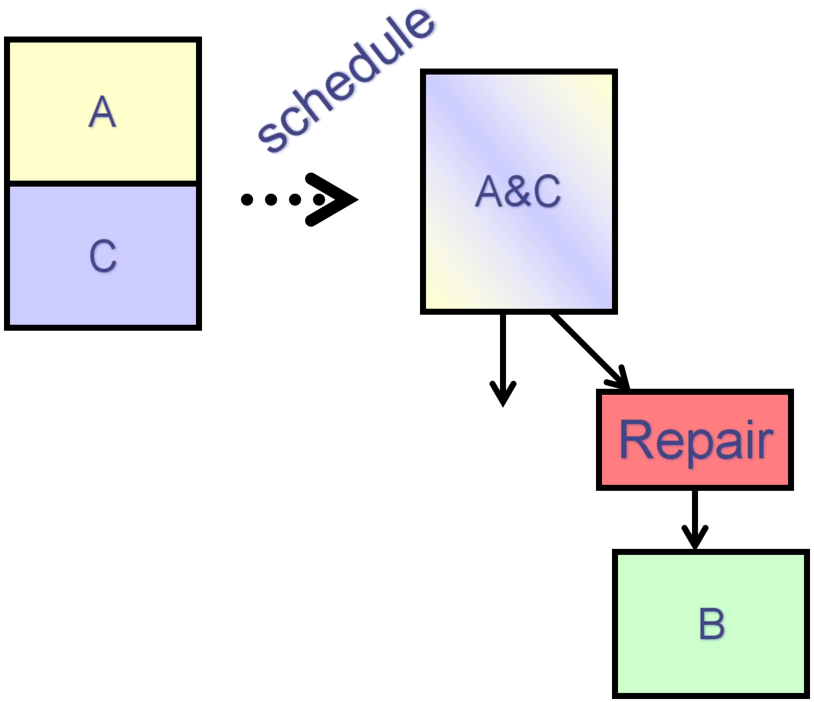
- Goal:
  - Create a large continuous piece or code
  - Schedule to the max: exploit parallelism
- Fact of life:
  - Basic blocks are small
  - Scheduling across BBs is difficult
- But:
  - while many control flow paths exist
  - There are few “hot” ones
- Trace Scheduling
  - Static control speculation
  - Assume specific path
  - Schedule accordingly
  - Introduce check and repair code where necessary



Scheduling Loop Unrolled Code

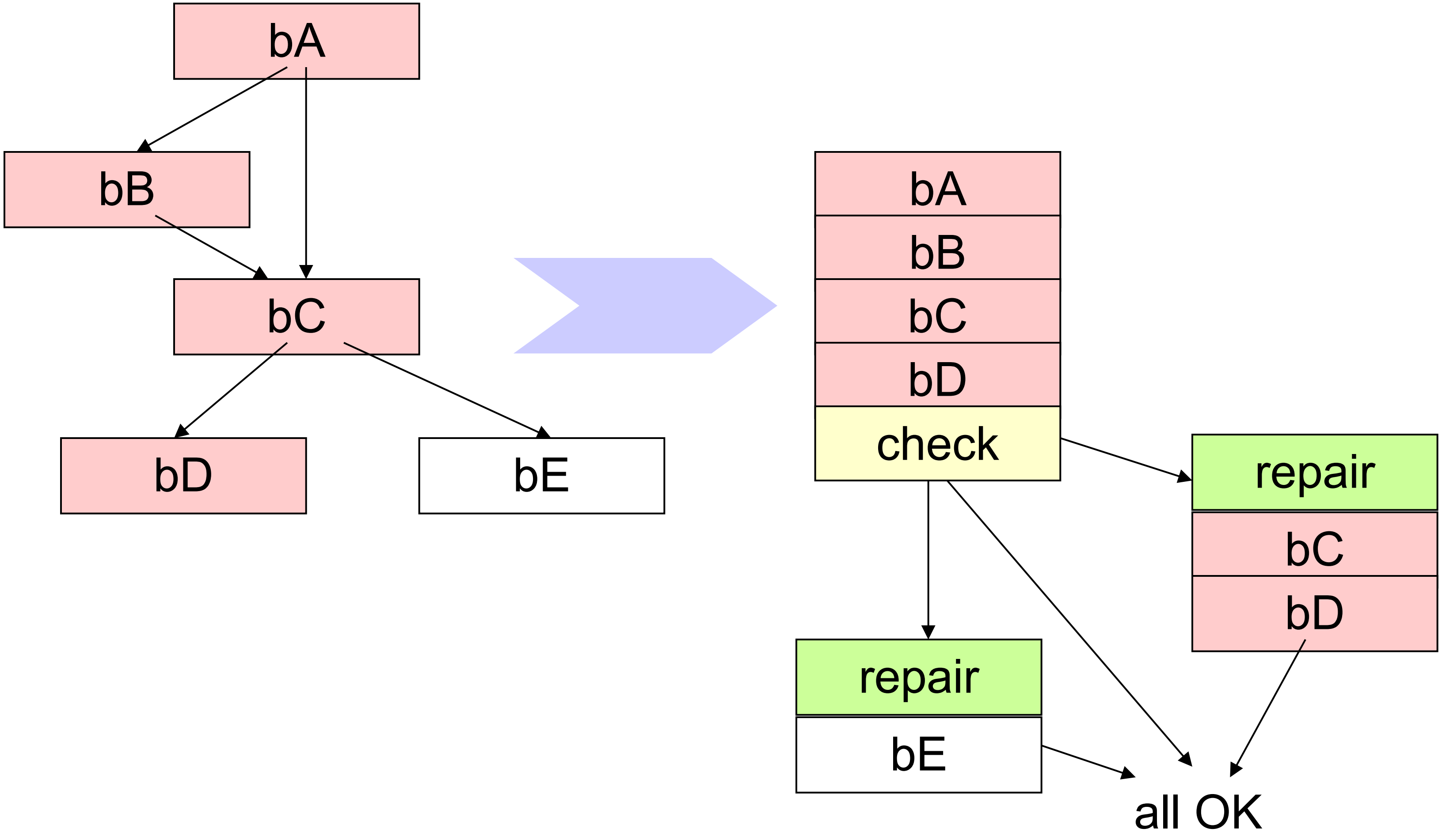


Assume A→C is the common path



- Expand the scope/flexibility of code motion

Trace Scheduling: Example #2



Trace Scheduling Example

```
test = a[i] + 20;
If (test > 0) then
      sum = sum + 10
else
      sum = sum + c[i]
c[x] = c[y] + 10
```

assume delay

Straight code

```
test = a[i] + 20
sum = sum + 10
c[x] = c[y] + 10
 if (test <= 0) then goto repair
…
```

```
repair:
            sum = sum – 10
            sum = sum + c[i]
```

Problems with Classic VLIW

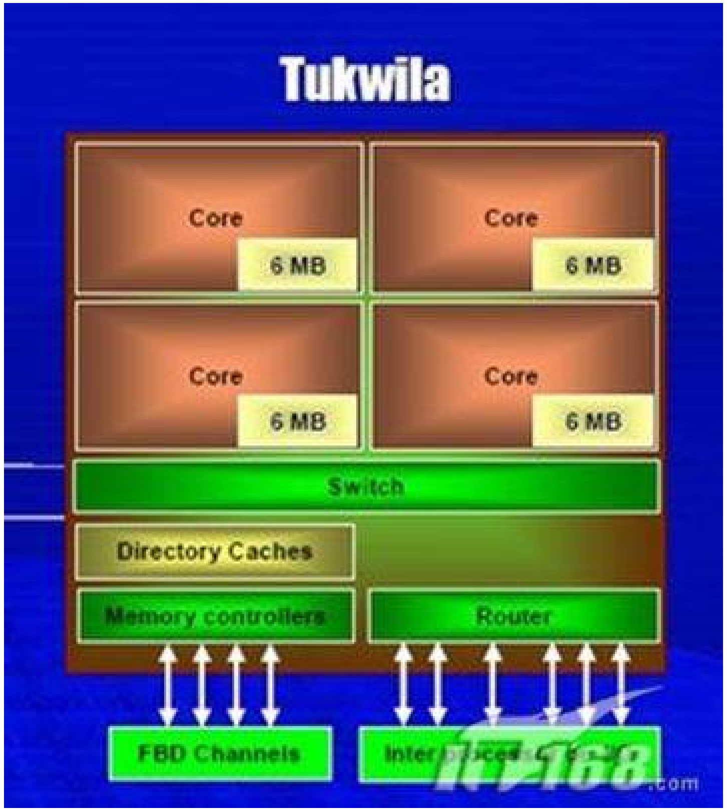
- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - 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 Example

Intel EPIC IA-64

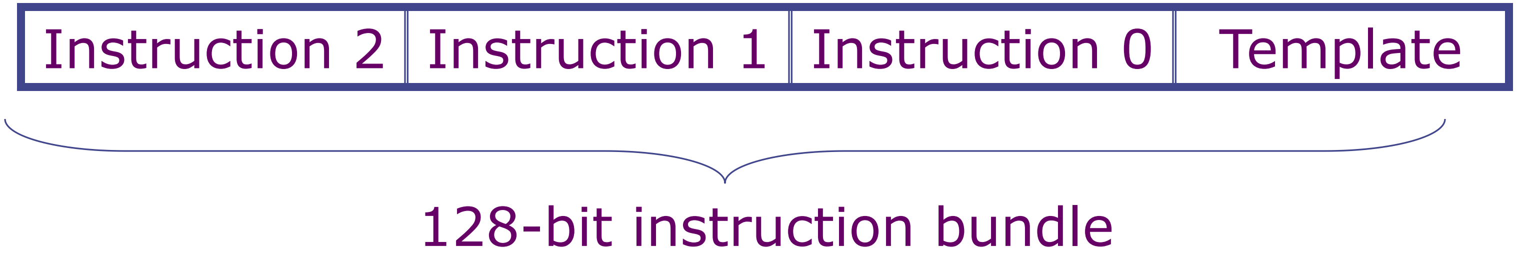
- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing
- IA-64 is Intel's chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code compatible VLIW
- Itanium (aka Merced) is first implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Recent version, Tukwila 2008, quad-cores, 65nm

Quad Core Itanium "Tukwila" [Intel 2008]



- 4 cores
- 6MB $/core, 24MB $ total
- ~2.0 GHz
- 698mm^2 in 65nm CMOS!!!!!
- 170W
- Over 2 billion transistors

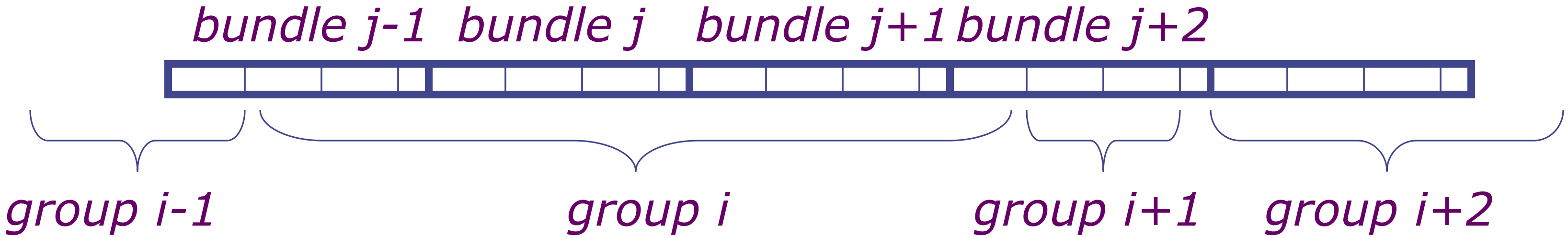
IA-64 Instruction Format



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*

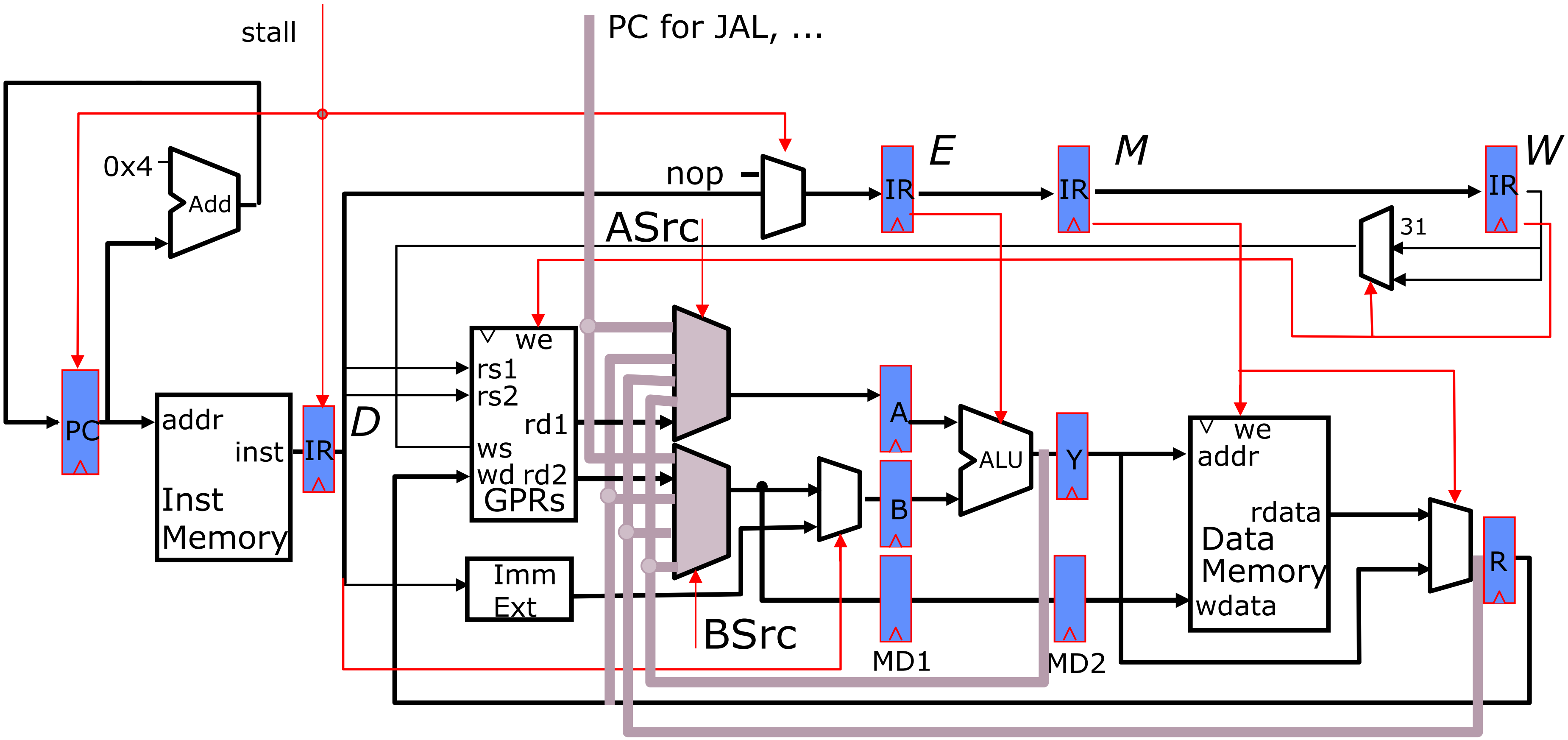
group i-1 *group i* *group i+1* *group i+2*

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

Fully Bypassed Datapath



Where does predication fit in?

Superscalar VS. VLIW

- Superscalar:
  - Relies on dynamic scheduling and out-of-order execution to exploit instruction-level parallelism.
  - Has complex hardware and scheduling logic.
  - Can handle dynamic and unpredictable instruction streams.
  - Provides flexibility but requires more hardware resources.
- VLIW:
  - Packs multiple instructions into a single long instruction word at compile-time.
  - Executes the packed instructions simultaneously in hardware.
  - Relies on static scheduling by the compiler to exploit instruction-level parallelism.
  - Has simpler hardware but relies heavily on compiler optimizations.
  - Suitable for static and predictable instruction streams.