CS 152 Computer Architecture and Engineering

Lecture 15 - VLIW Machines and Statically Scheduled ILP

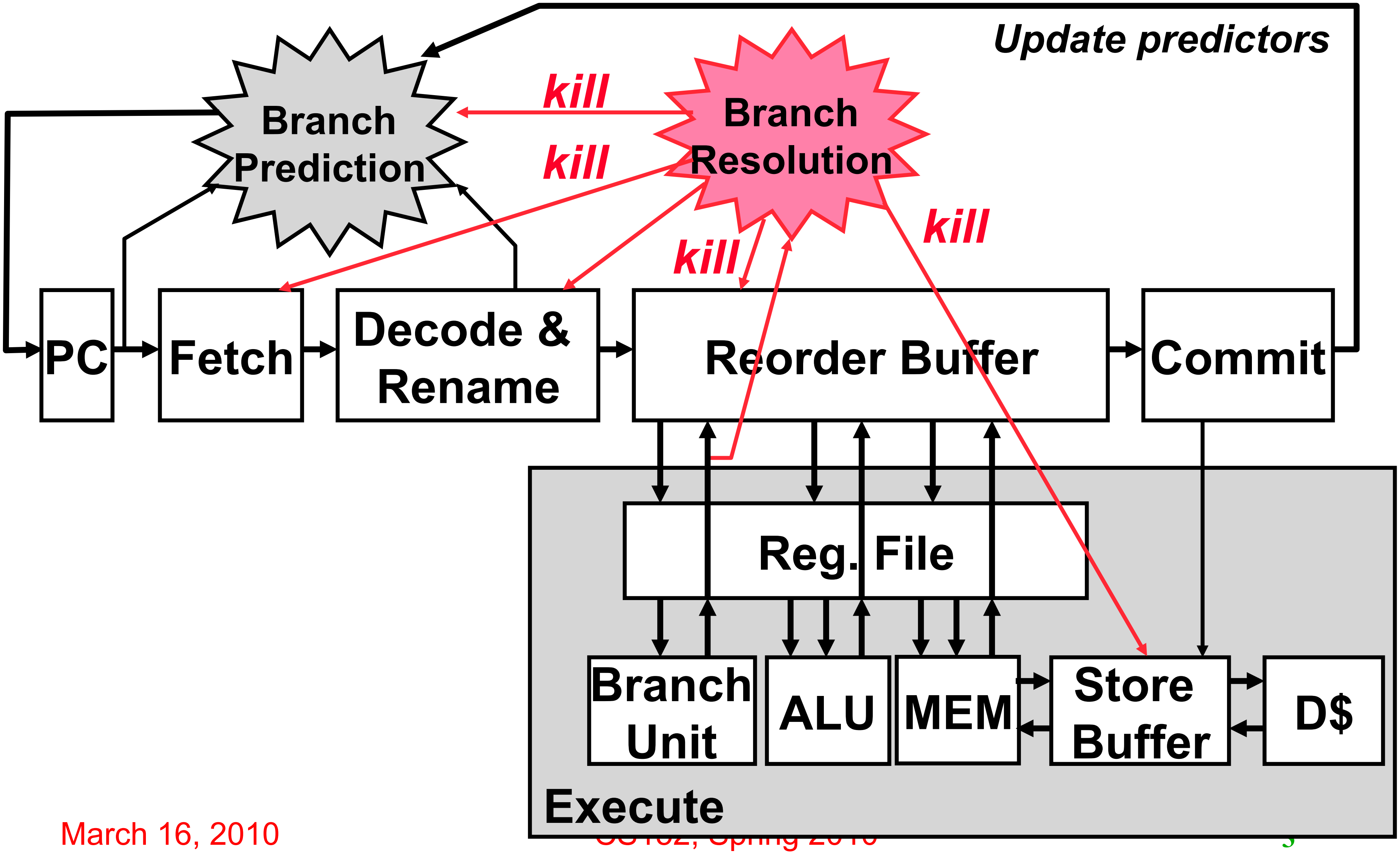
Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

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

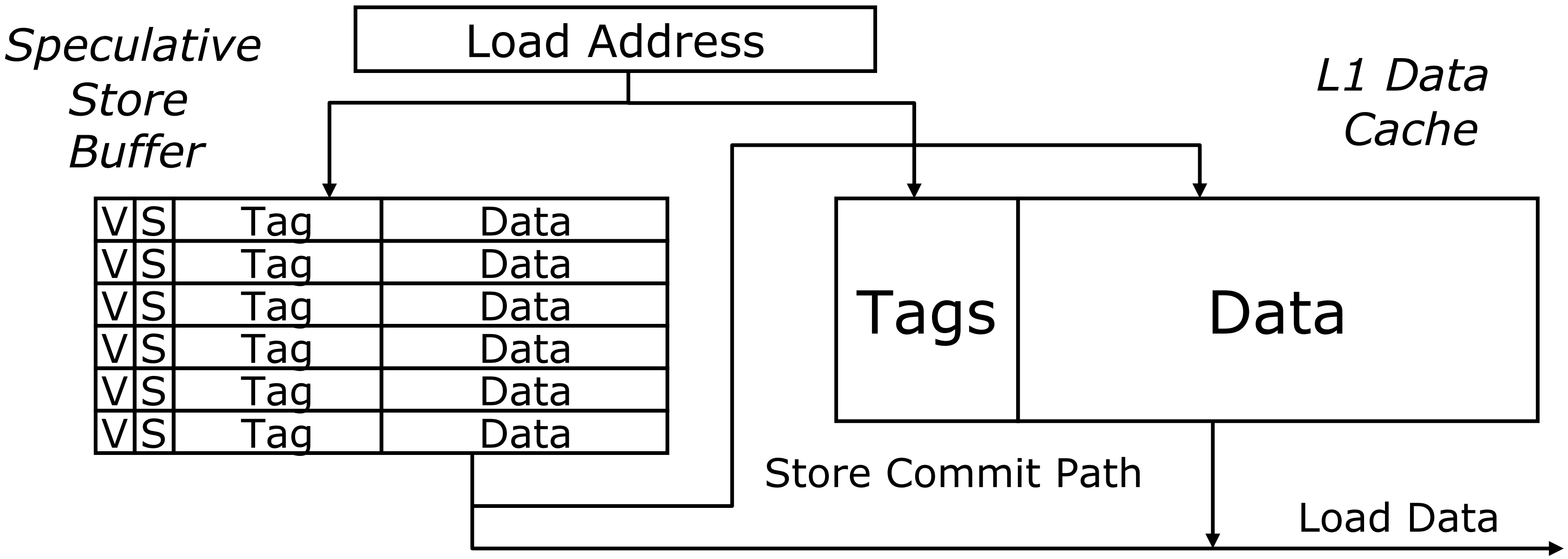
Last time in Lecture 14

- BTB allows prediction very early in pipeline
  - Also handles jump register, although return address stack handles subroutine returns better
- 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

Datapath: Branch Prediction and Speculative Execution



Speculative Store Buffer



Speculative Store Buffer

Load Address

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

L1 Data Cache

Tags

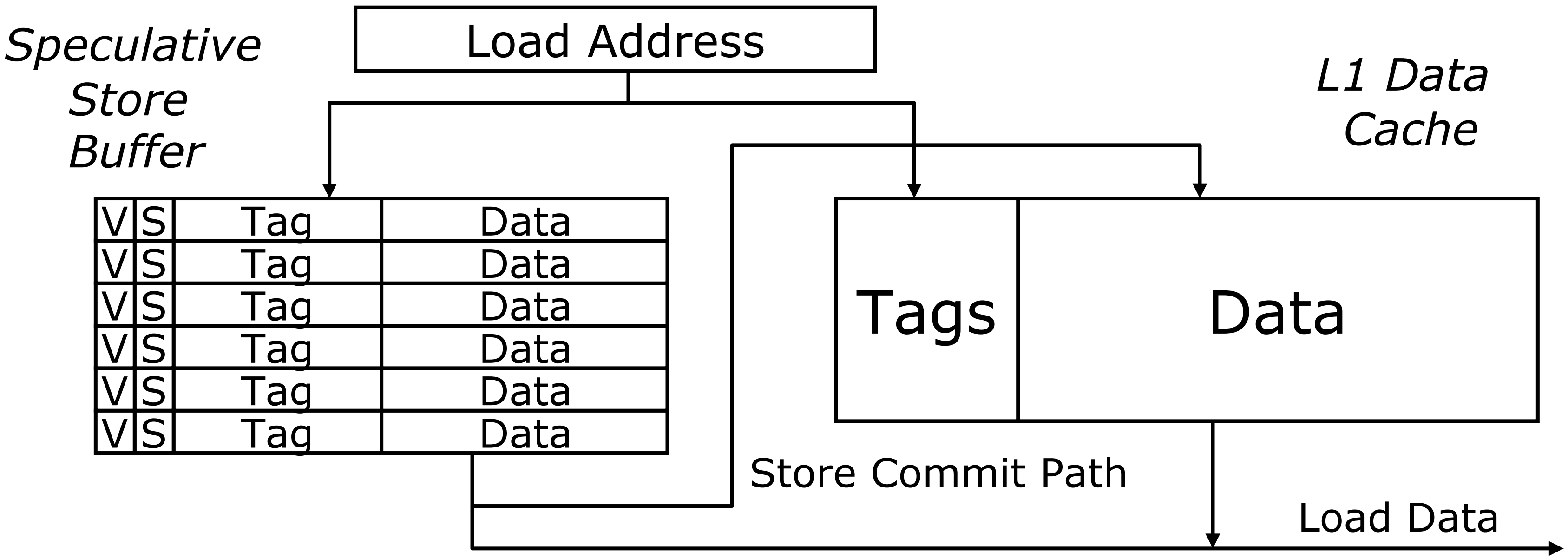
Data

Store Commit Path

Load Data

- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

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

- 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

Instruction Flow in Unified Physical Register File Pipeline

- Fetch
  - Get instruction bits from current guess at PC, place in fetch buffer
  - Update PC using sequential address or branch predictor (BTB)
- Decode/Rename
  - Take instruction from fetch buffer
  - Allocate resources to execute instruction:
    - Destination physical register, if instruction writes a register
    - Entry in reorder buffer to provide in-order commit
    - Entry in issue window to wait for execution
    - Entry in memory buffer, if load or store
  - Decode will stall if resources not available
  - Rename source and destination registers
  - Check source registers for readiness
  - Insert instruction into issue window+reorder buffer+memory buffer

Memory Instructions

- Split store instruction into two pieces during decode:
  - Address calculation, store-address
  - Data movement, store-data
- Allocate space in program order in memory buffers during decode
- Store instructions:
  - Store-address calculates address and places in store buffer
  - Store-data copies store value into store buffer
  - Store-address and store-data execute independently out of issue window
  - Stores only commit to data cache at commit point
- Load instructions:
  - Load address calculation executes from window
  - Load with completed effective address searches memory buffer
  - Load instruction may have to wait in memory buffer for earlier store ops to resolve

Issue Stage

- Writebacks from completion phase “wakeup” some instructions by causing their source operands to become ready in issue window
  - In more speculative machines, might wake up waiting loads in memory buffer
- Need to “select” some instructions for issue
  - Arbiter picks a subset of ready instructions for execution
  - Example policies: random, lower-first, oldest-first, critical-first
- Instructions read out from issue window and sent to execution

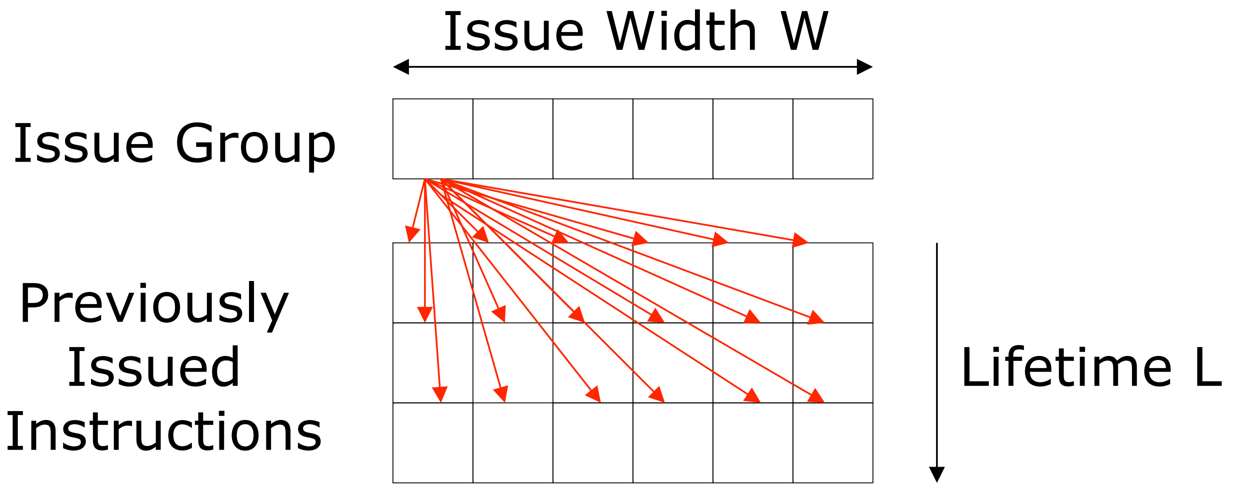
Execute Stage

- Read operands from physical register file and/or bypass network from other functional units
- Execute on functional unit
- Write result value to physical register file (or store buffer if store)
- Produce exception status, write to reorder buffer
- Free slot in instruction window

Commit Stage

- Read completed instructions in-order from reorder buffer
  - (may need to wait for next oldest instruction to complete)
- If exception raised
  - flush pipeline, jump to exception handler
- Otherwise, release resources:
  - Free physical register used by last writer to same architectural register
  - Free reorder buffer slot
  - Free memory reorder buffer slot

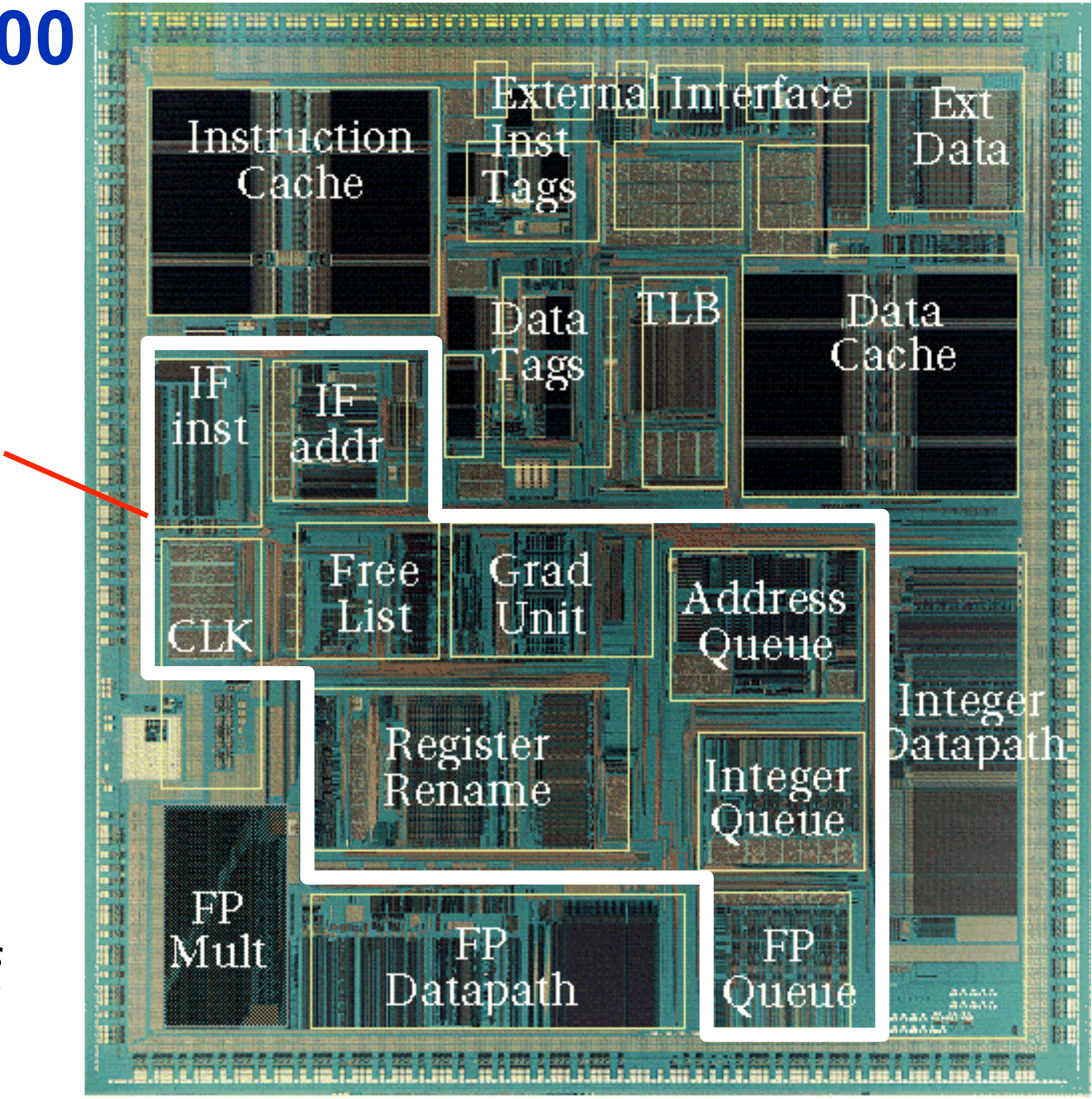
Superscalar Control Logic Scaling



- Each issued instruction must somehow check against W*L instructions, i.e., growth in hardware $\propto$ 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

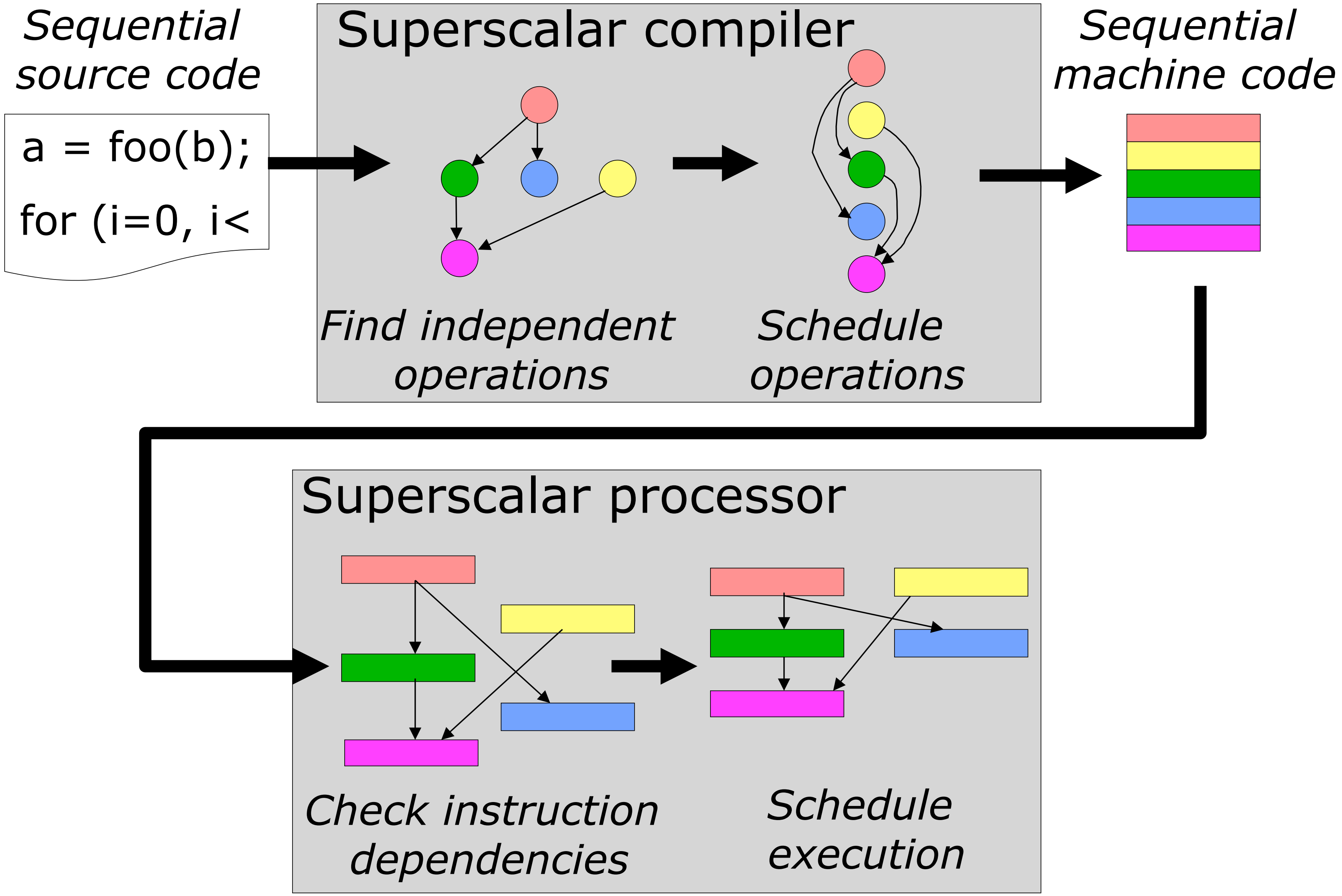
=> *Out-of-order control logic grows faster than W^2 (~W^3)*

Out-of-Order Control Complexity: MIPS R10000



[SGI/MIPS Technologies Inc., 1995]

Sequential ISA Bottleneck



Sequential source code

```
a = foo(b);

for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations

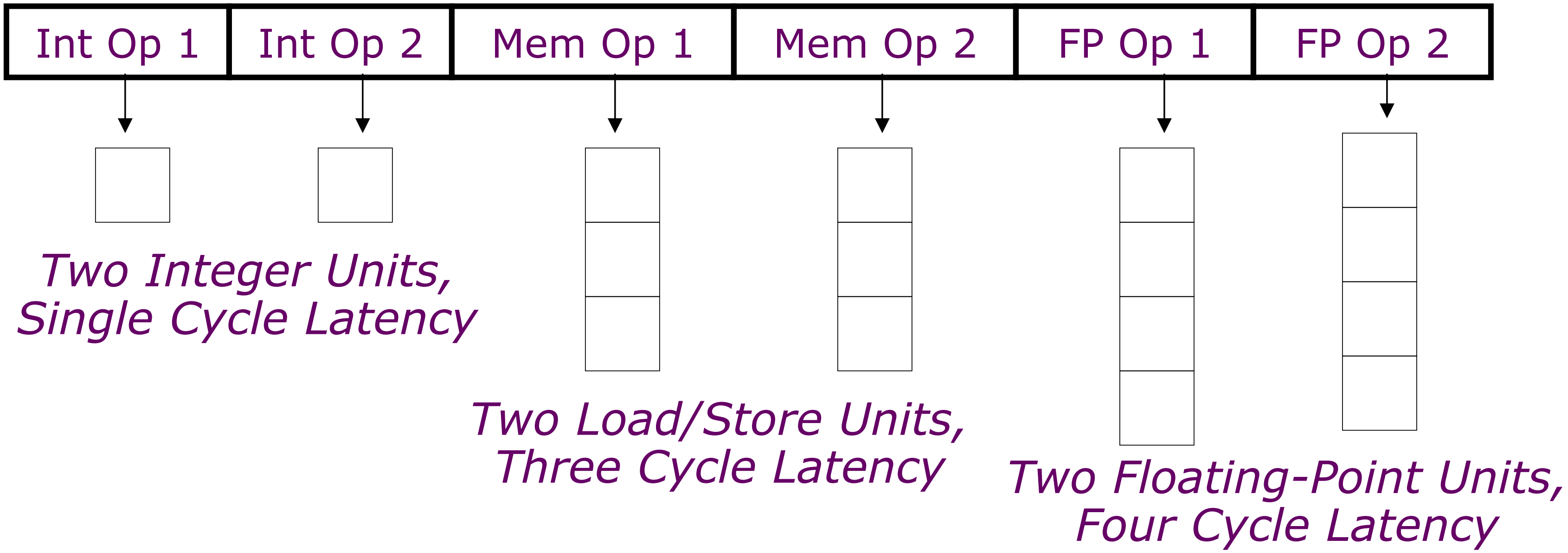
Sequential machine code

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

- 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 Compiler Responsibilities

- 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

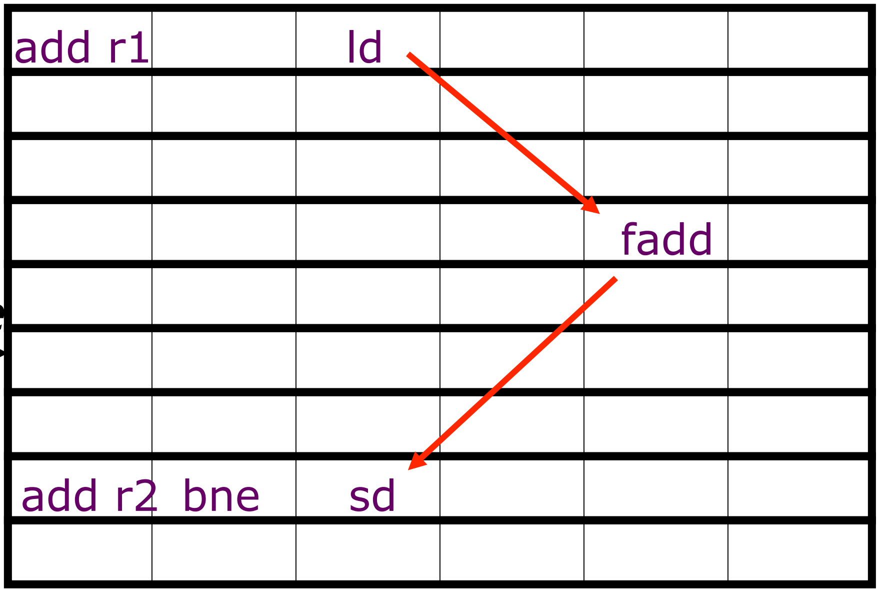
```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

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

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
```

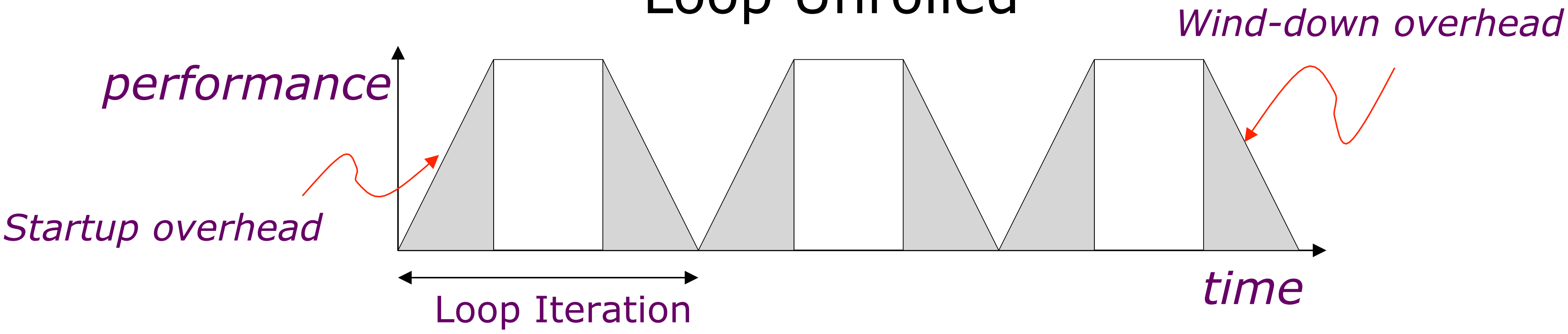
	Int1	Int 2	M1	M2	FP+	FPx
prolog			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
			ld f1		fadd f5	
			ld f2		fadd f6	
			ld f3		fadd f7	
	add r1		ld f4		fadd f8	
iterate loop:			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
	add r1	bne	ld f4	sd f8	fadd f8	
epilog			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

How many FLOPS/cycle?

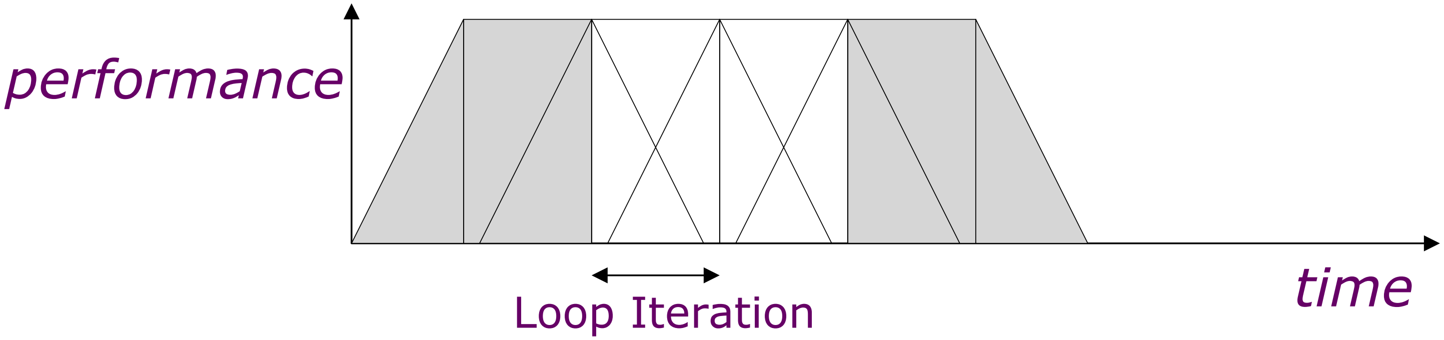
4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling

Loop Unrolled



Software Pipelined

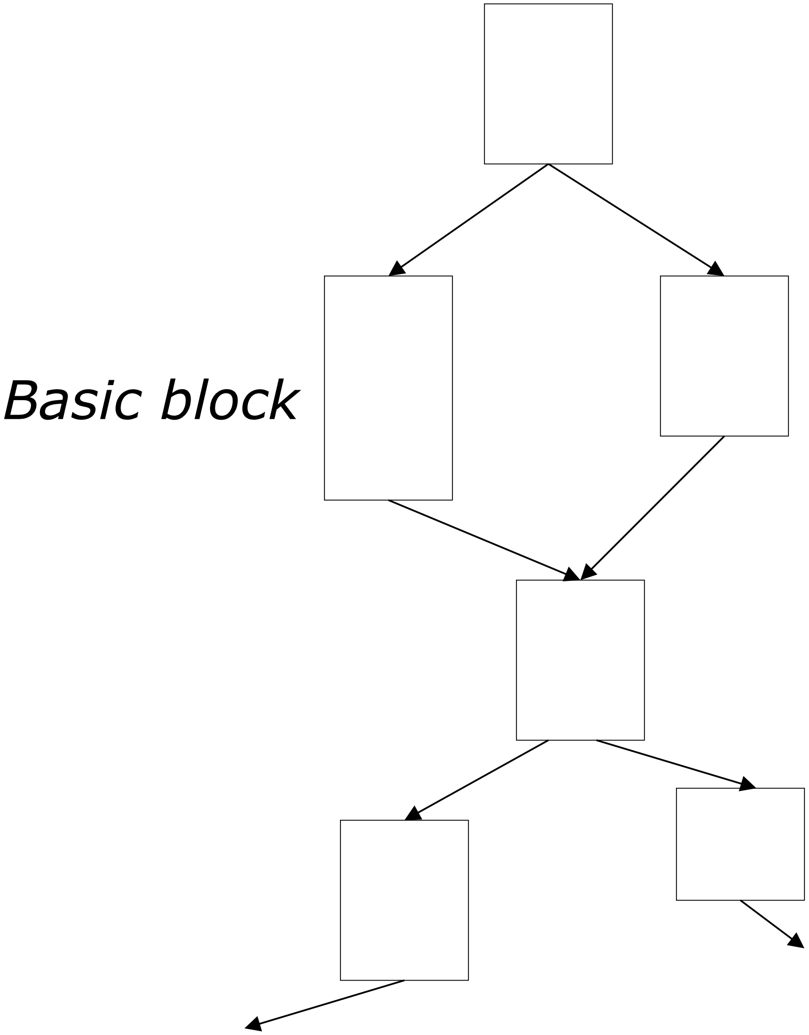


Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

CS152 Administrivia

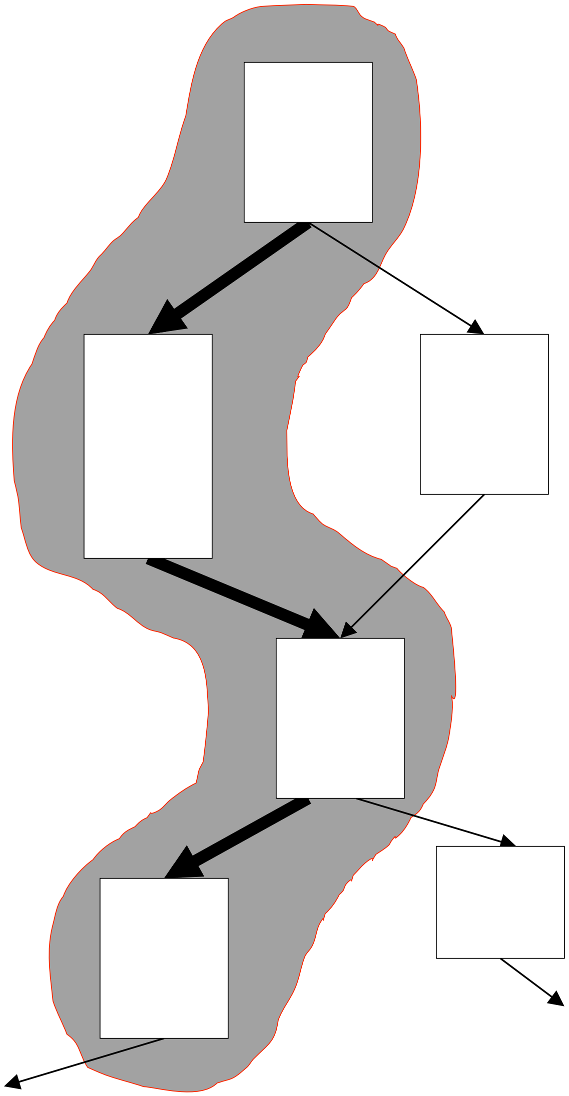
- Quiz 3, Tuesday March 30 (first class back after spring break)
  - All material on complex pipelining (L12-L14, plus review at start of L15), PS3, Lab 3

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]*

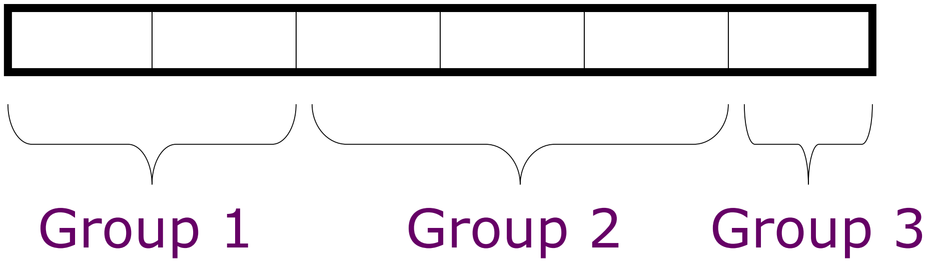


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

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



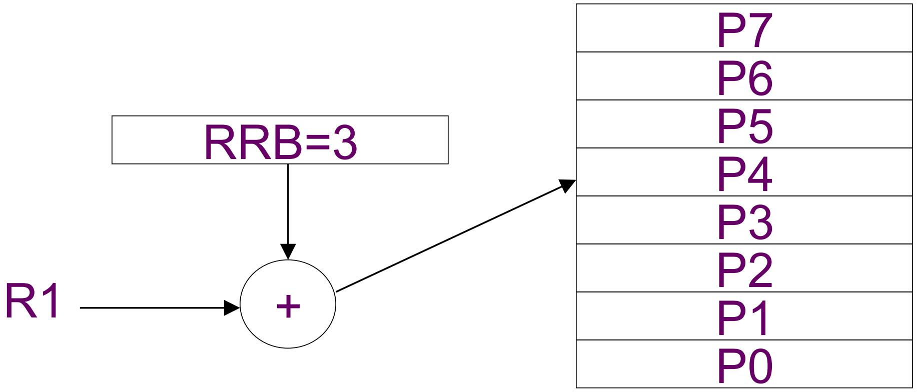
- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5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

Rotating Register Files

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

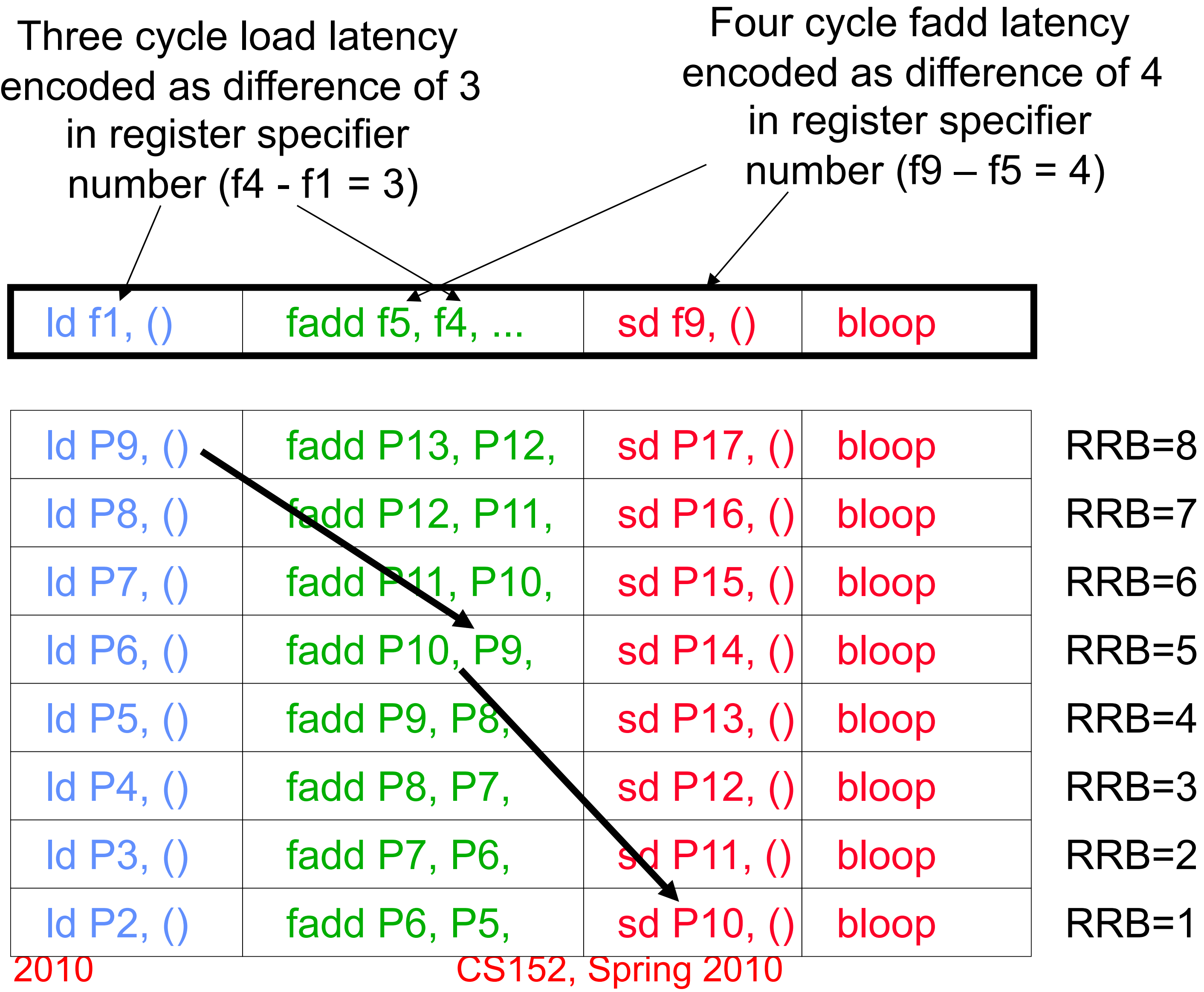
Solution: Allocate new set of registers for each loop iteration

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

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

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