ECE 252 / CPS 220
Advanced Computer Architecture I

Lecture 14
Very Long Instruction Word Machines

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html

ECE252 Administrivia

15 November – Homework #4 Due

Will be posted by end of week.

Project Proposals

Should have received comments already.

ECE 299 – Energy-Efficient Computer Systems

- www.duke.edu/~bcl15/class/class_ece299fall10.html
- Technology, architectures, systems, applications
- Seminar for Spring 2012.
- Class is paper reading, discussion, research project
- In Fall 2010, students read >35 research papers.
- In Fall 2012, read research papers.
- In Fall 2012, also considering textbook "The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines."

Last Time

Virtual Memory

- Enables multi-programming
- Programs operate in virtual memory space
- Programs are protected from each other

Virtual to Physical Address Translation

- Base&Bound
- Segmentation
- Paging
- Multi-level Translation (segmented paging, paged paging)

Translation Lookaside Buffer

- Accelerates virtual memory, address translation

OoO Superscalar Complexity

Out-of-order Superscalar

- Objective: Increase instruction-level parallelism.
- Cost: Hardware logic/mechanisms to track dependencies and dynamically schedule independent instructions.

Hardware Complexity

- Instructions can issue, complete out-of-order.
- Instructions must commit in-order
- Implement Tomasulo's algorithm with a variety of structures
- Example: Reservation stations, reorder buffer, physical register file

Very Long Instruction Word (VLIW)

- Objective: Increase instruction-level parallelism.
- Cost: Software compilers/mechanisms to track dependencies and statically schedule independent instructions.

Review of Fetch/Decode

Fetch

- Load instruction by accessing memory at program counter (PC)
- Update PC using sequential address (PC+4) or branch prediction (BTB)

Decode/Rename

- Take instruction from fetch buffer
- Allocate resources, which are necessary to execute instruction:
  - (1) Destination physical register – if instruction writes a register, rename
  - (2) Reorder buffer (ROB) entry – support in-order commit
  - (3) Issue queue entry – hold instruction as it waits for execution
  - (4) Memory buffer entry – resolve dependencies through memory (next slide)

- Stall if resources unavailable
- Rename source/destination registers
- Update reorder buffer, issue queue, memory buffer

Review of Memory Buffer

Allocate memory buffer entry

Store Instructions

- Calculate store-address and place in buffer
- Take store-data and place in buffer
- Instruction commits in-order when store-address, store-data ready

Load Instructions

- Calculate load-address and place in buffer
- Instruction searches memory buffer for stores with matching address
- Forward load data from in-flight stores with matching address
- Stall load if buffer contains stores with un-resolved addresses

Review of Issue/Execute

Issue

- Instruction commits from reorder buffer
- A commit wakes-up an instruction by marking its sources ready
- Select logic determines which ready instructions should execute
- Issue when by sending instructions to functional unit

Execute

- Read operands from physical register file and/or forwarding path
- Execute instruction in functional unit
- Write result to physical register file, store buffer
- Produce exception status
- Write to reorder buffer

Review of Commit

Commit

- Instructions can complete and update reorder buffer out-of-order
- Instructions commit from reorder buffer in-order

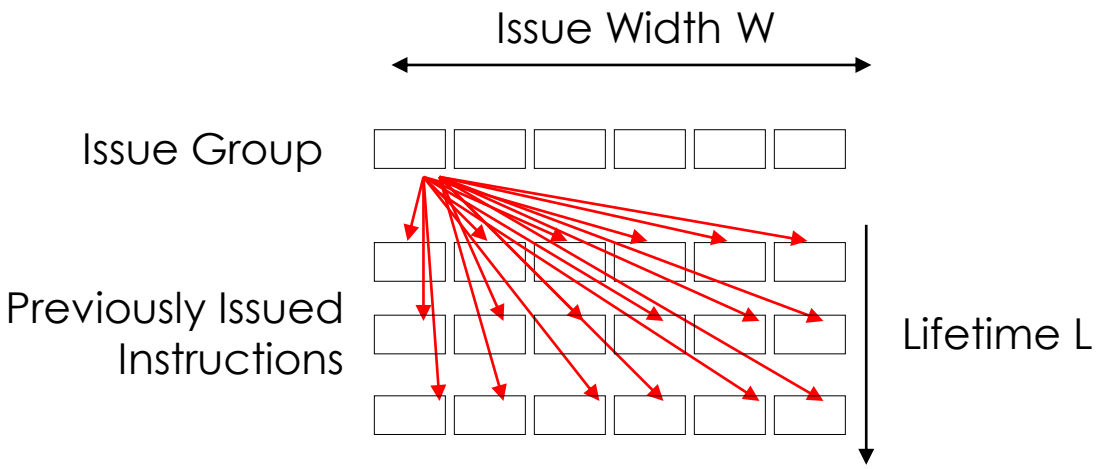
Exceptions

- Check for exceptions
- If exception raised, flush pipeline
- Jump to exception handler

Release Resources

- Free physical register used by last writer to same architected register
- Free reorder buffer slot
- Free memory buffer slot

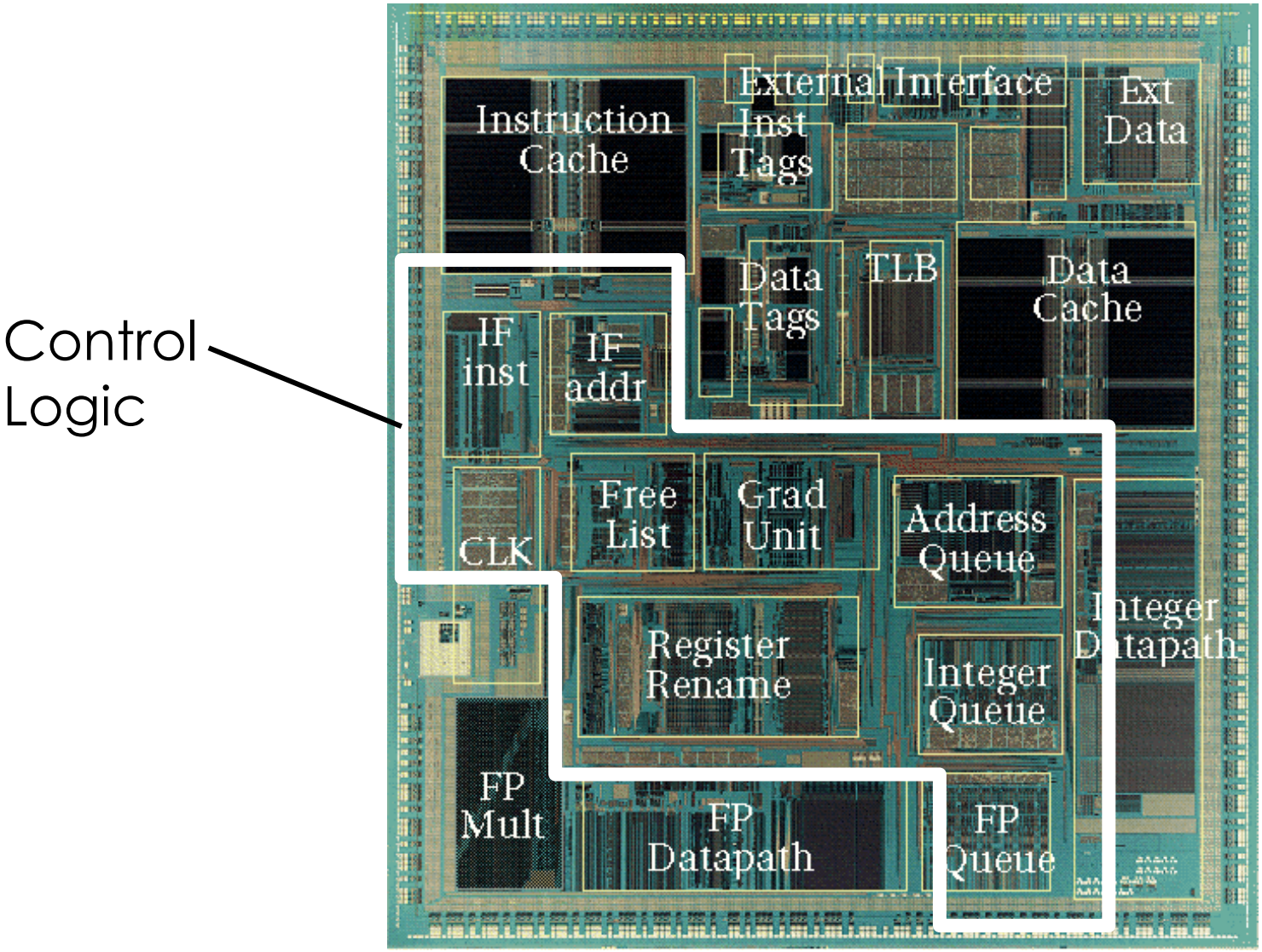
Control Logic Scaling



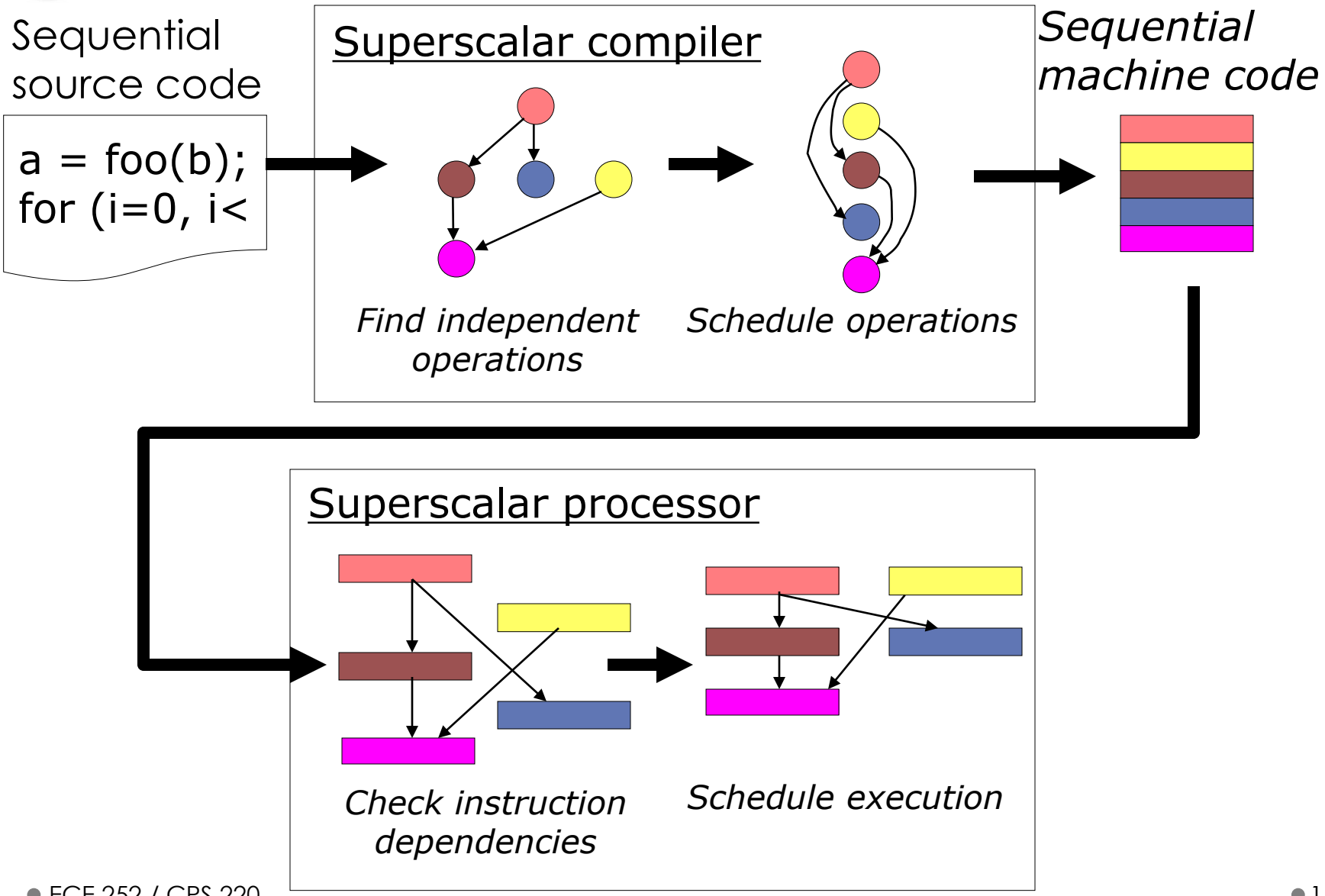
- Lifetime (L) – number of cycles an instruction spends in pipeline
- Lifetime depends on pipeline latency, time spent in reorder buffer
- Issue width (W) – maximum number of instructions issued per cycle

- As W increases, issue logic must find more instructions to execute in parallel and keep pipeline busy.
- More instructions must be fetched, decoded, and queued.
- **W x L** instructions can impact any of the **W** issuing instructions (e.g. forwarding) and growth in hardware proportional to **W x (W x L)**

Control Logic (MIPS R10000)



Sequential Instruction Sets



Sequential Instruction Sets

Superscalar Compiler

- Takes sequential code (e.g., C, C++)
- Check instruction dependencies
- Schedule operations to preserve dependencies
- Produces sequential machine code (e.g., MIPS)

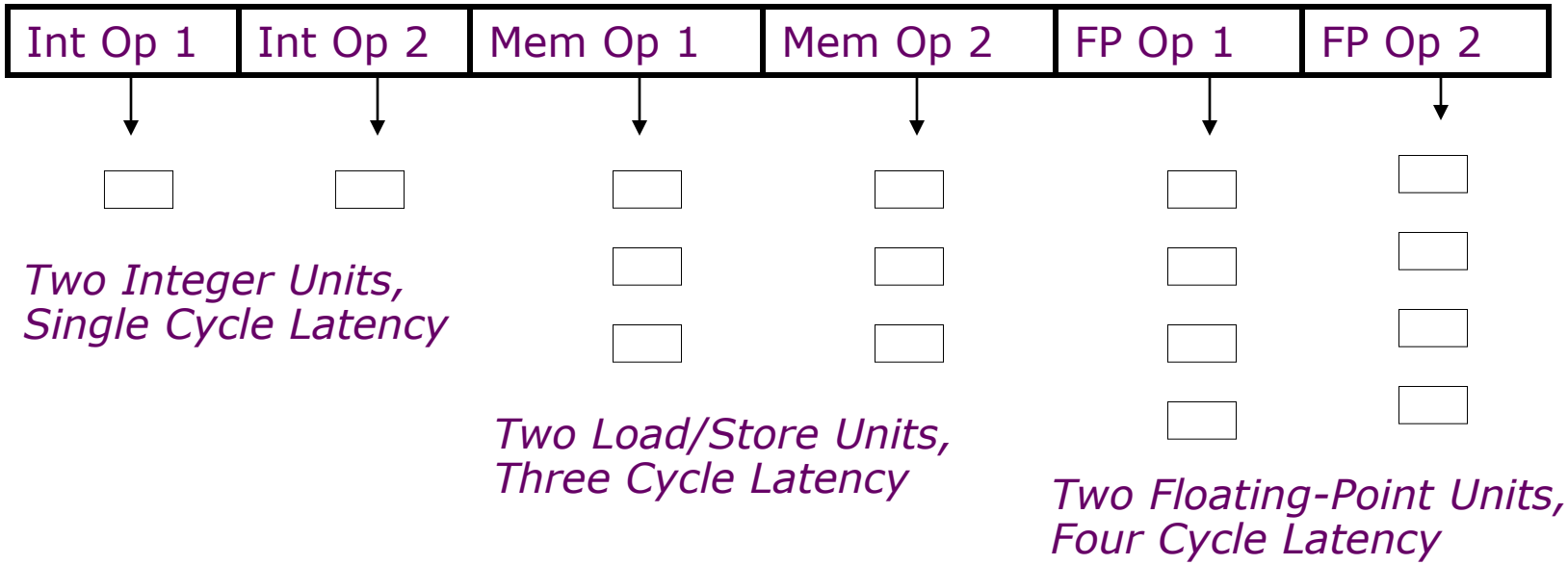
Superscalar Processor

- Takes sequential code (e.g., MIPS)
- Check instruction dependencies
- Schedule operations to preserve dependencies

Inefficiency of Superscalar Processors

- Performs dependency, scheduling dynamically in hardware
- Expensive logic rediscovers schedules that a compiler could have found

VLIW – Very Long Instruction Word



- Multiple operations packed into one instruction format
- Instruction format is fixed, each slot supports particular instruction type
- Constant operation latencies are specified (e.g., 1 cycle integer op)
- Software schedules operations into instruction format, guaranteeing
  (1) Parallelism within an instruction – no RAW checks between ops
  (2) No data use before ready – no data interlocks/stalls

VLIW Compiler Responsibilities

Schedule operations to maximize parallel execution

- Fill operation slots

Guarantee intra-instruction parallelism

- Ensure operations within same instruction are independent

Schedule to avoid data hazards

- Separate options with explicit NOPs

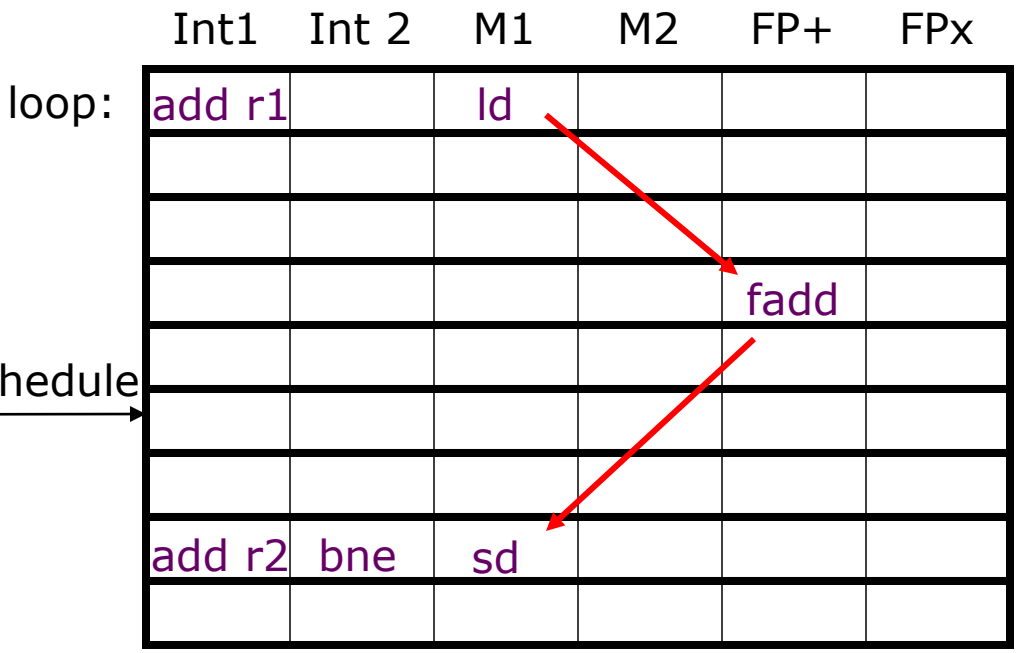
Loop Execution

```
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



- The latency of each instruction is fixed (e.g., 3 cycle ld, 4 cycle fadd)
- Instr-1: Load A[i] and increment i (r1) in parallel
- Instr-2: Wait for load
- Instr-3: Wait for add. Store B[i], increment i (r2), branch in parallel
- How many flops / 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 k iterations of computation at once.
- If N is not a multiple of unrolling factor k, insert clean-up code

- Example: unroll inner loop to perform 4 iterations at once

Scheduling Unrolled Loops

```
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			

- Unroll loop to execute 4 iterations
- Reduces number of empty operation slots
- How many flops/cycle? 4 fadds / 11 cycles = 0.36

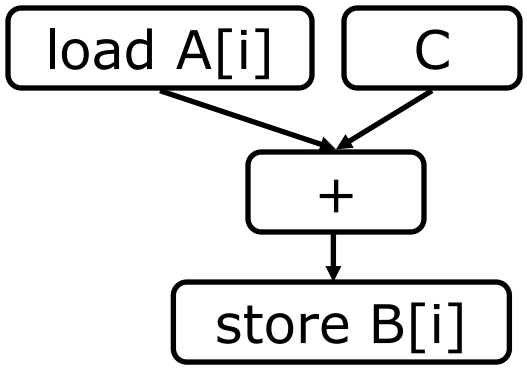
Software Pipelining

Exploit independent loop iterations

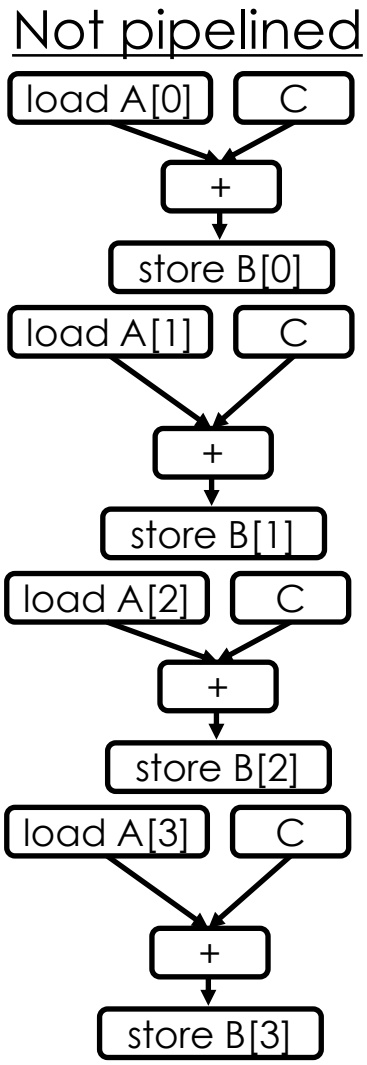
- If loop iterations are independent, then get more parallelism by scheduling instructions from different iterations

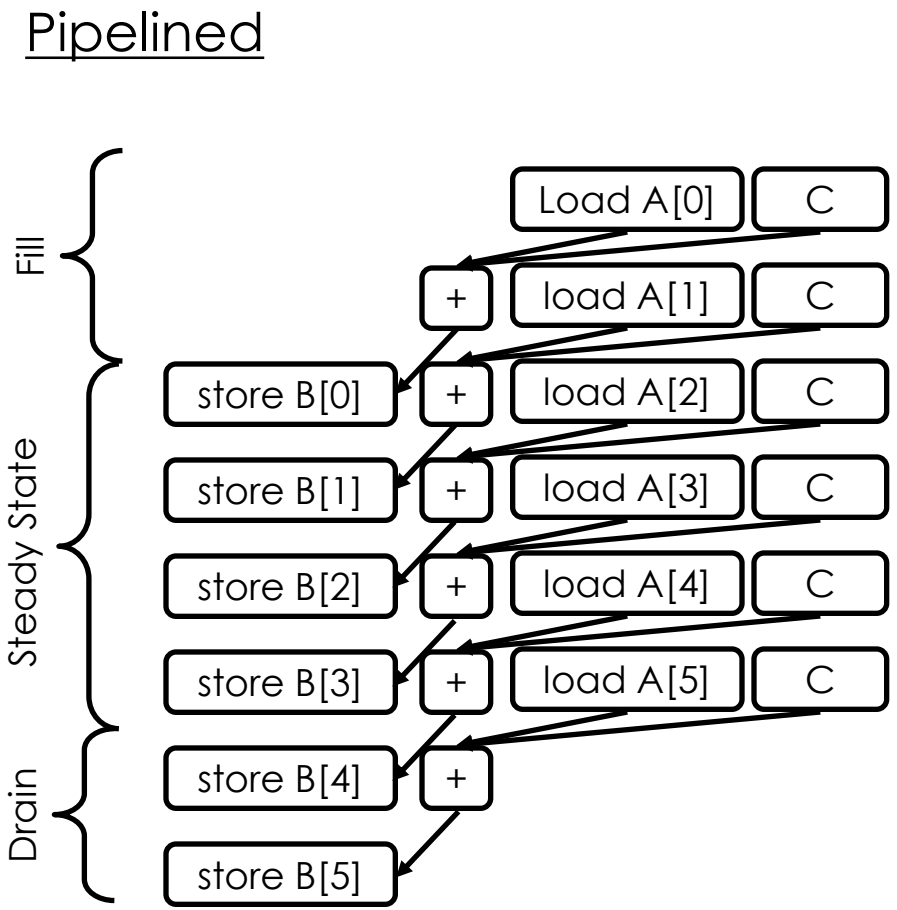
- Example: Loop iterations are independent in the code sequence below.
- Construct the data-flow graph for one iteration

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



Software Pipelining (Illustrated)





Scheduling SW Pipelined Loops

Unroll the loop to perform 4 iterations at once. Then SW pipeline.

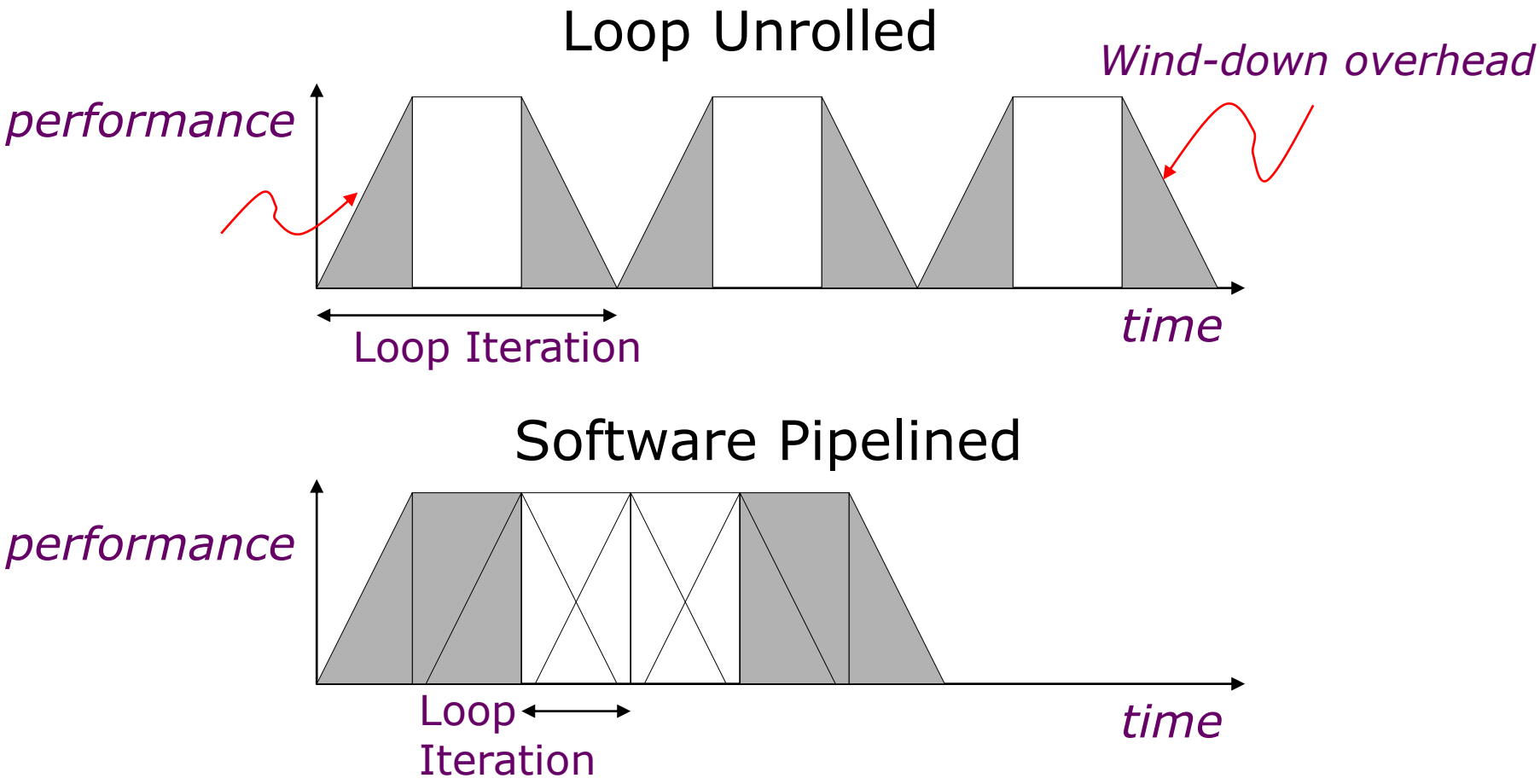
```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

```
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
```

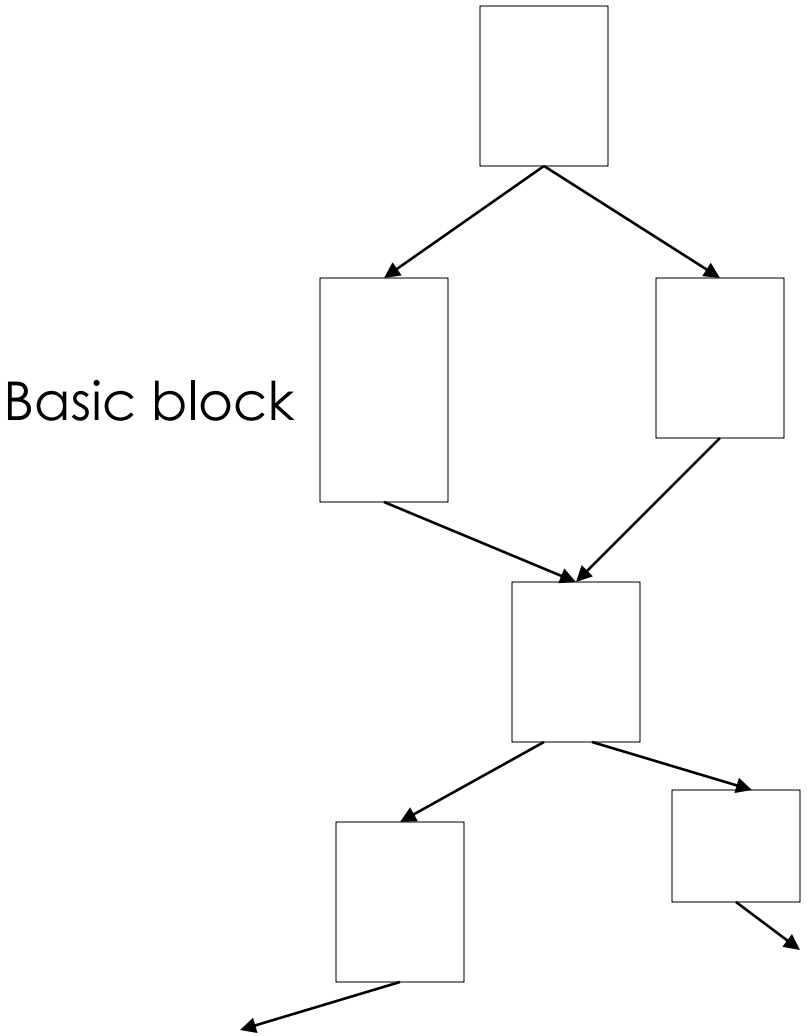
	Int1	Int 2	M1	M2	FP+	FPx
fill			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
			ld f1		fadd f5	
			ld f2		fadd f6	
			ld f3		fadd f7	
	add r1		ld f4		fadd f8	
loop: (steady state)			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
	add r1	bne	ld f4	sd f8	fadd f8	
drain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

Unrolling versus Pipelining

- Unrolled loops pay fill and drain costs once per loop iteration.
- SW pipelined loops pay (a.k.a. prologue) and drain (a.k.a. epilogue) costs only once per loop.

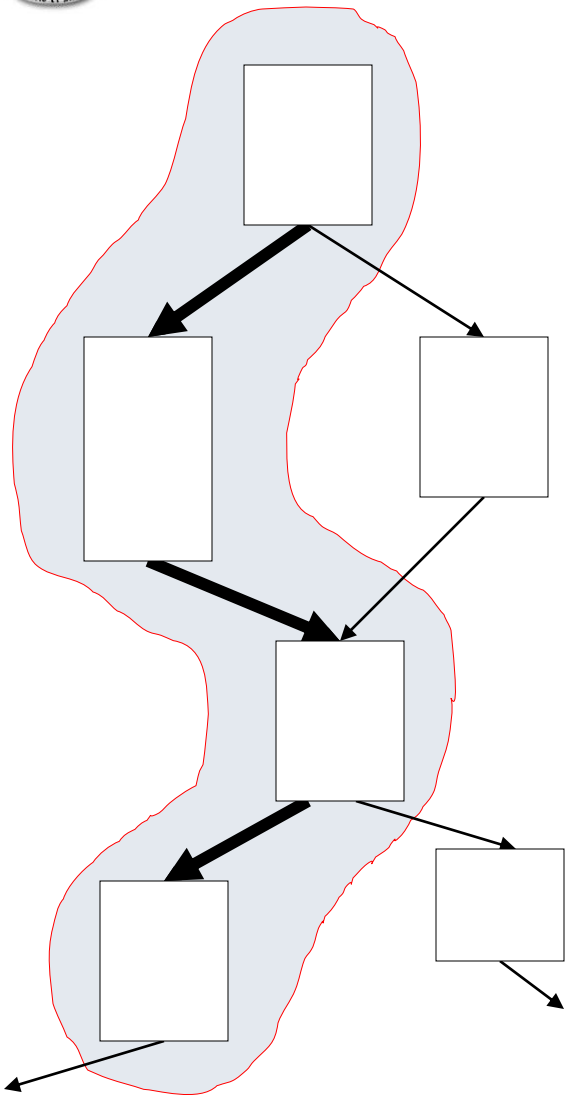


What if there are no loops?



- Basic block defined by sequence of consecutive instructions. Every basic block ends with a branch.
- Instruction-level parallelism is hard to find in basic blocks
- Basic blocks illustrated by control flow graph

Trace Scheduling



- A trace is a sequence of basic blocks (a.k.a., long string of straight-line code)
- Trace Selection: Use profiling or compiler heuristics to find common sequences/paths
- Trace Compaction: Schedule whole trace into few VLIW instructions.
- Add fix-up code to cope with branches jumping out of trace. Undo instructions if control flower diverges from trace.

Problems with "Classic" VLIW

Object Code Challenges

- Compatibility: Need to recompile code for every VLIW machine, even across generations.
- Compatibility: Code specific to operation slots in instruction format and latencies of operations

- Code Size: Instruction padding wastes instruction memory/cache with nops for unfilled slots.
- Code Size: Loop unrolling, software pipelining increases code footprint.

Scheduling Variable Latency Operations

- Effective schedules rely on known instruction latencies
- Caches, memories produce unpredictable variability

Problems with "Classic" VLIW

Object Code Challenges

- Compatibility: Need to recompile code for every VLIW machine, even across generations.
- Compatibility: Code specific to operation slots in instruction format and latencies of operations

- Code Size: Instruction padding wastes instruction memory/cache with nops for unfilled slots.
- Code Size: Loop unrolling, software pipelining increases code footprint.

Scheduling Variable Latency Operations

- Effective schedules rely on known instruction latencies
- Caches, memories produce unpredictable variability

Intel Itanium, EPIC IA-64

Explicitly Parallel Instruction Computing (EPIC)

- Computer architecture style (e.g., CISC, RISC, EPIC)

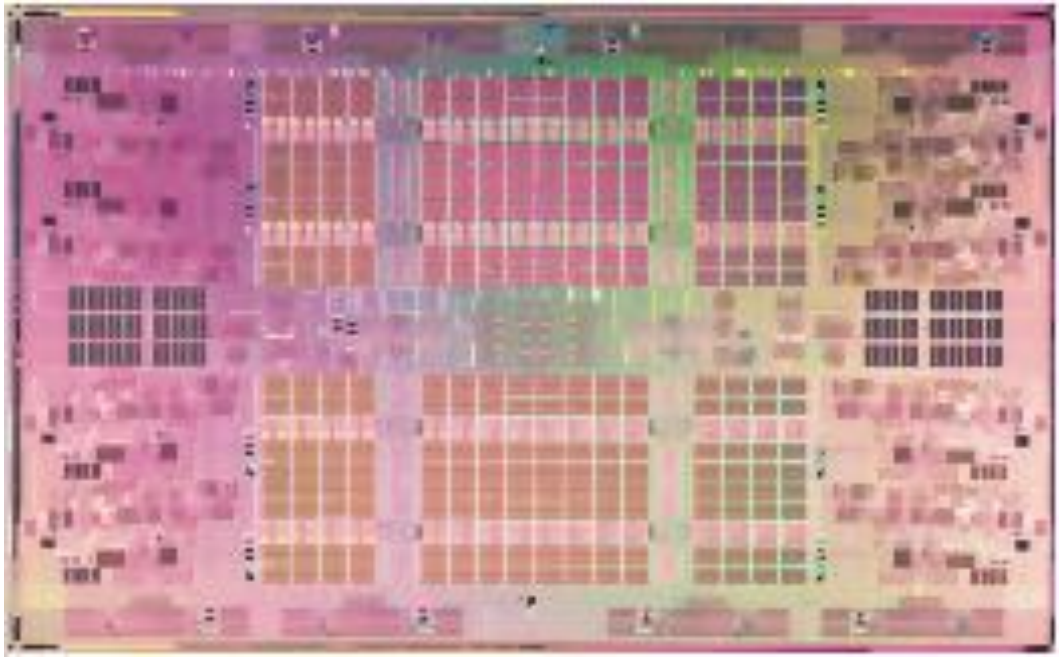
IA-64

- Instruction set architecture (e.g., x86, MIPS, IA-64)
- IA-64 – Intel Architecture 64-bit

Implementations

- Merced, first implementation, 2001
- McKinley, second implementation, 2002
- Poulson, recent implementation, 2011

Intel Itanium, EPIC IA-64



- Eight 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
- 3 billion transistors

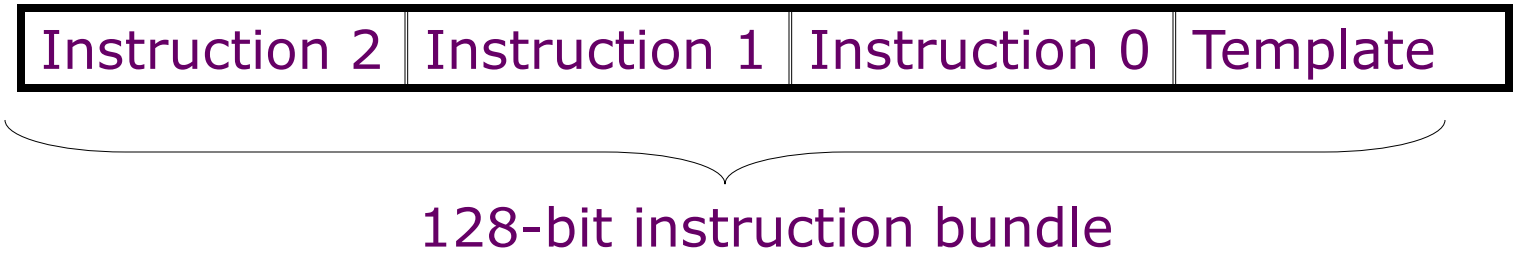
- Cores are 2-way multithreaded

- Each VLIW word is 128-bits, containing 3 instructions (op slots)

- Fetch 2 words per cycle → 6 instructions (op slots)

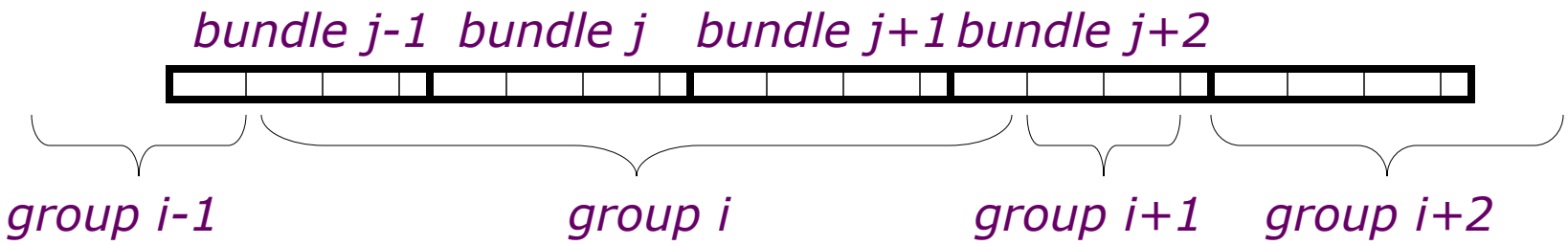
- Retire 4 words per cycle → 12 instructions (op slots)

Intel Itanium, EPIC IA-64



-Template bits describe grouping of these instructions with others in adjacent bundles

- Each group contains instructions that can execute in parallel



VLIW and Control Flow Challenges

Challenge

- Mispredicted branches limit ILP
- Trace selection groups basic blocks into larger ones
- Trace compaction schedules instructions into a single VLIW
- Requires fix-up code for branches that exit trace

Solution – Predicated Execution

- Eliminate hard to predict branches with predicated execution
- IA-64 instructions can be executed conditionally under predicate
- Instruction becomes a NOP if predicate register is false

VLIW and Control Flow Challenges

b0:
```
Inst 1
Inst 2
br a==b, b2
```
if

b1:
```
Inst 3
Inst 4
br b3
```
else

b2:
```
Inst 5
Inst 6
```
then

b3:
```
Inst 7
Inst 8
```

Four basic blocks

Predication

```
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
```

One basic block

On average >50% branches removed. Mahlke et al. 1995.

Limits of Static ILP

Software Instruction-level Parallelism (VLIW)

- Compiler complexity
- Code size explosion
- Unpredictable branches
- Variable memory latency and unpredictable cache behavior

Current Status

- Despite several attempts, VLIW has failed in general-purpose computing
- VLIW hardware complexity similar to in-order, superscalar hardware complexity. Limited advantage on large, complex applications
- Successful in embedded digital signal processing; friendly code

Summary

Out-of-order Superscalar

- Hardware complexity increases super-linearly with issue-width.

Very Long Instruction Word (VLIW)

- Compiler explicitly schedules parallel instructions
- Unrolling and software pipelining loops

Predication

- Mitigates branches in VLIW machines
- Add predicates to operations.
- If predicate is false, instruction does not affect architected state
- Mahlke et al. "A comparison of full and partial predicated execution support for ILP processors" 1995.

Acknowledgements

These slides contain material developed and copyright by

- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- Arvind Krishnamurthy (U. Washington)
- John Kubiatowicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)