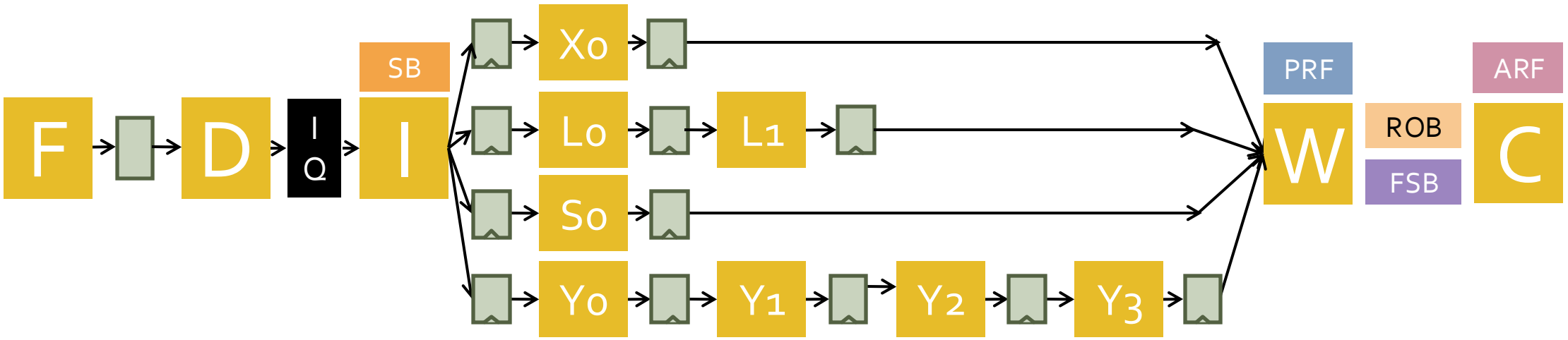
Computer Architecture
VLIW

Ting-Jung Chang

NYCU CS

Recap: Superscalar/OOO



Course Administration

- Lab2 is out

Exploiting ILP (Instruction-Level Parallelism)

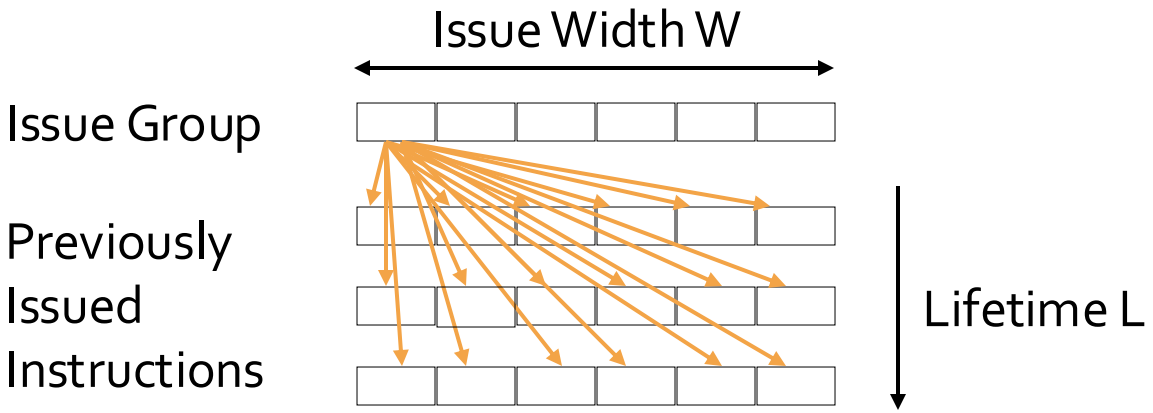
Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

H&P 6, Fig. 3.19

Reprise of Dynamic Scheduling

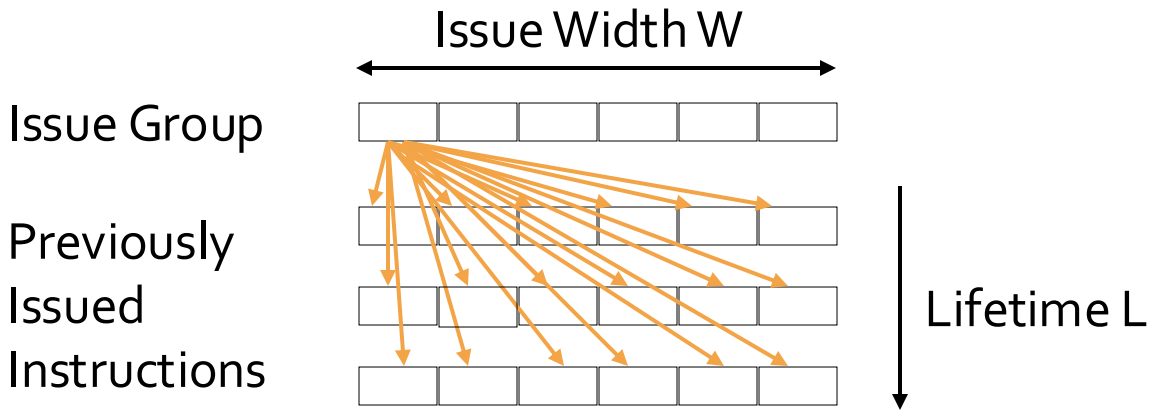
- Out-of-order (OOO) processors have key benefits
  - Overlap long-latency operations by…
  - Scheduling instructions over large window (speculation)
  - Responding to variable latencies (dynamism)
- This lets OOO processors keep the backend busy and sustain high ILP
- …But these mechanisms are complex & expensive
  - Overheads in superscalar structures
  - Complex scheduling logic & error conditions
    - E.g., detecting and rolling back speculative loads due to memory-memory RAW hazard

Superscalar 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

Superscalar Control Logic Scaling



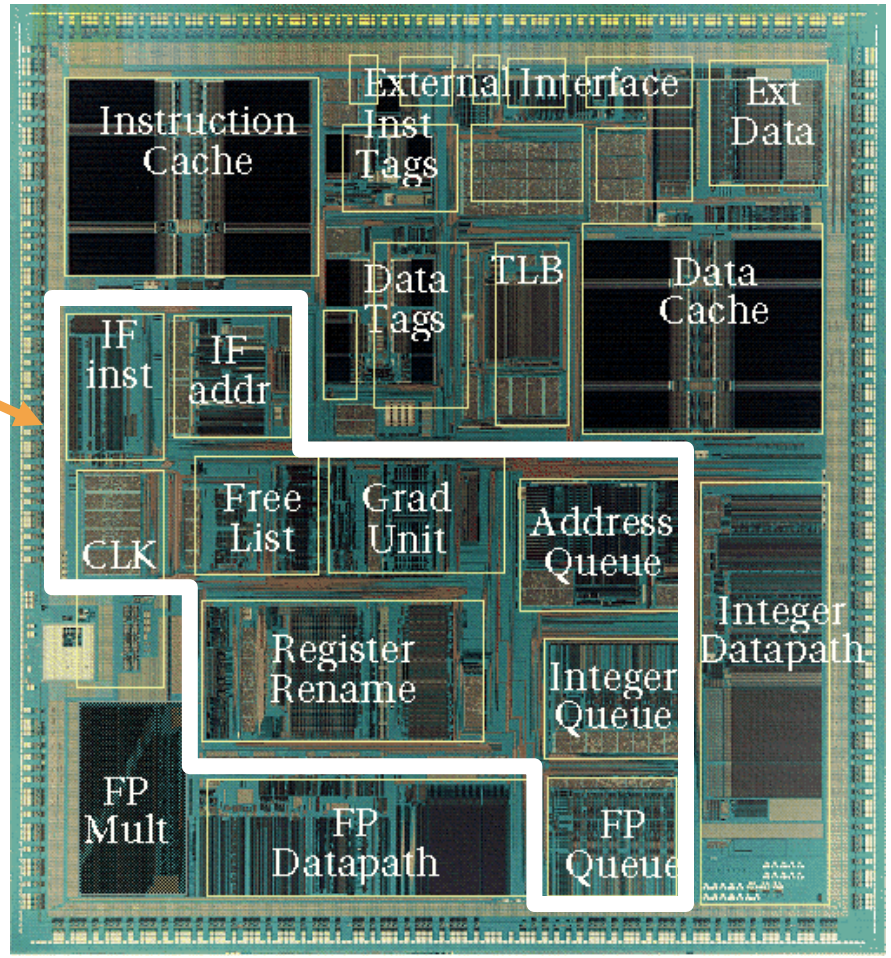
- Each issued instruction must somehow check against W*L instructions, i.e., growth in hardware $\propto$ W*(W*L)
- For in-order machines, L is related to pipeline latencies and check is done during issue (scoreboard)
- For out-of-order machines, L also includes time spent in IQ, SB, and check is done by broadcasting tags to waiting instructions at 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:

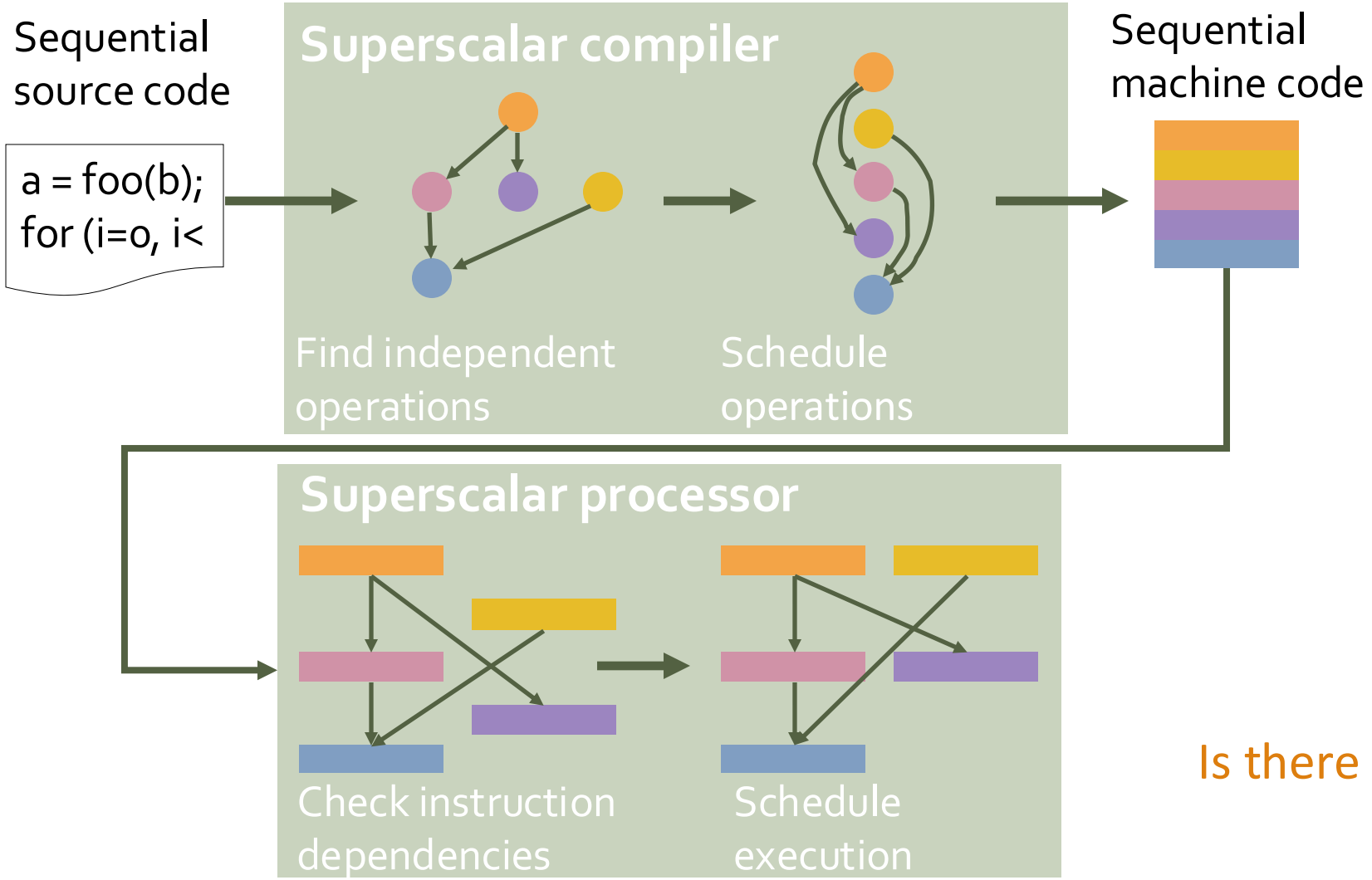
MIPS R10000

Control Logic



[A. Ahi et al., MIPS R10000 Superscalar Microprocessor, Hot Chips, 1995]
Image Credit: MIPS Technologies Inc. / Silicon Graphics Computer Systems

Sequential ISA Bottleneck

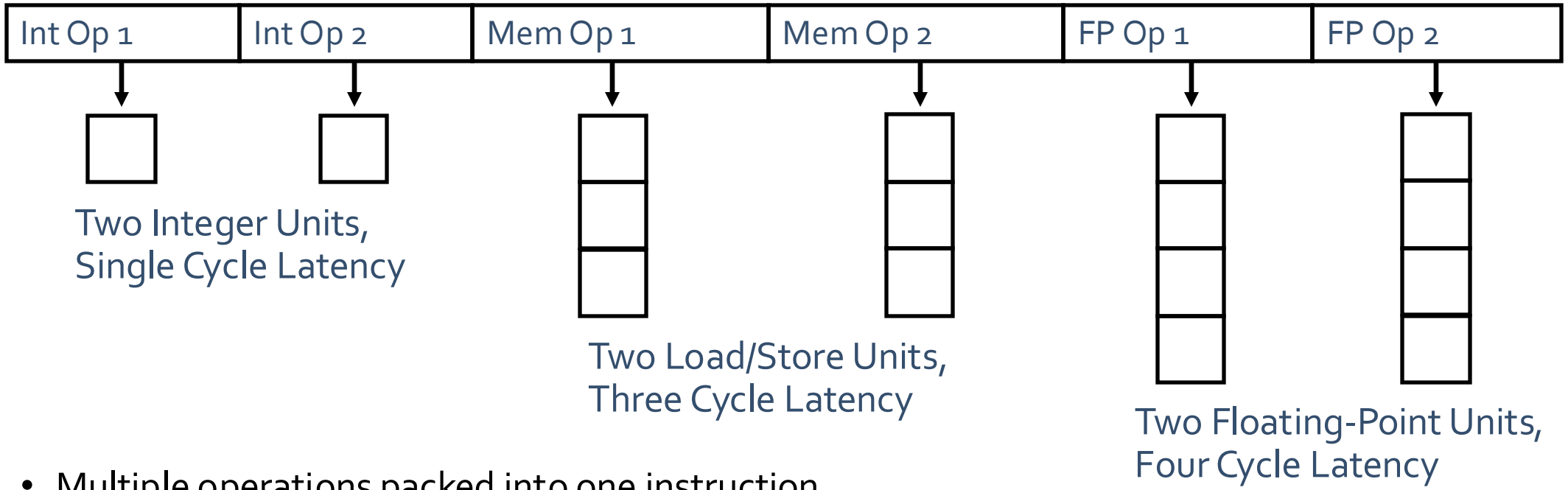


Is there a better way?

Sequential Instruction Sets

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

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

**Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau**

**Department of Computer Science, Yale University
New Haven, CT 06520**

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

VLIW

Equals (EQ) Scheduling Model

- Each operation takes exactly specified latency
  - The destination register will not be written until latency number of cycle
- Efficient register usage (Effectively more registers)
- No need for register renaming or buffering
  - Register writes whenever functional unit completes
  - Produce slightly shorter schedules due to register reuse
- Compiler depends on not having registers visible early

VLIW

Less-Than-or-Equals (LEQ) Scheduling Model

- Each operation may take less than or equal to its specified latency
  - Destination can be written any time after instruction issue
  - Dependent instruction still needs to be scheduled after instruction latency
- Precise interrupts simplified
- Binary compatibility preserved when latencies are reduced

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

VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution
  - Fill operation slots
- Guarantees intra-instruction parallelism
  - Ensure operations within same instruction are independent
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

- Burden on the compiler?

Burden on the Compiler

- Basic objective: Have the compiler mitigate the effects of data dependencies and control dependencies

- Need to have a large pool of instructions from which we can schedule. Need large regions!
- Need to have a lot of registers to overcome false dependencies

Loop Execution

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

Compile

```
loop:  fld  f1, 0(x1)
       addi x1, x1, 8
       fadd f2, f0, f1
       fsd  f2, 0(x2)
       addi x2, x2, 8
       bne  x1, x3, loop
```

Schedule

loop:

Int1	Int2	M1	M2	FP+	FPx

The latency of each instruction is fixed (e.g., 3 cycle ld, 4 cycle fadd)

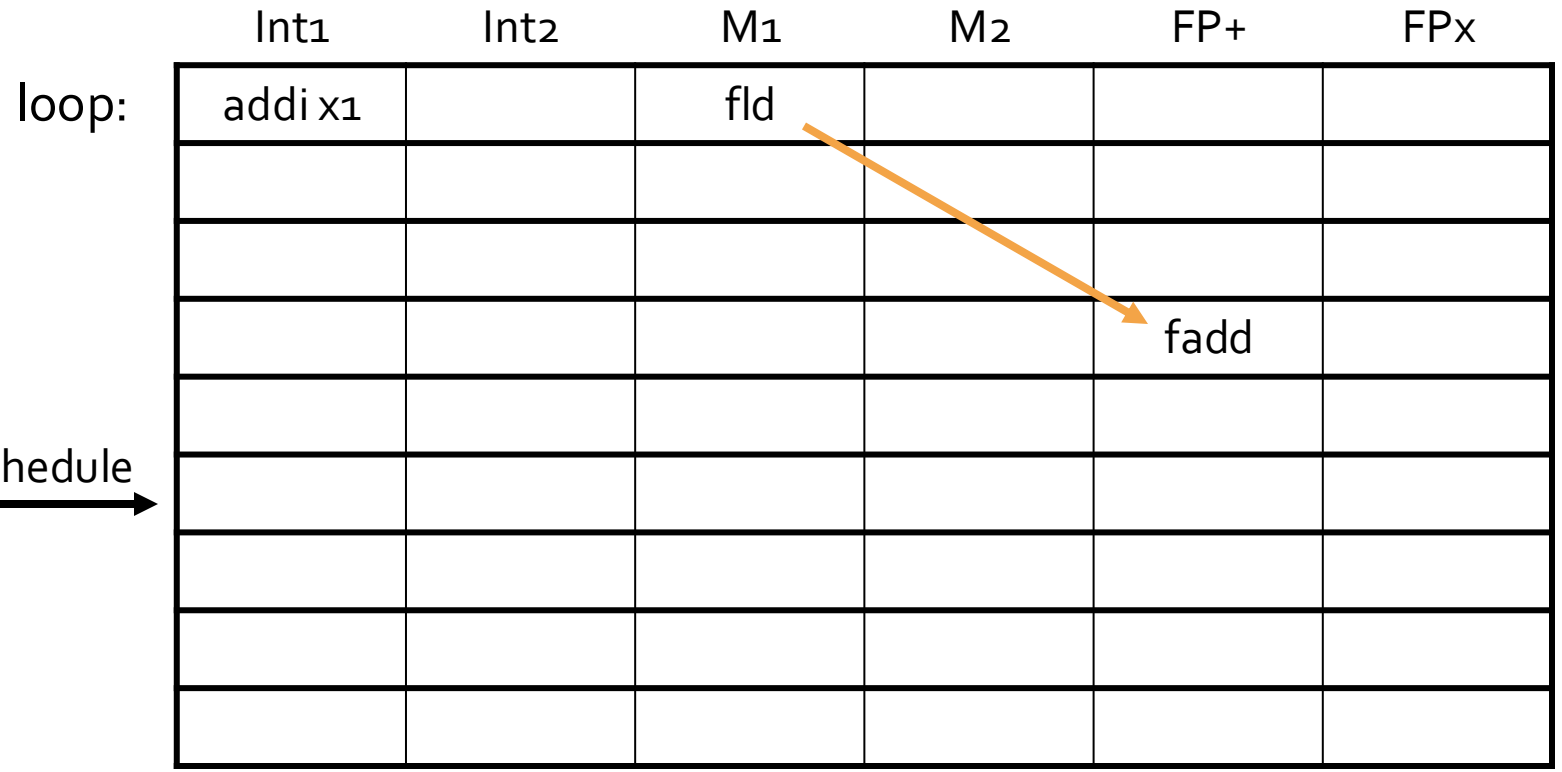
Loop Execution

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

Compile

```
loop:  fld  f1, 0(x1)
       addi x1, x1, 8
       fadd f2, f0, f1
       fsd  f2, 0(x2)
       addi x2, x2, 8
       bne  x1, x3, loop
```

Schedule



Instr-1: Load A[i] and increment i (x1) in parallel
Instr-2: Wait for load

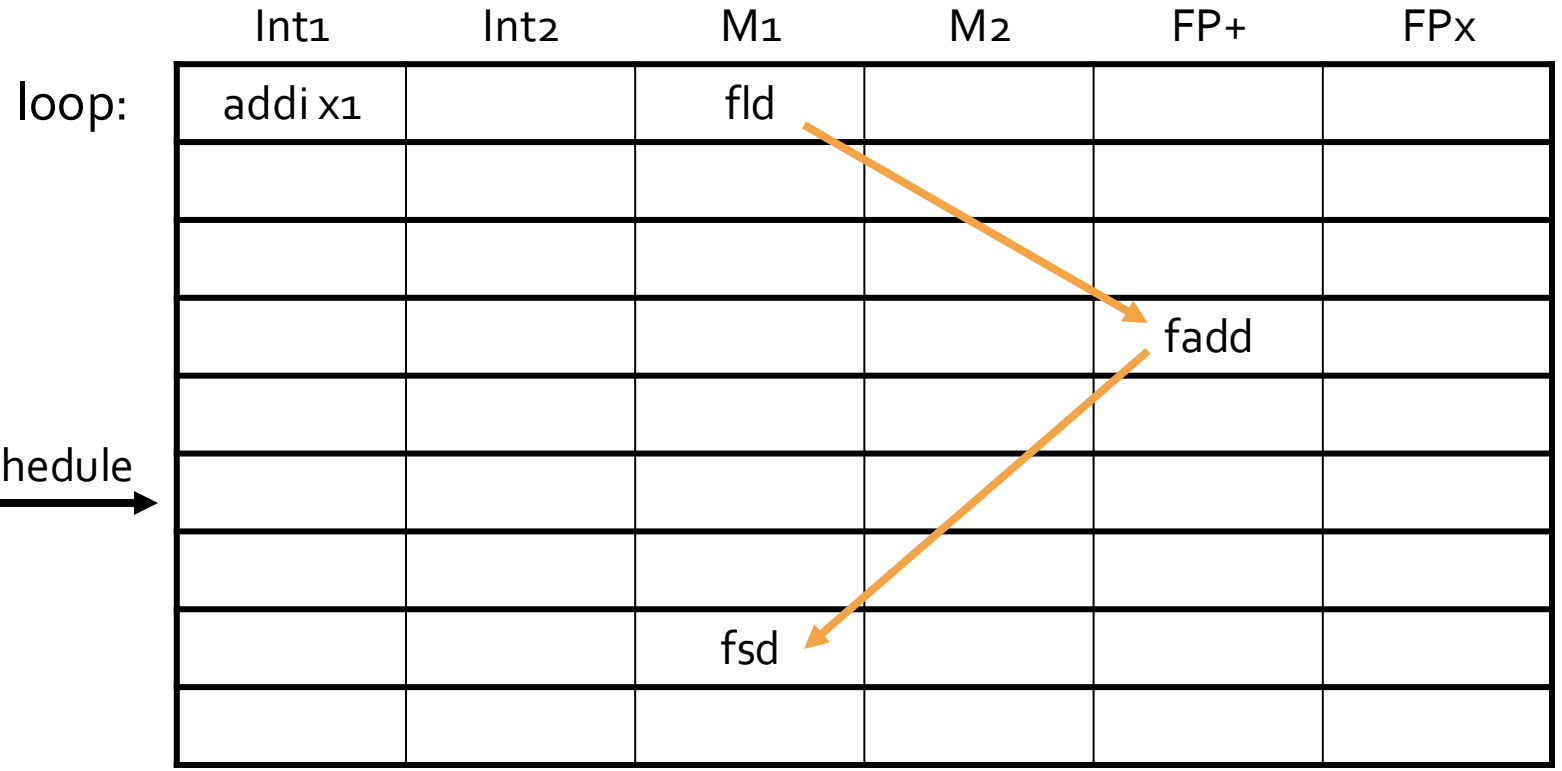
Loop Execution

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

Compile

```
loop:  fld  f1, 0(x1)
       addi x1, x1, 8
       fadd f2, f0, f1
       fsd  f2, 0(x2)
       addi x2, x2, 8
       bne  x1, x3, loop
```

Schedule



Instr-3: Wait for add. Store B[i]

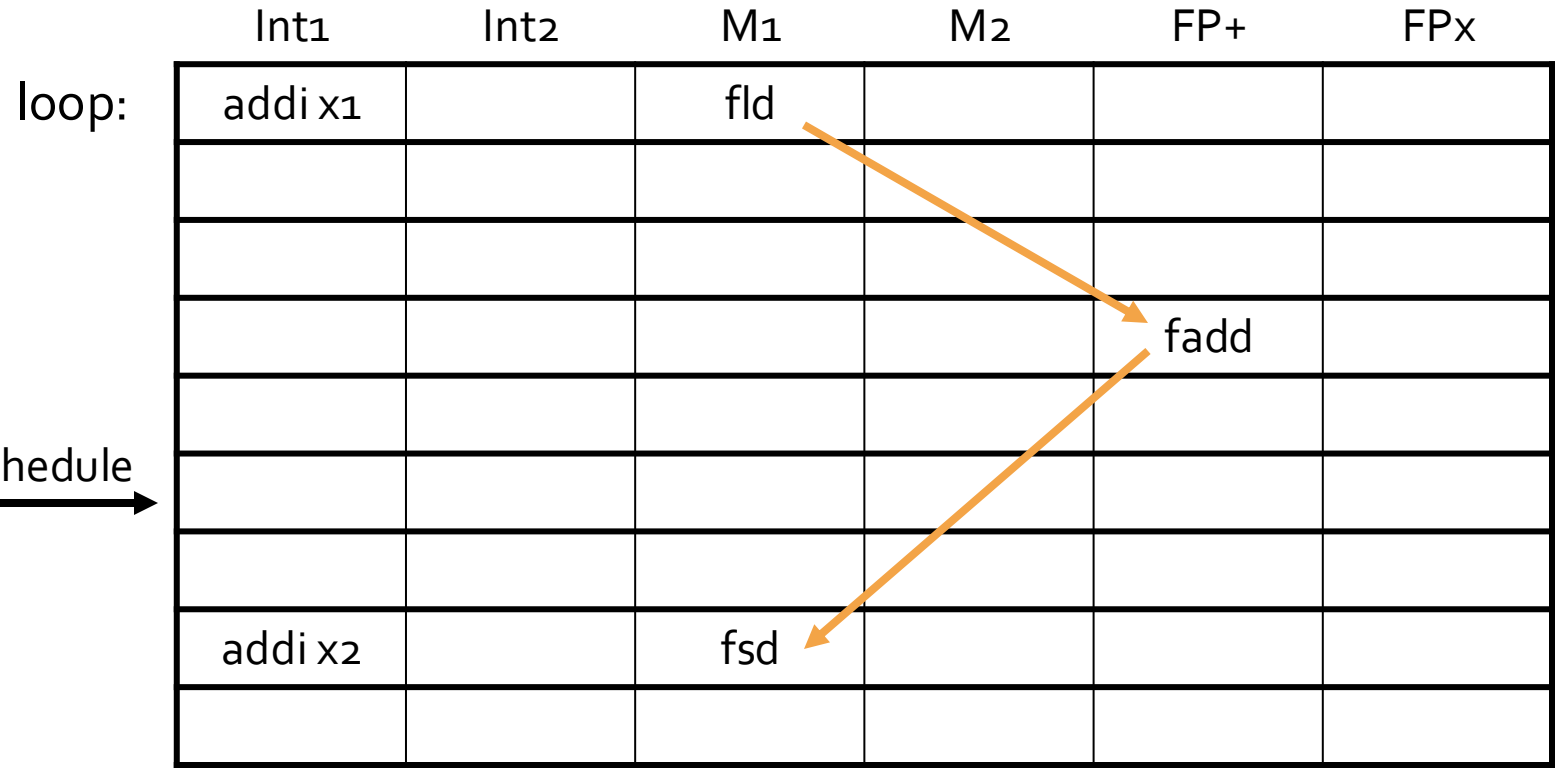
Loop Execution

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

Compile

```
loop:  fld  f1, 0(x1)
       addi x1, x1, 8
       fadd f2, f0, f1
       fsd  f2, 0(x2)
       addi x2, x2, 8
       bne  x1, x3, loop
```

Schedule



Instr-3: Wait for add. Store B[i], increment i (x2)

Loop Execution

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

Compile

```
loop:  fld  f1, 0(x1)
       addi x1, x1, 8
       fadd f2, f0, f1
       fsd  f2, 0(x2)
       addi x2, x2, 8
       bne  x1, x3, loop
```

Schedule

	Int1	Int2	M1	M2	FP+	FPx
loop:	add x1		fld			
					fadd	
	addi x2	bne	fsd			

How many FP ops/cycle?

1 fadd/ 8 cycles = 0.125

Instr-3: Wait for add. Store B[i], increment i (x2), branch in parallel

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;
}
```

Example: unroll inner loop to perform 4 iterations at once

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: fld  f1, 0(x1)
      fld  f2, 8(x1)
      fld  f3, 16(x1)
      fld  f4, 24(x1)
      addi x1, x1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      fsd  f5, 0(x2)
      fsd  f6, 8(x2)
      fsd  f7, 16(x2)
      fsd  f8, 24(x2)
      addi x2, x2, 32
      bne  x1, x3, loop
```

Schedule

	Int1	Int2	M1	M2	FP+	FPx
loop:			fld f_1			
			fld f_2			
			fld f_3			
	addi x1		fld f_4			

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop: fld  f1, 0(x1)
      fld  f2, 8(x1)
      fld  f3, 16(x1)
      fld  f4, 24(x1)
      addi x1, x1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      fsd  f5, 0(x2)
      fsd  f6, 8(x2)
      fsd  f7, 16(x2)
      fsd  f8, 24(x2)
      addi x2, x2, 32
      bne  x1, x3, loop
```

Schedule

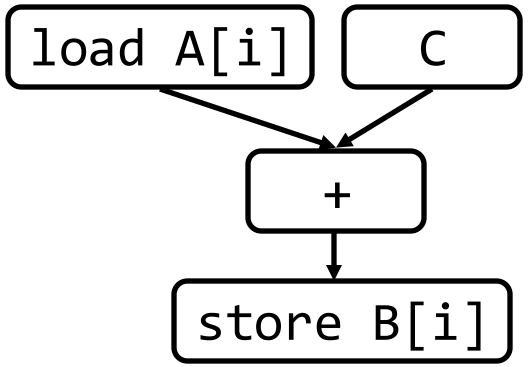
	Int1	Int2	M1	M2	FP+	FPx
loop:			fld f1			
			fld f2			
			fld f3			
	addi x1		fld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			fsd f5			
			fsd f6			
			fsd f7			
	addi x2	bne	fsd f8			

How many FP ops/cycle? 4 fadds / 11 cycles = 0.36

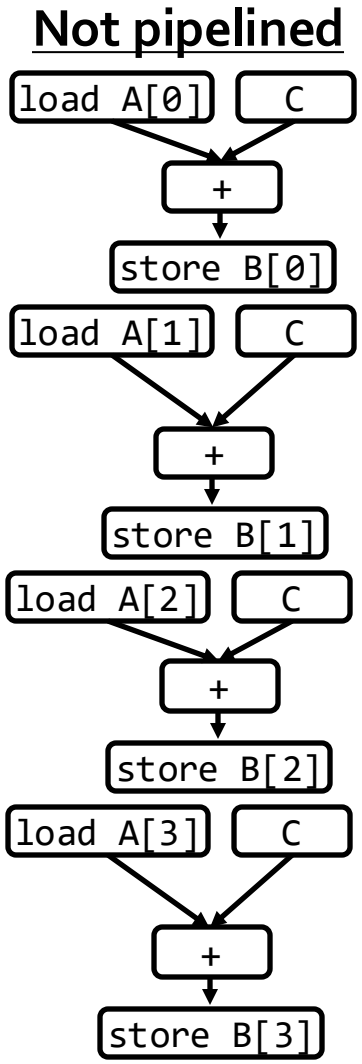
Software Pipelining

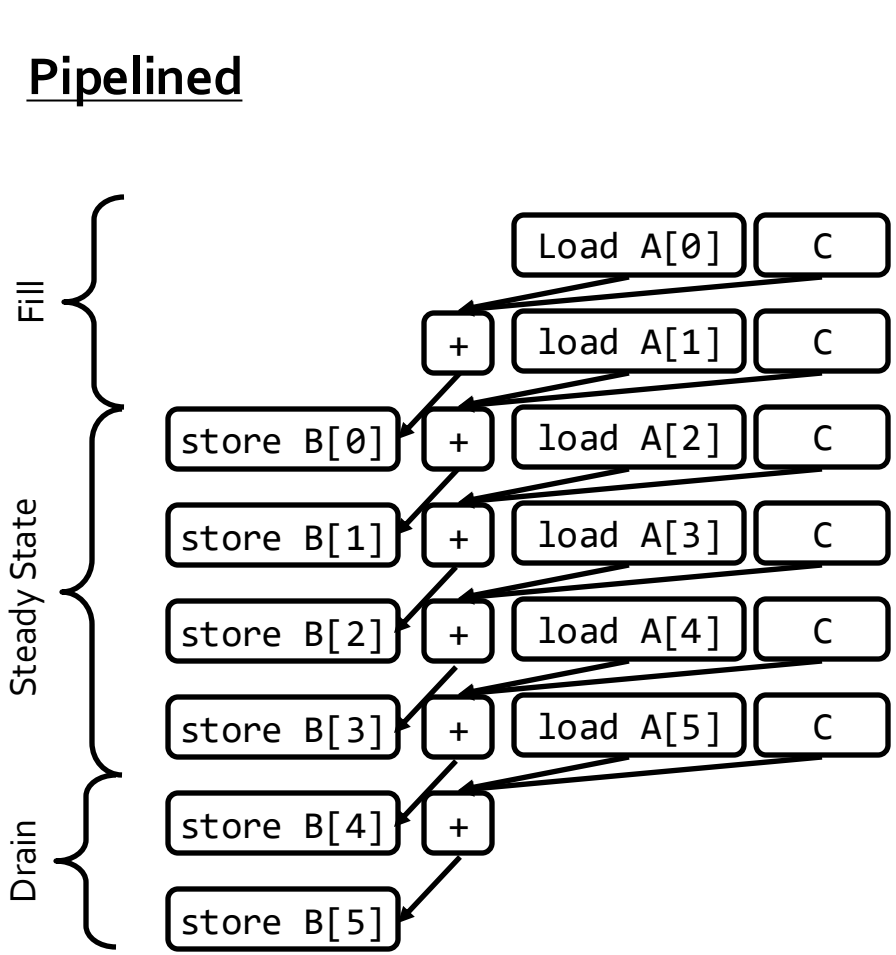
- Exploit independent loop iterations
  - If loop iterations are independent, then get more parallelism by scheduling instructions from different iterations
  - Example: Loop iterations are independent in the code sequence below.
  - Construct the data-flow graph for one iteration

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



Software Pipelining (Illustrated)





Software Pipelining

Unroll 4 ways first

```
loop: fld  f1, 0(x1)
      fld  f2, 8(x1)
      fld  f3, 16(x1)
      fld  f4, 24(x1)
      addi x1, x1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      fsd  f5, 0(x2)
      fsd  f6, 8(x2)
      fsd  f7, 16(x2)
      addi x2, x2, 32
      fsd  f8, -8(x2)
      bne  x1, x3, loop
```

Schedule →

Int1	Int2	M1	M2	FP+	FPx

Software Pipelining

Unroll 4 ways first

```
loop: fld  f1, 0(x1)
      fld  f2, 8(x1)
      fld  f3, 16(x1)
      fld  f4, 24(x1)
      addi x1, x1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      fsd  f5, 0(x2)
      fsd  f6, 8(x2)
      fsd  f7, 16(x2)
      addi x2, x2, 32
      fsd  f8, -8(x2)
      bne  x1, x3, loop
```

Schedule →

Int1	Int2	M1	M2	FP+	FPx
		fld f1			
		fld f2			
		fld f3			
addi x1		fld f4			
				fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
			fsd f5		
			fsd f6		
	addi x2		fsd f7		
	bne		fsd f8		

Software Pipelining

Unroll 4 ways first, then SW pipeline.

```
loop: fld  f1, 0(x1)
      fld  f2, 8(x1)
      fld  f3, 16(x1)
      fld  f4, 24(x1)
      addi x1, x1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      fsd  f5, 0(x2)
      fsd  f6, 8(x2)
      fsd  f7, 16(x2)
      addi x2, x2, 32
      fsd  f8, -8(x2)
      bne  x1, x3, loop
```

Schedule →

Int1	Int2	M1	M2	FP+	FPx
		fld f1			
		fld f2			
		fld f3			
addi x1		fld f4			
		fld f1		fadd f5	
		fld f2		fadd f6	
		fld f3		fadd f7	
addi x1		fld f4		fadd f8	
			fsd f5	fadd f5	
			fsd f6	fadd f6	
	addi x2		fsd f7	fadd f7	
	bne		fsd f8	fadd f8	
			fsd f5		
			fsd f6		
	addi x2		fsd f7		
	bne		fsd f8		

Software Pipelining

Unroll 4 ways first, then SW pipeline.

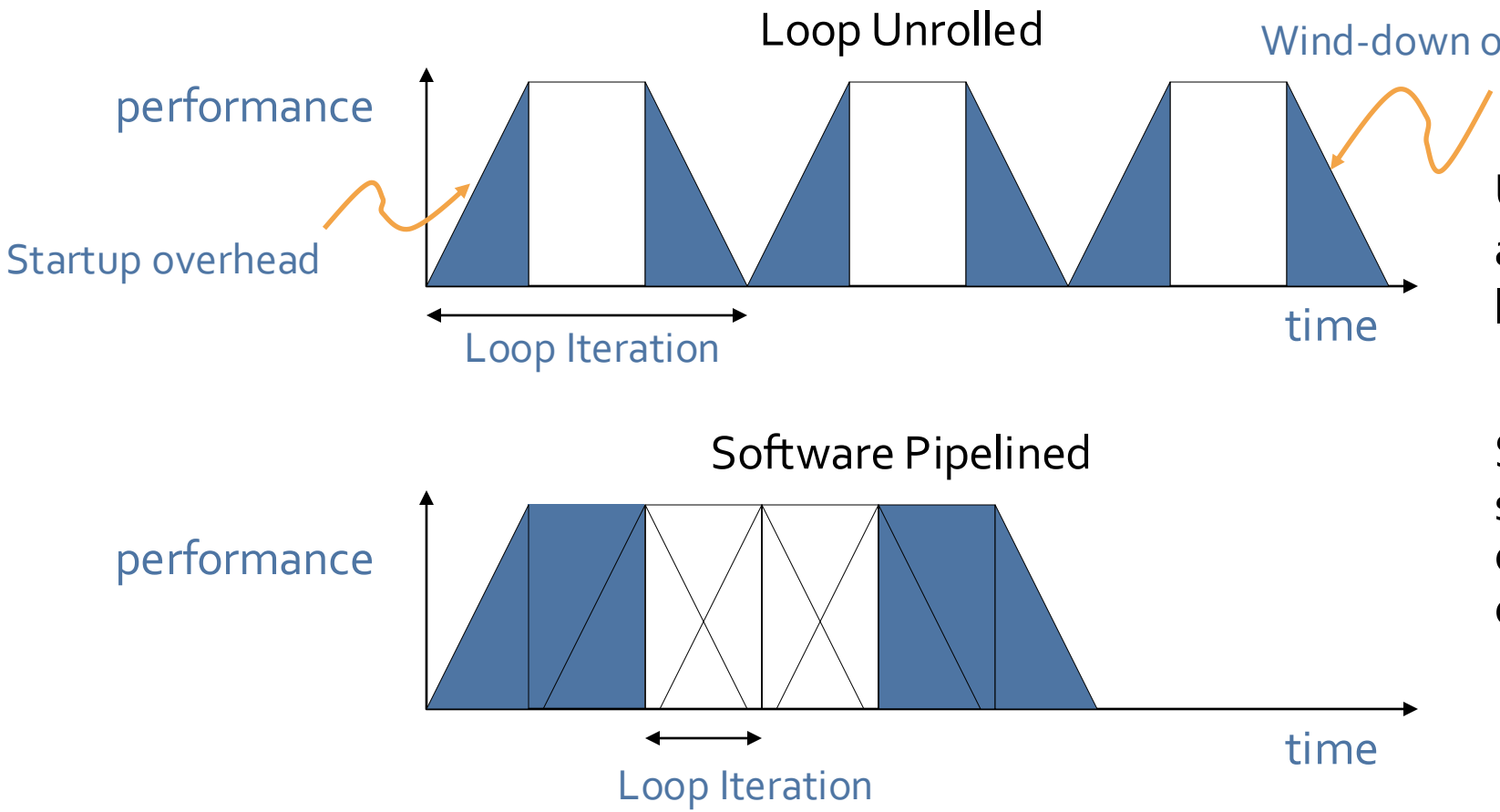
```
loop: fld  f1, 0(x1)
      fld  f2, 8(x1)
      fld  f3, 16(x1)
      fld  f4, 24(x1)
      addi x1, x1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      fsd  f5, 0(x2)
      fsd  f6, 8(x2)
      fsd  f7, 16(x2)
      addi x2, x2, 32
      fsd  f8, -8(x2)
      bne  x1, x3, loop
```

Schedule →

	Int1	Int2	M1	M2	FP+	FPx
prolog			fld f1			
			fld f2			
			fld f3			
	addi x1		fld f4			
			fld f1		fadd f5	
			fld f2		fadd f6	
			fld f3		fadd f7	
	addi x1		fld f4		fadd f8	
iterate (loop:)			fld f1	fsd f5	fadd f5	
			fld f2	fsd f6	fadd f6	
		addi x2	fld f3	fsd f7	fadd f7	
	addi x1	bne	fld f4	fsd f8	fadd f8	
epilog				fsd f5	fadd f5	
				fsd f6	fadd f6	
		addi x2		fsd f7	fadd f7	
		bne		fsd f8	fadd f8	
				fsd f5		

How many FP ops/cycle? 4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling

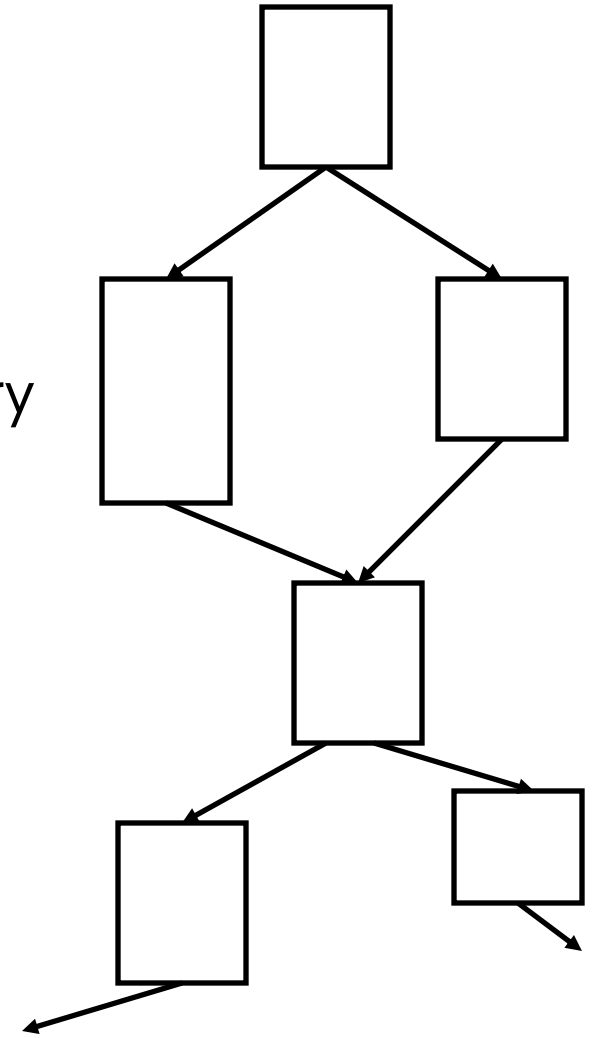


Unrolled loops pay fill and drain costs once per loop iteration.

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

What if there are no loops?

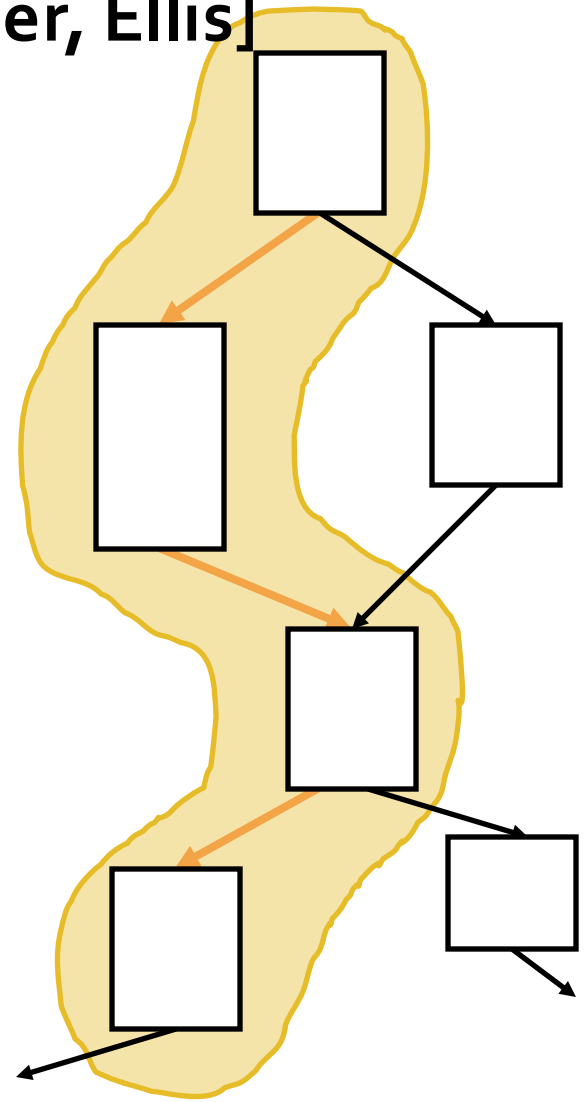
Basic block
- Single entry
- Single exit



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

Trace Scheduling

[Fisher, Ellis]

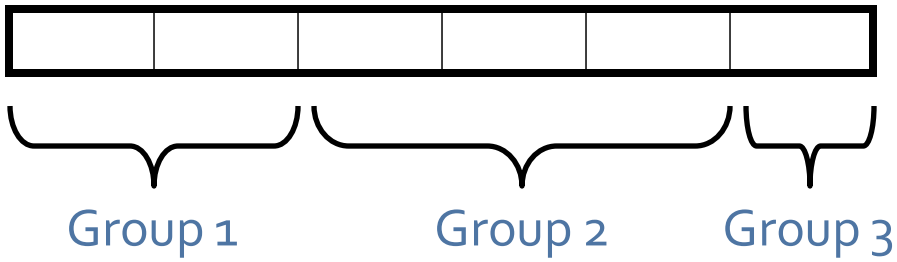


- Pick string of basic blocks, a trace, that represents most frequent branch path
- **Trace Selection**: profiling feedback or compiler heuristics to find common branch paths
- **Trace Compaction**: Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace. Undo instructions if control flower diverges from 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 a significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path
- Precise Interrupts can be challenging
  - Does fault in one portion of bundle fault whole bundle?
  - EQ Model has problem with single step, etc.

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

Predication

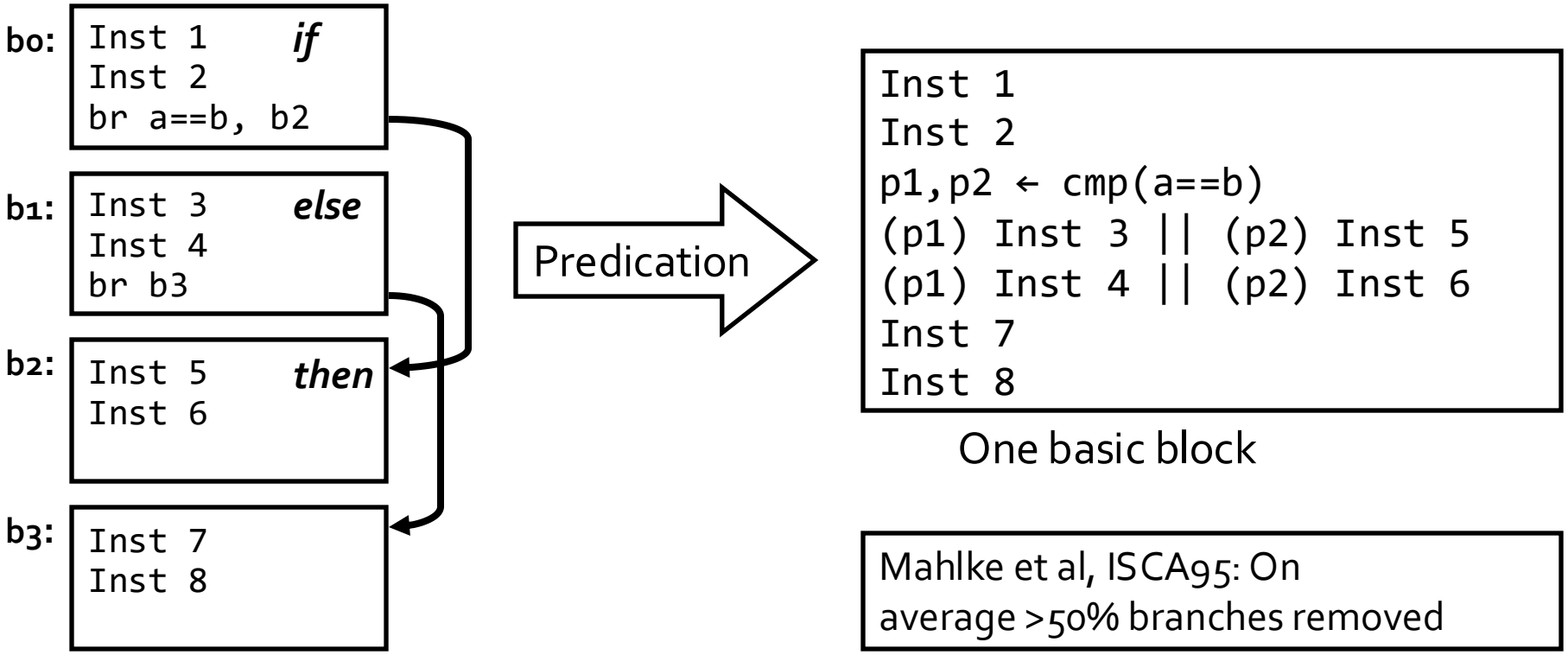
- Problem: Mispredicted branches limit ILP
- Solution: Eliminate hard to predict branches with predicated execution
- Predication helps with small branch regions and/or branches that are hard to predict by turning control flow into data flow
- Most basic form of predication: conditional moves
  - `movz rd, rs1, rs2 if ( R[rs2] == 0 ) then R[rd] <- R[rs1]`
  - `movn rd, rs1, rs2 if ( R[rs2] != 0 ) then R[rd] <- R[rs1]`

```
if (a<b)     slt x1, x2, x3   slt x1, x2, x3
 x=a         beq x1, x0, L1   movz x4, x3, x1
else         move x4, x2      movn x4, x2, x1
 x=b         j L2
          L1:move x4, x3
          L2:
```

What if-then-else has many instructions?
What if unbalanced?

Full Predication

- Almost all instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



Leveraging Speculative Execution and Reacting to Dynamic Events in a VLIW

Speculation:

- Moving instructions across branches
- Moving memory operations past other memory operations

Dynamic Events:

- Cache Miss
- Exceptions
- Branch Mispredict

Code Motion

Before Code Motion

```
mul  x1,  x2,  x3
addi x11, x10, 1
mul  x5,  x1,  x4
mul  x7,  x5,  x6
sw   x7,  0(x16)
addi x12, x11, 1
lw   x14, 0(x9)
add  x13, x12, x14
add  x14, x12, x13
bne  x16, x0, target
```

After Code Motion

```
lw   x14, 0(x9)
mul  x1,  x2,  x3
addi x11, x10, 1
addi x12, x11, 1
mul  x5,  x1,  x4
add  x13, x12, x14
mul  x7,  x5,  x6
add  x14, x12, x13
sw   x7,  0(x16)
bne  x16, x0, target
```

Scheduling and Bundling

Before Bundling

```
lw   x14, 0(x9)
addi x11, x10, 1
mul  x1,  x2,  x3
addi x12, x11, 1
mul  x5,  x1,  x4
add  x13, x12, x14
mul  x7,  x5,  x6
add  x14, x12, x13
sw   x7,  0(x16)
bne  x16, x0, target
```

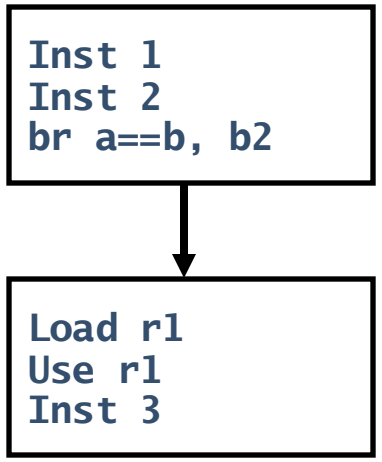
After Bundling

```
{lw   x14, 0(x9)
 addi x11, x10, 1
 mul  x1,  x2,  x3}
{addi x12, x11, 1
 mul  x5,  x1,  x4}
{add  x13, x12, x14
 mul  x7,  x5,  x6}
{add  x14, x12, x13
 sw   x7,  0(x16)
 bne  x16, x0, target}
```

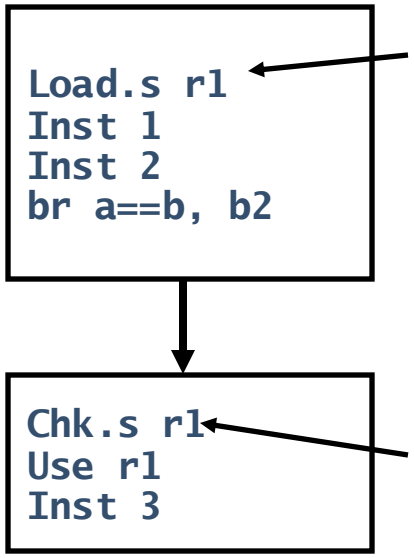
VLIW Speculative Execution

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

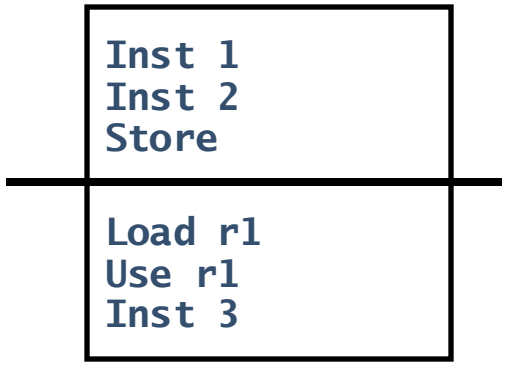
Check for exception in original home block jumps to fixup code if exception detected

Particularly useful for scheduling long latency loads early

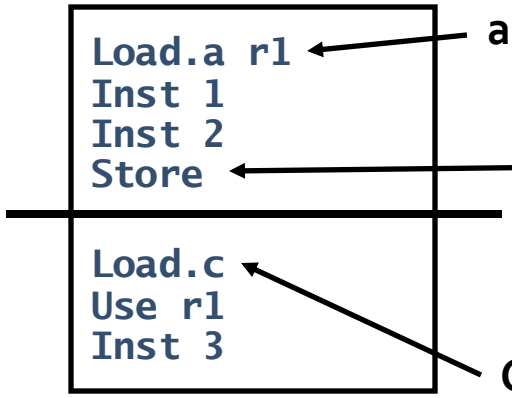
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check 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 (ALAT)

Check if load invalid (or missing), jump to fixup code if so

Requires associative hardware in address check table

ALAT (Advanced Load Address Table)

Store
CAM on Address

Register Number	Address	Size

Chk.a/ld.c
CAM on Register Number

- Load.a adds entry to ALAT
- Store removes entry if address/size match
- Check instruction (chk.a or ld.c) checks to make sure that address is still in ALAT and intermediary store did not push it out.
  - If not found, run recovery code (ex. re-execute load)

VLIW Multi-Way Branches

Problem: Long instructions provide few opportunities for branches

Solution: Allow one instruction to branch multiple directions

```
{ .mii
      cmp.eq P1, P2, R1, R2 (R1==R2)
      cmp.ne P3, P4, R4, R5 (R4!=R5)
      cmp.lt P5, P6, R8, R9 (R8<R9)
}
{ .bbb
 (P1) br.cond label1
 (P2) br.cond label2
 (P5) br.cond label3
}
// fall through code here
```

Scheduling Around Dynamic Events

- Cache Miss
  - Informing loads (loads nullify subsequent instructions)
  - Elbrus (Soviet/Russian) processor investigated branch on cache miss
- Branch Mispredict
  - Delay slots with predicated instructions
- Exceptions
  - Hard on superscalar also…

Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (CISC, RISC)
  - Explicitly Parallel Instruction Computing
- IA-64 was Intel's chosen ISA (x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code-compatible VLIW
- Merced was first Itanium implementation (8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Recent version, Poulson, eight cores, 32nm, announced 2011

Eight Core Itanium "Poulson" [Intel 2011]

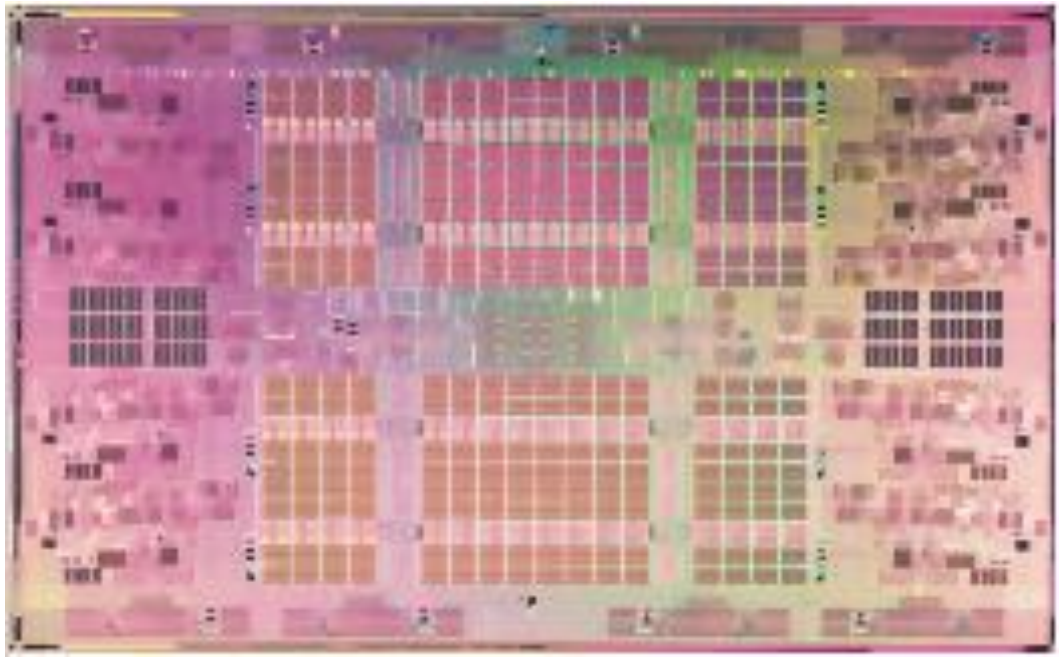
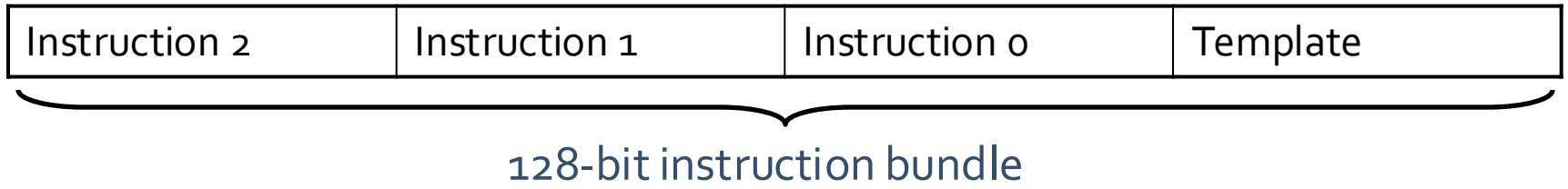


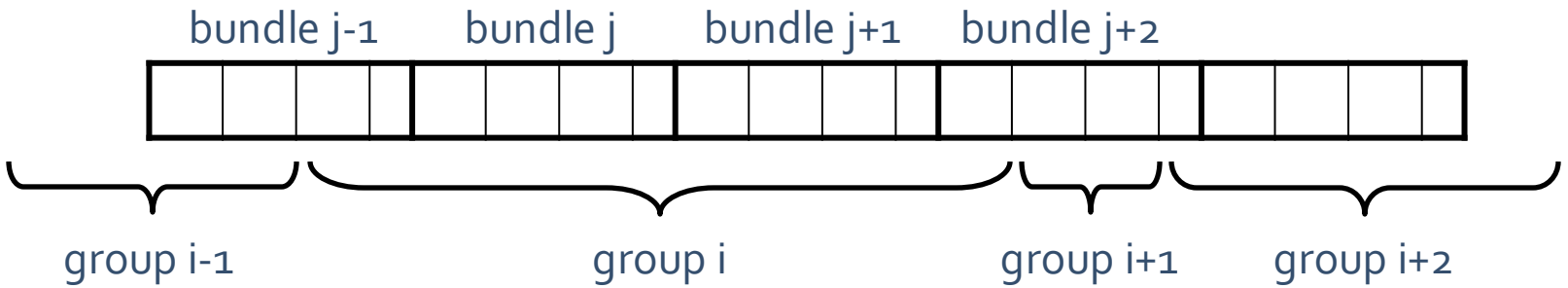
Image Credit: Intel

- 8 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
- **Over 3 billion transistors**
- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
  - Two 128-bit bundles
- Up to 12 insts/cycle execute
- 1.7GHz - 2.53GHz
- Updated to Kittson in 2017 with slight performance increases to 2.66GHz

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

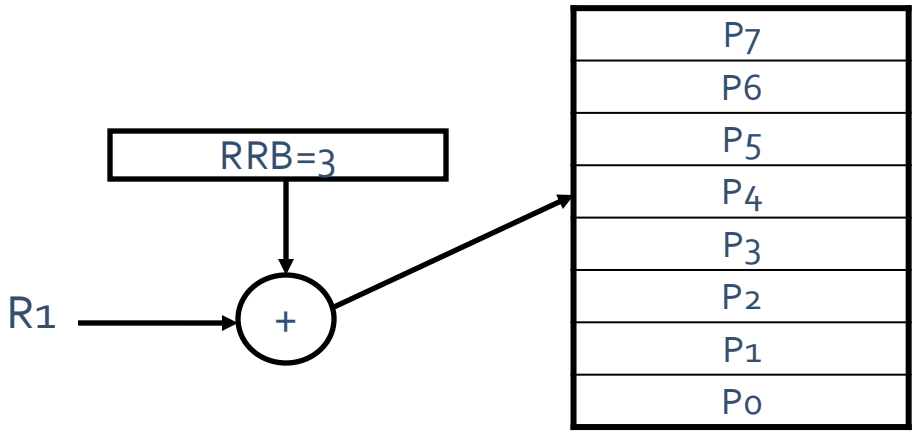
- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

- GPRs "rotate" to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration

IA-64 Rotating Register File

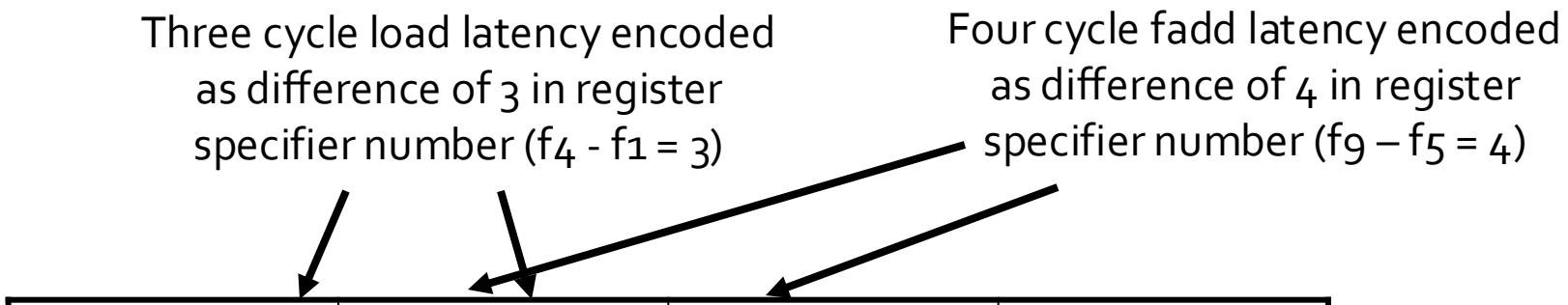
Problem: Scheduling of loops requires many register names and duplicated code in prolog and epilog

Solution: Allocate a new set of registers for each loop iteration



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)



fld f1, ()	fadd f5, f4, …	fsd f9, ()	bloop	
fld P9, ()	fadd P13, P12,	fsd P17, ()	bloop	RRB=8
fld P8, ()	fadd P12, P11,	fsd P16, ()	bloop	RRB=7
fld P7, ()	fadd P11, P10,	fsd P15, ()	bloop	RRB=6
fld P6, ()	fadd P10, P9,	fsd P14, ()	bloop	RRB=5
fld P5, ()	fadd P9, P8,	fsd P13, ()	bloop	RRB=4
fld P4, ()	fadd P8, P7,	fsd P12, ()	bloop	RRB=3
fld P3, ()	fadd P7, P6,	fsd P11, ()	bloop	RRB=2
fld P2, ()	fadd P6, P5,	fsd P10, ()	bloop	RRB=1

Intel Kills Itanium

- Donald Knuth "… Itanium approach that was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write."
- "Intel officially announced the end of life and product discontinuance of the Itanium CPU family on January 30th, 2019", Wikipedia

Why Itanium (IA-64) Failed..?

- Many **Architectural (ISA)** visible widgets tied hands of microarchitect designers
  - ALAT
  - Predication (increases inter-dependency of instructions)
  - Rotating Register file
- First implementations had very low clock rate
- Complex encoding
- Code size bloat
- Did not fundamentally solve some of the dynamic scheduling
- Large compiler complexity
  - Profiling needed for really good performance
- Limited Static Instruction Level Parallelism (ILP)
- People **did** build more complex superscalars. Area was not the most critical constraint. (Code compatibility and Power were critical)
- AMD64!

Limits of Static Scheduling

- Statically unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena.
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.
  - Google's Tensor Processing Unit is a VLIW machine.

VLIW in DSA

- TPU v2/v3: 322 bits VLIW bundle
  - 2 scalar slots
  - 4 vector slots (2 for load/store)
  - 2 matrix slots (push, pop)
  - 1 misc. slot
  - 6 immediates
- TPU v4i: 400 bits (25% wider)

Ten Lessons From Three Generations Shaped Google's TPUv4i

Industrial Product

Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson, Google LLC

Superscalar vs. VLIW

Superscalar Processors	VLIW Processors
Hardware make decisions	Compiler make decisions
Dynamic issue capability	Static issue capability
Complexity in hardware	Complexity in the compiler
Dynamic knowledge	Static knowledge (profiling)
Expensive for many FUs	Cheaper for many FUs
Legacy code	Reschedule (not so true..?)
Memory disambiguation	Code growth
Most current processors	IA-64, many DSPs

Recap: Exploiting ILP (Instruction-Level Parallelism)

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

H&P 6, Fig. 3.19

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)
  - Christopher Batten (Cornell)
  - David Wentzlaff (Princeton)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- Cornell material derived from course ECE 4750
- Princeton material derived from course ECE 475