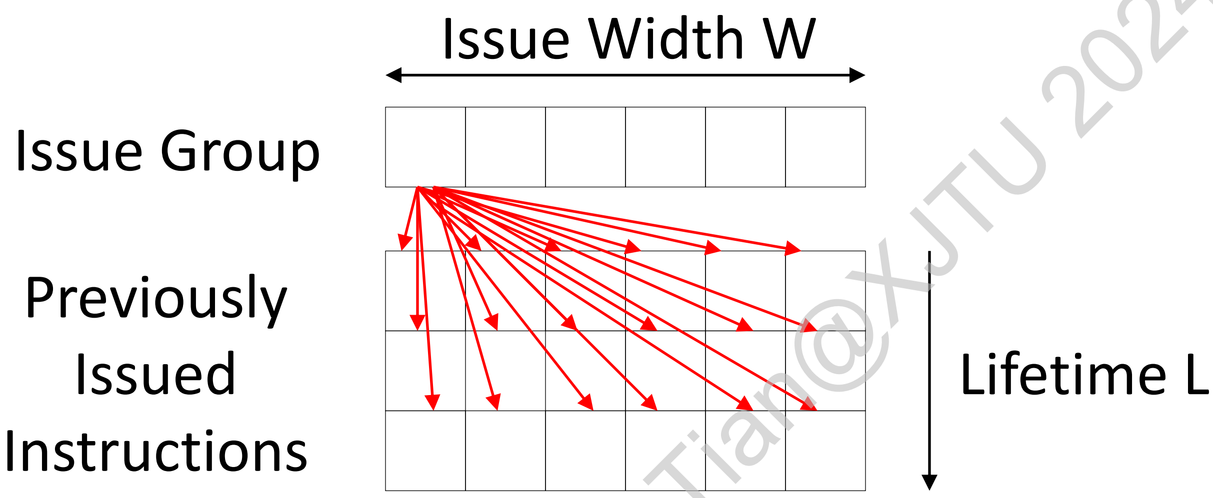
Computer Architecture

Lecture 09 – VLIW (Instruction level Parallelism)

Tian Xia
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

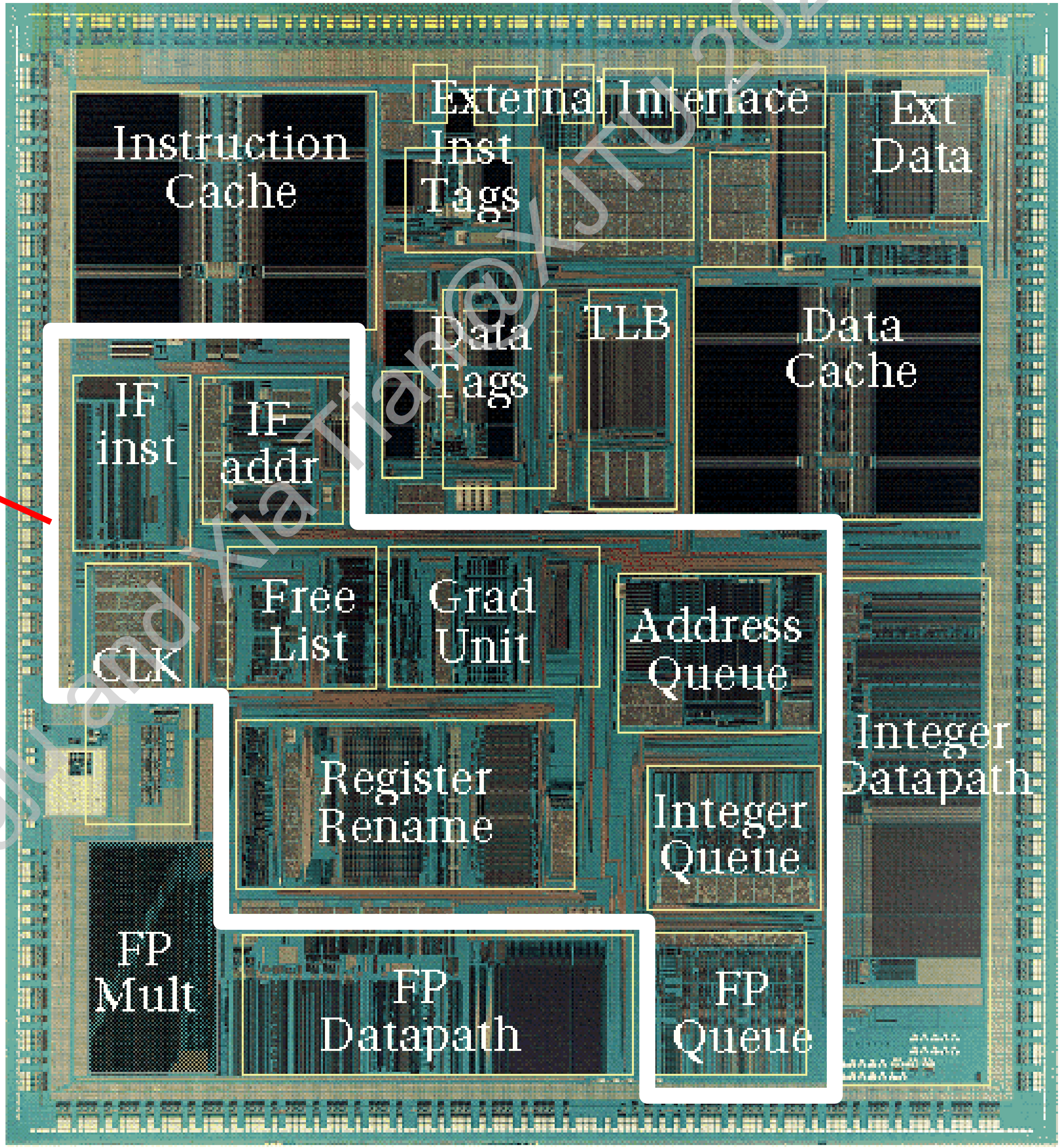
Superscalar Control Logic Scaling



- Each issued instruction must somehow check against **W*L** instructions to wake them up, i.e., growth in **hardware ∝ 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 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)

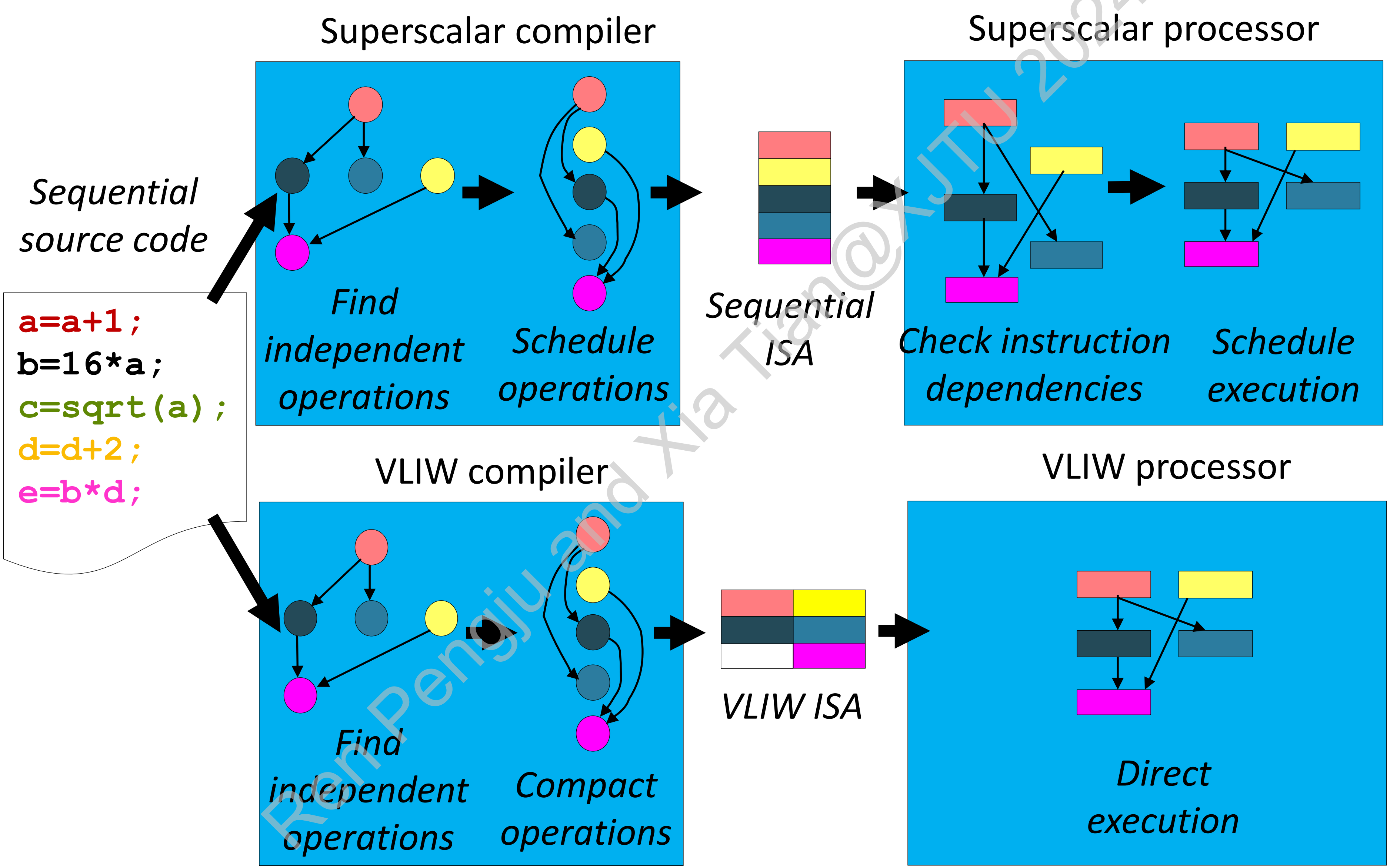
Out-of-Order Control Complexity: MIPS R10000



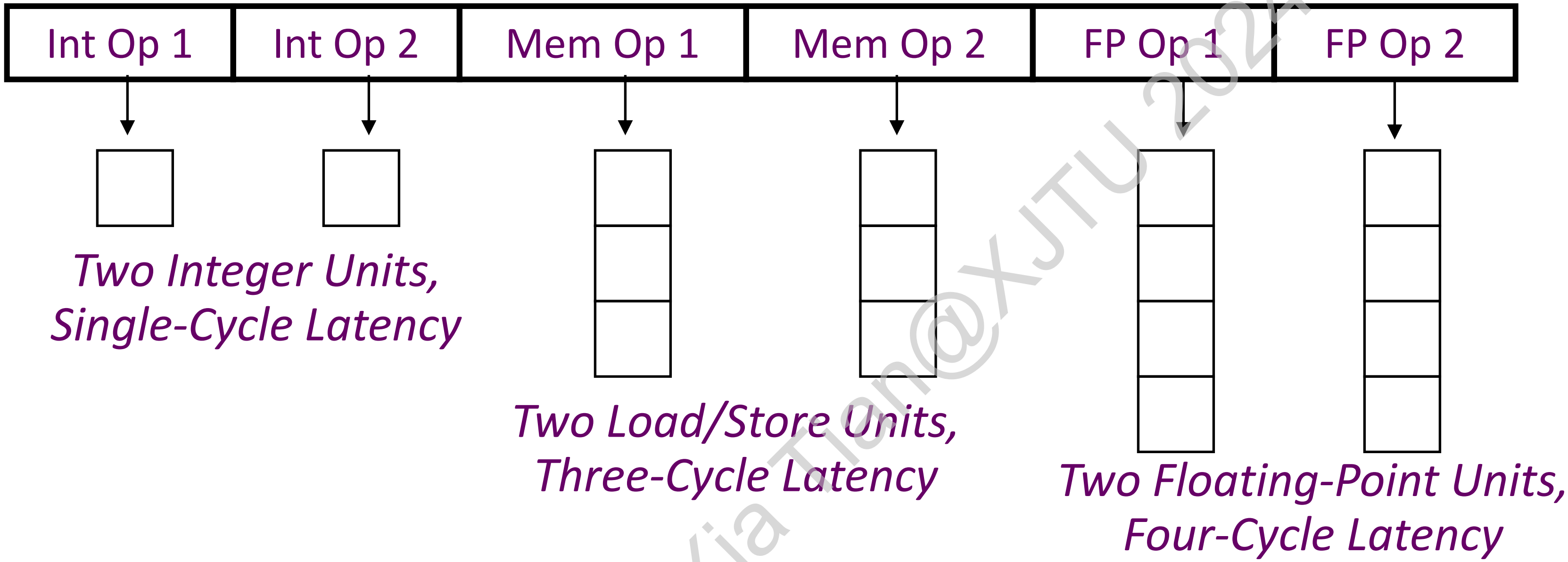
Control Logic

[SGI/MIPS Technologies Inc., 1995]

Sequential ISA Bottleneck



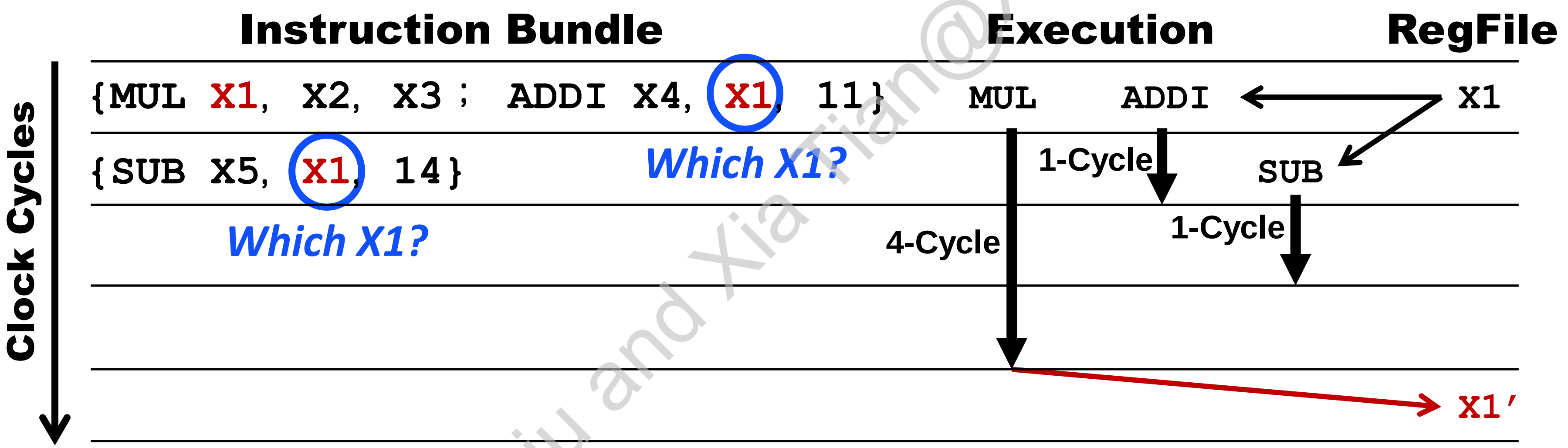
VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction (**bundle**)
- Each operation slot is for a **fixed function unit type**
- Fill unused slots with **paddling instructions (NOP)**
- **Constant operation latency** is specified(e.g. 3 for Mem Op)
- Architecture requires compiler to guarantee:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks

VLIW: Very Long Instruction Word

- Architecture requires guarantee of:
  - Parallelism within an instruction => **no cross-operation RAW check**
  - No data use before data ready => **no data interlocks**

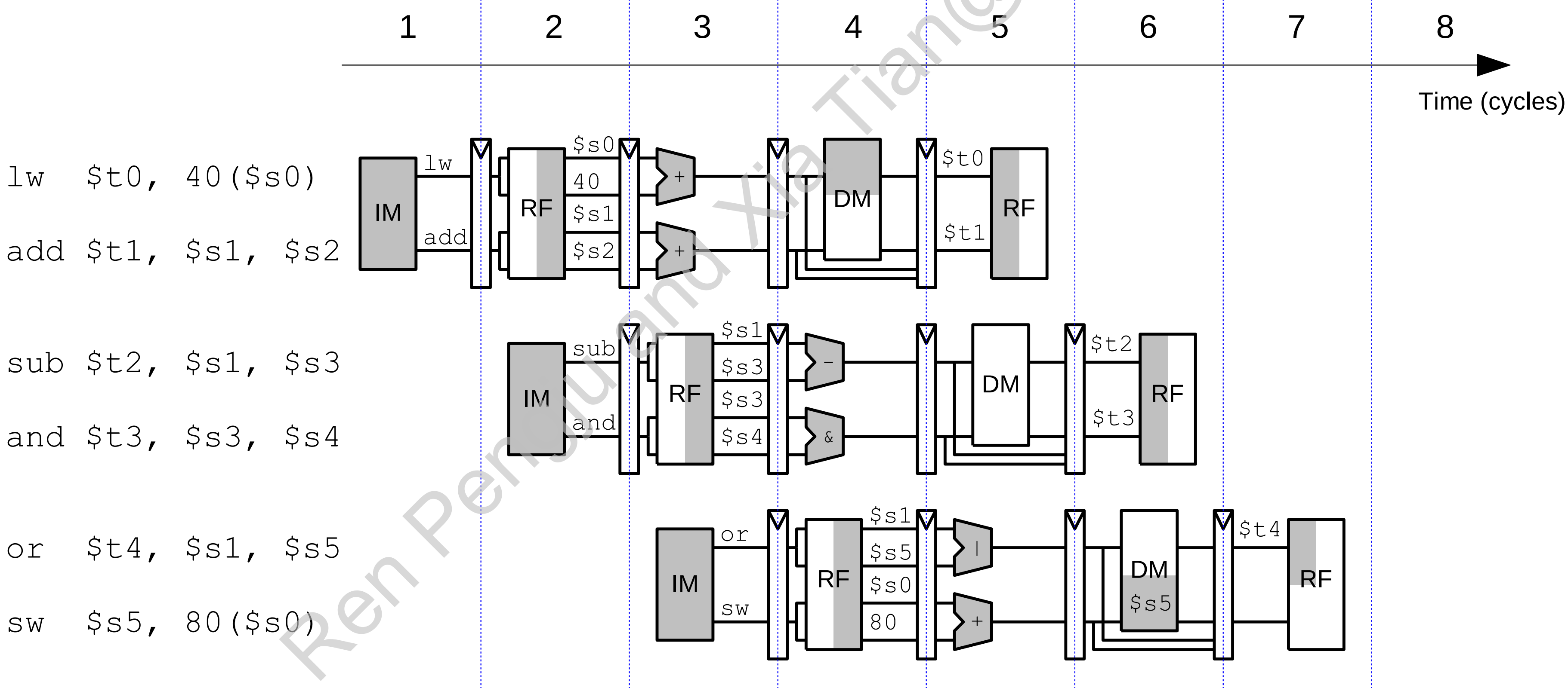


- Hardware **assumes compiler** is aware of all the **data dependency** and instruction **latency** of bundles

VLIW Performance Example (2-wide bundles)

```
lw  $t0, 40($s0)    add $t1, $s1, $s2
sub $t2, $s1, $s3   and $t3, $s3, $s4
or  $t4, $s1, $s5   sw  $s5, 80($s0)
```

Ideal IPC = 2



Actual IPC = 2 (6 instructions issued in 3 cycles)

VLIW Compiler Responsibilities

- **Statically schedule** (reorder) operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid **data hazards** (no interlocks)
  - Typically separates operations with **explicit NOPs**
- What if data **cache miss** occurs?
  - (**Scratchpad**) Usually control cache **manually** by programmer
  - Could stop the pipeline and **wait**

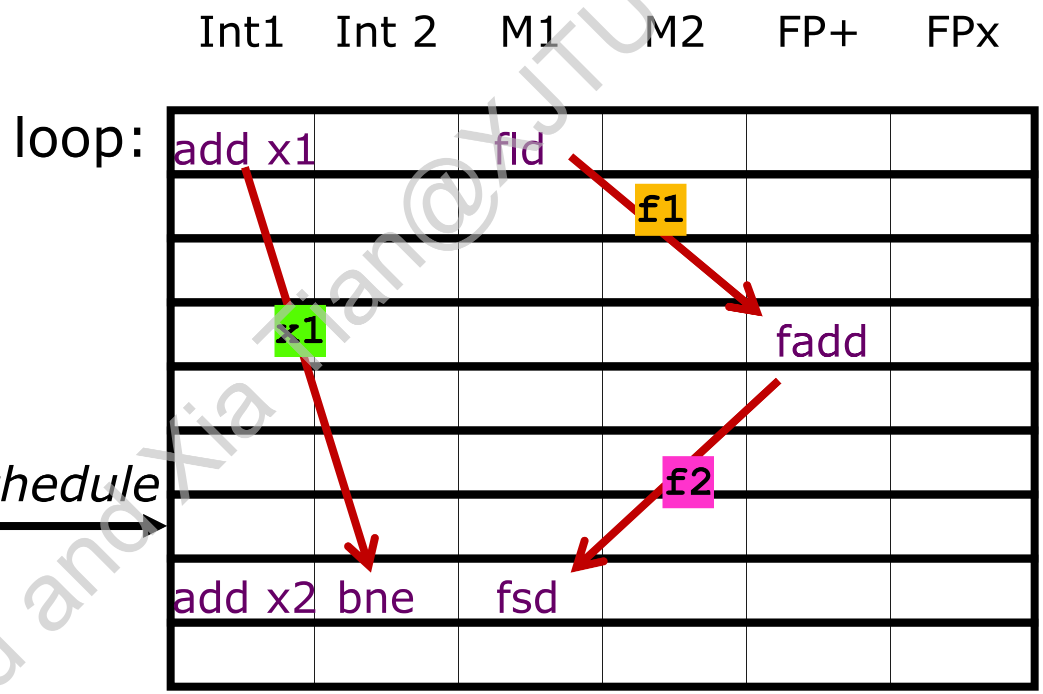
Loop Execution

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

Compile

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

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;
}
```

Is this code correct ?

- Need to handle values of N that are not multiples of unrolling factor with **final cleanup** loop
- More instructions (→ larger I cache)

Scheduling Loop Unrolled Code

Unroll into 4 ways

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add 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)
       add x2, 32
       bne x1, x3, loop
```

Schedule

loop:

Int1	Int 2	M1	M2	FP+	FPx
		fld f1			
		fld f2			
		fld f3			
add x1		fld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		fsd f5			
		fsd f6			
		fsd f7			
add x2	bne	fsd f8			

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

Scheduling Loop Unrolled Code

Unroll into 4 ways

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add 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)
       add x2, 32
       bne x1, x3, loop
```

Schedule →

loop:

Int1	Int 2	M1	M2	FP+	FPx
		fld f1	fld f2		
		fld f3	fld f4		
add x1				fadd f5	fadd f6
				fadd f7	fadd f8
		fsd f5	fsd f6		
add x2	bne	fsd f7	fsd f8		

How many FLOPS/cycle?

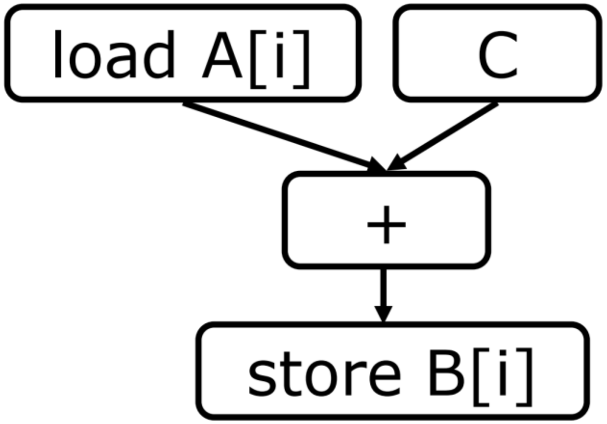
4 fadds / 9 cycles = 0.44

Software Pipelining

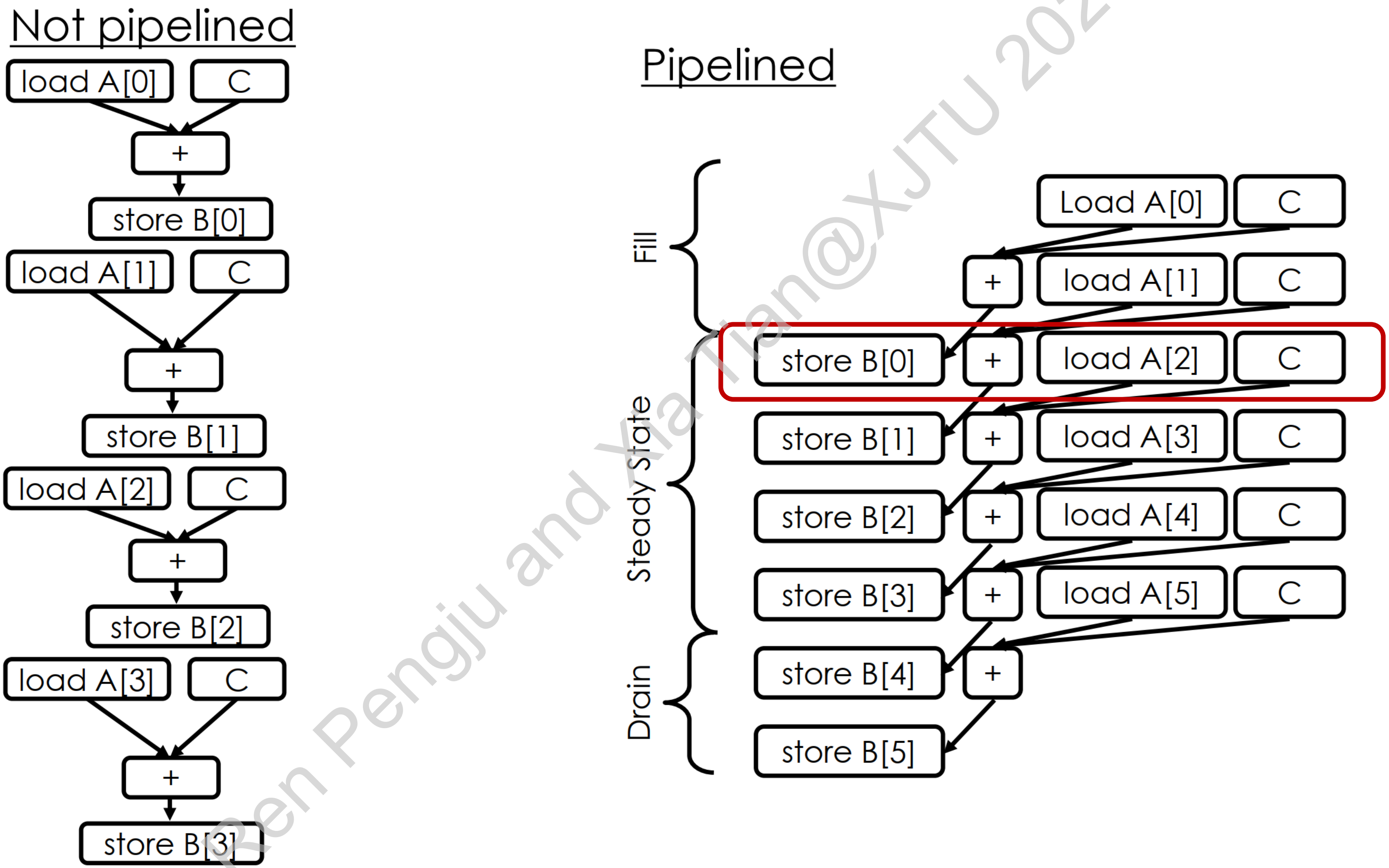
Exploit independent loop iterations

- If loop iterations are **independent**, then get more parallelism by scheduling instructions from **different iterations**
- Construct the **data-flow graph** for one iteration

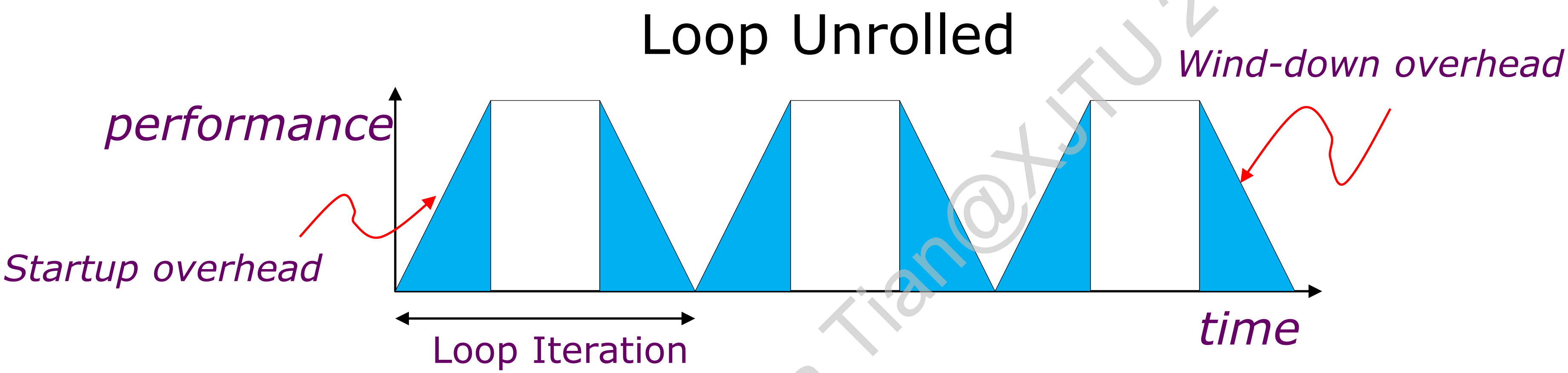
```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

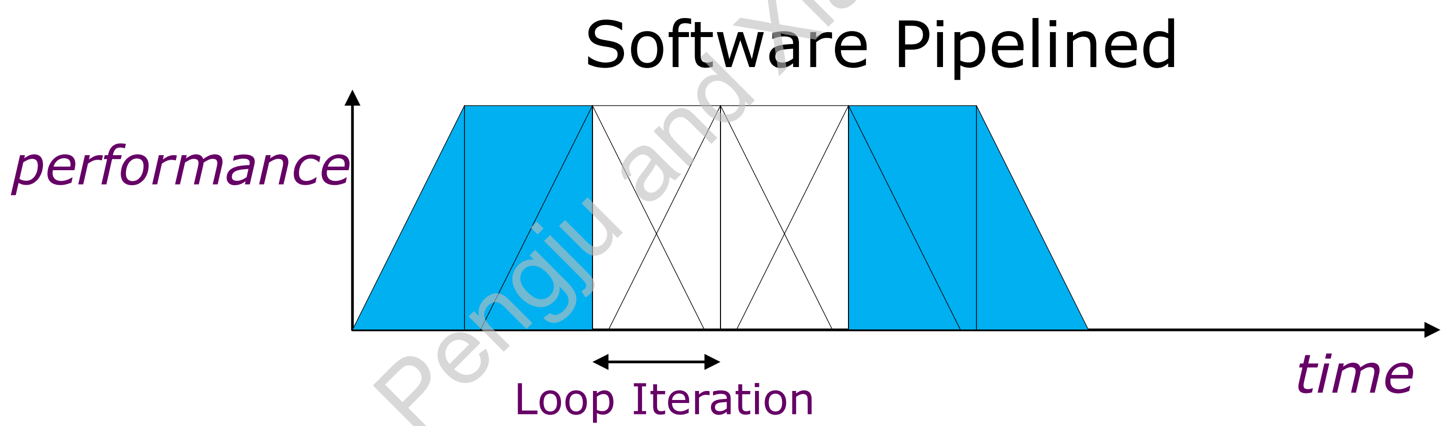


Software Pipelining



Software Pipelining vs. Loop Unrolling





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

Software Pipelining

Unroll 4 ways first

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add 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)
       add x2, 32
       bne x1, x3, loop
```

	Int1	Int 2	M1	M2	FP+	FPx
			fld f1			
			fld f2			
			fld f3			
	add x1		fld f4			
			fld f1		fadd f5	
			fld f2		fadd f6	
			fld f3		fadd f7	
	add x1		fld f4		fadd f8	
loop:			fld f1	fsd f5	fadd f5	
			fld f2	fsd f6	fadd f6	
			fld f3	fsd f7	fadd f7	
add x1	add x2	bne	fld f4	fsd f8	fadd f8	
				fsd f5	fadd f5	
				fsd f6	fadd f6	
				fsd f7	fadd f7	
	add x2	bne		fsd f8	fadd f8	
				fsd f5		

Conflict: put BNE in the next bundle

Software Pipelining

Unroll 4 ways first

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add 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)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```

	Int1	Int 2	M1	M2	FP+	FPx
			fld f1			
			fld f2			
			fld f3			
	add x1		fld f4			
			fld f1		fadd f5	
			fld f2		fadd f6	
			fld f3		fadd f7	
	add x1		fld f4		fadd f8	
loop:			fld f1	fsd f5	fadd f5	
			fld f2	fsd f6	fadd f6	
		add x2	fld f3	fsd f7	fadd f7	
	add x1	bne	fld f4	fsd f8	fadd f8	
				fsd f5	fadd f5	
				fsd f6	fadd f6	
		add x2		fsd f7	fadd f7	
		bne		fsd f8	fadd f8	
				fsd f5		

Software Pipelining

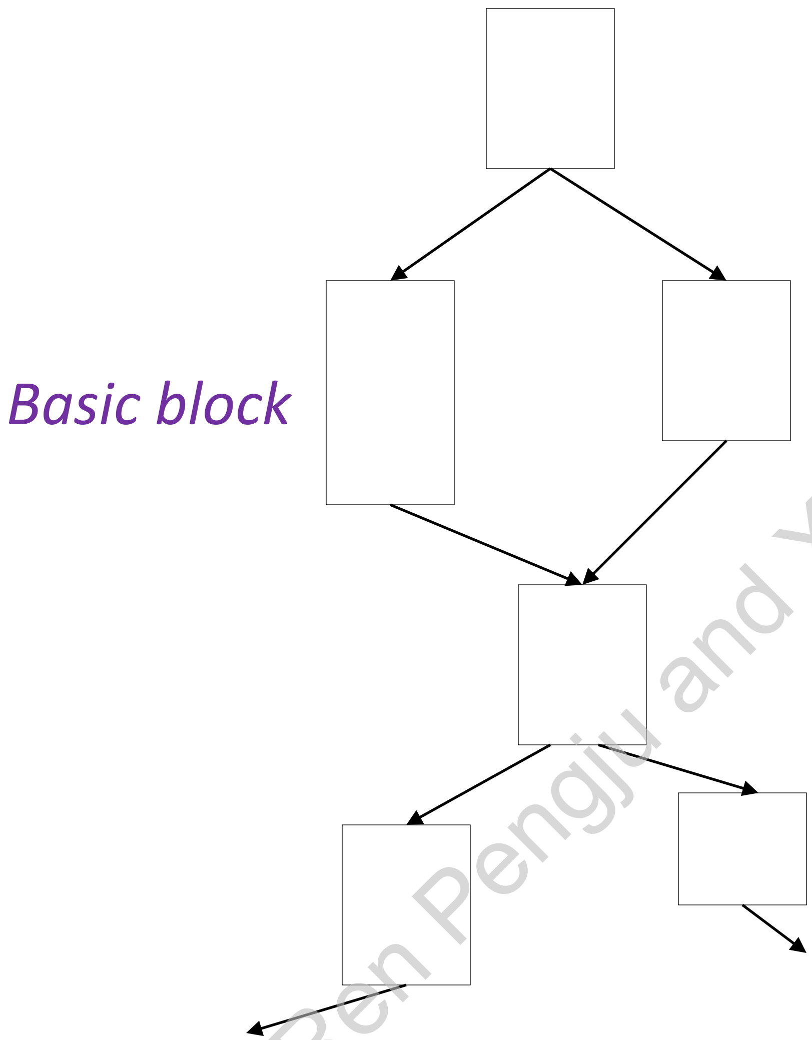
Unroll 4 ways first

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add 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)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```

	Int1	Int 2	M1	M2	FP+	FPx
prolog			fld f1			
			fld f2			
			fld f3			
	add x1		fld f4			
			fld f1		fadd f5	
			fld f2		fadd f6	
			fld f3		fadd f7	
	add x1		fld f4		fadd f8	
iterate **loop:**			fld f1	fsd f5	fadd f5	
			fld f2	fsd f6	fadd f6	
		add x2	fld f3	fsd f7	fadd f7	
	add x1	bne	fld f4	fsd f8	fadd f8	
epilog				fsd f5		
				fsd f6		
		add x2		fsd f7		
		bne		fsd f8		

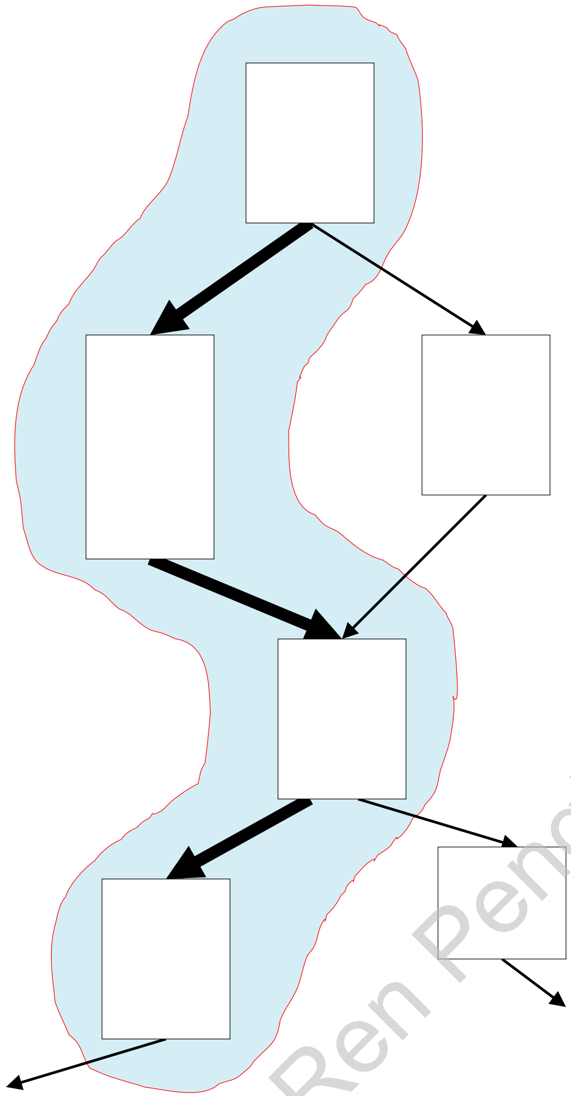
= 1 FLOPS/cycle

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 because of the limited scope of instructions

Trace Scheduling *[Fisher,Ellis]*



- A trace is a **possible 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 **fixup code** to cope with branches jumping out of trace

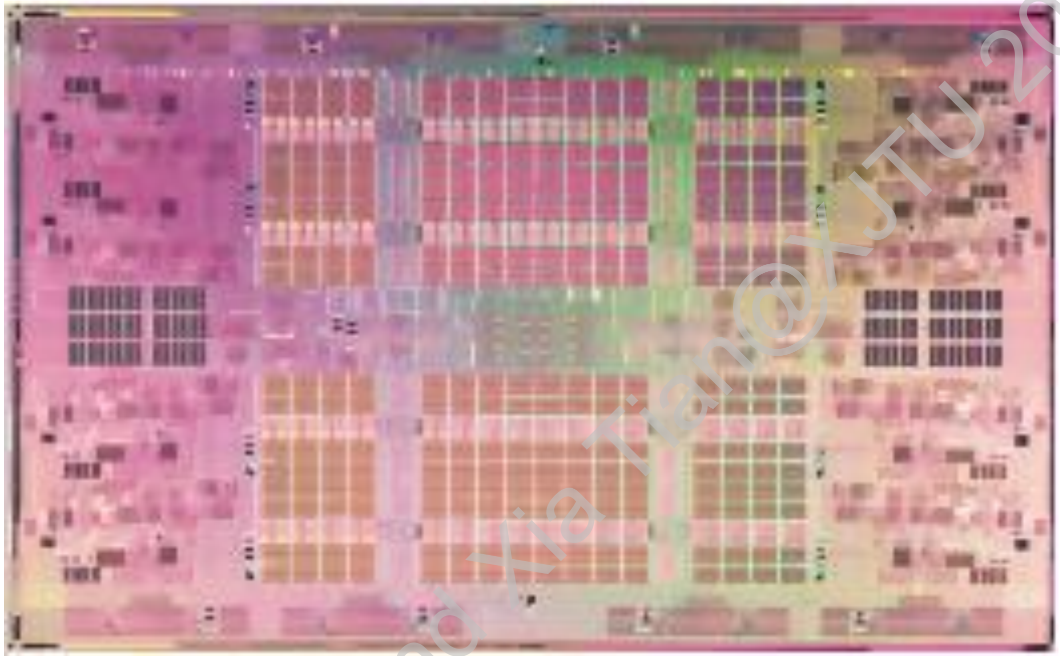
Problems with "Classic" VLIW

- **Object-code compatibility**
  - Have to **recompile** all code for every machine, even for two machines in same generation
- **Knowing branch probabilities**
  - **Profiling** requires an significant extra step in build process
- **Scheduling for statically unpredictable branches**
  - optimal schedule varies with branch path
- **Object code size**
  - **Instruction padding (NOPs)** wastes instruction memory/cache
  - **loop unrolling**/**software pipelining** replicates code
- **Scheduling variable latency memory operations**
  - **Cache miss** and/or **memory bank conflicts** impose statically unpredictable variability
  - **Uncertainty about addresses** limit code reordering

Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
  - Initially proposed by HP (in 1989)
  - **Explicitly Parallel Instruction Computing** (really just **VLIW**)
- IA-64 is Intel's chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - Intel version of VLIW (cooperated with HP)
  - **An object-code-compatible VLIW**
  - Designed to handle **complex tasks** (e.g. scientific computing and simulations, high-volume stock trading, airline reservation systems and secure internet transactions).
- Merced was first Itanium implementation (cf. 8086)
  - First customer shipment expected **1997** (actually **2001**)
  - McKinley, second implementation shipped in **2002**
  - Recent version, **Poulson**, eight cores, 32nm, announced **2011**
  - **Retired in 2019**, after nearly 20 years of production.

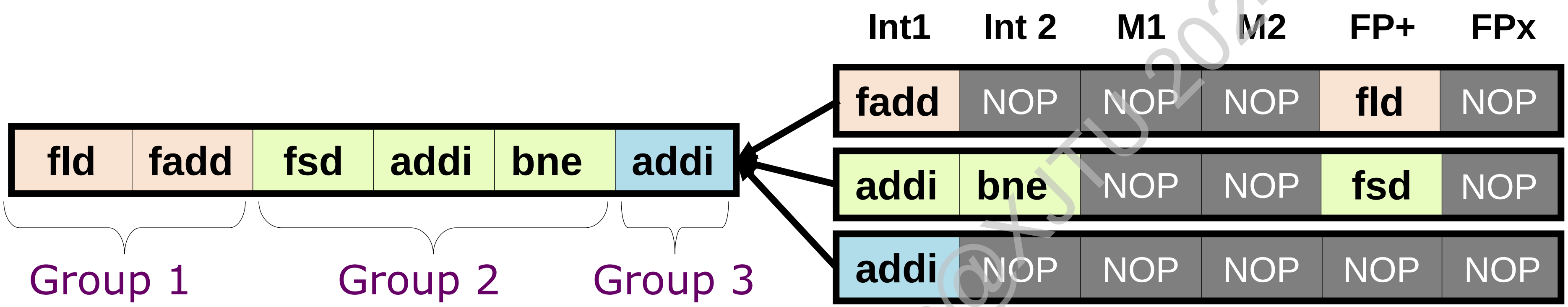
Eight Core Itanium "Poulson" *[Intel 2011]*



- 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

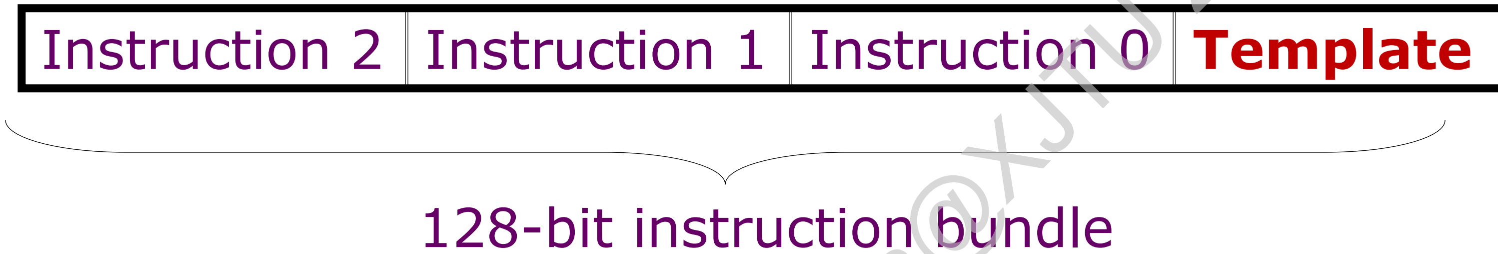
- Cores are 2-way multithreaded
- **6 instruction/cycle fetch**
  - **Two 128-bit bundles**
- **Up to 12 insts/cycle execute**

VLIW Instruction Encoding

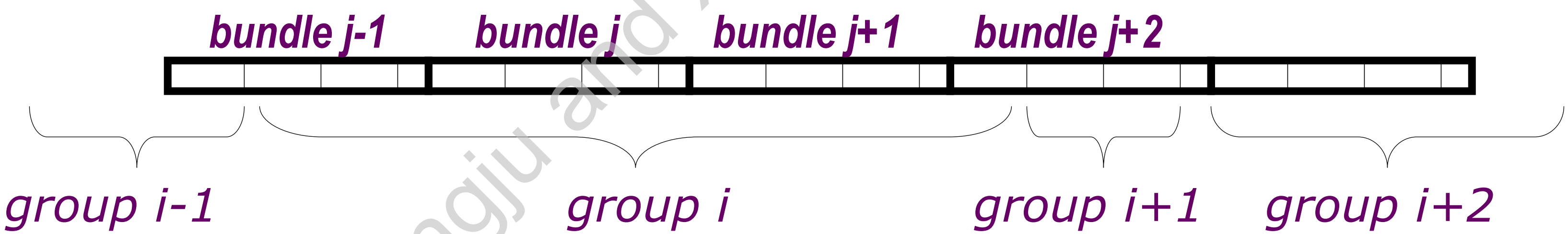


- **Schemes to reduce effect of unused fields (NOPs)**
  - **Compressed format** in memory, expand (**uncompress**) on I-cache refill
    - Used in Multi-flow Trace
    - Introduces instruction addressing challenge
  - Mark **parallel groups**
    - Used in TMS320C6x DSPs, Intel IA-64
  - Provide a **single-op VLIW instruction**
    - Cydra-5 UniOp instructions

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

Recap: Soft-Pipelining

Problems: Scheduled loops require **lots of registers → duplicated codes**

		Int1	Int 2	M1	M2	FP+	FPx
prolog				fld f1			
				fld f2			
				fld f3			
		add x1		fld f4			
				fld f1		fadd f5	
				fld f2		fadd f6	
				fld f3		fadd f7	
		add x1		fld f4		fadd f8	
iterate	**loop:**			fld f1	fsd f5	fadd f5	
				fld f2	fsd f6	fadd f6	
			add x2	fld f3	fsd f7	fadd f7	
		add x1	bne	fld f4	fsd f8	fadd f8	
epilog					fsd f5		
					fsd f6		
			add x2		fsd f7		
			bne		fsd f8		

IA-64 Registers

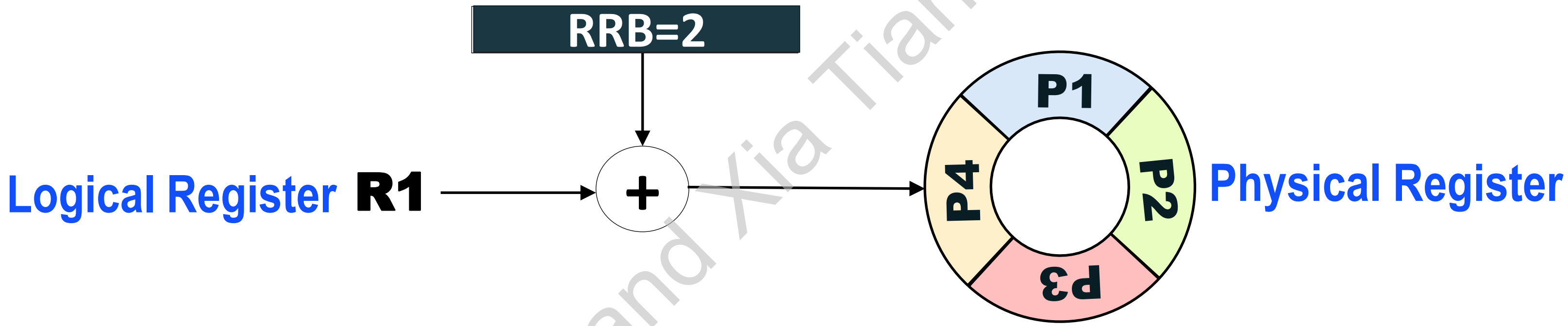
- 128 **General Purpose** 64-bit **Integer Registers**
- 128 **General Purpose** 64/80-bit **Floating Point Registers**
- 64 1-bit **Predicate Registers**

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

Rotating Register Files

Problems: Scheduled loops require **lots of registers → duplicated codes**

Solution: Automatically use 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 (static)** registers.

Rotating Register Files (Software/Hardware Co-design)

Register rotation is used for **optimizing loops** that are both counted or data-terminated.

- **Counted loops** are loops whose iterations are **known prior** to entering the loop
- **Data-terminated loops** are dependent upon values **calculated inside the loop**.

Register Set	Static	Rotating
General Registers (GR)	0-31	32-127
Floating Point Registers (FR)	0-31	32-127
Predicate Registers (PR)	0-15	16-63

The general, floating point and predicate registers are divided into subsets of **static** and **rotating** sets. The above is the subdivision

Recap: Soft-Pipelining

Problems: Scheduled loops require **lots of registers → duplicated codes**

prolog

Int1	Int 2	M1	M2	FP+	FPx
		fld f1			
		fld f2			
		fld f3			
add x1		fld f4			
		fld f1		fadd f5	
		fld f2		fadd f6	
		fld f3		fadd f7	
add x1		fld f4		fadd f8	

iterate

loop:

Int1	Int 2	M1	M2	FP+	FPx
		fld f1	fsd f5	fadd f5	
		fld f2	fsd f6	fadd f6	
	add x2	fld f3	fsd f7	fadd f7	
add x1	bne	fld f4	fsd f8	fadd f8	

Static (Int1, Int 2) **Rotating** (M1, M2, FP+)

Solution: Reduce loop codes to **one instruction**

epilog

Int1	Int 2	M1	M2	FP+	FPx
			fsd f5		
			fsd f6		
	add x2		fsd f7		
	bne		fsd f8		

Rotating Register File

(Previous Loop Example)

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

3 cycles later, f1 is renamed as f4

Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

4 cycles later, f5 is renamed as f9

fld f1, ()	fadd f5, f4, ...	fsd f9, ()	bloop

Recognized by hardware decode

F1=P9

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

F5=P10

F4=P9

F9=P10

Only need **1 instruction** in loop code!

IA-64 Predicated Execution

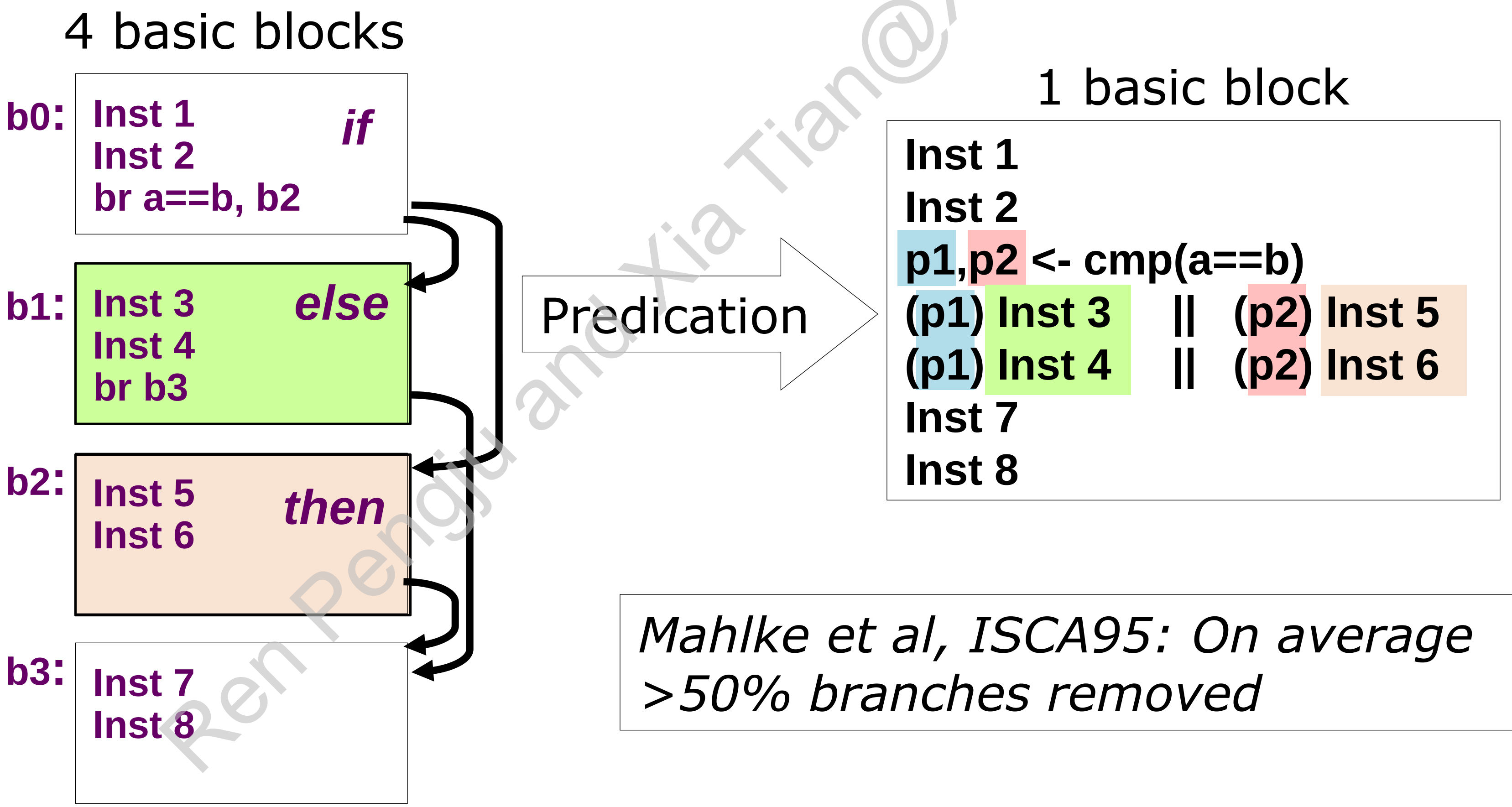
- **Predication** is the **conditional execution** of instructions.
- In **traditional** architectures, conditional execution is implemented through **branches**.
- In **VLIW** machine, **predicated execution** avoids branches, and simplifies compiler optimization by converting a **control dependency** to a **data dependency**.

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

IA-64 Normal Compares

Compare operation works on a **pair of predicate registers**.

- Compare operations play a key role in IA-64, and particularly in relation to predication.

Normal Compare instruction evaluates the expression and then:

- Set **the first predicate register** to the result of the comparison
- Set **the second predicate register** to the complement of the comparison.

cmp.eq p_1, p_2, r_a, r_b

- **cmp.eq** → Normal Cmp Instruction
- p_1, p_2 → Target Predicate Registers
- r_a, r_b → Source Registers

```
If (a == b) {
    c++;
} else {
    d++;
}
```

Predication →

```
{
  cmp.eq p1, p2 = ra, rb
  (p1)  add  rc = rc, 1
  (p2)  add  rd = rd, 1
}
```

IA-64 Unconditional Compares

Unconditional Compare instructions are **predicated themselves**:

- When its **self-predicate is 1**, the compare executes normally and writes to its target registers as would a normal compare.
- When its **self-predicate is 0**, **write 0 to both** of it's target registers.

[p_s], **cmp.eq.unc**, p_1, p_2, r_a, r_b

- [p_s] → Self Predicate Register
- cmp.eq.unc → Unconditional Cmp Instruction
- p_1, p_2 → Target Predicate Registers
- r_a, r_b → Source Registers

Unconditional Compare is useful in **nested if-conversion.**

```
If (a > b) {
   c++;
} else {
   d+=c;
   if (e==f) {
      g++;
   } else    {
      h--;
   }
}
```

Predication →

```
{
  cmp.gt p1, p2 = ra, rb
}{
  (p1)  add  rc = rc, 1
  (p2)  add  rd = rd, rc
  (p2)  cmp.eq.unc  p3, p4 = re, rf
}{
  (p3) add rg = rg, 1
  (p4) add rh = rh, -1
}
```

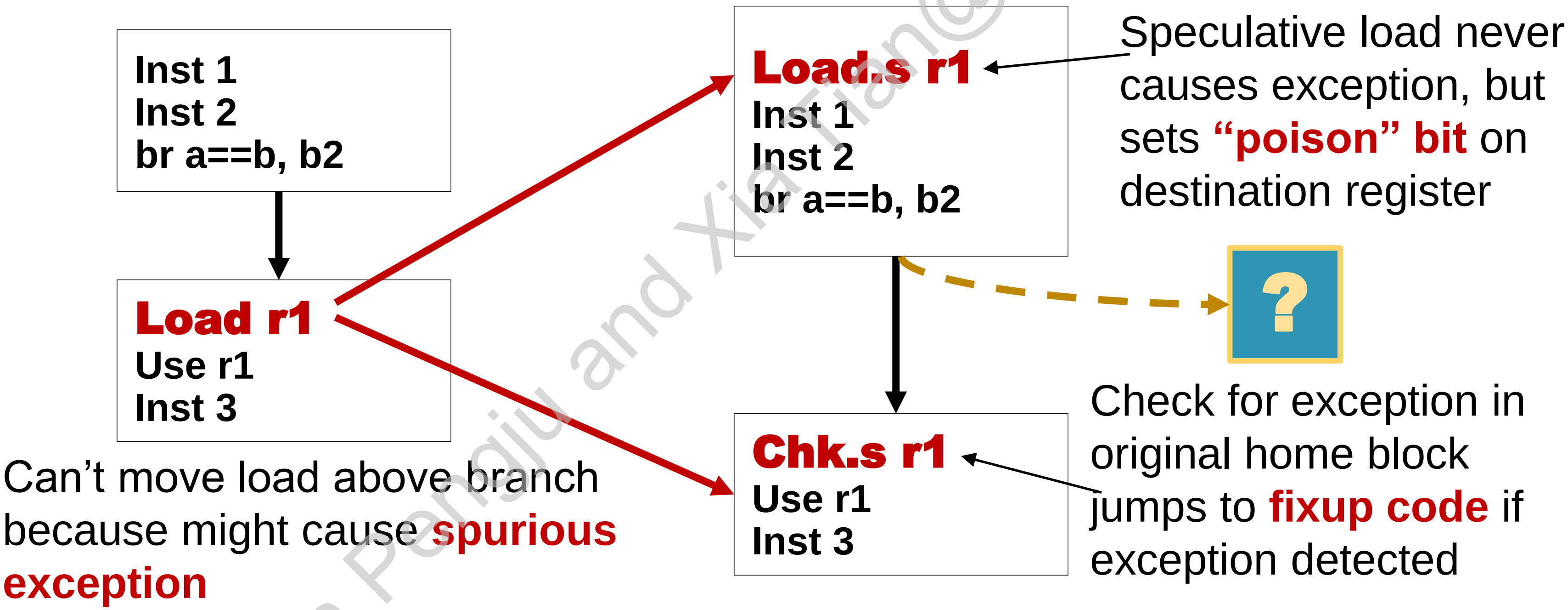
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions

-- Requires associative hardware in register **poison bit**

-- Particularly useful for scheduling **long latency loads** early



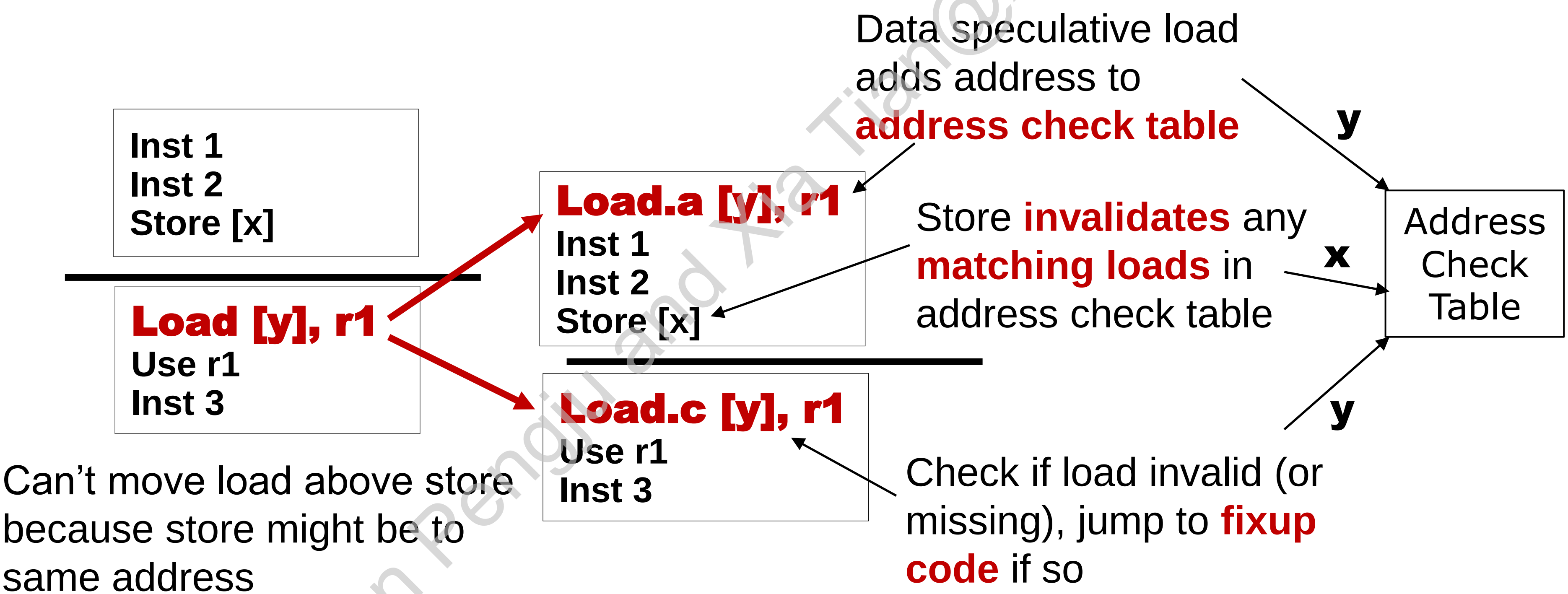
- Compiler guarantees the **next reference** of r1 is **not READ** (**except for check**)
- If branch is not taken and check is never executed, **clear poison bit** when next time r1 is modified.

IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards

-- Requires associative hardware in **address check table**



Limits of Static Scheduling

- **Unpredictable branches**
  - **Solved** by: predicated, speculate
- **Code size explosion**
  - **Solved** by: compression, rotate register
- **Variable memory latency** (e.g. cache miss)
  - **Solved** by: manually managed scratchpad
- **Compiler complexity** (**unsolved**)
- **Poor compatibility** (**unsolved**)

VLIW Today

- Despite several attempts, VLIW has **failed in general-purpose computing** arena (so far).
  - 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**, friendly code.
  - E.g. Texas Instrument (TI) DSP series

Next Lecture : Vectors and SIMD

(Data Level Parallel)

Acknowledgements

- **Some slides contain material developed and copyright by:**
  - **Arvind (MIT)**
  - **Krste Asanovic (MIT/UCB)**
  - **Joel Emer (Intel/MIT)**
  - **James Hoe (CMU)**
  - **David Patterson (UCB)**
  - **David Wentzlaff (Princeton University)**

- **MIT material derived from course 6.823**
- **UCB material derived from course CS252 and CS 61C**