

EE 660: Computer Architecture

VLIW Introduction

Yao Zheng

Department of Electrical Engineering

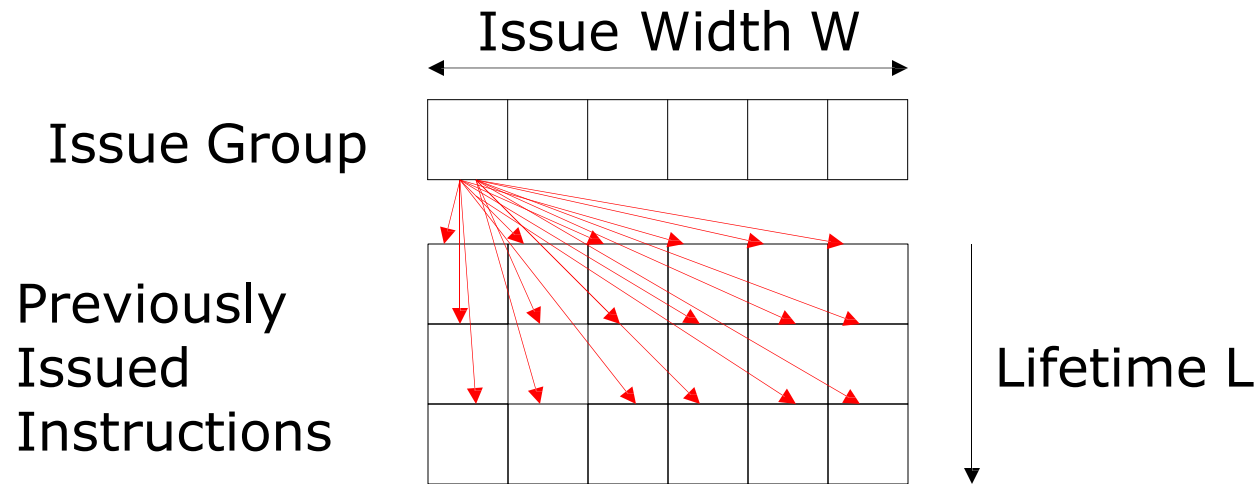
University of Hawai'i at Mānoa



UNIVERSITY
of HAWAI'I®
MĀNOA

Based on the slides of Prof. David Wentzlauff

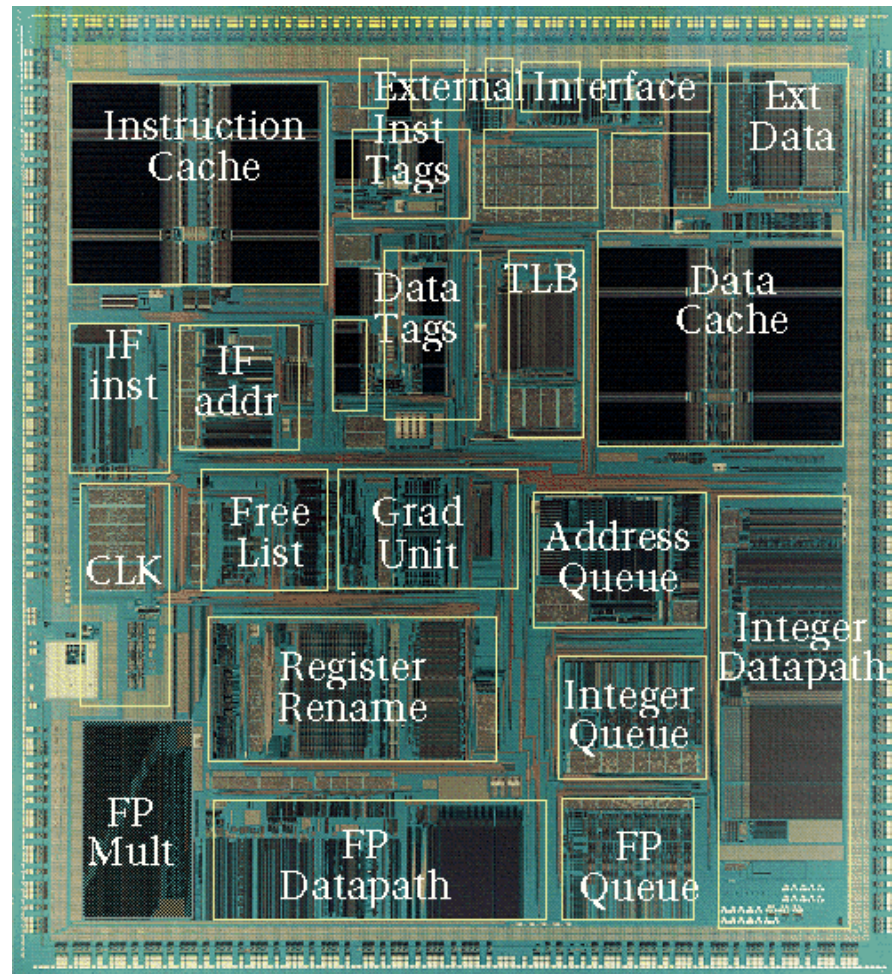
Superscalar Control Logic Scaling



- Each issued instruction must somehow check against $W \cdot L$ instructions, i.e., growth in hardware $\propto W \cdot (W \cdot L)$
- 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 \Rightarrow greater L

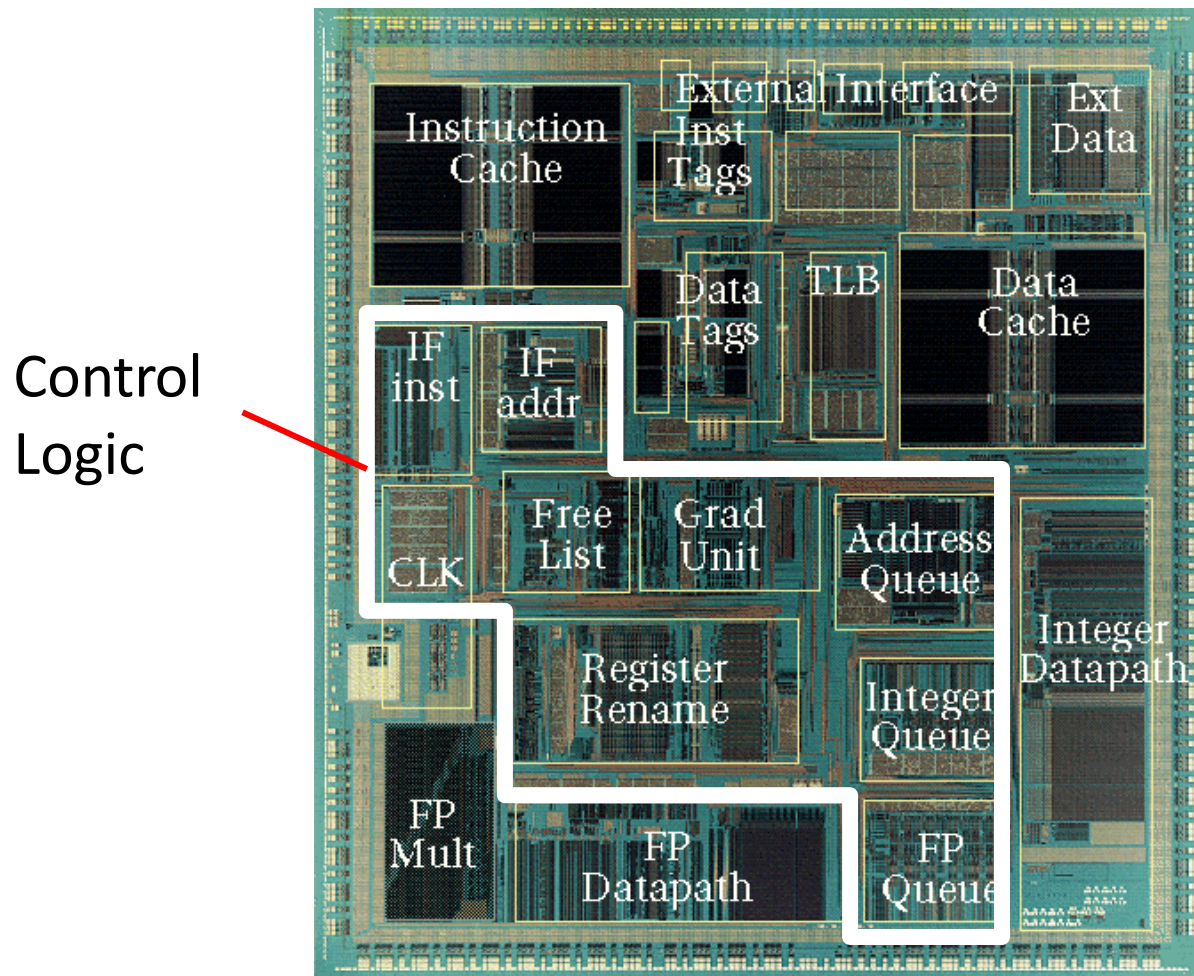
\Rightarrow Out-of-order control logic grows faster than W^2 ($\sim W^3$)

Out-of-Order Control Complexity: MIPS R10000



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

Out-of-Order Control Complexity: MIPS R10000



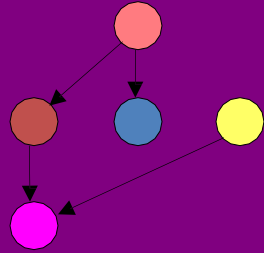
[A. Ahi et al., MIPS R10000 Superscalar Microprocessor, Hot Chips, 1995]
Image Credit: MIPS Technologies Inc. / Silicon Graphics Computer Systems

Sequential ISA Bottleneck

Sequential
source code

```
a = foo(b);  
for (i=0, i<
```

Superscalar compiler

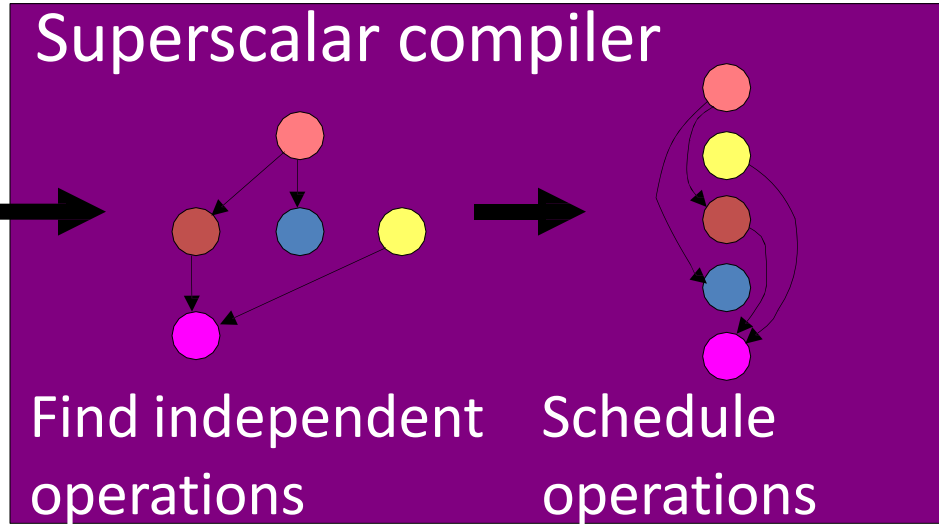


Find independent
operations

Sequential ISA Bottleneck

Sequential
source code

```
a = foo(b);  
for (i=0, i<
```

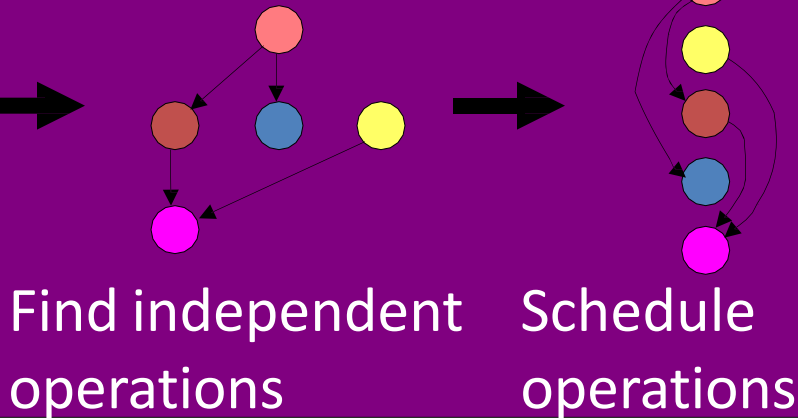


Sequential ISA Bottleneck

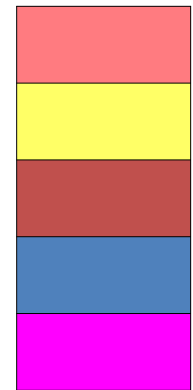
Sequential source code

```
a = foo(b);  
for (i=0, i<
```

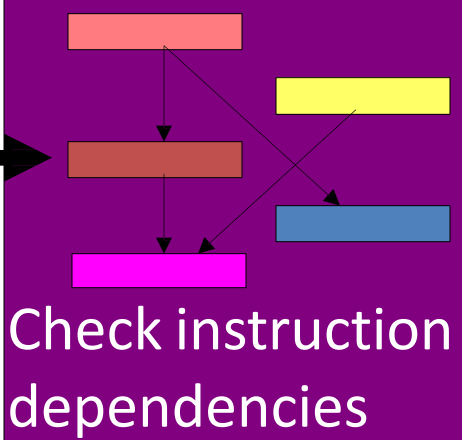
Superscalar compiler



Sequential machine code



Superscalar processor



Sequential ISA Bottleneck

Sequential source code

```
a = foo(b);  
for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations

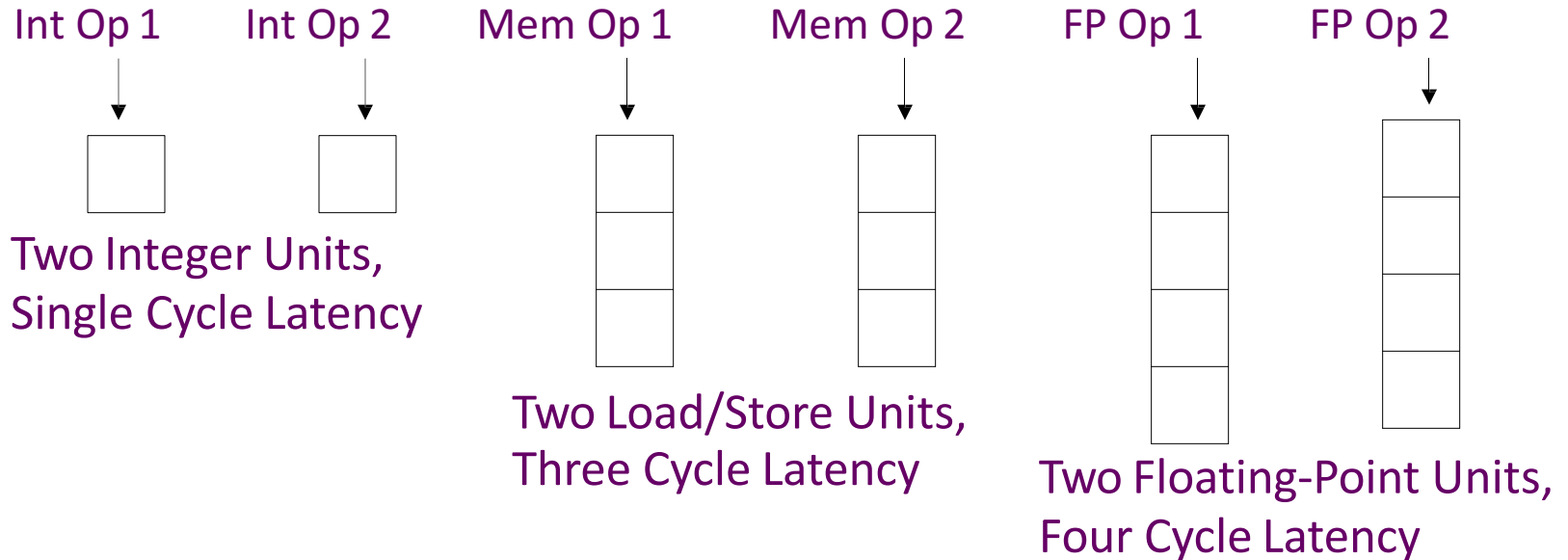
Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

VLIW: Very Long Instruction Word



- 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

```
{MUL R1, R2, R3; ADD R7, R8, R9}  
{AND R10, R1, R5}
```

X0

Y0 Y1 Y1 Y2 Y2 W

M0 M1

VLIW Equals (EQ) Scheduling Model

- Each operation takes exactly specified latency
- Efficient register usage (Effectively more registers)
- No need for register renaming or buffering
 - Bypass from functional unit output to inputs
 - Register writes whenever functional unit completes
- 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
- Guarantees intra-instruction parallelism
- Schedule 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: lw F1, 0(R1)  
      addiu R1, R1, 4  
      add.s F2, F0, F1  
      sw F2, 0(R2)  
      addiu R2, R2, 4  
      bne R1, R3, loop
```

Schedule

Int1 Int 2 M1 M2 FP+ FPx

loop:

Loop Execution

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

Compile

```
loop: lw F1, 0(R1)
      addiu R1, R1, 4
      add.s F2, F0, F1
      sw F2, 0(R2)
      addiu R2, R2, 4
      bne R1, R3, loop
```

Schedule

loop:

	Int1	Int 2	M1	M2	FP+	FPx
add R1			lw			
					add.s	

Loop Execution

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

Compile

```
loop: lw F1, 0(R1)
      addiu R1, R1, 4
      add.s F2, F0, F1
      sw F2, 0(R2)
      addiu R2, R2, 4
      bne R1, R3, loop
```

Schedule

loop:

	Int1	Int 2	M1	M2	FP+	FPx
add R1			lw			
					add.s	
			sw			

Loop Execution

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

Compile

```
loop: lw F1, 0(R1)
      addiu R1, R1, 4
      add.s F2, F0, F1
      sw F2, 0(R2)
      addiu R2, R2, 4
      bne R1, R3, loop
```

Schedule

Int1 Int 2 M1 M2 FP+ FPx

loop:

add R1		lw				
					add.s	
add R2		sw				

Loop Execution

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

Compile

```
loop: lw F1, 0(R1)
      addiu R1, R1, 4
      add.s F2, F0, F1
      sw F2, 0(R2)
      addiu R2, R2, 4
      bne R1, R3, loop
```

Schedule

Int1 Int 2 M1 M2 FP+ FPx

loop:

add R1		lw				
					add.s	
add R2	bne	sw				

How many FP ops/cycle?

Loop Execution

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

Compile

```
loop: lw F1, 0(R1)
      addiu R1, R1, 4
      add.s F2, F0, F1
      sw F2, 0(R2)
      addiu R2, R2, 4
      bne R1, R3, loop
```

Schedule

Int1 Int 2 M1 M2 FP+ FPx

loop:

add R1		lw				
					add.s	
add R2	bne	sw				

How many FP ops/cycle?

1 add.s / 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;  
}
```

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: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

Schedule

Int1 Int 2 M1 M2 FP+ FPx

loop:

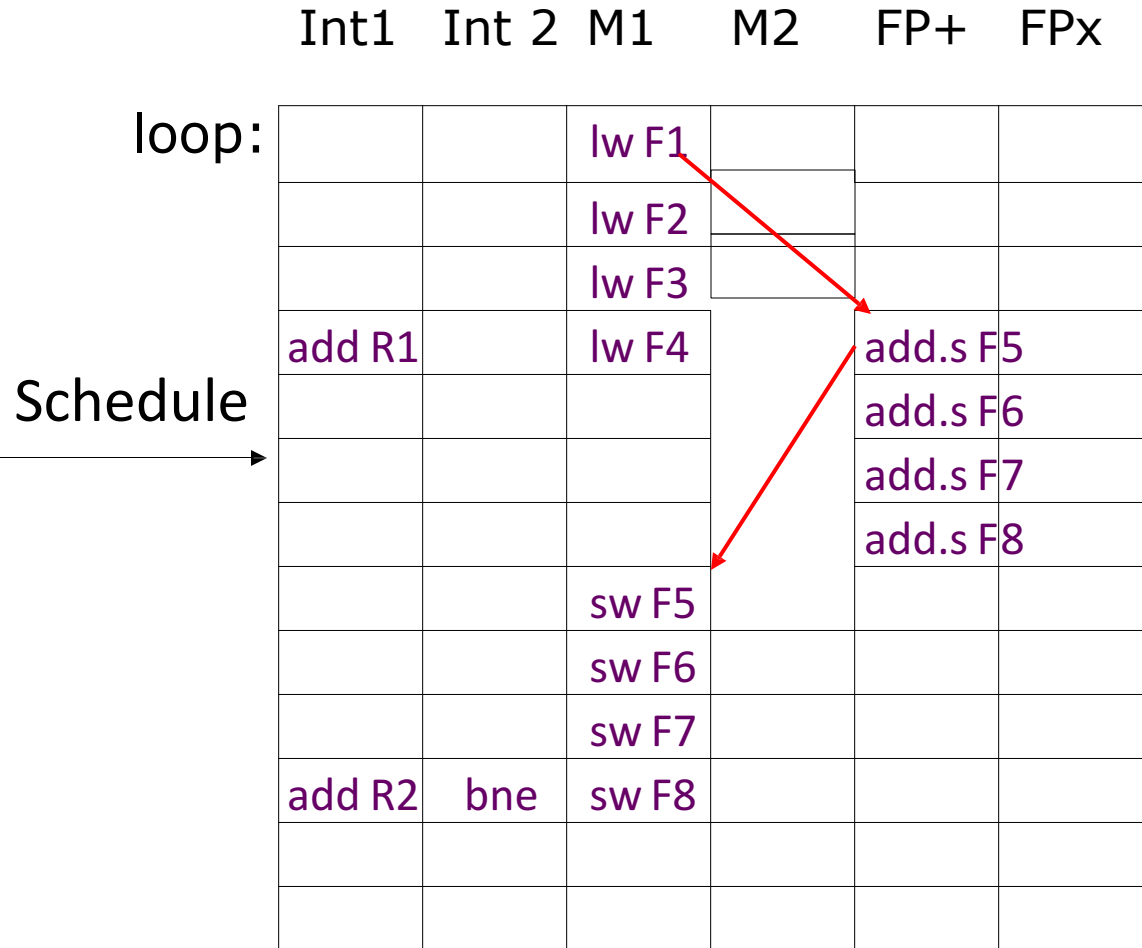
	Int1	Int 2	M1	M2	FP+	FPx
			lw F1			
			lw F2			
			lw F3			
add R1			lw F4			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```



Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

Schedule →

Int1 Int 2 M1 M2 FP+ FPx

loop:

		lw F1			
		lw F2			
		lw F3			
add R1		lw F4		add.s F5	
				add.s F6	
				add.s F7	
				add.s F8	
		sw F5			
		sw F6			
		sw F7			
add R2	bne	sw F8			

How many FLOPS/cycle?

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
loop:			lw F1			
			lw F2			
			lw F3			
	add R1		lw F4		add.s F5	
					add.s F6	
					add.s F7	
					add.s F8	
			sw F5			
			sw F6			
			sw F7			
	add R2	bne	sw F8			

How many FLOPS/cycle?

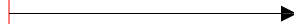
4 add.s / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```



Int1	Int 2	M1	M2	FP+	FPx
		lw F1			
		lw F2			
		lw F3			
add R 1		lw F4			
				add.s F5	
				add.s F6	
				add.s F7	
				add.s F8	
			sw F5		
			sw F6		
	add R2		sw F7		
	bne		sw F8		

Software Pipelining

Unroll 4 ways first

```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

Int1	Int 2	M1	M2	FP+	FPx
		lw F1			
		lw F2			
		lw F3			
add R1		lw F4			
		lw F1		add.s F5	
		lw F2		add.s F6	
		lw F3		add.s F7	
add R1		lw F4		add.s F8	
			sw F5	add.s F5	
			sw F6	add.s F6	
	add R2		sw F7	add.s F7	
	bne		sw F8	add.s F8	
			sw F5		
			sw F6		
	add R2		sw F7		
	bne		sw F8		

Software Pipelining

Unroll 4 ways first

```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

Int1	Int 2	M1	M2	FP+	FPx
		lw F1			
		lw F2			
		lw F3			
add R 1		lw F4			
		lw F1		add.s F5	
		lw F2		add.s F6	
		lw F3		add.s F7	
add R 1		lw F4		add.s F8	
		lw F1	sw F5	add.s F5	
		lw F2	sw F6	add.s F6	
	add R2	lw F3	sw F7	add.s F7	
add R 1 bne		lw F4	sw F8	add.s F8	
			sw F5	add.s F5	
			sw F6	add.s F6	
	add R2		sw F7	add.s F7	
	bne		sw F8	add.s F8	
			sw F5		

Software Pipelining

Unroll 4 ways first

```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

	Int1	Int 2	M1	M2	FP+	FPx
prolog			lw F1			
			lw F2			
			lw F3			
	add R 1		lw F4			
			lw F1		add.s F5	
			lw F2		add.s F6	
			lw F3		add.s F7	
	add R 1		lw F4		add.s F8	
iterate			lw F1	sw F5	add.s F5	
			lw F2	sw F6	add.s F6	
		add R2	lw F3	sw F7	add.s F7	
	add R 1 bne		lw F4	sw F8	add.s F8	
epilog				sw F5	add.s F5	
				sw F6	add.s F6	
		add R2		sw F7	add.s F7	
		bne		sw F8	add.s F8	
				sw F5		

Software Pipelining

Unroll 4 ways first

```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

	Int1	Int 2	M1	M2	FP+	FPx
prolog			lw F1			
			lw F2			
			lw F3			
	add R 1		lw F4			
			lw F1		add.s F5	
			lw F2		add.s F6	
			lw F3		add.s F7	
	add R 1		lw F4		add.s F8	
iterate			lw F1	sw F5	add.s F5	
			lw F2	sw F6	add.s F6	
		add R2	lw F3	sw F7	add.s F7	
	add R 1 bne		lw F4	sw F8	add.s F8	
epilog				sw F5	add.s F5	
				sw F6	add.s F6	
		add R2		sw F7	add.s F7	
		bne		sw F8	add.s F8	
				sw F5		

How many FLOPS/cycle?

Software Pipelining

Unroll 4 ways first

```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
    
```

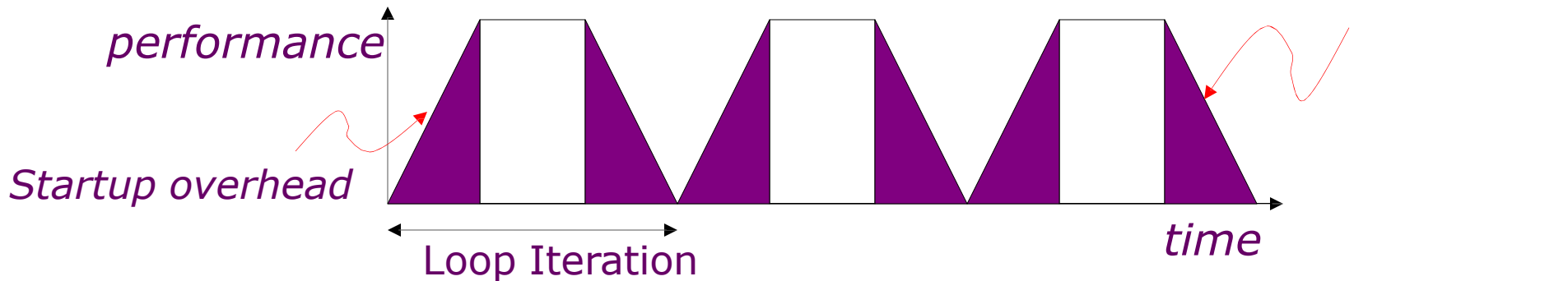
	Int1	Int 2	M1	M2	FP+	FPx
prolog			lw F1			
			lw F2			
			lw F3			
	add R 1		lw F4			
			lw F1		add.s F5	
			lw F2		add.s F6	
			lw F3		add.s F7	
			lw F4		add.s F8	
iterate	add R 1		lw F4		add.s F8	
			lw F1	sw F5	add.s F5	
			lw F2	sw F6	add.s F6	
		add R2	lw F3	sw F7	add.s F7	
epilog	add R 1 bne		lw F4	sw F8	add.s F8	
				sw F5	add.s F5	
				sw F6	add.s F6	
		add R2		sw F7	add.s F7	
		bne		sw F8	add.s F8	
				sw F5		

How many FLOPS/cycle?

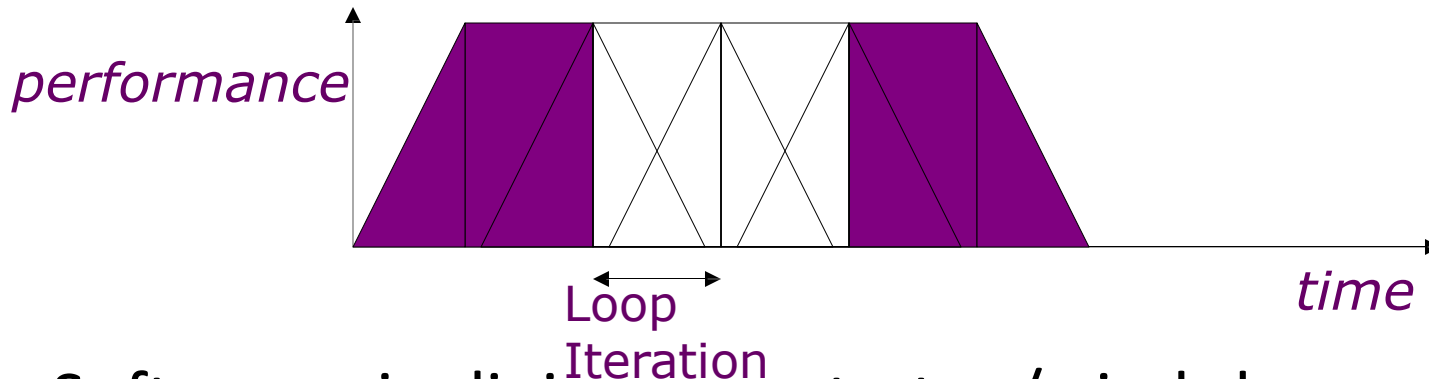
4 add.s / 4 cycles = 1

Software Pipelining vs. Loop Unrolling

Loop Unrolled

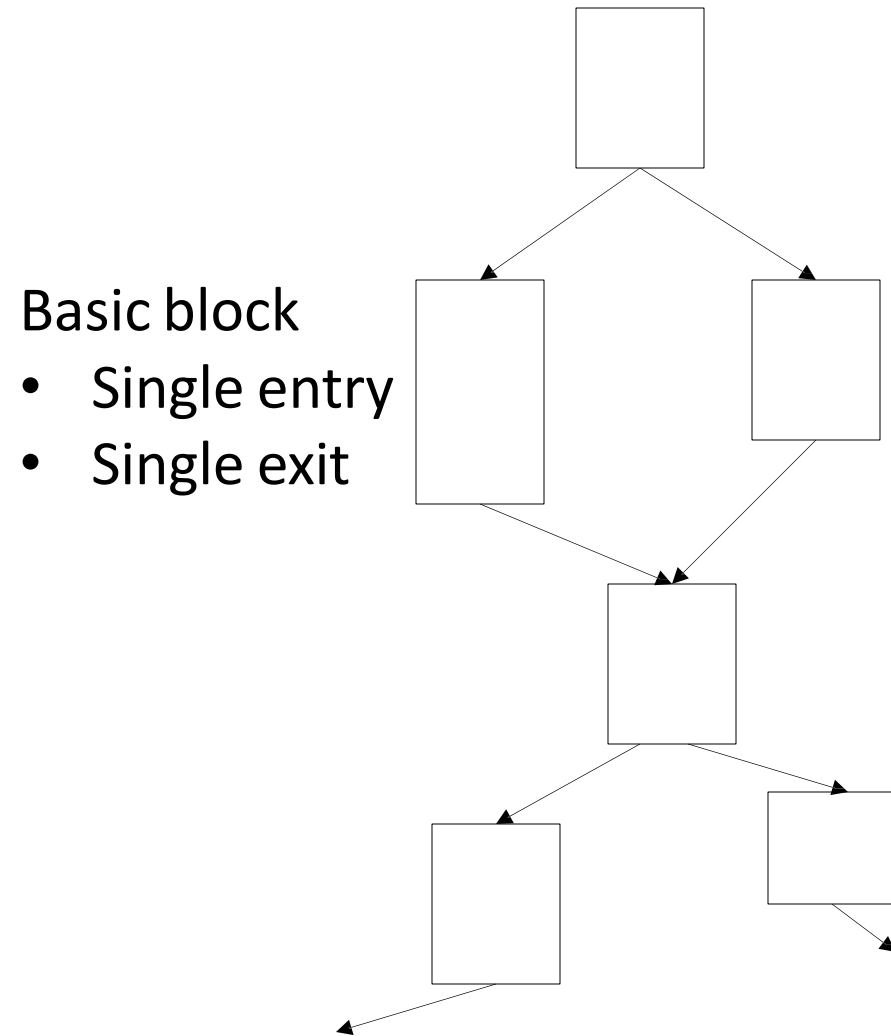


Software Pipelined



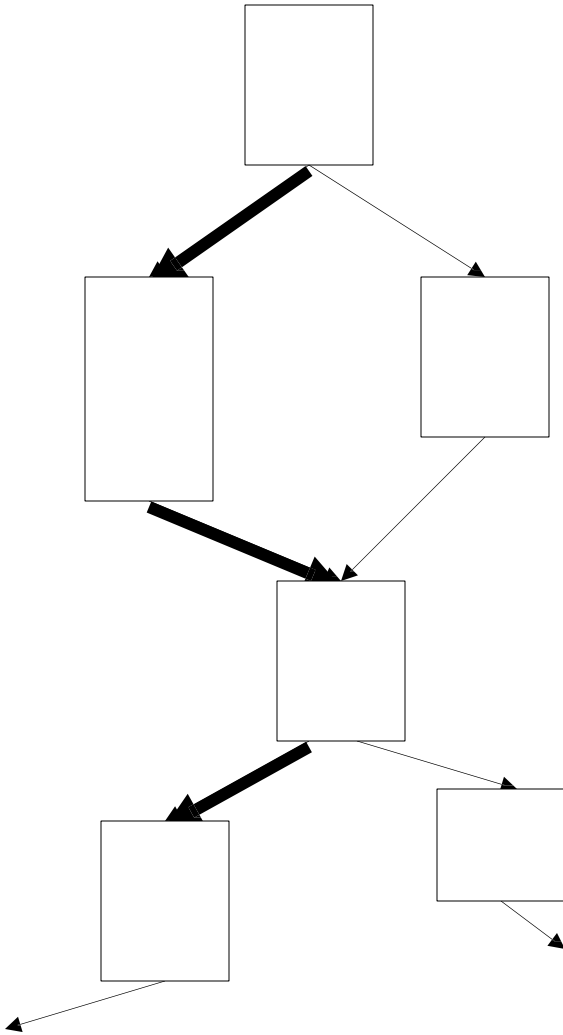
Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

What if there are no loops?



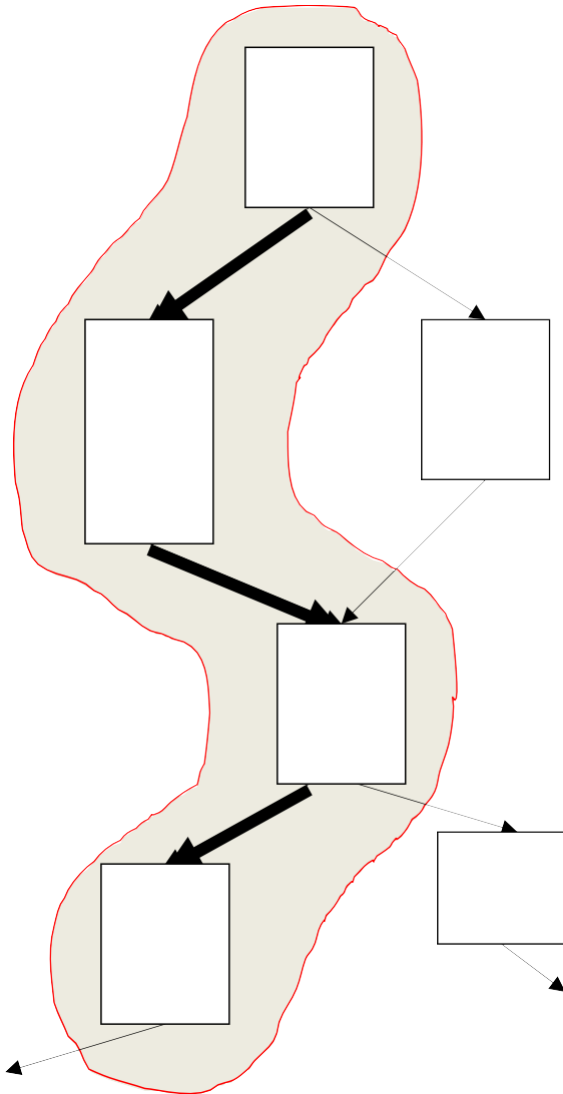
- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

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

Trace Scheduling [Fisher, Ellis]

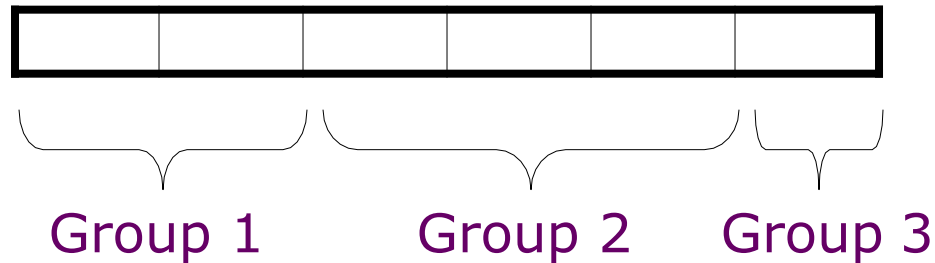


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

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
- 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

Partial 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, rs, rt if (R[rt] == 0) then R[rd] <- R[rs]`
- `movn rd, rs, rt if (R[rt] != 0) then R[rd] <- R[rs]`

<code>if (a<b)</code>	<code>slt R1, R2, R3</code>	<code>slt R1, R2, R3</code>
<code>x=a</code>	<code>beq R1, R0, L1</code>	<code>movz R4, R2, R1</code>
<code>else</code>	<code>move R4, R2</code>	<code>movn R4, R3, R1</code>
<code>x=b</code>	<code>j L2</code>	
	<code>L1:move R4, R3</code>	
	<code>L2:</code>	

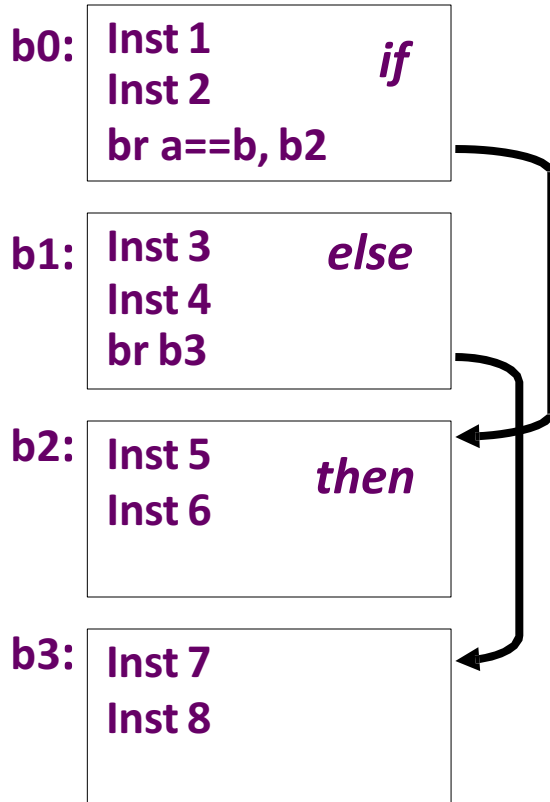
What if-then-else has many instructions? What if unbalanced?

Full Predication

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

Full Predication

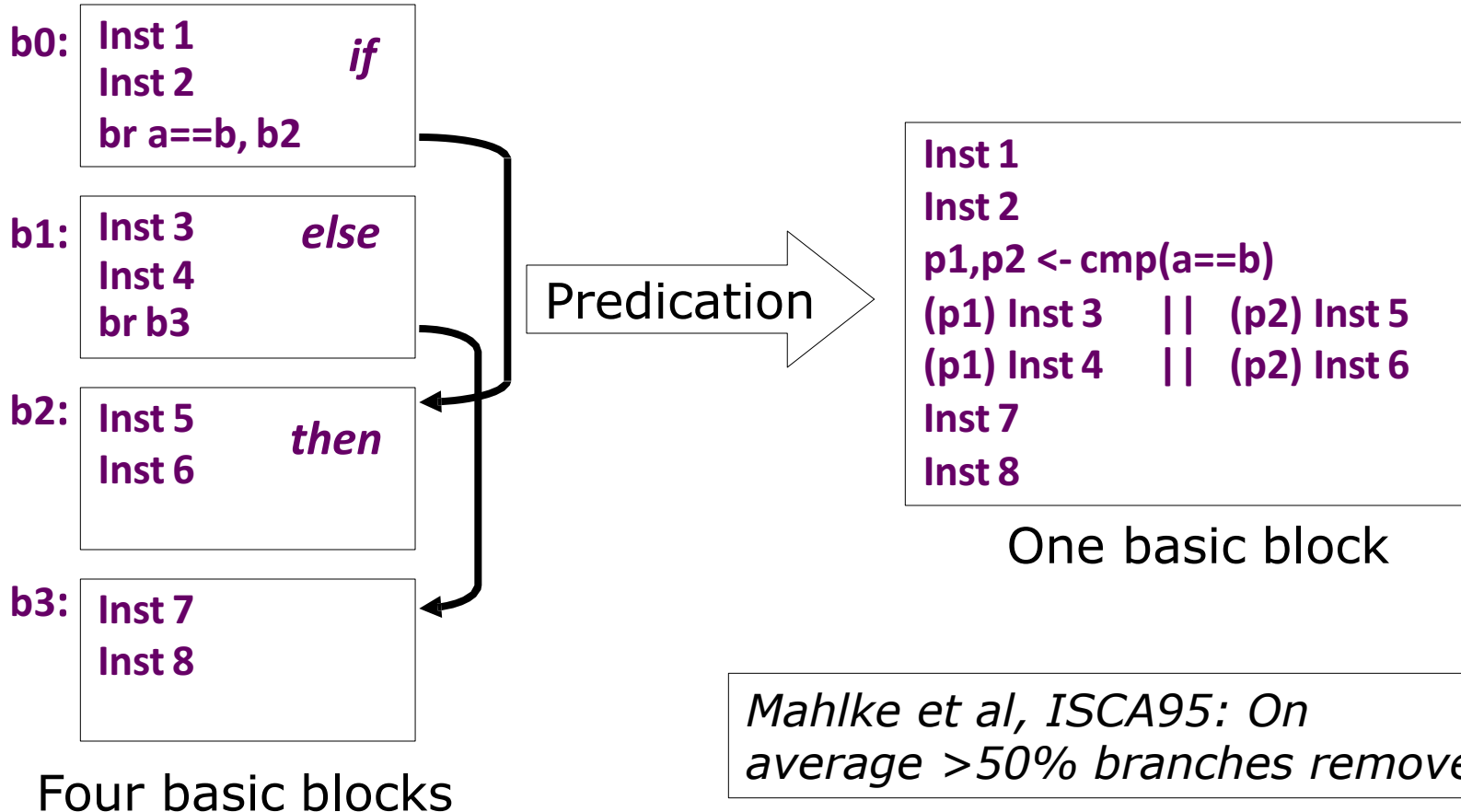
- Almost all instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



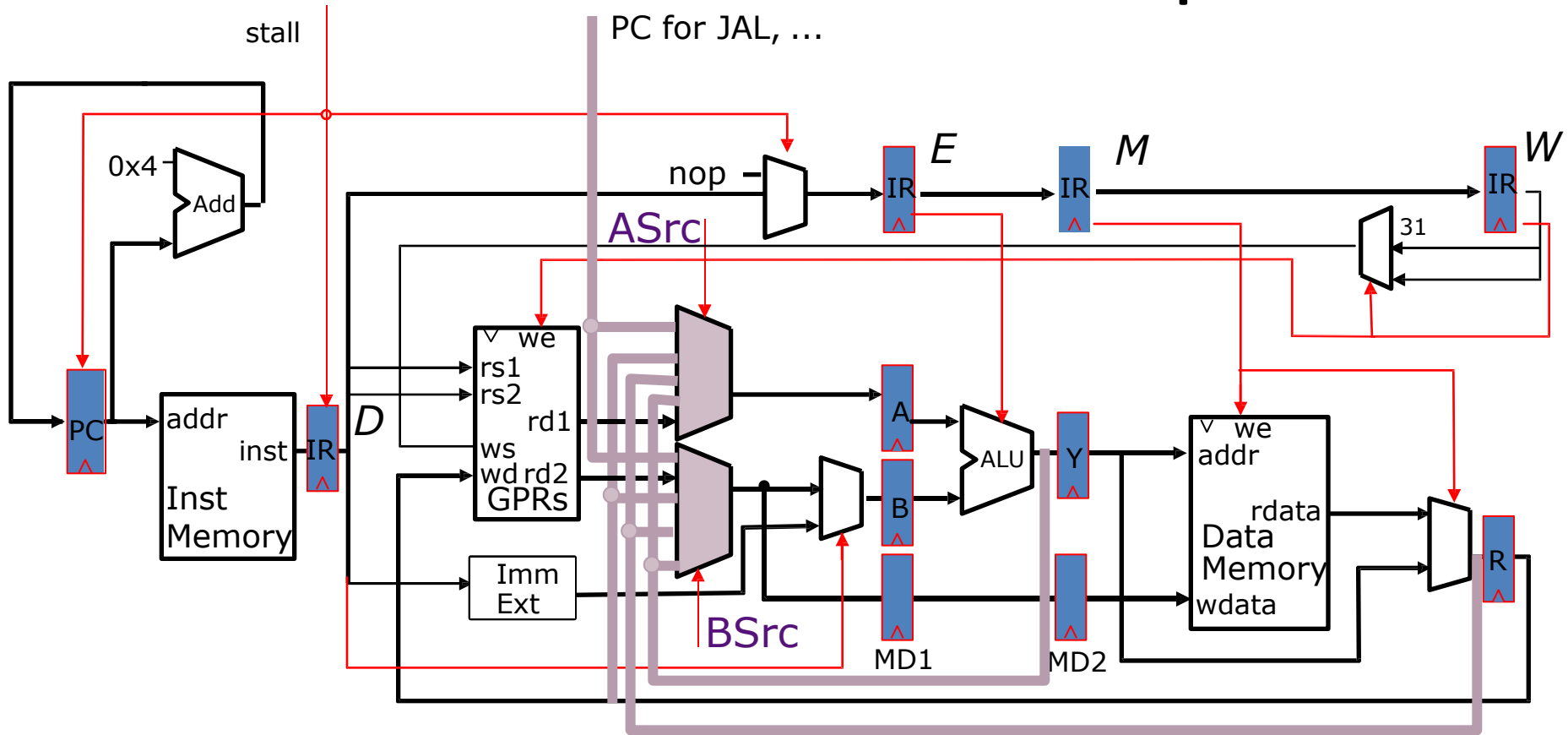
Four basic blocks

Full Predication

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

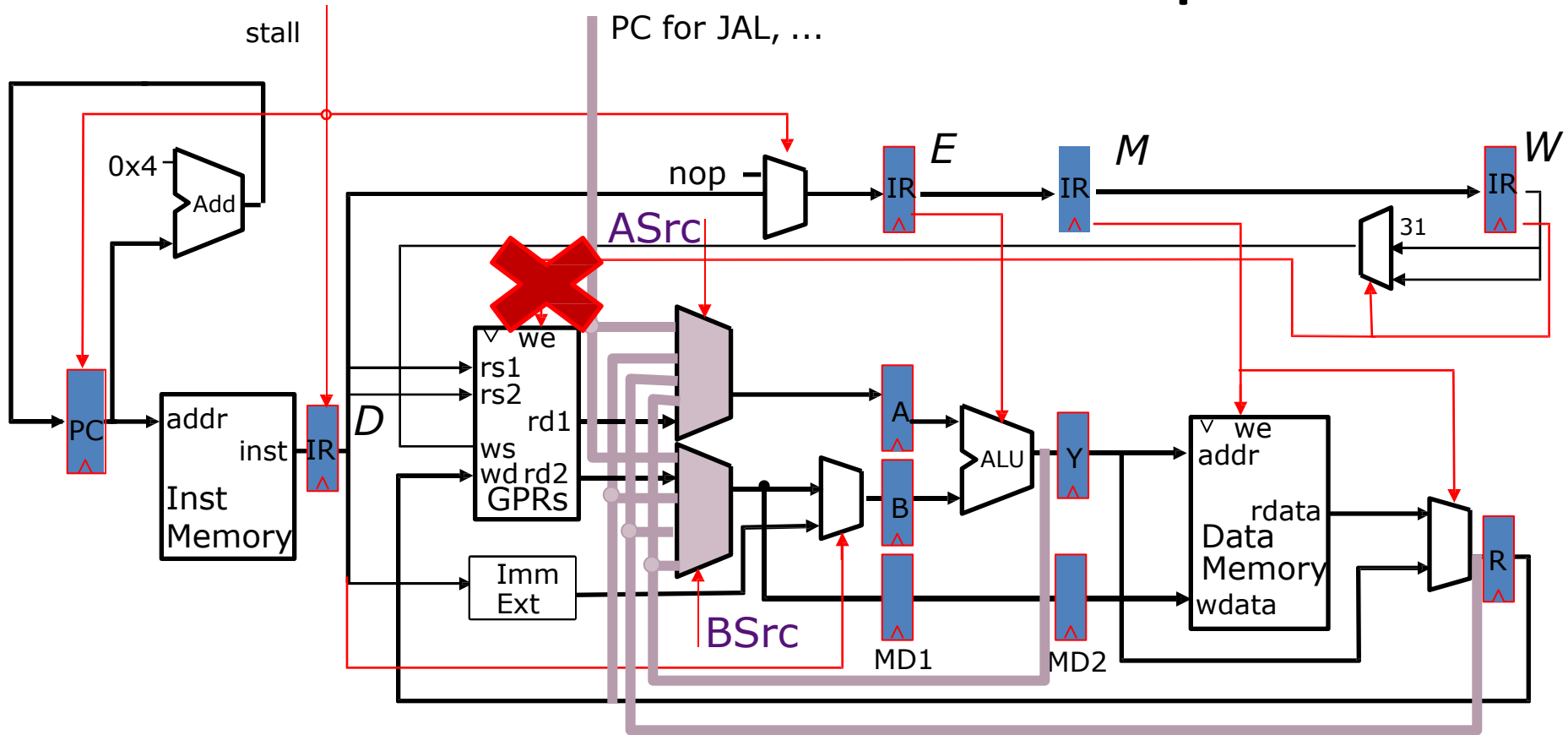


Predicates and the Datapath



`movz rd, rs, rt if (R[rt] == 0) then R[rd] <- R[rs]`

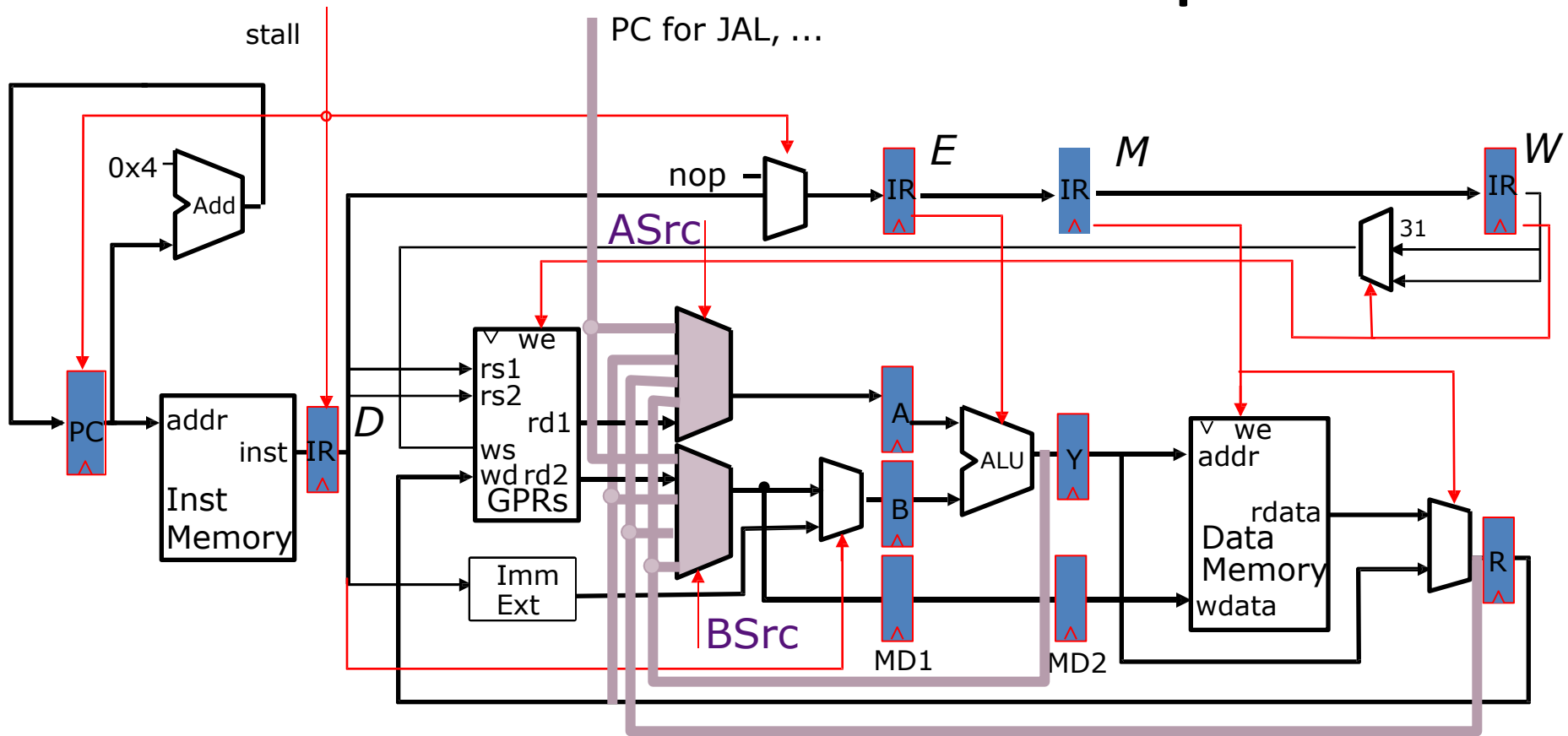
Predicates and the Datapath



movz rd, rs, rt if (R[rt] == 0) then R[rd] <- R[rs]

- Suppress Writeback

Predicates and the Datapath



`movz rd, rs, rt if (R[rt] == 0) then R[rd] <- R[rs]`

- Suppress Writeback
- Bypassing value doesn't work!
 - Need initial value (Extra read port on RF)

Problems with Full Predication

- Adds another register file
- Need to bypass predicates
- Need original value to use data value bypassing

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    R1, R2, R3
ADDIU  R11,R10,1
MUL    R5, R1, R4
MUL    R7, R5, R6
SW     R7, 0(R16)
ADDIU  R12,R11,1
LW     R14, 0(R9)
ADD    R13,R12,R14
ADD    R14,R12,R13
BNEQ  R16, target
```

After Code Motion

```
LW     R14, 0(R9)
ADDIU  R11,R10,1
MUL    R1, R2, R3
ADDIU  R12,R11,1
MUL    R5, R1, R4
ADD    R13,R12,R14
MUL    R7, R5, R6
ADD    R14,R12,R13
SW     R7, 0(R16)
BNEQ  R16, target
```

Scheduling and Bundling

Before Bundling

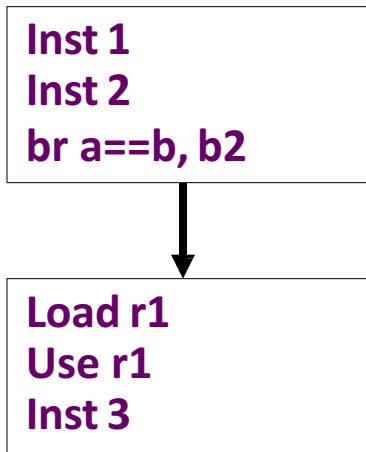
```
LW      R14, 0(R9)
ADDIU   R11, R10, 1
MUL     R1,  R2,  R3
ADDIU   R12, R11, 1
MUL     R5,  R1,  R4
ADD     R13, R12, R14
MUL     R7,  R5,  R6
ADD     R14, R12, R13
SW      R7,  0(R16)
BNEQ   R16, target
```

After Bundling

```
{ LW      R14, 0(R9)
  ADDIU   R11, R10, 1
  MUL     R1,  R2,  R3 }
{ ADDIU   R12, R11, 1
  MUL     R5,  R1,  R4 }
{ ADD     R13, R12, R14
  MUL     R7,  R5,  R6 }
{ ADD     R14, R12, R13
  SW      R7,  0(R16)
  BNEQ   R16, target }
```

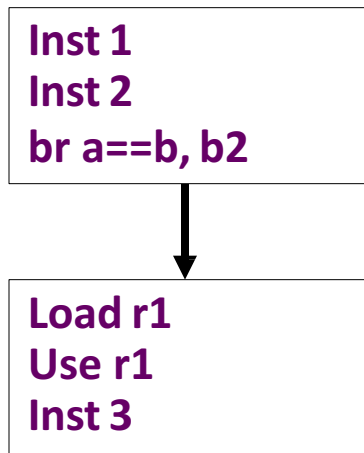
VLIW Speculative Execution

Problem: Branches restrict compiler code motion



VLIW Speculative Execution

Problem: Branches restrict compiler code motion

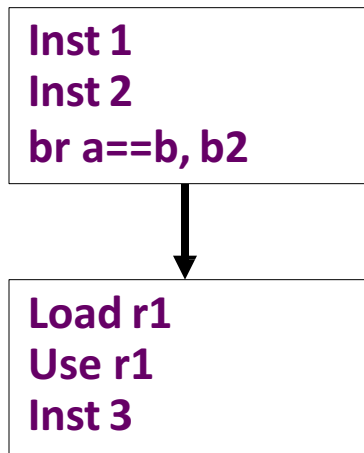


*Can't move load above branch
because might cause spurious
exception*

VLIW Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions

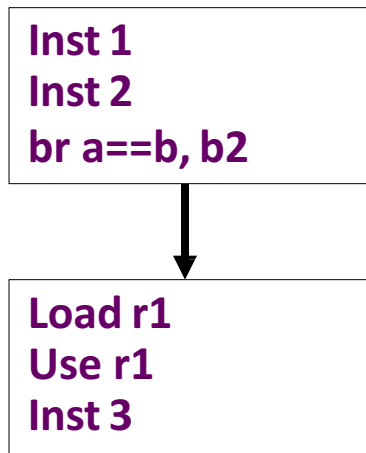


*Can't move load above branch
because might cause spurious
exception*

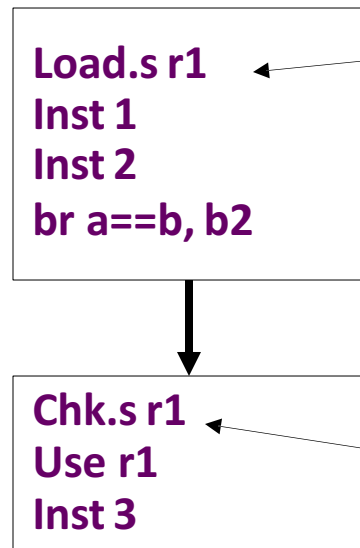
VLIW Speculative Execution

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



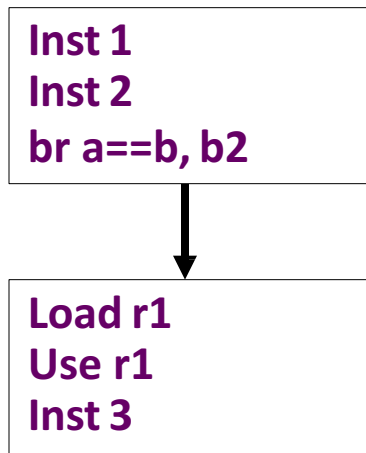
Speculative load never causes exception, but sets "poison" bit on destination register

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

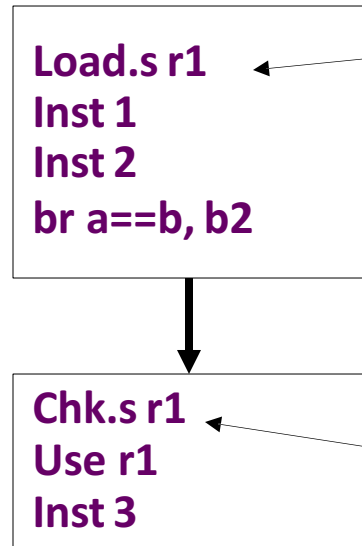
VLIW Speculative Execution

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

VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards

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*

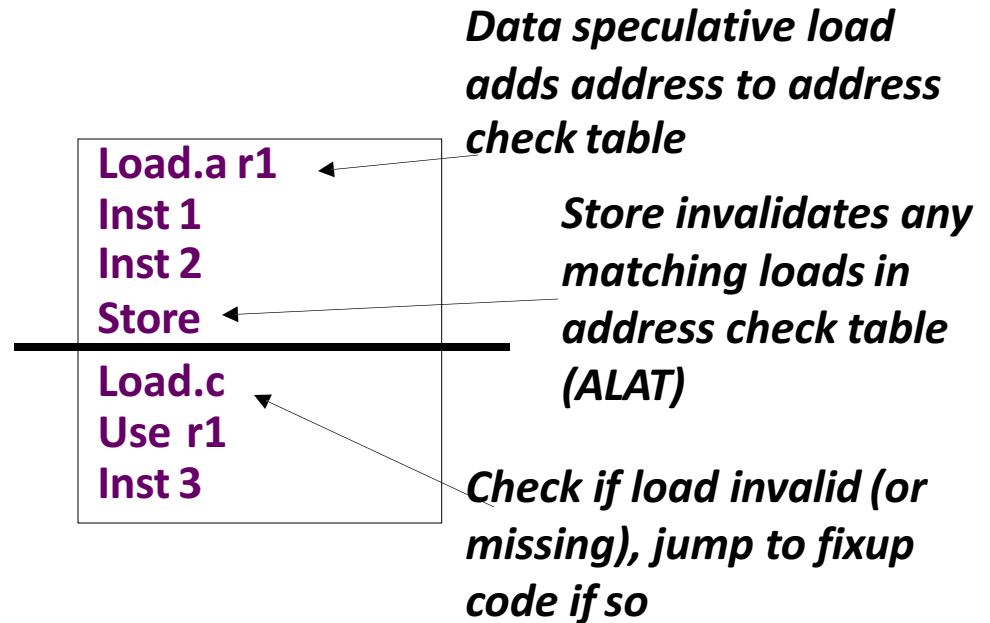
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards



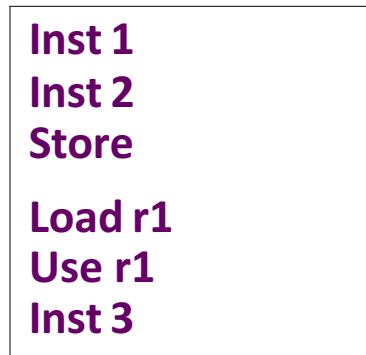
*Can't move load above store
because store might be to same
address*



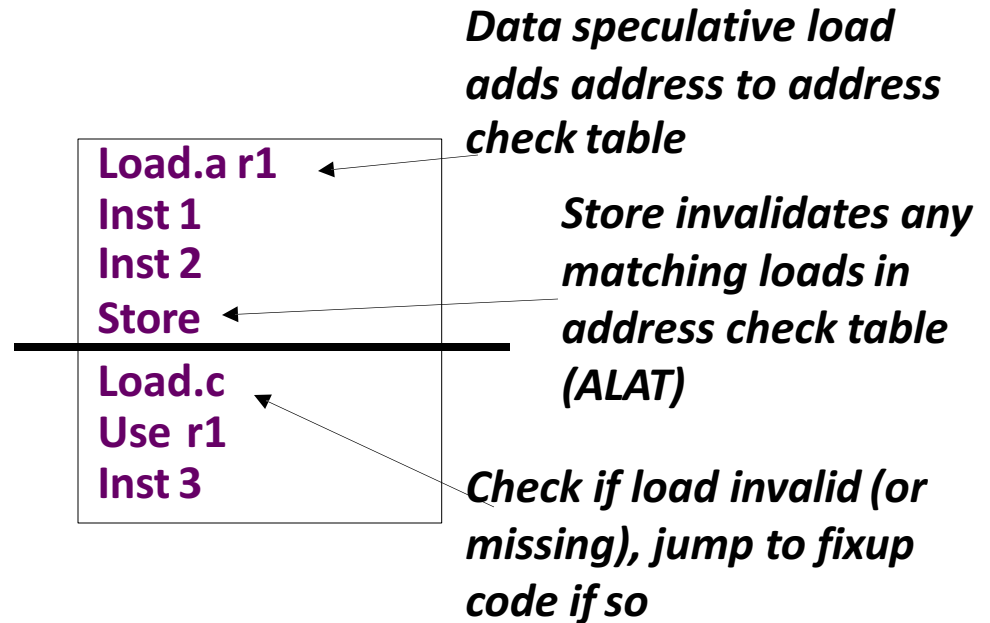
VLIW Data Speculation

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*



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

VLIW Multi-Way Branches

Problem: Long instructions provide few opportunities for branches

Solution: Allow one instruction to branch multiple directions

VLIW Multi-Way Branches

Problem: Long instructions provide few opportunities for branches

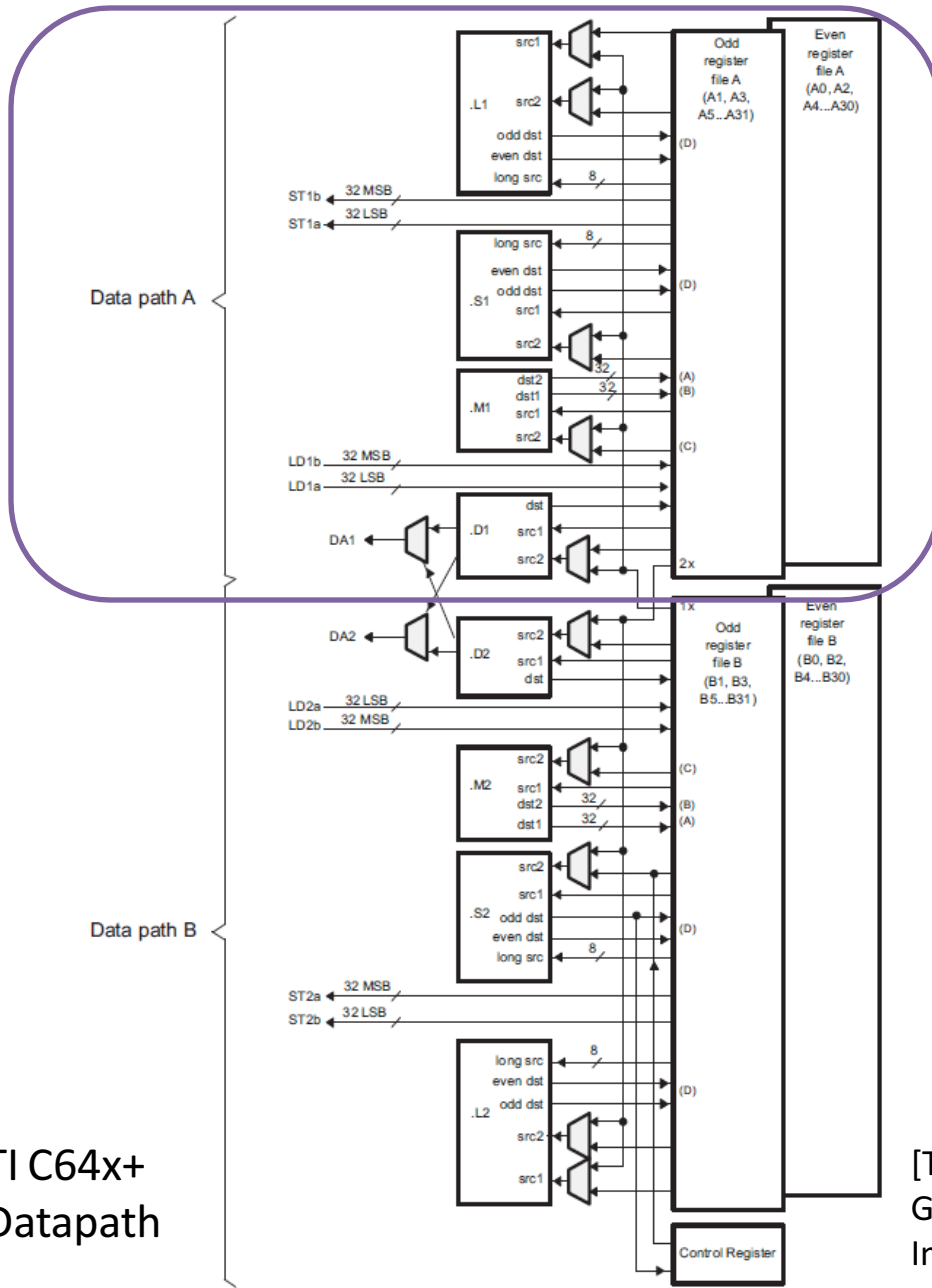
Solution: Allow one instruction to branch multiple directions

```
{ .mii
    cmp.eq P1, P2 = R1, R2
    cmp.ne P3,P4 = R4, R5
    cmp.lt P5,P6, 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 had branch on cache miss
- Branch Mispredict
 - Delay slots with predicated instructions
- Exceptions
 - Hard on superscalar also...

Clustered VLIW



- Divide machine into clusters (local register files and local functional units)
- Lower bandwidth between clusters/Higher latency between clusters
- Used in:
 - HP/ST Lx Processor (printers)
 - TI C6x Series DSP

[TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, TI, 2010, SPRU732J]

Image Credit: Texas Instruments Incorporated

Intel Itanium, EPIC IA-64

Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (CISC, RISC)
 - Explicitly Parallel Instruction Computing

Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (CISC, RISC)
 - Explicitly Parallel Instruction Computing
- IA-64 is Intel's chosen ISA (x86, MIPS)
 - IA-64 = Intel Architecture 64-bit
 - An object-code-compatible VLIW

Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (CISC, RISC)
 - Explicitly Parallel Instruction Computing
- IA-64 is 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]*

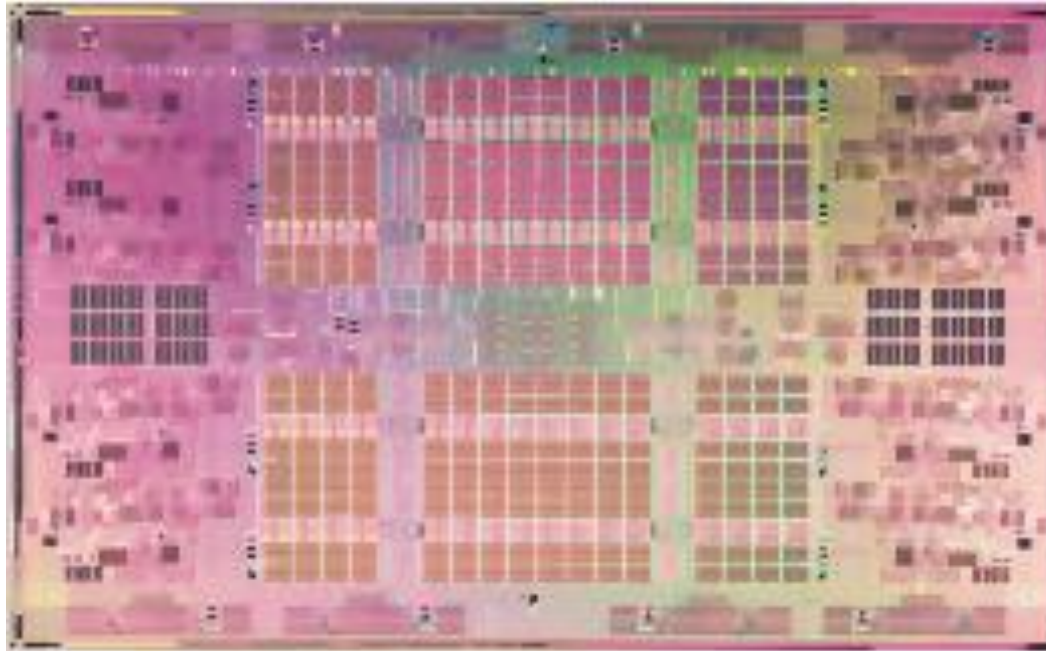


Image Credit: Intel

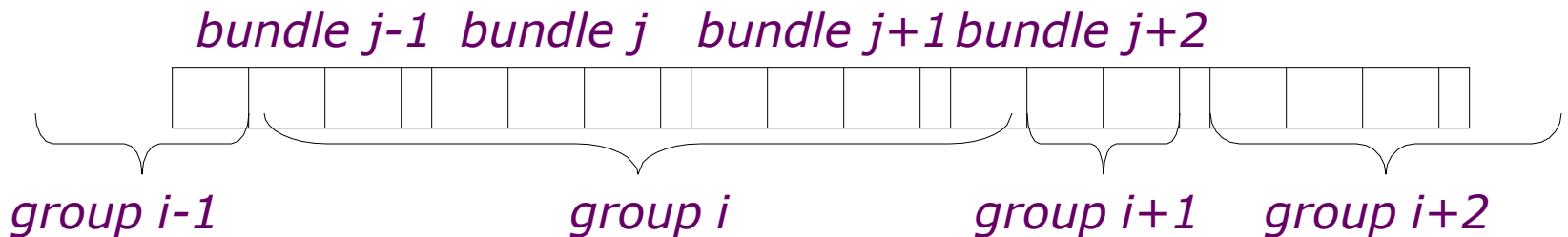
- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm² in 32nm CMOS
- **Over 3 billion transistors**
- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
 - Two 128-bit bundles
- Up to 12 insts/cycle execute

IA-64 Instruction Format



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



IA-64 Registers

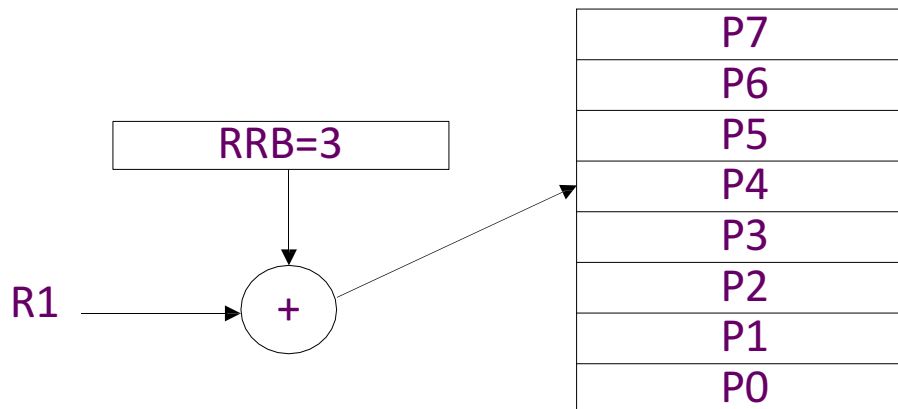
- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

- 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)

lw f1, ()	add.s f5, f4, ...	sw f9, ()	bloop
lw P9, ()	add.s P13, P12,	sw P17, ()	bloop
lw P8, ()	add.s P12, P11,	sw P16, ()	bloop
lw P7, ()	add.s P11, P10,	sw P15, ()	bloop
lw P6, ()	add.s P10, P9,	sw P14, ()	bloop
lw P5, ()	add.s P9, P8,	sw P13, ()	bloop
lw P4, ()	add.s P8, P7,	sw P12, ()	bloop
lw P3, ()	add.s P7, P6,	sw P11, ()	bloop
lw P2, ()	add.s P6, P5,	sw P10, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

RRB=1

Rotating Register File

(Previous Loop Example)

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

Four cycle `add.s` latency
encoded as difference of 4 in
register specifier number ($f9 - f5 = 4$)

<code>lw f1, ()</code>	<code>add.s f5, f4, ...</code>	<code>sw f9, ()</code>	<code>bloop</code>
<code>lw P9, ()</code>	<code>add.s P13, P12,</code>	<code>sw P17, ()</code>	<code>bloop</code>
<code>lw P8, ()</code>	<code>add.s P12, P11,</code>	<code>sw P16, ()</code>	<code>bloop</code>
<code>lw P7, ()</code>	<code>add.s P11, P10,</code>	<code>sw P15, ()</code>	<code>bloop</code>
<code>lw P6, ()</code>	<code>add.s P10, P9,</code>	<code>sw P14, ()</code>	<code>bloop</code>
<code>lw P5, ()</code>	<code>add.s P9, P8,</code>	<code>sw P13, ()</code>	<code>bloop</code>
<code>lw P4, ()</code>	<code>add.s P8, P7,</code>	<code>sw P12, ()</code>	<code>bloop</code>
<code>lw P3, ()</code>	<code>add.s P7, P6,</code>	<code>sw P11, ()</code>	<code>bloop</code>
<code>lw P2, ()</code>	<code>add.s P6, P5,</code>	<code>sw P10, ()</code>	<code>bloop</code>

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

RRB=1

Rotating Register File

(Previous Loop Example)

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

Four cycle `add.s` latency
encoded as difference of 4 in
register specifier number ($f9 - f5 = 4$)

<code>lw f1, ()</code>	<code>add.s f5, f4, ...</code>	<code>sw f9, ()</code>	<code>bloop</code>
------------------------	--------------------------------	------------------------	--------------------

<code>lw P9, ()</code>	<code>add.s P13, P12,</code>	<code>sw P17, ()</code>	<code>bloop</code>
<code>lw P8, ()</code>	<code>add.s P12, P11,</code>	<code>sw P16, ()</code>	<code>bloop</code>
<code>lw P7, ()</code>	<code>add.s P11, P10,</code>	<code>sw P15, ()</code>	<code>bloop</code>
<code>lw P6, ()</code>	<code>add.s P10, P9,</code>	<code>sw P14, ()</code>	<code>bloop</code>
<code>lw P5, ()</code>	<code>add.s P9, P8,</code>	<code>sw P13, ()</code>	<code>bloop</code>
<code>lw P4, ()</code>	<code>add.s P8, P7,</code>	<code>sw P12, ()</code>	<code>bloop</code>
<code>lw P3, ()</code>	<code>add.s P7, P6,</code>	<code>sw P11, ()</code>	<code>bloop</code>
<code>lw P2, ()</code>	<code>add.s P6, P5,</code>	<code>sw P10, ()</code>	<code>bloop</code>

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

RRB=1

Rotating Register File

(Previous Loop Example)

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

Four cycle `add.s` latency
encoded as difference of 4 in
register specifier number ($f9 - f5 = 4$)

<code>lw f1, ()</code>	<code>add.s f5, f4, ...</code>	<code>sw f9, ()</code>	<code>bloop</code>
<code>lw P9, ()</code>	<code>add.s P13, P12,</code>	<code>sw P17, ()</code>	<code>bloop</code>
<code>lw P8, ()</code>	<code>add.s P12, P11,</code>	<code>sw P16, ()</code>	<code>bloop</code>
<code>lw P7, ()</code>	<code>add.s P11, P10,</code>	<code>sw P15, ()</code>	<code>bloop</code>
<code>lw P6, ()</code>	<code>add.s P10, P9,</code>	<code>sw P14, ()</code>	<code>bloop</code>
<code>lw P5, ()</code>	<code>add.s P9, P8,</code>	<code>sw P13, ()</code>	<code>bloop</code>
<code>lw P4, ()</code>	<code>add.s P8, P7,</code>	<code>sw P12, ()</code>	<code>bloop</code>
<code>lw P3, ()</code>	<code>add.s P7, P6,</code>	<code>sw P11, ()</code>	<code>bloop</code>
<code>lw P2, ()</code>	<code>add.s P6, P5,</code>	<code>sw P10, ()</code>	<code>bloop</code>

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

RRB=1

Why Itanium (IA-64) Failed (My Opinion Only)

- 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 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!

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)
- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750

Copyright © 2013 David Wentzlaff