# Today's Plan

**A Brief History Lesson**

- RISC vs. CISC
- Speculation and predication

# Hidden vs. Exposed Technology: RISC vs. CISC

**CISC: Complex Instruction Set Computers**

- VAX, Intel x86, . . .
- Instruction density was important, so architects were motivated to provide many features, i.e., lots of instructions

**Led to complex, highly specialized instructions**

- MNEGF Move negated floating point
- FFS Find first set bit
- INSQHI Insert at head of queue, interlocked
- POLYD Polynomial evaluation

**The VAX had over 300 instructions**

- Variable length
- Multiple complex addressing modes
  - eg. Load indirect off of an offset

# Problems with CISC Architectures

**Collapsing under its own weight**

- Increasingly complex control logic $\Rightarrow$ increasing use of microcode
- Microcode is flexible: essentially "interpreted control unit"
  - Easier to implement than hardwired control
  - Slower than hardwired control

**Change in context**

- Fewer hand-coded assembly programs– more compiled code
- Compilers unable to effectively use complex instructions
- Large memories decrease the benefit of compact code

# Rise of RISC (1980s)

**RISC: Reduced Instruction Set Computers**

- Fixed-length instructions
- Simple addressing modes
- Load/Store architecture
  - Only Loads and Stores access memory

# Advantages of RISC Architectures

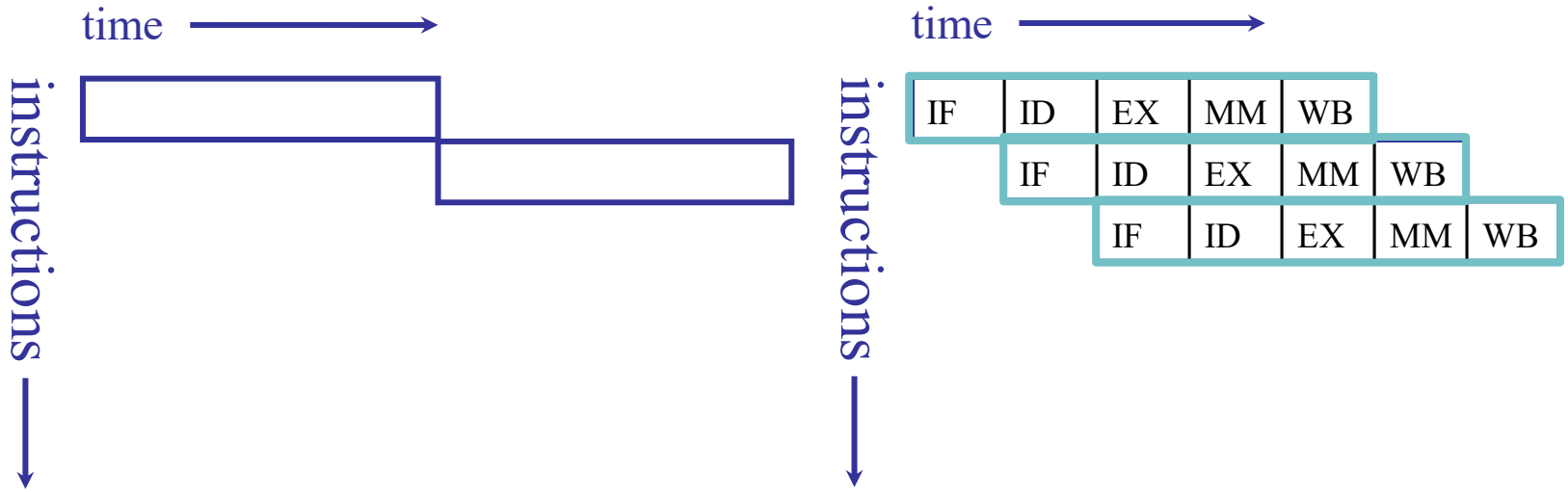
**Regularity and simplicity provide implementation advantages**

- Less complex control logic
- Faster clock rates
- Easier to pipeline operations
  - Easier to schedule multiple concurrent operations
  - Recall pipelined processors. . .

# Pipelined Microarchitectures

**A form of task parallelism**

- Divide instruction processing into multiple stages



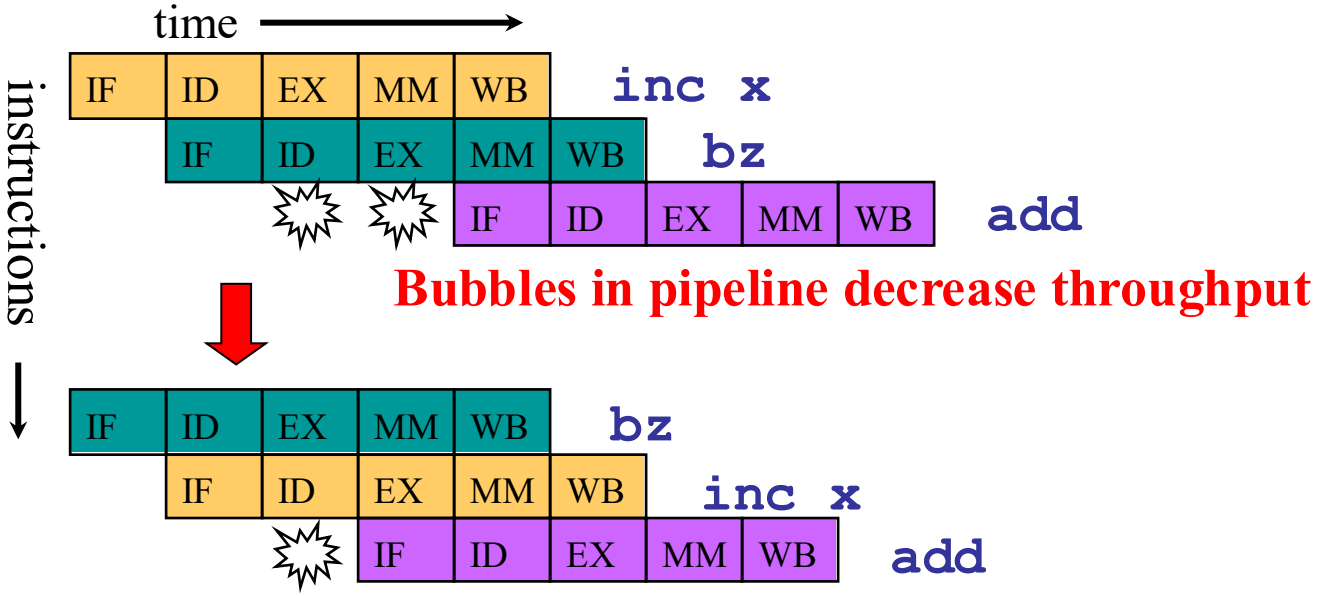

- Increases latency of an individual instruction
- Decreases cycle time
- Increases throughput—ideally completes one instruction per cycle
- Increases overall throughput

# Scheduling to Improve Performance

**Code Fragment**

```
inc x
bz $r1, label        // if $r1==0, branch to label
add $r2,$r3,$r4
```

**Pipelined Execution**



# Advantages of RISC Architectures (cont)

**Compilation advantages**

- Easier to compile for
- Simpler optimization model
  - No variable length instructions
  - Long latency instructions (loads, stores, branches) are exposed
  - Can re-order code to hide the latency of loads, stores, and branches

**Don't pay for what you don't need**

# Lessons from RISC vs. CISC

**If you want parallelism**

- Design it into the language, in this case, the ISA

**If you want effective use of your system**

- Expose costs
  - Load/Store architecture exposes costs
  - Complex instructions hide costs

**These lessons apply to languages and programming models as well**

# Today's Plan

**A Brief History Lesson**

- RISC vs. CISC
- Speculation and predication

# Parallelism and the System Stack

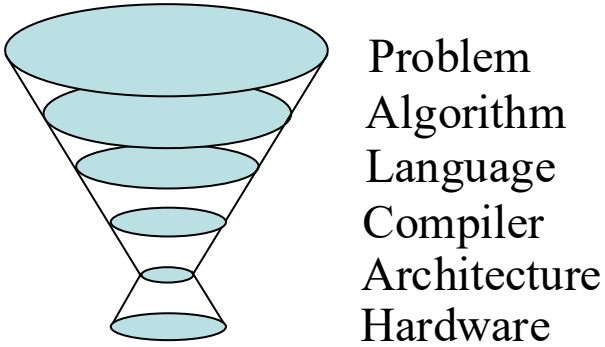
**Superscalar view of the world**

- The ISA is implicitly sequential
- The hardware then dynamically figures out what can execute in parallel

**The problem**

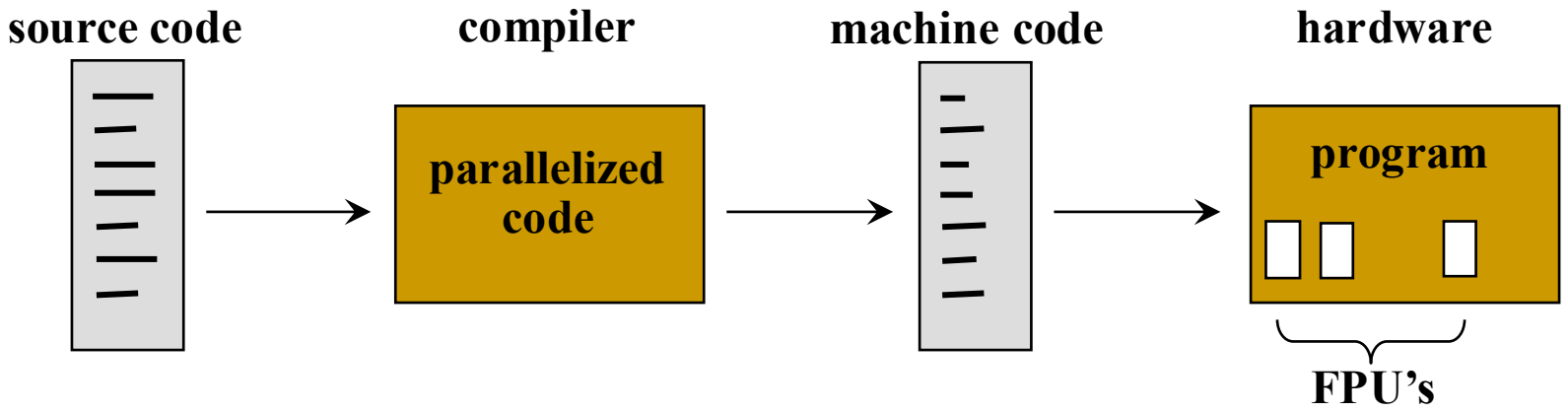
- Each level of the system reduces the amount of available parallelism

**Moore's Funnel***




* [Chuck Moore, c 2003]

# Implicitly Sequential Instruction Stream

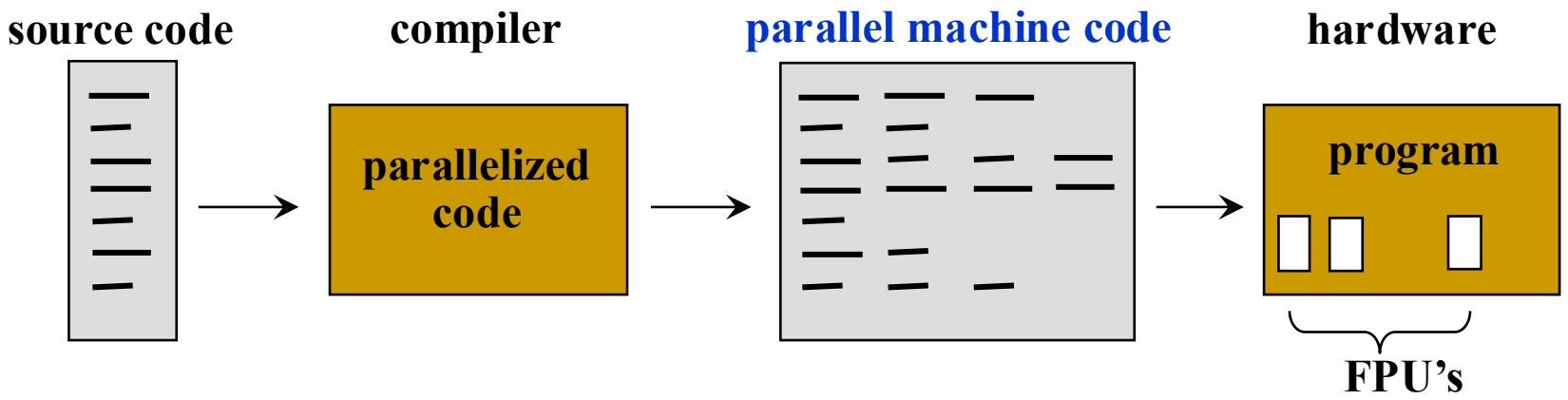



**Problems**

- Compilers can expose parallelism
- Compilers must eventually emit linear code
- Hardware must then re-analyze code to perform OoO execution
  - Hardware loses information available to the compiler
  - Compiler and hardware can only communicate through the sequential stream of instructions, so hardware does redundant work

**How can we solve this problem?**

# Explicitly Parallel Instruction Stream




**A solution**

- Hardware does not need to re-analyze code to detect dependences
- Hardware does not perform OoO execution

**VLIW: Very Long Instruction Word**

- Each instruction controls multiple functional units
- Each instruction is explicitly parallel

# VLIW

**Basic idea**

- Each instruction controls multiple functional units
- Rely on compilers to perform scheduling and to identify parallelism
- Simplified hardware implementations

**Benefits**

- Compiler can look at a larger window of instructions than hardware
- Can improve the scheduler even after a chip has been fabricated

**Problems**

- Slow compilation times
- No binary compatibility
  - Code is implementation-specific
- Difficult for compilers to deal with aliasing and long latencies

# IA-64 (2001)

**A return to hardware "simplicity"**

- Revisit the ideas of VLIW
- Simplify the hardware to make it faster
- Spend larger percentage of cycles doing actual work
- Spend larger percentage of hardware on registers, caches, and FPU's
- Use larger number of registers to support more parallelism

# Two Key Performance Bottlenecks

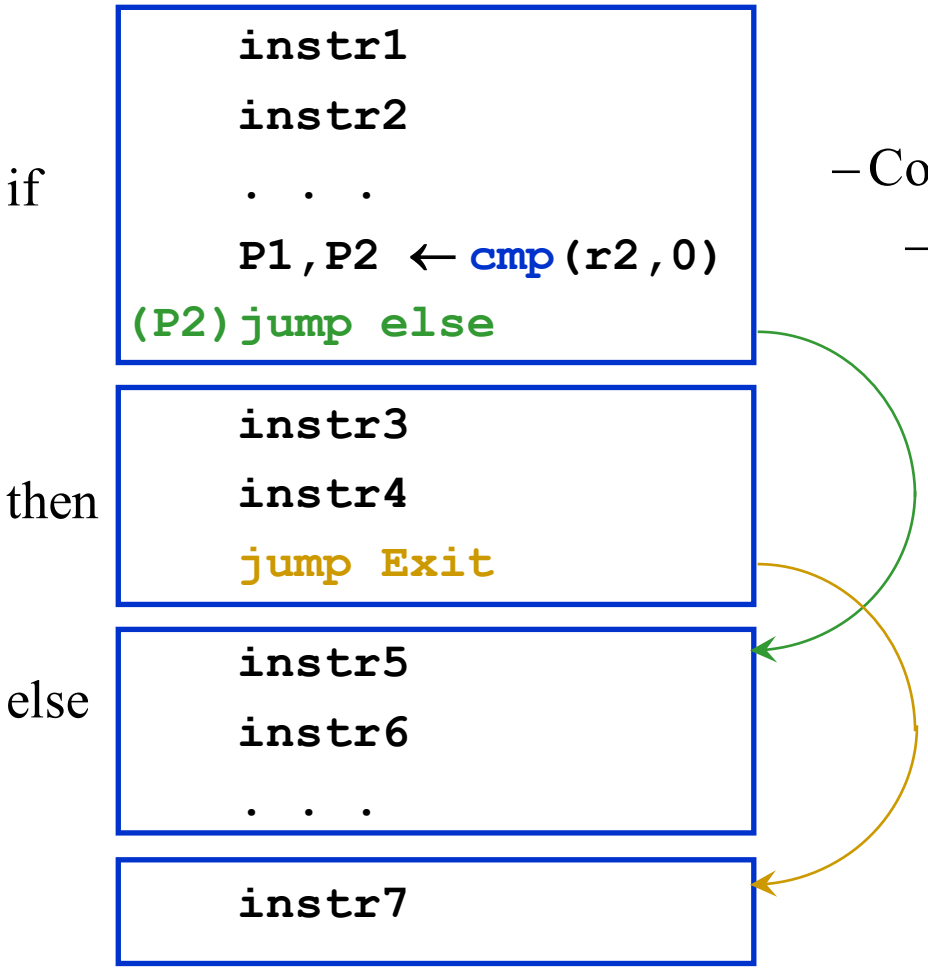
**Branches**

- Modern microprocessors perform good branch prediction
- But when they mispredict, the penalty is high
- Estimates: 20-30% of performance goes to branch mispredictions [Intel98]
- Branches also lead to small basic blocks, which restrict latency hiding opportunities

**Memory latency**

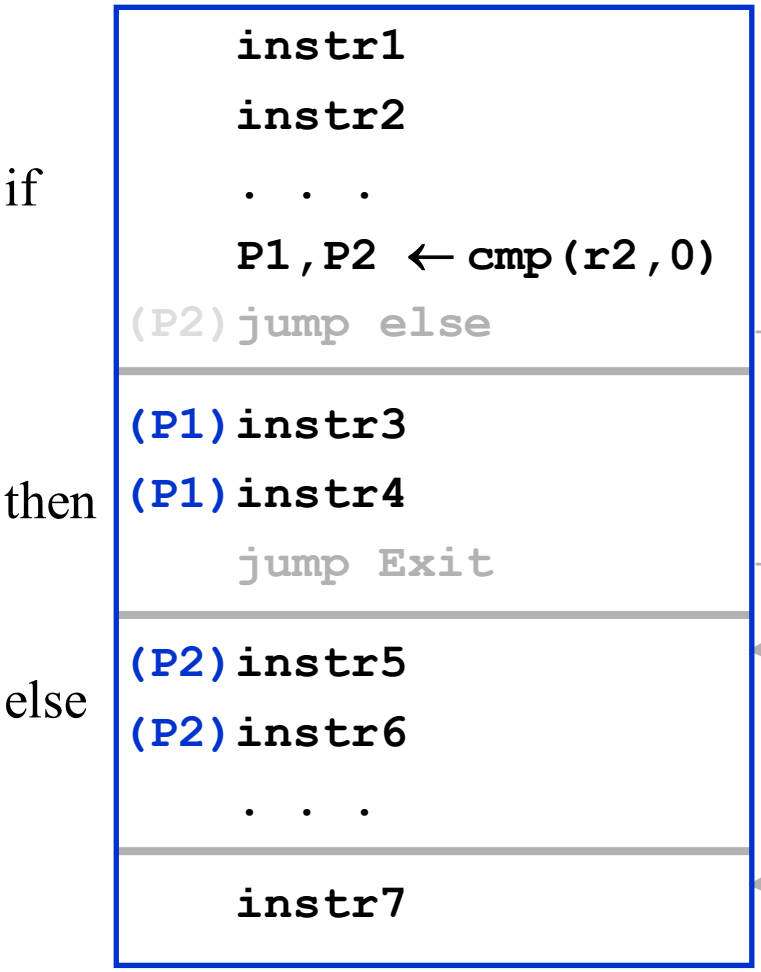
- Even with cache hierarchies, memory access takes multiple cycles in the best case

# Branches Limit Performance




- Control dependences inhibit parallelism
  - Don't know whether to execute instr3 or instr5 until the cmp is completed

# Predicated Execution




**This is called *if-conversion***

**Idea**

- Add a predicate flag to each instruction
  - If predicate is true, the instruction is executed
  - If predicate is false, the instruction is not executed
  - Predicates are simply bits in a register
- Converts control flow into data flow
  - Exposes parallelism
  - With predicate flags, instr3 – instr7 can all be fetched in parallel

**Benefits**

- Fewer branches (fewer mispredictions)
- Larger basic blocks
- More parallelism

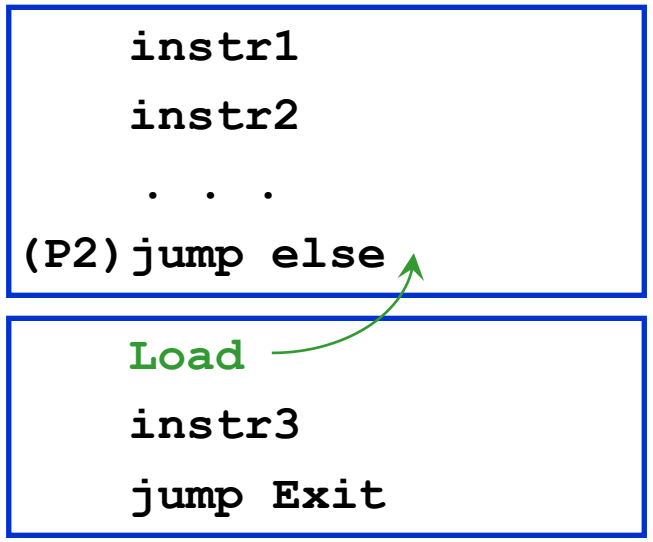
# The Memory Latency Problem

**Memory Latency**

- Writes can be done out of order and can be buffered
- Loads are the problem: processor must wait for loads to complete before using the loaded value
- Standard latency-hiding trick: issue non-blocking load as early as possible to hide latency

**The Problem**

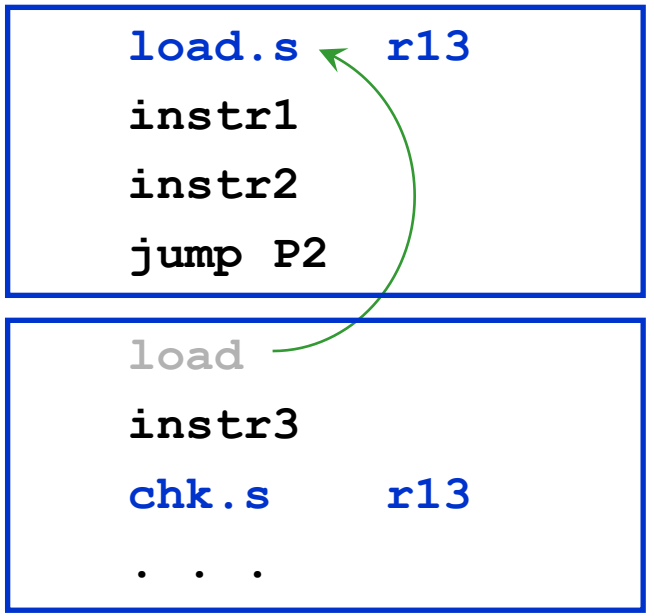
- Loads typically issued at beginning of basic block
- Can't move the Load outside the basic block
  - If the Load were to cause an exception when the basic block is not executed, then the early Load causes an erroneous exception



# (Control) Speculative Loads

**Split-phase operation**

- Issue the load (load.s) as early as you wish
- Detect any exception and record it somewhere with the target of the load
- Can later check to see whether the load completed successfully: chk.s



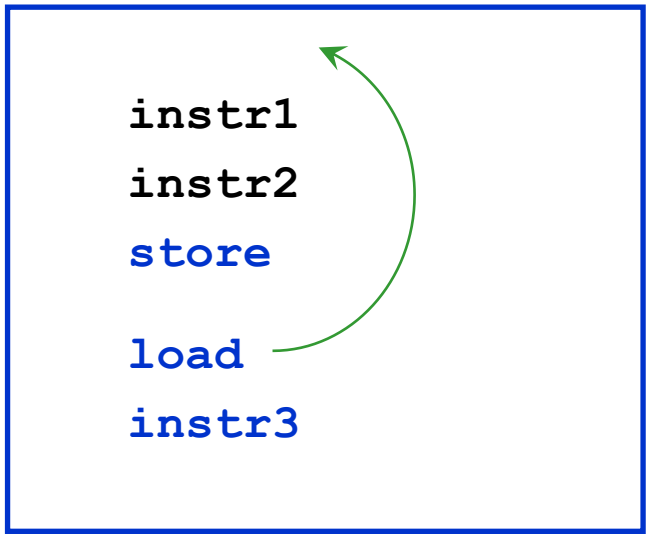

**Benefits**

- More freedom to move code– can now move Loads above branches as long as the check is in the original basic block
- Complication: We have an exception if chk.s is issued without a corresponding load.s

# (Data) Speculative Loads

**Another Issue**

- Suppose that `instr3` depends on the value of a `load`
- Would like to move the load as early as possible to hide its latency
- Cannot move load before the `store` because there is a data dependence if they use the same address



# (Data) Speculative Loads (cont)

**Solution**

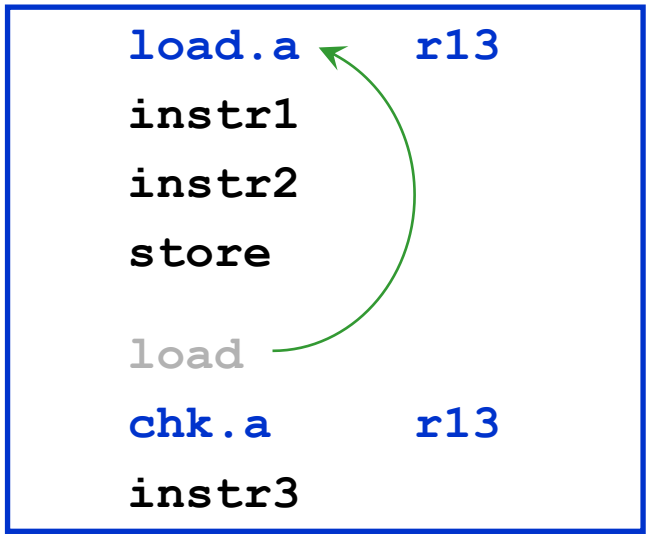
- Speculate that the load is not data dependent on the preceding store

**Split-phase operation**

- Issue early advanced load
- Check for completion before using the result (**`r13`**)
- Hardware needs to keep track of these advanced loads and their dependences

**Note**

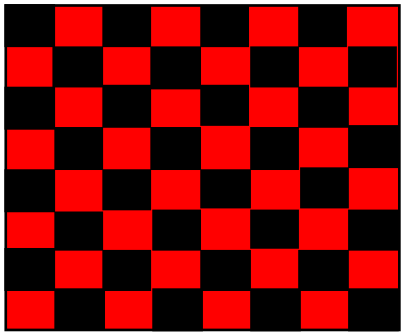
- Can speculate instructions that depend on load, too



# N-Queens Example

**The Problem**

- Place N Queens on a chessboard so they don't attack each other



**The Solution**

- March through columns with a recursive procedure
  - B array: check the row
  - A and C arrays: check the two diagonals
  - Code to test if the $(i,j)^{th}$ square is legal:

```
if ((b[i]==0) && (a[i+j-1]==0) && c[i+j+N]==0)
```

# N-Queens Solution

```
if ((b[i]==0) && (a[i+j-1]==0) && c[i+j+N]==0)
```

```
    r1 = &b[i]
1   r3 = &a[i+j-1]
    r5 = &c[i+j+N]
2   load r2 = [r1]
4   P1,P2 ← cmp(r2,0)
5   <P2> jump Else

6   load r4 = [r3]
8   P3,P4 ← cmp(r4,0)
9   <P4> jump Else

10  load r6 = [r5]
12  P5,P6 ← cmp(r6,0)
13  <P5> jump Then

    Else . . .
```

Load the addresses into registers

De-reference the arrays

Test for legality

(Assume we can issue 3 instructions per cycle)

Parallelism boundary

Basic block boundary

**Summary**

3 basic blocks

12 instructions

13 cycles (assuming 1 cycle per instruction except 2 cycles per load)

3 conditional branches

**Almost no parallelism!**

# N-Queens Solution: Adding Speculation

**Original Code**

```
    r1 = &b[i]
1   r3 = &a[i+j-1]
    r5 = &c[i+j+N]
2   load r2 = [r1]
4   P1,P2 ← cmp(r2,0)
5   <P2> jump Else

6   load r4 = [r3]
8   P3,P4 ← cmp(r4,0)
9   <P4> jump Else

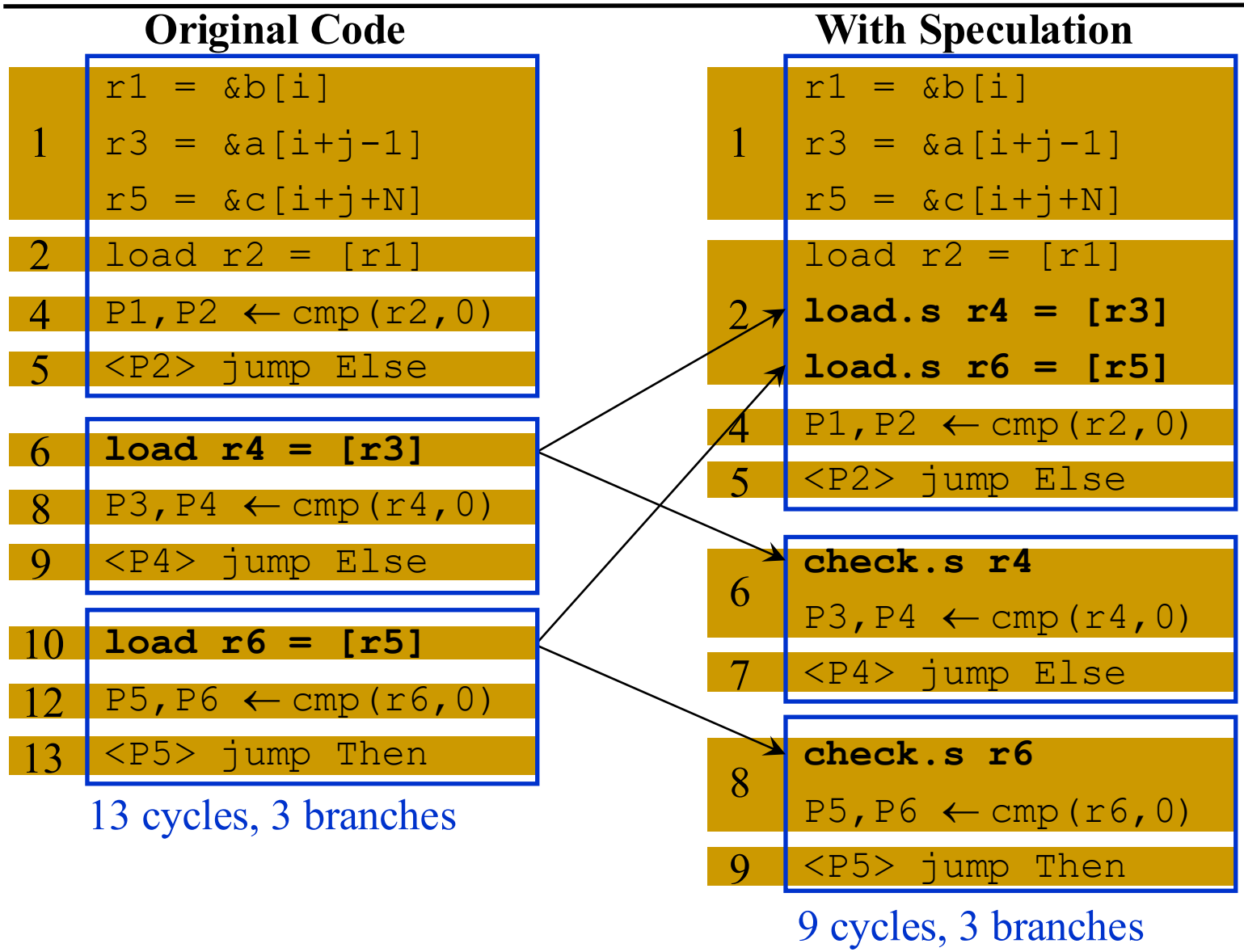
10  load r6 = [r5]
12  P5,P6 ← cmp(r6,0)
13  <P5> jump Then
```

13 cycles, 3 branches

Each of these loads takes two cycles

Each of these loads is issued in isolation

# N-Queens Solution: Adding Speculation



# N-Queens Solution: Adding Predication

**With Speculation**

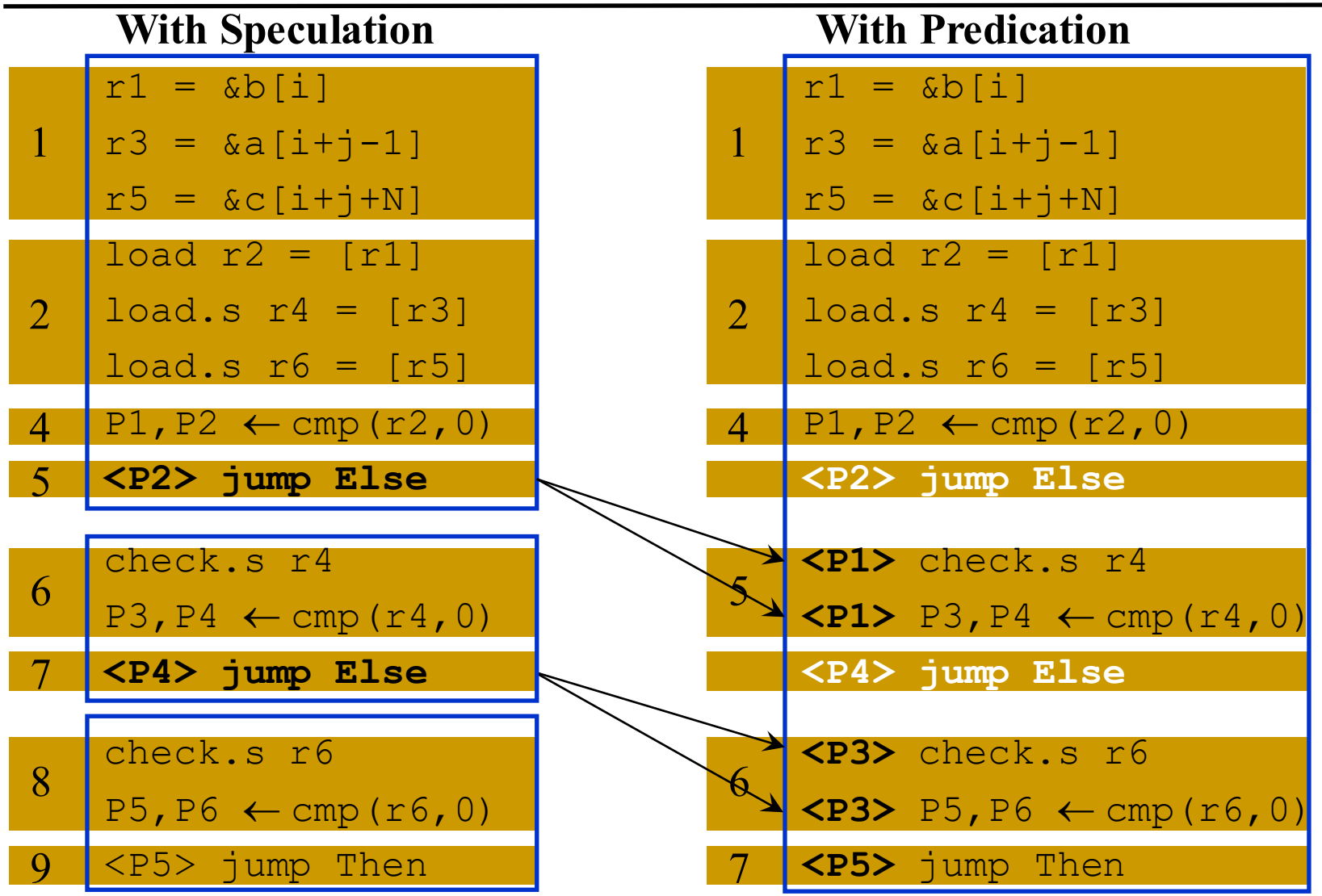
```
1   r1 = &b[i]
    r3 = &a[i+j-1]
    r5 = &c[i+j+N]
2   load r2 = [r1]
    load.s r4 = [r3]
    load.s r6 = [r5]
4   P1,P2 ← cmp(r2,0)
5   <P2> jump Else
6   check.s r4
    P3,P4 ← cmp(r4,0)
7   <P4> jump Else
8   check.s r6
    P5,P6 ← cmp(r6,0)
9   <P5> jump Then
```

Good parallelism

Control flow boundary

9 cycles, 3 branches

# N-Queens Solution: Adding Predication




9 cycles, 3 branches

# N-Queens Solution: Summary

**Original Code**

```
1   r1 = &b[i]
    r3 = &a[i+j-1]
    r5 = &c[i+j+N]
2   load r2 = [r1]
4   P1,P2 ← cmp(r2,0)
5   <P2> jump Else

6   load r4 = [r3]
8   P3,P4 ← cmp(r4,0)
9   <P4> jump Else

10  load r6 = [r5]
12  P5,P6 ← cmp(r6,0)
13  <P5> jump Then
```

13 cycles, 3 branches

**Predication**

```
1   r1 = &b[i]
    r3 = &a[i+j-1]
    r5 = &c[i+j+N]
2   load r2 = [r1]
    load.s r4 = [r3]
    load.s r6 = [r5]
4   P1,P2 ← cmp(r2,0)

5   <P1> check.s r4
    <P1> P3,P4 ← cmp(r4,0)

6   <P3> check.s r6
    <P3> P5,P6 ← cmp(r6,0)
7   <P5> jump Then
```

7 cycles, 1 branch

# Predication is an Old Idea

**High performance computing**

- SIMD machines (Single Instruction Multiple Data)
  - All processors operate in lock-step but operate on different data
  - What do you do with control flow?

```
if (A[i][j] < 0)
    A[i][j] = -A[i][j]
```

  - Compute a mask of 0's and 1's
  - Execute both halves of the control flow using the appropriate mask
  - Can do this in either hardware or software

```
Mask[i][j] = (A[i][j] < 0)
A[i][j] -=  Mask[i][j] * 2 * A[i][j]
```

# Is Speculation a Good Idea?

**Disadvantages of speculation**

- Wasted work
- Increased complexity

**Who should perform speculation?**

- The hardware can exploit runtime information
- The compiler can exploit a much larger scope

**Speculation**

- An example of a split-phase operation
  - Split-phase communication

**Security concerns**

# Implications

**IA64**

- The ideas are not new
- The willingness to change the ISA was new and significant

**Implications for compilers**

- Increased role of the compiler
- More control over sequencing, prefetching, stores, branch prediction
  - Hardware doesn't "undo" the compiler's work

**Future systems**

- What is the right division of labor between the compiler and the hardware?
- How else can compilers be used to simplify the hardware and make the hardware more effective?
- Can we improve the communication between the compiler and hardware?