IA-64 Architecture Innovations

Carole Dulong
Principal Engineer
IA-64 Computer Architect
Intel Corporation

February 19, 1998

intel®

Today's Objectives

- Provide background on key IA-64 architecture concepts
  - Today's focus on architecture
- Demonstrate IA-64 architecture innovation benefits
  - A few key features
- Describe what the IA-64 architecture means for developers
  - New levels of performance
  - Better utilization of wider machines

IA-64 Architecture Innovations
Outline

- Background
  - IA-64 architecture objectives
  - Explicit Parallelism
  - Predication
  - Control Speculation
- IA-64 at work: Code Examples
- Summary

IA-64 Architecture Objectives

- Enable industry leading system performance
  - Overcome traditional architectures' performance limiters
- Provide end user investment protection
  - Enable full binary compatibility with IA-32 software
- Allow scalability over a wide range of implementations
- Full 64-bit computing

intel®

IA-64 Architecture Applies EPIC Design Philosophy

Explicitly Parallel Instruction Computing

- **Explicit parallelism**
  - ILP is explicit in machine code
  - Compiler schedules across a wide scope
  - Binary compatibility across all family members
- **Features that enhance Instruction Level Parallelism**
  - Predication
  - Speculation
  - Others...
- **Resources for parallel execution**
  - Many registers
  - Many functional units
  - Inherently scalable

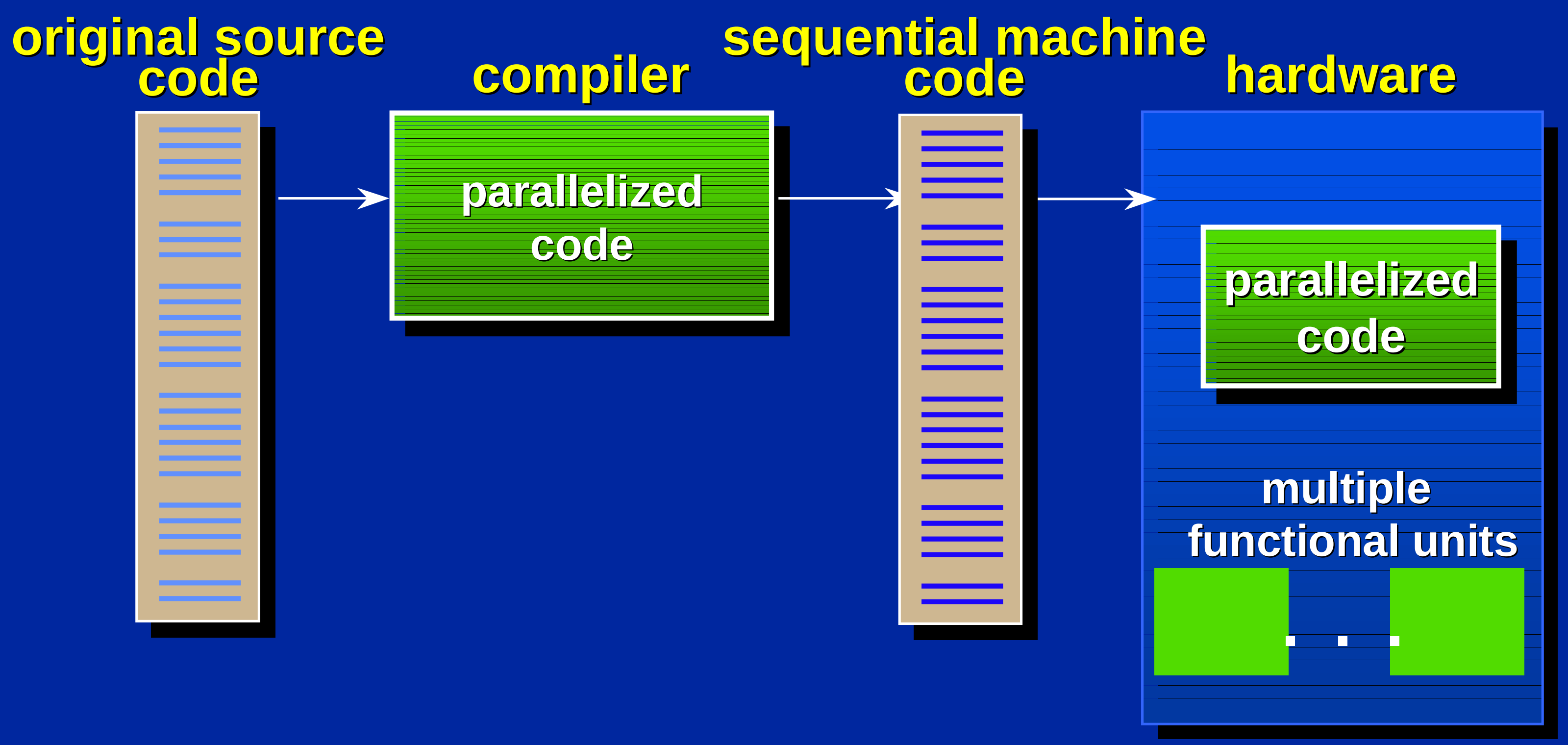
intel®

A Few Key IA-64 Architecture Innovations

- **Explicit Parallelism**
  - Enables compiler to expose Instruction Level Parallelism
- **Predication**
  - Removes branches & mispredicts
- **Speculation**
  - Minimizes the effect of memory latency

Extracting Parallelism

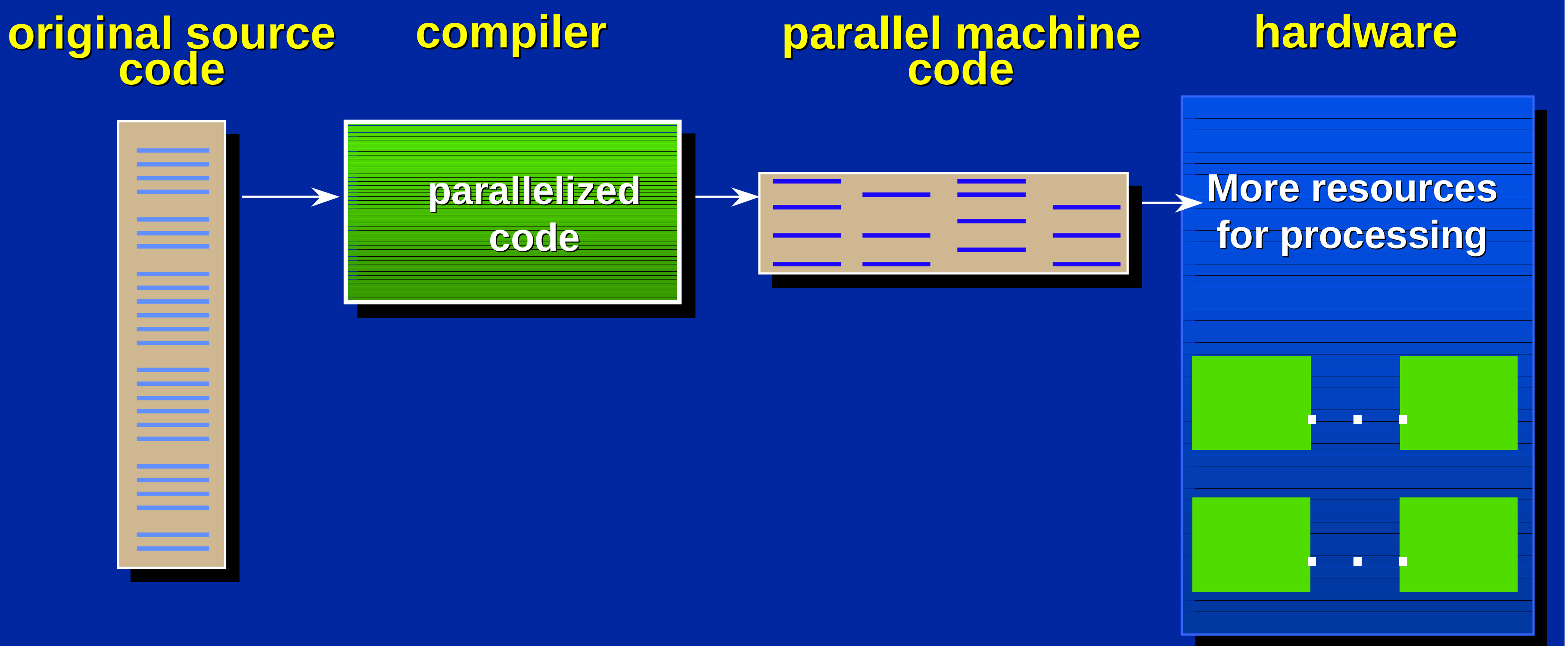
- Sequential execution model



- Compiler has limited, indirect view of hardware

Traditional architectures enable limited parallelism

Better Strategy: Explicit Parallelism



Compiler exposes, enhances, and exploits parallelism in the source program and makes it <u>explicit</u> in the machine code.

Explicit parallelism increases ILP

intel®

Branches Limit Performance

if	Load a[i].ptr p1, p2 = cmp a[i].ptr != 0 branch if p2
then	Load a[i].l store b[i] branch
else	Load a[i].r store b[i]
	i = i + 1

```
If a[i].ptr != 0
        b[i] = a[i].l;
else
        b[i] = a[i].r;
i = i + 1
```

Traditional Architectures: 4 basic blocks

Control flow introduces branches

Predication

if	Load a[i].ptr p1, p2 = cmp a[i].ptr != 0 ~~branch~~ if p2
then	**<p1>** Load a[i].l **<p1>** store b[i] ~~branch~~
else	**<p2>** Load a[i].r **<p2>** store b[i]
	i = i + 1

```
If a[i].ptr != 0
        b[i] = a[i].l;
else
        b[i] = a[i].r;
i = i + 1
```

Predication removes branches and eliminates mispredicts

Predication Enhances Parallelism

Traditional Architectures: 4 basic blocks

if	Load a[i].ptr p1, p2 = cmp a[i] != 0 jump if p2
then	Load a[i].l store b[i] jump
else	Load a[i].r store b[i]
	i = i + 1

IA-64™ Architecture: 1 basic block

Load a[i].ptr
p1, p2 = cmp a[i] != 0

<p1> Load a[i].l <p2> Load a[i].r
<p1> store b[i] <p2> store b[i]

i = i + 1

Predication enables more effective use of parallel hardware

Memory Latency Causes Delays

- **Loads significantly affect performance**
  - **Often first instruction in dependency chain of instructions**
  - **Can incur high latencies**

Traditional Architectures

```
Add  t1 + 1
comp t1 > t2
branch
```

Barrier

```
Load a[t1-t2]
Load b[j]
add b[j] + 1
```

```
t1 = t1 + 1
If  t1 > t2
      j = a[t1 - t2]
      b[j] ++
```

Loads can cause exceptions

Speculation with IA-64 Architecture

```
Add  t1 + 1
load.s a[t1-t2]
comp t1 > t2
jump
```

;Exception Detection

Propagate Exception

```
Check.s
Load b[j]
add b[j] + 1
```

;Exception Delivery

- **Separate load behavior from exception behavior**
  - Speculative load instruction (load.s) initiates a load operation and detects exceptions
  - Propagate an exception "token" (stored with destination register) from load.s to check.s
  - Speculative check instruction (check.s) delivers any exceptions detected by load.s

Speculation Minimizes the Effect of Memory Latency

Traditional Architectures

```
Add  t1 + 1
comp t1 > t2
jump
```

Barrier

```
Load a[t1-t2]
Load b[j]
add b[j] + 1
```

IA-64 Architecture

```
Add  t1 + 1
load.s a[t1-t2]
comp t1 > t2
jump
```

;Exception Detection

Propagate Exception

```
Check.s
Load b[j]
add b[j] + 1
```

;Exception Delivery

- **Give scheduling freedom to the compiler**
  - Allows load.s to be scheduled above branches
  - check.s remains in home block, branches to fixup code if an exception is propagated

intel®

Predication & Speculation

With Predication

```
        Load a[i].ptr
        p1, p2 = cmp a[i].ptr != 0

<p1> Load a[i].l<p2> Load a[i].r
<p1> store b[i] <p2> store b[i]

        i = i + 1
```

With Predication & Speculation

```
        Load a[i]
        load.s a[I].l load.s a[I].r
        p1, p2 = cmp a[i] != 0
<p1> check.s   <p2> check.s
<p1> store b[i] <p2> store b[i]

        i = i + 1
```

Predication and Speculation = higher ILP

```
If a[i].ptr != 0
        b[i] = a[i].l;
else
        b[i] = a[i].r;
i = i + 1
```

IA-64: Next Generation Architecture

- A unique combination of innovative features:
  - Explicit Parallelism
  - Predication
  - Speculation

  :

IA-64 Architecture: Performance, Scalability, and Compatibility

intel®

IA-64 Architecture Innovations
Outline

- Background
- IA-64 at work: Code Examples
  - xlxgetvalue from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - treeins code fragment
    - classic if-then-else statement
- Summary

Key IA-64 Architecture Features Applied

Demonstrate the IA-64 architecture benefits on representative code fragments

- **Pointer chasing**
  - control speculation to load data before pointer safety check
- **Loop with conditional statement**
  - control speculation to unroll loop
  - predication to remove hard to predict branches
- **If-then-else statements**
  - predication to execute all paths in parallel

Classic Control Flow Structures in General Purpose Code

intel®

IA-64 Architecture Innovations
Outline

- Background
- IA-64 at work: Code Examples
  - xlxgetvalue from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - treeins code fragment
    - classic if-then-else statement
- Summary

Xlxgetvalue in a Nutshell

- Code fragment from SpecInt95 benchmark LI
  - representative of pointer chasing code
- Technique used: Control Speculation
- Benefits:
  - hide memory latency
  - expose Instruction Level Parallelism allowing parallel execution

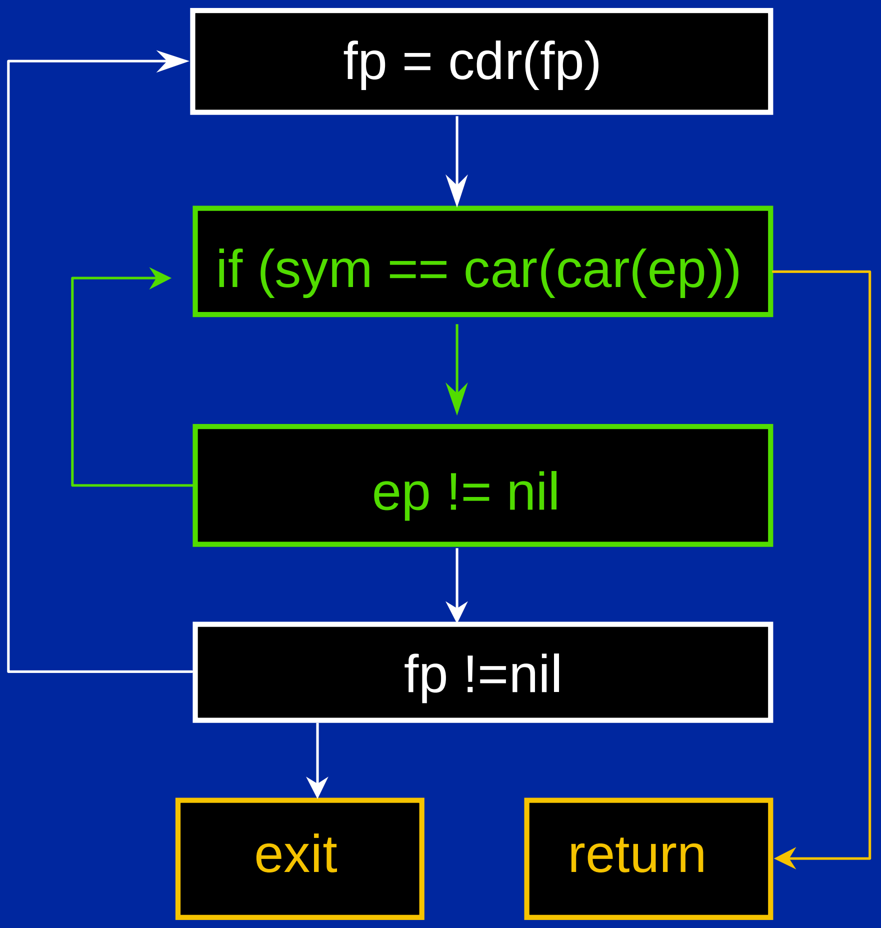
Xlxgetvalue Step by Step

- Compile one iteration
  - use control speculation to issue loads as early as possible
- Unroll the loop
  - use control speculation to start next iteration before it is safe to do so
  - take advantage of the machine width to execute several iterations in parallel

Expose ILP with Control Speculation in pointer chasing code

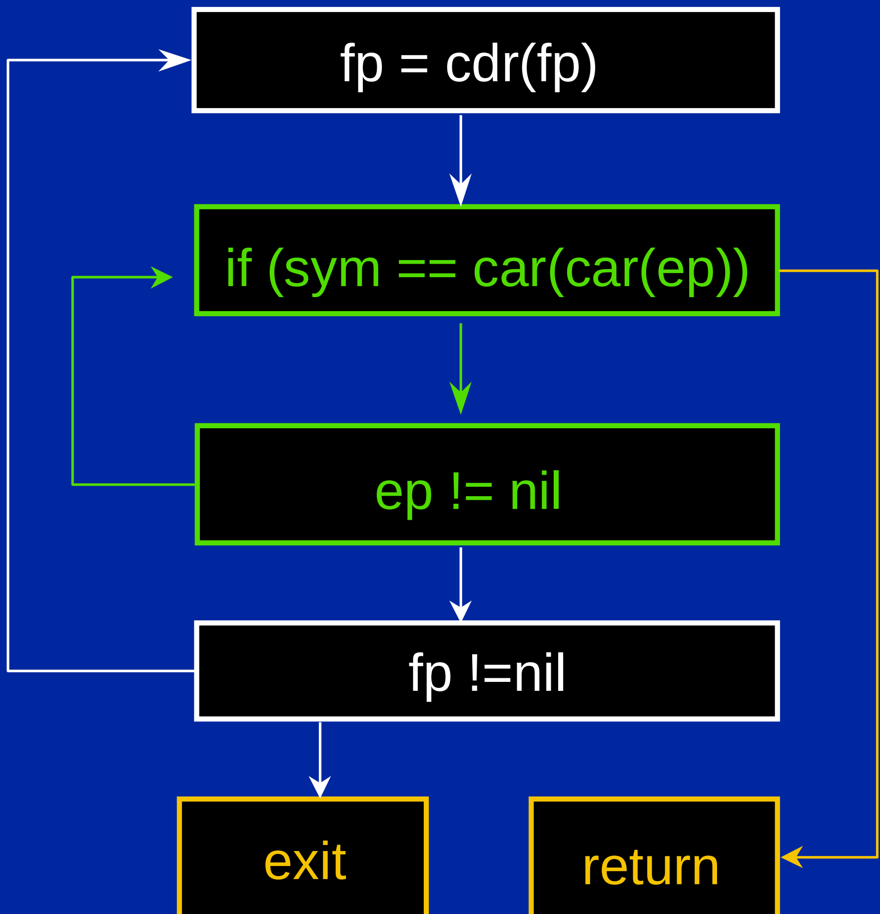
Xlxgetvalue Code Fragment

```
for (fp = xlenv; fp;
                fp = cdr(fp))
  for (ep = car(fp); ep;
             ep = cdr(ep))
     if (sym ==
                car(car(ep)))
          return (cdr(car(ep)));
```



intel®

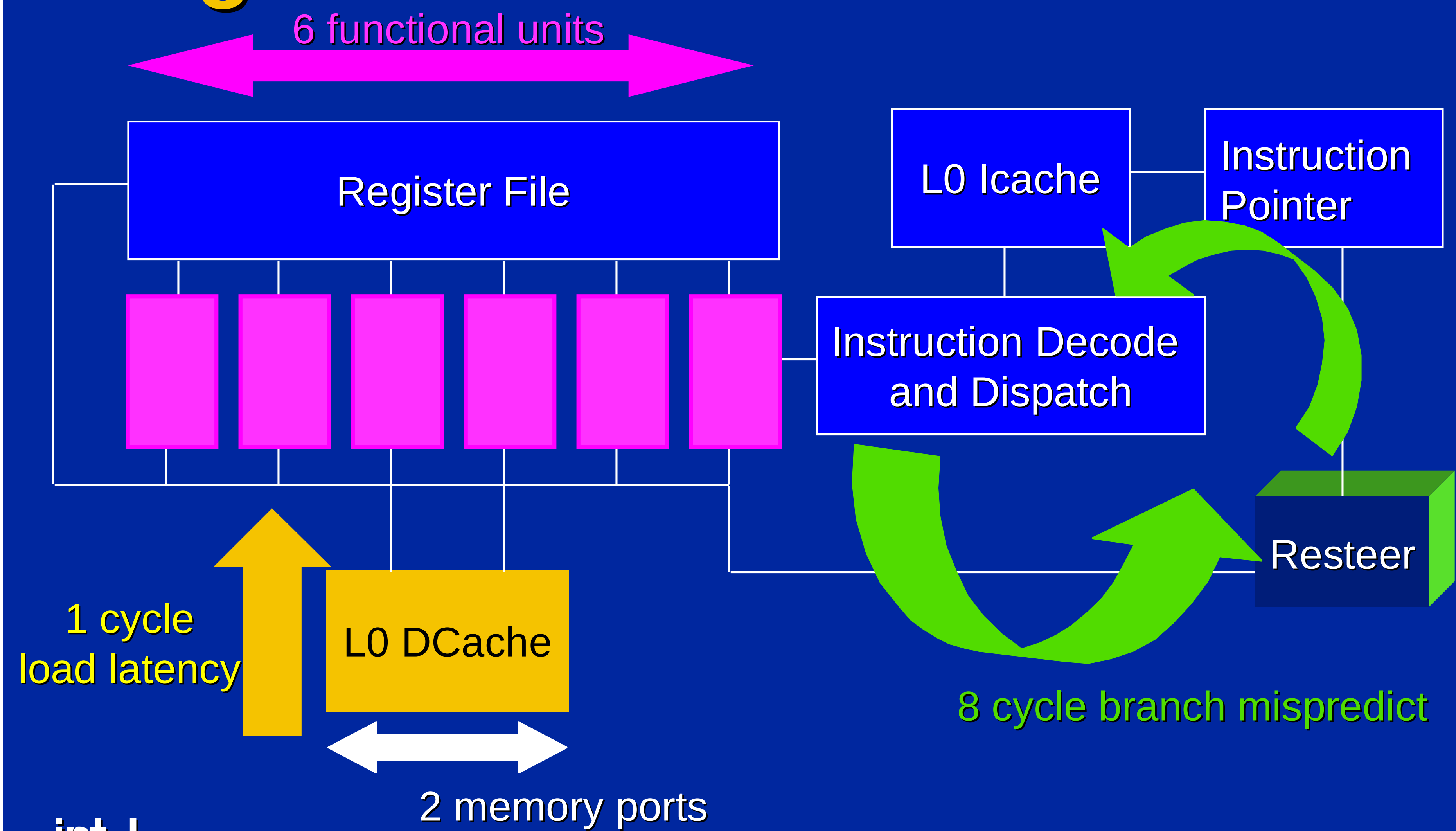
Compiling Serial Code . . .




```
Ld fp
cmp fp == nil
br to exit if true
Load ep
Cond1 = (cmp ep ==nil)
br nxt_fp if Cond1
load car(ep)
load x= car(car(ep))
Cond2 = (comp sym== x)
br to return if Cond2
br nxt_ep
```

intel

Example Machine Model for xlxgetval



6 Execution units, 2 memory ports, 1 cycle load latency

Compiling . . .

```
for (ep = car(fp); ep; ep = cdr(ep))
    if (sym == car(car(ep)))
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load ep1					
2						
3						
4						
5						

intel®

Compiling . . .

```
for (ep = car(fp); ep; ep = cdr(ep))
    if (sym == car(car(ep)))
```

Cycle	Unit 1	Uni t 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load ep1					
2	Cond1 = ep1 == nil					
3						
4						
5						

intel®

Compiling . . .

```
for (ep = car(fp); ep; ep = cdr(ep))
    if (sym == car(car(ep)))
```

Cycle	Unit 1	Uni t 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load ep1					
2	Cond1 = ep1 == nil	Load.s car(ep1)				
3	check.s					
4						
5						

intel®

Compiling . . .

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Uni t 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load ep1					
2	Cond1 = ep1 == nil	Load.s car(ep1)				
3	check.s	Load x=car (car(ep1))				
4						
5						

intel®

Compiling . . .

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Uni t 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load ep1					
2	Cond1 = ep1 == nil	Load.s car(ep1)				
3	check.s	Load x=car (car(ep1))				
4	Cond2 = sym== x					
5						

intel®

First Iteration

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Uni t 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load ep1					
2	Cond1 = ep1 == nil	Load.s car(ep1)	Br nxt_fp if cond1			
3	check.s	Load x=car (car(ep1))				
4	Cond2 = sym== x	Br return if cond2	Br nxt_ep			
5						

Speculation allows the loads to be started early

Second Iteration: Unrolling . . .

for (ep = car(fp); ep; ep = cdr(ep))

if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	U. 6
1	Ld ep1					
2	Ld.s car(ep1)	Cond1 =Cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
3	Check.s	Ld car car(ep1)				
4	Cond2 =Cmp == symm			Br return if cond2		
5						

intel®

Second Iteration: Unrolling . . .

for (ep = car(fp); ep; ep = cdr(ep))

if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	U. 6
1	Ld ep1					
2	Ld.s car(ep1)	Cond1 = Cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
3	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3= Cmp ep2==nil		
4	Cond2 = Cmp == symm	Check.s		Br return if cond2	Br nxt_fp if cond3	
5						

Only 1 check for 2 dependent loads

Second Iteration: Unrolling . . .

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	U. 6
1	Ld ep1					
2	Ld.s car(ep1)	Cond1 = Cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
3	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3 = Cmp ep2==nil		
4	Cond2 = Cmp == symm	Check.s	Ld car car(ep2)	Br return if cond2	Br nxt_fp if cond3	
5						

intel®

Second Iteration: Unrolling . . .

for (ep = car(fp); ep; ep = cdr(ep))

if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	U. 6
1	Ld ep1					
2	Ld.s car(ep1)	Cond1 = Cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
3	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3 = Cmp ep2==nil		
4	Cond2 = Cmp == symm	Check.s	Ld car car(ep2)	Br return if cond2	Br nxt_fp if cond3	
5		Cond4 = Cmp == sym	Br return if cond4	Br nxt_ep		

intel®

Speculation enables efficient machine utilization

Optimized Code

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	U. 6
0	Ld ep1	(Done outside of the loop)				
1	Ld.s car(ep1)	Cond1 = Cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
2	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3 = Cmp ep2==nil		
3	Cond2 = Cmp == symm	Check.s	Ld car car(ep2)	Br return if cond2	Br nxt_fp if cond3	
4	Ld nxt ep1 = cdr(ep2)	Cond4 = Cmp == sym	Br return if cond4	Br nxt_ep		

First load can be done at the bottom of the loop

Scheduled without Control Speculation

Loads are delayed by one clock

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	U. 6
0	Ld ep1	Done outside of the loop				
1	Ld.s car(ep1)	Cond1 = Cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
2	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3 = Cmp ep2==nil		
3	Cond2 = Cmp == symm	Check.s	Ld car car(ep2)	Br return if cond2	Br nxt_fp if cond3	
4	Ld nxt ep1 = cdr(ep2)	Cond4 = Cmp == sym	Br return if cond4	Br nxt_ep		

int

Recompiled without Speculation

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1		Cond1 = Cmp ep == nil		Br nxt_fp if cond1		
2	Ld car(ep1)	Ld ep2 = cdr(ep1)				
3	Ld car car(ep1)		Cond3 = Cmp ep2==nil			
4	Cond2 = Cmp == symm	Ld car(ep2)	Br return if cond2	Br nxt_fp if cond3		
5		Ld car car(ep2)				
6	Ld nxt ep1 = cdr(ep2)	Cond4 = Cmp == sym	Br return if cond4	Br nxt_ep		

Inefficient use of machine width

xlxgetvalue Conclusions

- **Control speculation Benefits:**
  - hides memory latency through
    - loading data before knowing if the address is a valid pointer
    - loading data before knowing if the next loop iteration is valid
  - enables the compiler to expose parallelism in pointer chasing code

On average over 50% of loads can be executed speculatively

intel®

Scoreboard

With speculation

Without speculation

xlxgetvalue

2 cycles per iteration

3 cycles per iteration

Speculation provides a significant performance advantage

IA-64 Architecture Innovations
Outline

- Background
- IA-64 at work: Code Examples
  - xlxgetvalue from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - treeins code fragment
    - classic if-then-else statement
- Summary

intel®

Puzzle in a Nutshell

- Classic Code fragment
  - representative of loop with nested if statements
- Technique used: Predication and Control Speculation
- Benefits:
  - remove hard to predict branches & mispredicts
  - expose Instruction Level Parallelism allowing parallel execution

Puzzle Step by Step

- Compile one iteration
  - use predication to remove a hard to predict branch
- Unroll the loop
  - use control speculation to start next iteration before it is safe to do so
  - take advantage of the width of the machine to execute several iterations in parallel

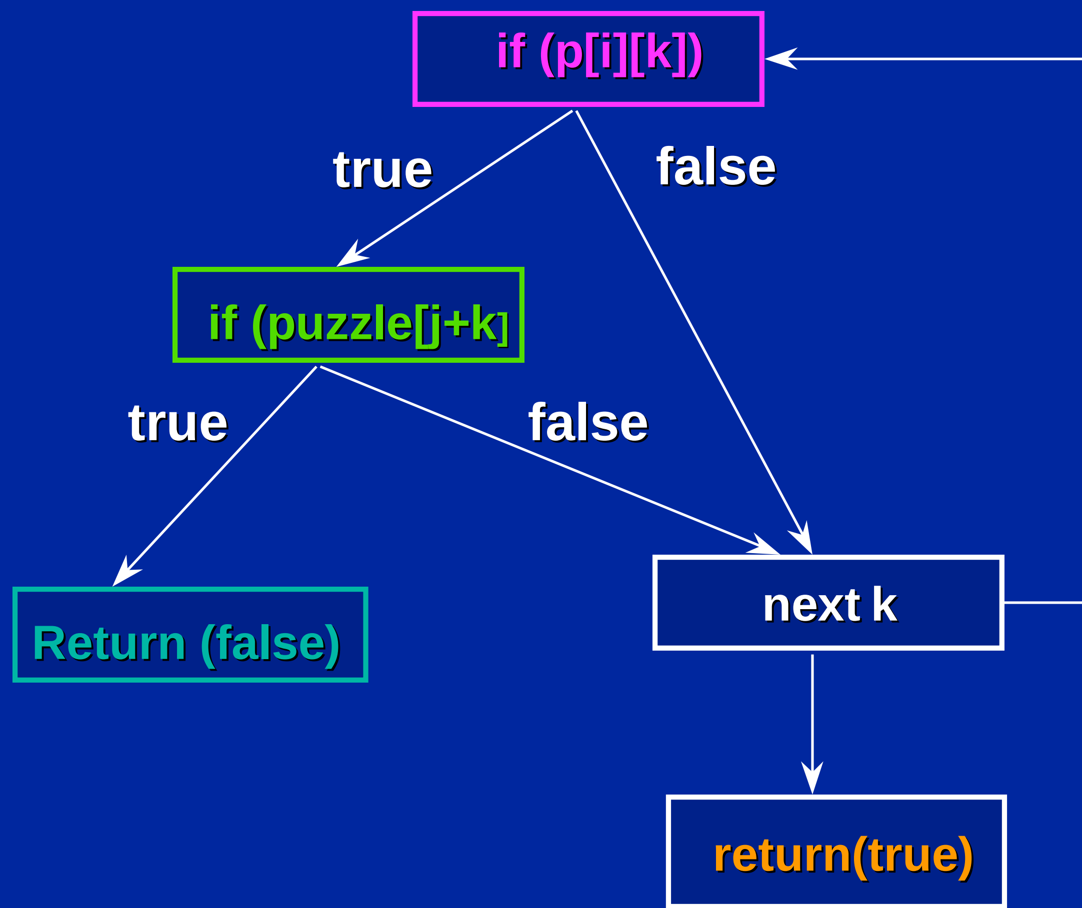
Expose ILP with predication in loop with nested if statements

intel®

Puzzle Code Fragment

```
int fit(i,j)
int i,j;
{
    int k;

    for (k = 0;
            k <= kmax;
            k++)
        if (p[i][k])
            if (puzzle[j+k])
                return (false);
return(true);
}
```

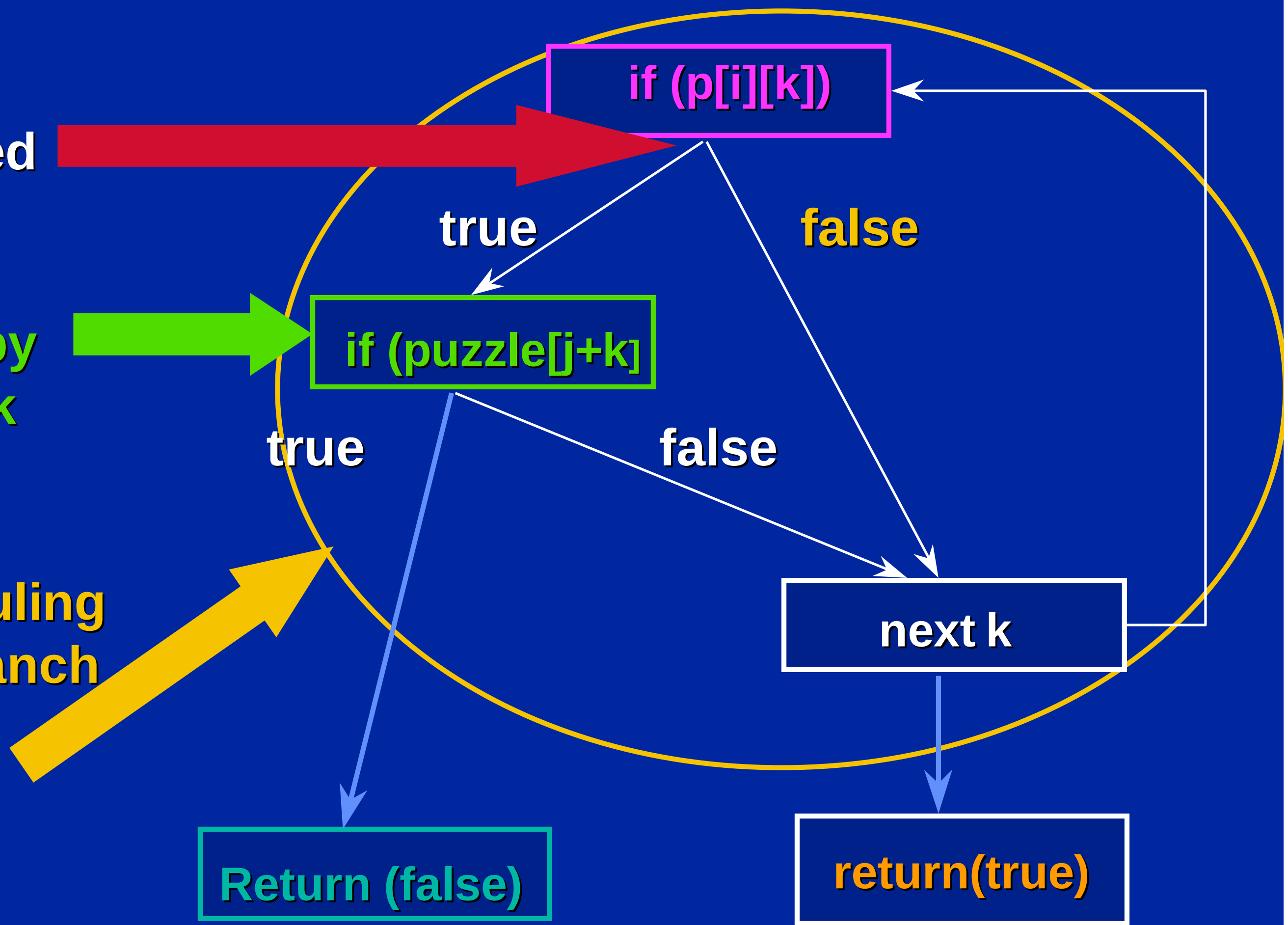


Hard to Predict Branch

Highly Mispredicted Branch

Remove the branch by predicating this block

New Compiler Scheduling Scope without the branch



Remaining Branches are easy to predict

Compiling Serial Code . . .

```
int fit(i,j)
int i,j;
{
    int k;

    for (k = 0;
             k <= kmax;
             k++)
        if (p[i][k])
            if (puzzle[j+k])
               return (false);
return(true);
}
```

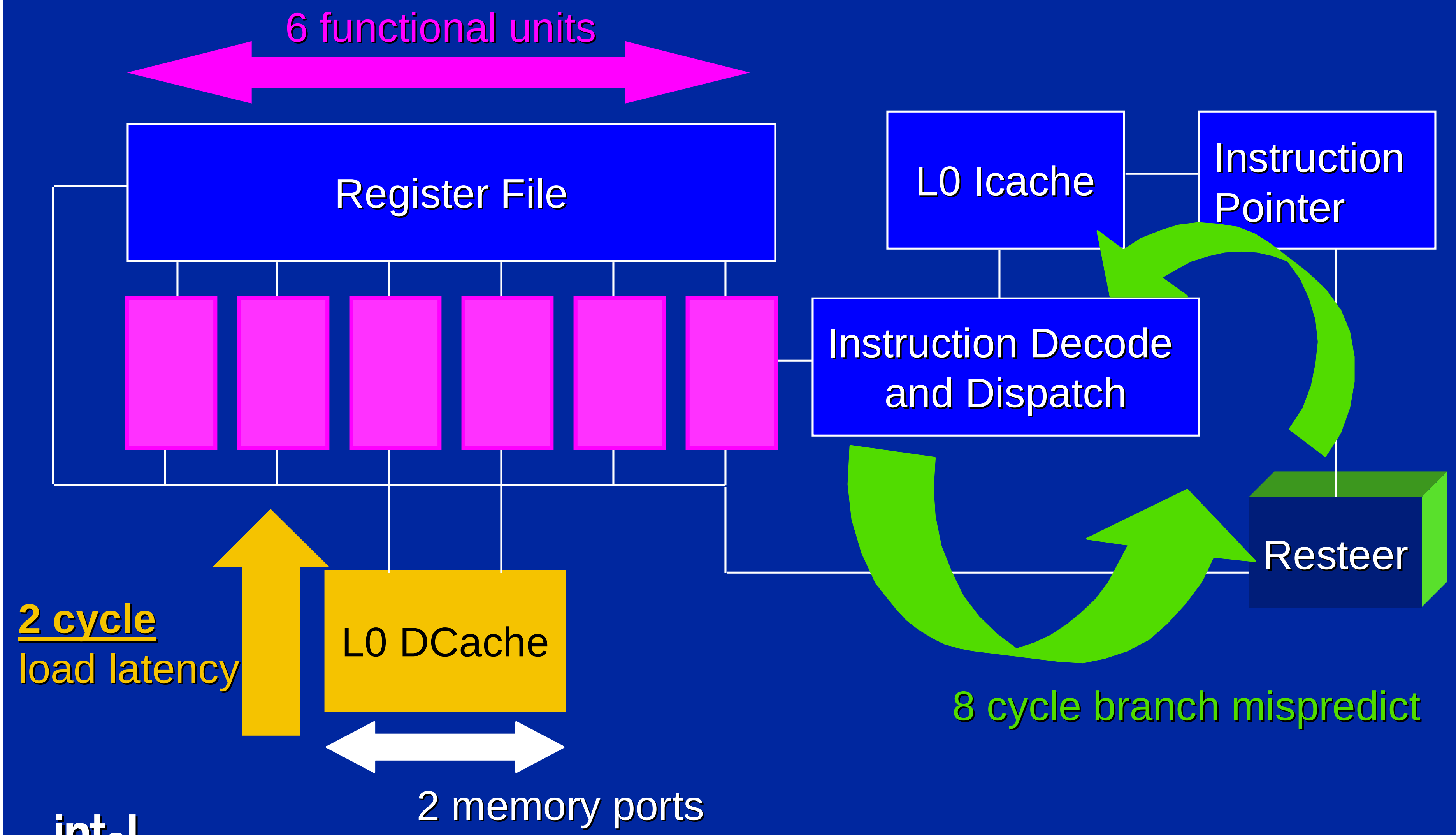
```
@p +=1
Load x= p[I][k]
cond1 = cmp x== 1
br to k+1 if !cond1
@puzzle += 1
Load y= puzzle[j+k]
cond2 = cmp y == 1
br to return cond2
k ++
cmp k<= kmax
br next_k
```

Highly mispredicted branch

Branch prediction incurs performance penalties.

intel®

Example Machine Model for Puzzle



6 Execution units, 2 memory ports, 2 cycle load latency

Compiling . . .

```
for (k = 0;  k <= kmax  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load x= p[I][k]		@p += 1			
2						
3	P1 = Cmp x==0					
4						

Compiling . . .

```
for (k = 0;  k <= kmax;  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load x= p[I][k]	Load y= puzzle[j+k]	@p +=1	@puzzle +=1		
2						
3	P1 = Cmp x==0					
4	<p1> p2=Cmp y==0				<p2>Br to return	

First Iteration

```
for (k = 0;  k <= kmax;  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load x= p[I][k]	Load y = puzzle[j+k]	@p +=1	@puzzle +=1		
2						
3	P1 = Cmp x==0	K++			~~<p1> br to k+1~~	
4	<p1> p2=Cmp y==0	P3=Cmp k<=kmax			<p2>Br to return	<p3>Br to next_k

Using predication to remove a hard to predict branch

Second Iteration Unrolling . . .

```
for (k = 0;  k <= kmax;  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load x= p[I][k]	Load y= puzzle[j+k]	@p +=1	@puzzle +=1		
2	Load.s x2=p[I][k]	Load.s y2= puzzle[k+k]				
3	P1 = Cmp x1==0	K++				
4	<p1> p2=Cmp y==0	P3=Cmp k <=kmax			<p2>Br to return	<p3> br to exit
5	Check.s	Check.s		P6 = cmp k<=kmax	<p5> br to return	<p6> br next_k

Using speculation to unroll the loop

Optimized Code

```
for (k = 0;  k <= kmax  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Load x= p[I][k]	Load y= puzzle[j+ k]	@p +=1	@puzzle +=1		
2	Load.s x2=p[I][k]	Load.s y2=puzzl e[j}k]	@p +=1	@puzzle +=1		
3	P1 = Cmp x==0	K++				
4	<p1> p2=Cmp y==0	P3=Cmp k<=kmax	P4 = cmp x2==0	K + =1	<p2>Br to return	<p3> br to exit
5	Check.s	Check.s	<p4> p5 = cmp y2==0	P6 = cmp k<=kmax	<p5> br to return	<p6> br next_k

Scheduled without Predication or Speculation

```
for (k = 0;  k <= kmax;  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	P3=Cmp k<=kmax	@p +=1	@puzzle +=1	Br to exit if p3		
2	K + +	Load x2=p[I][k]	Load y2=puzzle	P1=Cmp x==0	Br to 3 if p1	Loop to 1
3	p2=Cmp y==0				Br to return if p2	Loop to 1

A hard to predict branch cannot be avoided, mispredict penalties add execution time

intel®

Puzzle Code Fragment Conclusions

- Predication Benefits
  - remove difficult to predict branches
  - remove high penalty for branch mispredicts
- Control Speculation Benefits
  - unroll the loop and execute loads of next iteration
  - unroll as much as needed to fill wide machine
- Benefit over Out of Order Execution:
  - on average, predication can eliminate over 50% of branches

Combining Predication & Speculation results in a significant performance advantage

intel®

Comparing Bills

	With speculation and predication	Without speculation nor predication
Schedule for 2 iteration	5 cycles	5 cycles
		+
Branch Mispredict Penalty 25% mispredict rate		X 2 branches, 2 clocks — 4 cycles
	5 cycles	9 cycles

Scoreboard

	With speculation and predication	Without speculation or predication
xlxgetvalue	2 cycles per iteration	3 cycles per iteration
puzzle	2.5 cycles per iteration	4.5 cycles per iteration

IA-64 Architecture Innovations
Outline

- Background
- IA-64 at work: Code Examples
  - xlxgetvalue from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - treeins code fragment
    - classic if-then-else statement
- Summary

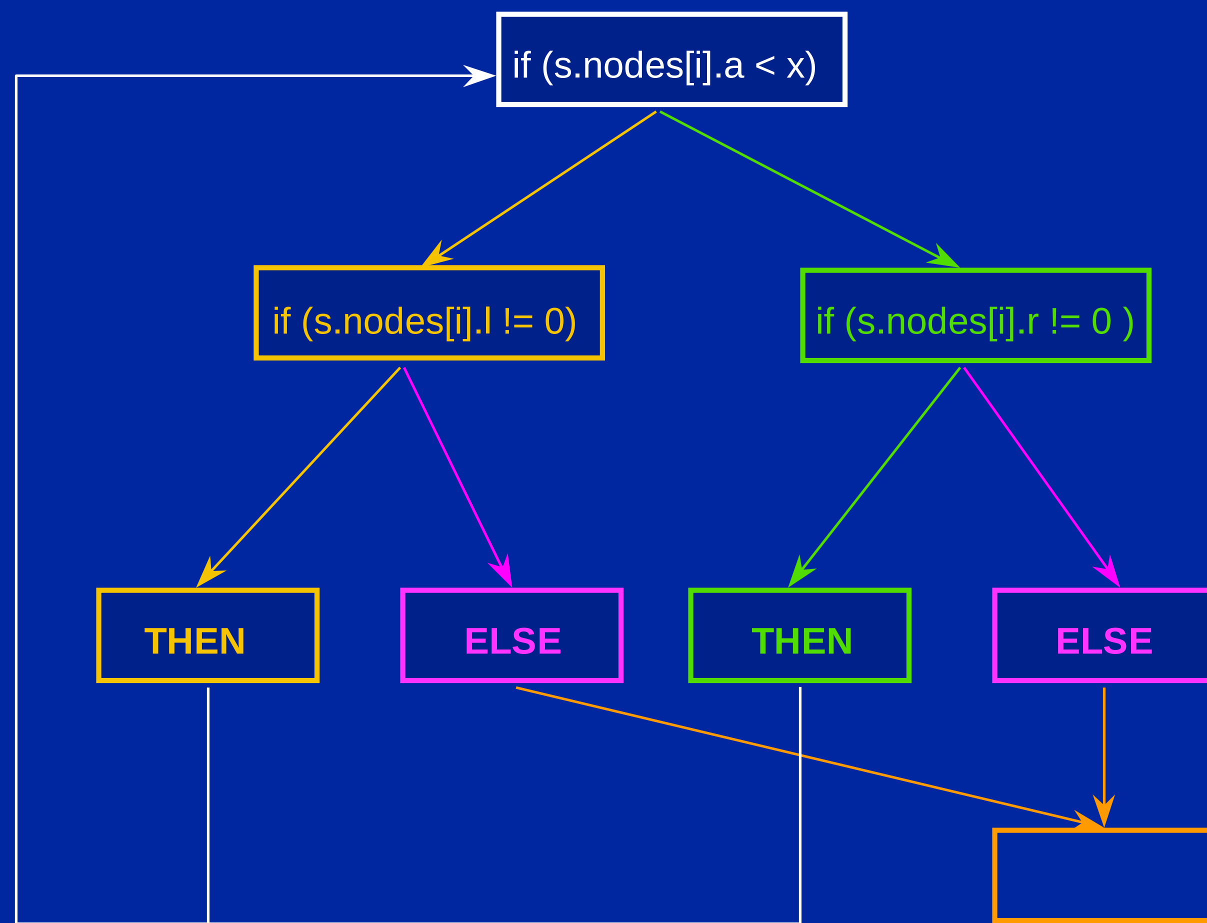
intel®

Treeins in a Nutshell

- Classic Code fragment
  - representative of if-then-else control structures
- Technique used: Predication
- Benefits:
  - remove hard to predict branches
  - execute all paths in parallel
  - expose Instruction Level Parallelism allowing parallel execution

Treeins Code Fragment

```
L10: /* compare */
      if (s.nodes[i].a < x)
         if (s.nodes[i].l != 0 )  {
            i = s.nodes[i].l;
            goto L10;}
         else {
            s.nodes[i].l = j;
            goto L20;}
      else{
         if (s.nodes[i].r != 0) {
            i = s.nodes[i].r;
            goto L10;}
         else{
            s.nodes[i].r = j;
            goto L20;}
       }
   L20:  /* insert */
         s.nodes[j].a = x;
         s.nodes[j].l = 0;
         s.nodes[j].r = 0;
         s.nxt_avail = j+1;
```

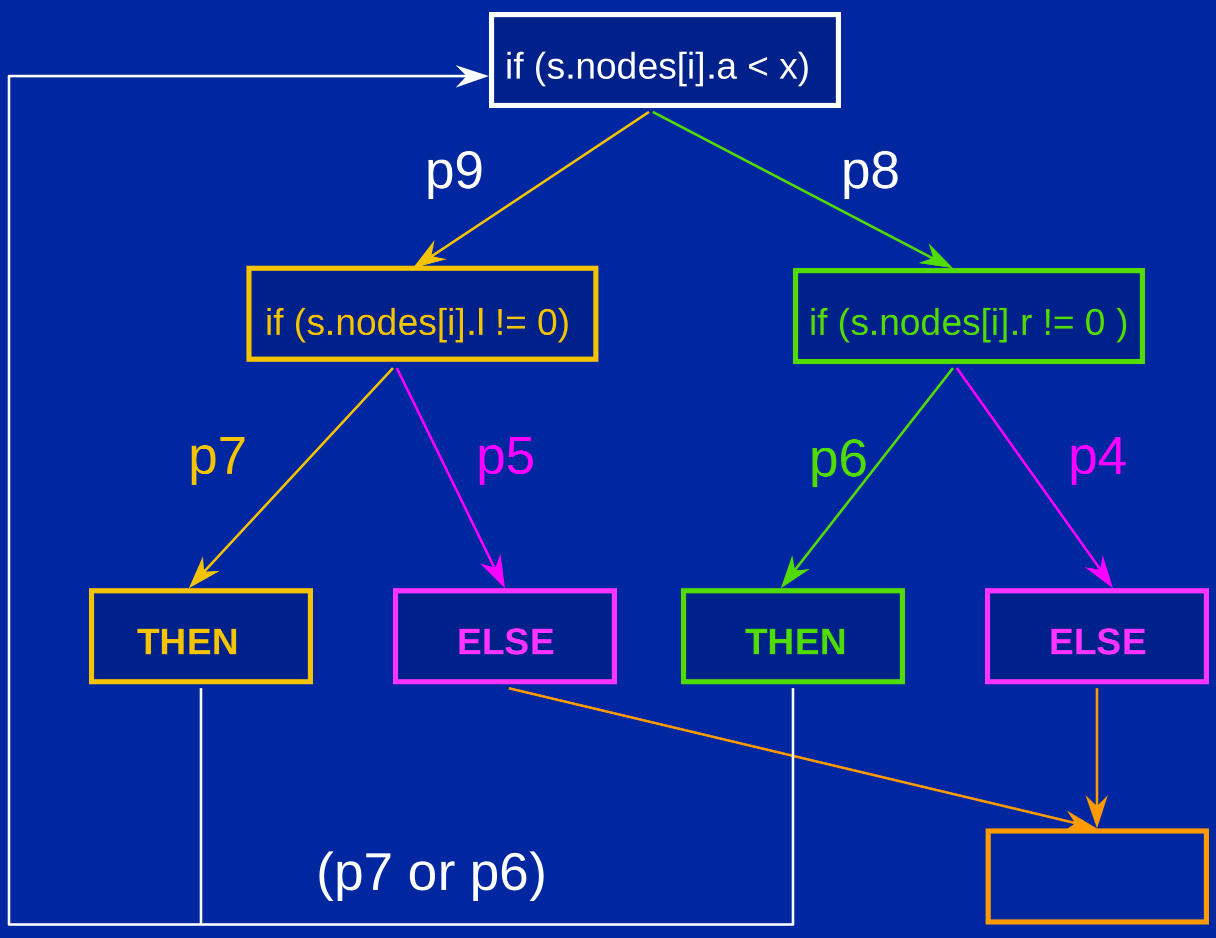


Treeins Step by Step

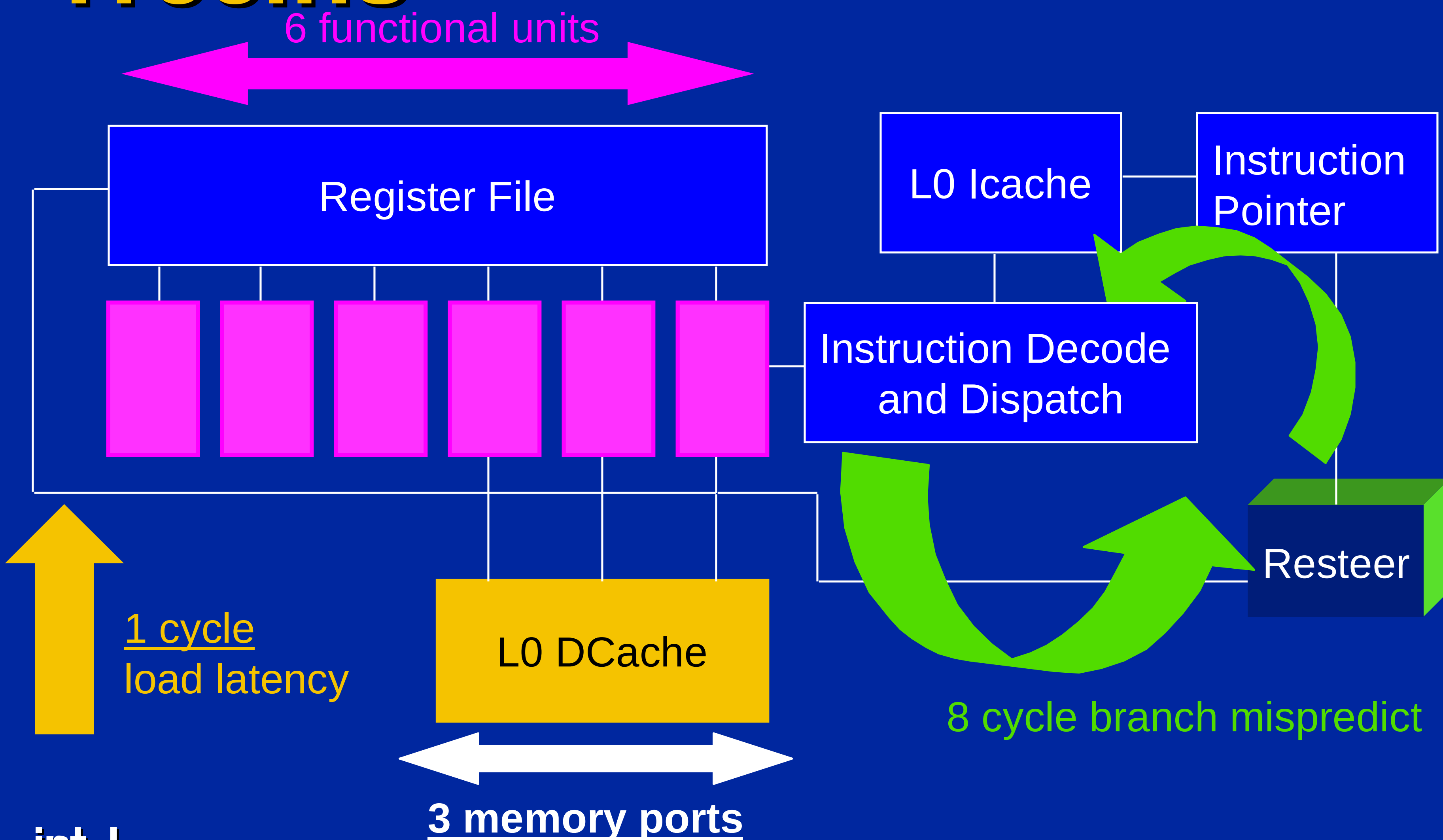
- Assign a predicate to each block
- Schedule the left most "then path"
- Schedule the second "then path"
- Schedule the "else paths" in parallel with the "then paths"

Predicate Usage

```
L10: /* compare */
      if (s.nodes[i].a < x)
         if (s.nodes[i].l != 0 )  {
            i = s.nodes[i].l;
            goto L10;}
         else {
            s.nodes[i].l = j;
            goto L20;}
      else{
         if (s.nodes[i].r != 0) {
            i = s.nodes[i].r;
            goto L10;}
         else{
            s.nodes[i].r = j;
            goto L20;}
       }
   L20:  /* insert */
         s.nodes[j].a = x;
         s.nodes[j].l = 0;
         s.nodes[j].r = 0;
         s.nxt_avail = j+1;
```

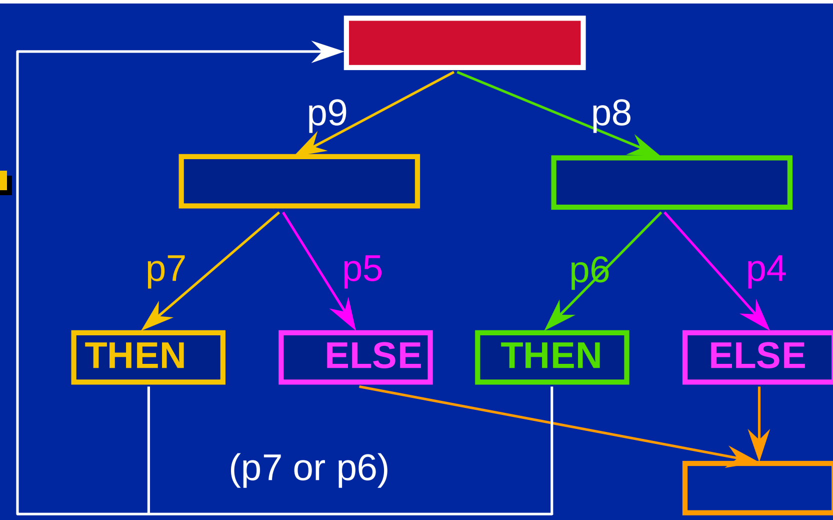


Example Machine model for Treeins



6 Execution units, 3 memory ports, 1 cycle load latency

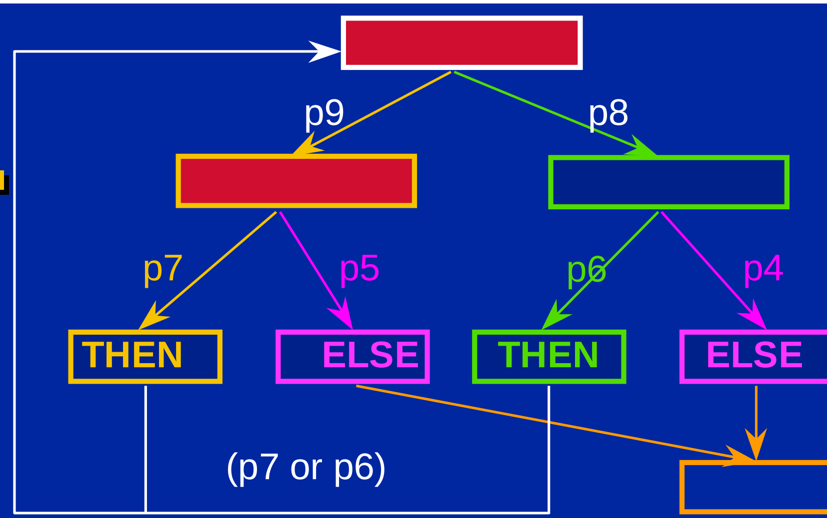
Compiling . . .




```
L10: /* compare */
    if (s.nodes[i].a < x)
      if (s.nodes[i].l != 0 )  {
        i = s.nodes[i].l;
        goto L10;}
      else {
        s.nodes[i].l = j;
        goto L20;}
    else{
      if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;}
      else{
        s.nodes[i].r = j;
        goto L20;}
    }
  L20:  /* insert */
      s.nodes[j].a = x;
      s.nodes[j].l = 0;
      s.nodes[j].r = 0;
      s.nxt_avail = j+1;
```

	Unit 1	Unit 2	Unit 3	Unit 4
1	shladd			
2	Shladd			
3				
4				
5				
6				
7				

Compiling . . .

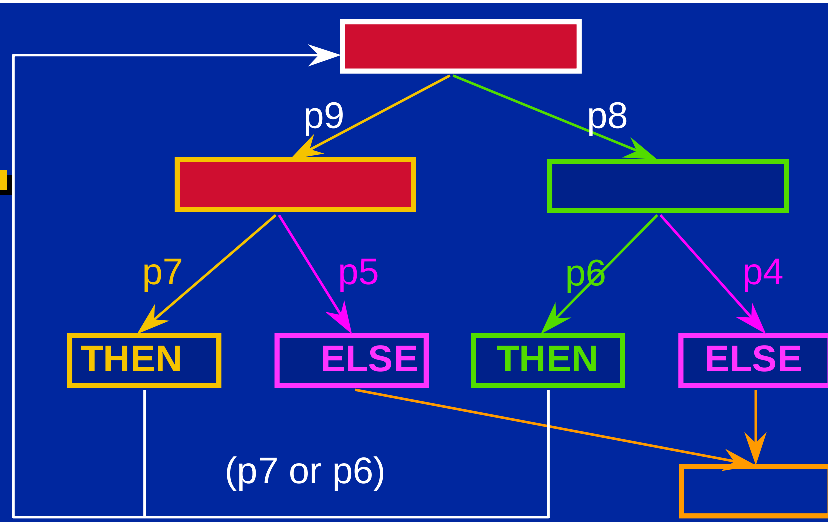



```
L10: /* compare */
    if (s.nodes[i].a < x)
      if (s.nodes[i].l != 0 )  {
        i = s.nodes[i].l;
        goto L10;}
      else {
        s.nodes[i].l = j;
        goto L20;}
    else{
      if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;}
      else{
        s.nodes[i].r = j;
        goto L20;}
    }
  L20:  /* insert */
      s.nodes[j].a = x;
      s.nodes[j].l = 0;
      s.nodes[j].r = 0;
      s.nxt_avail = j+1;
```

	Unit 1	Unit 2	Unit 3	Unit 4
1	shladd			
2	Shladd			
3	add	add		
4	Ld a	Ld l		
5				
6				
7				

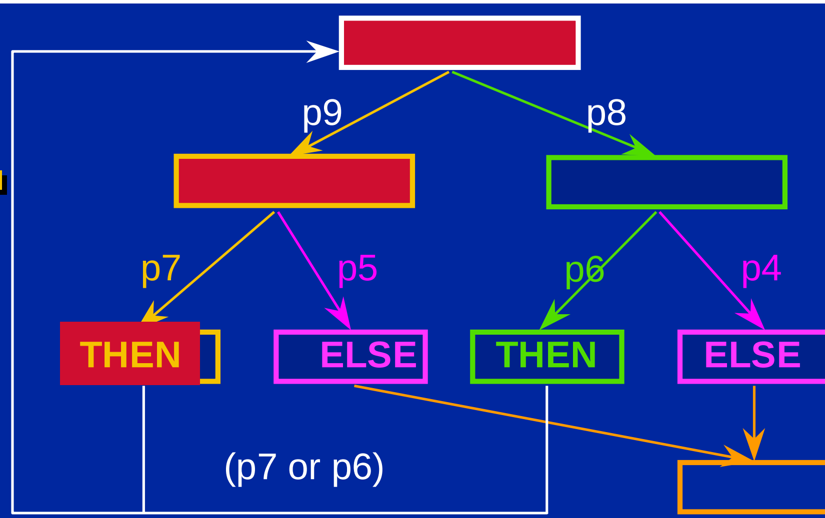
Compiling . . .

```
L10: /* compare */
    if (s.nodes[i].a < x)
      if (s.nodes[i].l != 0 )  {
        i = s.nodes[i].l;
        goto L10;}
      else {
        s.nodes[i].l = j;
        goto L20;}
    else{
      if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;}
      else{
        s.nodes[i].r = j;
        goto L20;}
     }
  L20:  /* insert */
      s.nodes[j].a = x;
      s.nodes[j].l = 0;
      s.nodes[j].r = 0;
      s.nxt_avail = j+1;
```



	Unit 1	Unit 2	Unit 3	Unit 4
1	shladd			
2	Shladd			
3	add	add		
4	Ld a	Ld l		
5	Cmp p9, p8= a<x			
6	<p9> cmp p7,p5= l!=0			
7				

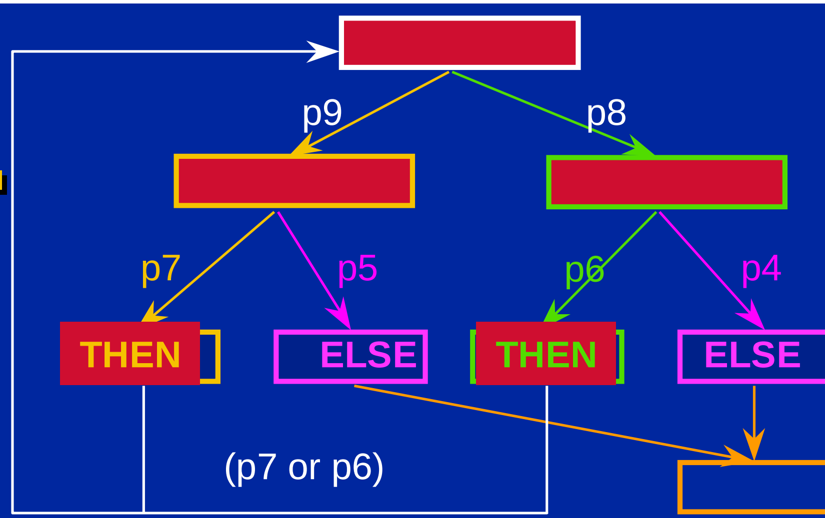
Compiling . . .




```
L10: /* compare */
   if (s.nodes[i].a < x)
     if (s.nodes[i].l != 0 )  {
       i = s.nodes[i].l;
       goto L10;}
     else {
       s.nodes[i].l = j;
       goto L20;}
   else{
     if (s.nodes[i].r != 0) {
       i = s.nodes[i].r;
       goto L10;}
     else{
       s.nodes[i].r = j;
       goto L20;}
   }
 L20:  /* insert */
     s.nodes[j].a = x;
     s.nodes[j].l = 0;
     s.nodes[j].r = 0;
     s.nxt_avail = j+1;
```

	Unit 1	Unit 2	Unit 3	Unit 4
1	shladd			
2	Shladd			
3	add	add		
4	Ld a	Ld l		
5	Cmp p9, p8= a<x			
6	<p9> cmp p7,p5= l!=0			
7	<p7> mov			

Compiling . . .




```
L10: /* compare */
    if (s.nodes[i].a < x)
      if (s.nodes[i].l != 0 )  {
        i = s.nodes[i].l;
        goto L10;}
      else {
        s.nodes[i].l = j;
        goto L20;}
    else{
      if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;}
      else{
        s.nodes[i].r = j;
        goto L20;}
    }
  L20:  /* insert */
      s.nodes[j].a = x;
      s.nodes[j].l = 0;
      s.nodes[j].r = 0;
      s.nxt_avail = j+1;
```

	Unit 1	Unit 2	Unit 3	Unit 4
1	shladd			
2	Shladd			
3	add	add		
4	Ld a	Ld l	Ld r	
5	Cmp p9, p8= a<x			
6	<p9> cmp p7,p5= l!=0	<p8> cmp p6, p4 r!=0		
7	<p7> mov	<p6> mov		

Compiling the “then” paths

```
if (s.nodes[i].a < x)
    if (s.nodes[i].l != 0 ) {
      i = s.nodes[i].l;
      goto L10;}
else{
    if (s.nodes[i].r != 0) {
      i = s.nodes[i].r;
      goto L10; }
```

	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	shladd					
2	Shladd					
3	add	add				
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9> cmp p7,p5= l!=0	<p8> cmp p6, p4 r!=0				
7	<p7> mov i=l	<p6> mov i=r	<p7> branch next_loop	<p6> branch next_loop		

intel®

Control flow is now scheduled

Compiling the “then” paths

if (s.nodes[i].a < x)
    if (s.nodes[i].l != 0) {
      i = s.nodes[i].l;
      goto L10;}

else{
    if (s.nodes[i].r != 0) {
      i = s.nodes[i].r;
      goto L10; }

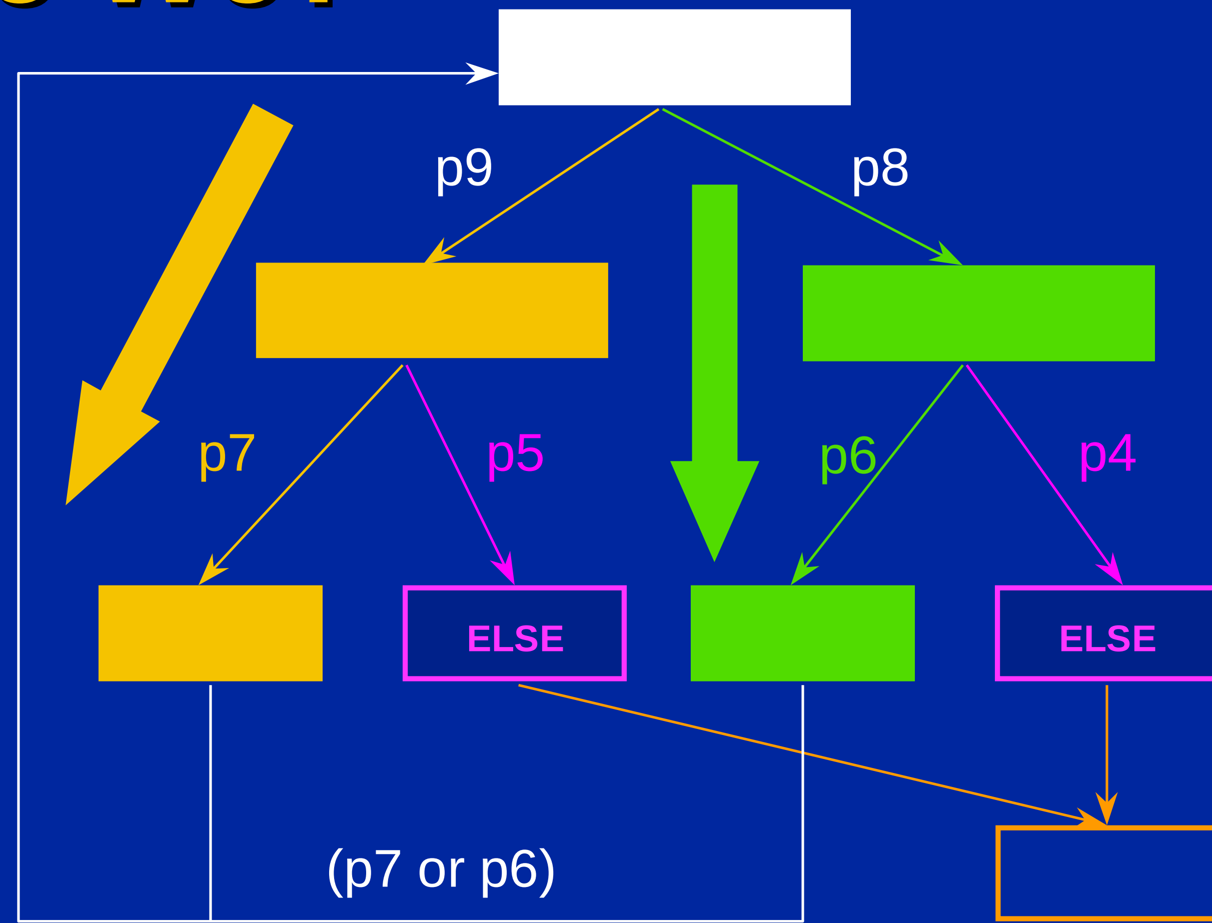
	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	shladd					
2	Shladd					
3	add	add				
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9> cmp p7,p5= l!=0	<p8> cmp p6, p4 r!=0	<p9> mov i=l	<p8> mov i=r	<p6> branch next_loop	<p7> branch next_loop
7						

intel®

Using predicate promotion to shorten the loop path

Where Are We?

```
L10: /* compare */
      if (s.nodes[i].a < x)
         if (s.nodes[i].l != 0 )  {
            i = s.nodes[i].l;
            goto L10;}
         else {
            s.nodes[i].l = j;
            goto L20;}
      else{
         if (s.nodes[i].r != 0) {
            i = s.nodes[i].r;
            goto L10;}
         else{
            s.nodes[i].r = j;
            goto L20;}
       }
   L20:  /* insert */
         s.nodes[j].a = x;
         s.nodes[j].l = 0;
         s.nodes[j].r = 0;
         s.nxt_avail = j+1;
```



Then paths are scheduled
Else paths are not

Compiling the “else” paths

```
else {
        s.nodes[i].l = j;
        goto L20;}
```

```
else{
        s.nodes[i].r = j;
        goto L20;}
```

	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	shladd					
2	Shladd					
3	add	add				
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9> cmp p7,p5= l!=0	<p8> cmp p6, p4=r!=0	<p9> mov i=l	<p8> mov i=r	<p7> branch next_loop	<p6> branch next_loop
7	<p5> store l	<p4> store r				

Compiling . . .

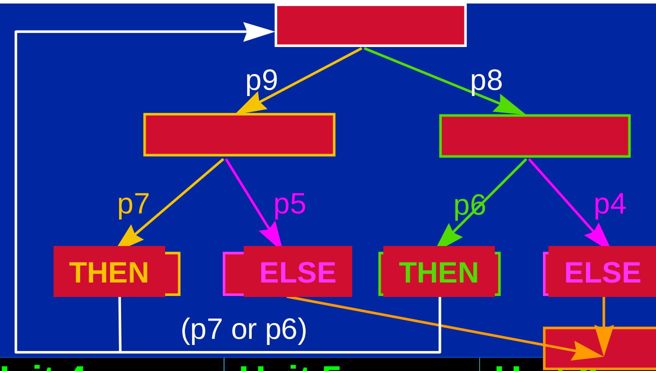
```
L20:  /* insert */
      s.nodes[j].a = x;
      s.nodes[j].l = 0;
      s.nodes[j].r = 0;
      s.nxt_avail = j+1;
```

	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	shladd		shladd			
2	Shladd		shladd			
3	add	add	add	add	add	add
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9> cmp p7,p5= l!-0	<p8> cmp p6, p4 r!=0	<p9> mov i=l	<p8> mov i=r	<p7> branch next_loop	<p6> branch next_loop
7	<p5> store [l].l	<p4> store [l].r	Store [j].a			
8	Store [j].l	Store [j]r	Store nxt_avail			

intel®

Scheduling the else path does not slow the then paths

Treeins Schedule



	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	shladd		shladd			
2	Shladd		shladd			
3	add	add	add	add	add	add
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9> cmp p7,p5= l!-0	<p8> cmp p6, p4 r!=0	<p9> mov i=l	<p8> mov i=r	<p7> branch next_loop	<p6> branch next_loop
7	<p5> store [l].l	<p4> store [l].r	Store [j].a			
8	Store [j].l	Store [j]r	Store nxt_avail			

All paths scheduled in parallel

Scheduled Without Predication

2 difficult to predict branches

Separate paths

	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	shladd		shladd			
2	Shladd		shladd			
3	add	add	add	add	add	add
4	ld	ld	ld			
5	Cmp p9, p8=			BRANCH		
6	<p9> cmp p7,p5=	<p8> cmp p6, p4	<p9> mov	<p8> mov	<p7> branch next_loop	BRANCH
7	<p5> store	<p4> store	store			
8	store	store	store			

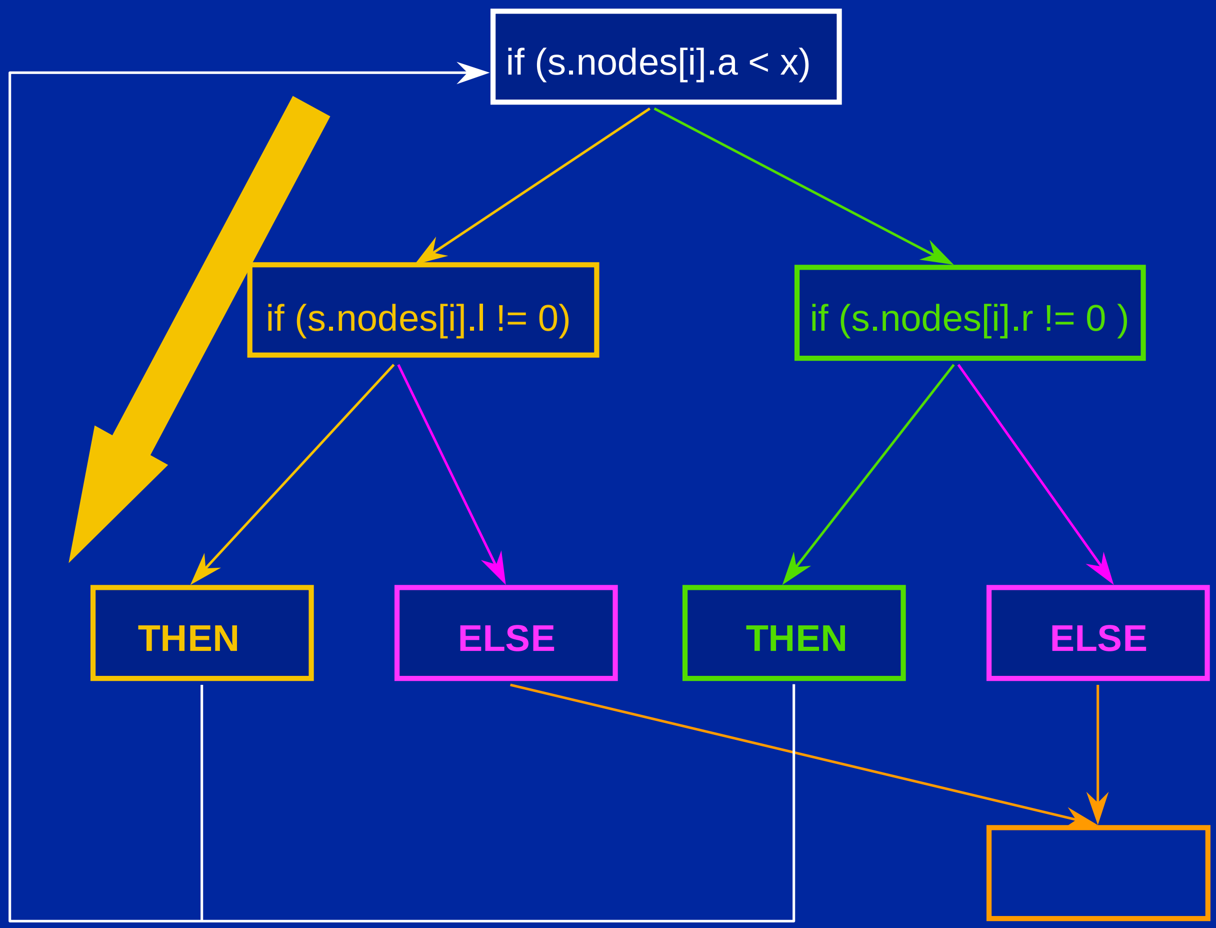
Separate paths

OOO Must Choose One Path

Branches are predicted.

Instructions are speculated along one path.

All instructions are thrown away if branches are mispredicted



Mispredicts have significant performance penalties

intel®

Treeins Code Fragment Conclusions

- IA-64 processor executes instructions along 4 control flow paths in parallel
  - advantage over Out of Order Execution which can only follow one path
  - take advantage of a wide machine

IA-64 architecture enables significant performance advantages

intel®

Comparing Bills

	With speculation and predication	Without speculation nor predication
Schedule	6 or 8 cycles	7 cycles
Branch Mispredict Penalty 25% mispredict rate		+ 2 branches X 2 clocks (4 cycles)
	7 cycles	11 cycles

intel®

Scoreboard

	With speculation and predication	Without speculation nor predication
xlxgetvalue	2 cycles per iteration	3 cycles per iteration
puzzle	2.5 cycles per iteration	4.5 cycles per iteration
treeins	7 cycles per iteration	11 cycles per iteration

IA-64 at Work Summary

	Speculation and predication benefits
Pointer chasing	Execute loads before pointer safety check
Loop with conditional	Unroll loop, remove hard to predict branch
If-then-else	Execute all paths in parallel Eliminate branch mispredict penalties

IA-64 Architecture Innovations
Outline

- Background
- IA-64 at work: Code Examples
- Summary
  - What IA-64 architecture means for developers
    - New levels of performance
    - Wider machines: better utilization and inherent scalability

IA-64 Architecture Benefits Summary

- Explicit parallelism leads to greater ILP
- Efficient use of large resources
- Benefits over out-of-order execution
  - Exposes greater parallelism through compiler's larger scheduling scope
  - Executes multiple paths in parallel by removing branches & mispredicts
  - Enables parallel execution by minimizing memory latency impact

IA-64: Enabling Industry Leading Performance for Server and Workstation Workloads