Advanced Techniques for Avoiding Misprediction

Branch Mispredictions Limit ILP

Advanced superscalar techniques

- Fast decoding
- Branch prediction
- Cache prefetch

Long speculative instruction sequences in cache

- Misprediction $\Rightarrow$ long recovery process
  - Cancel large block of executed instructions
  - Evict and replace many instruction cache lines
  - Decode/reschedule many instructions for instruction pool

Misprediction $\rightarrow$ predict-prefetch-decode bottleneck

- Predict-prefetch-decode
  - Stop advance
  - Switch to recovery
- EUs starve (stall)
- Instruction pool fill disrupted

Trace Cache

Trace

- Long instruction sequence in dynamic execution order
  - Multiple branch actions (taken/not-taken)
  - Dynamic scheduling (actual execution order in CPU)

Trace cache

- Cache of instruction traces
- Constructed dynamically on first run of trace
  - Fetch → branch predict → decode → reschedule → execute
  - Instructions come from multiple instruction cache lines

Rapid recovery

- Misprediction → hit in trace cache
  - Copy trace to instruction pool from cache
  - Skips fetch → branch predict → decode → reschedule

Trace Cache Construction

Trace generation

- Load cache lines → fetch instructions
- Predict branches
- Decode instructions → instruction pool
- Instructions dynamically scheduled for execution in EUs

Trace cache entry

- Start on target or fall-through of branch
- Trace embeds next few branch actions
  - Validated taken / not-taken execution paths

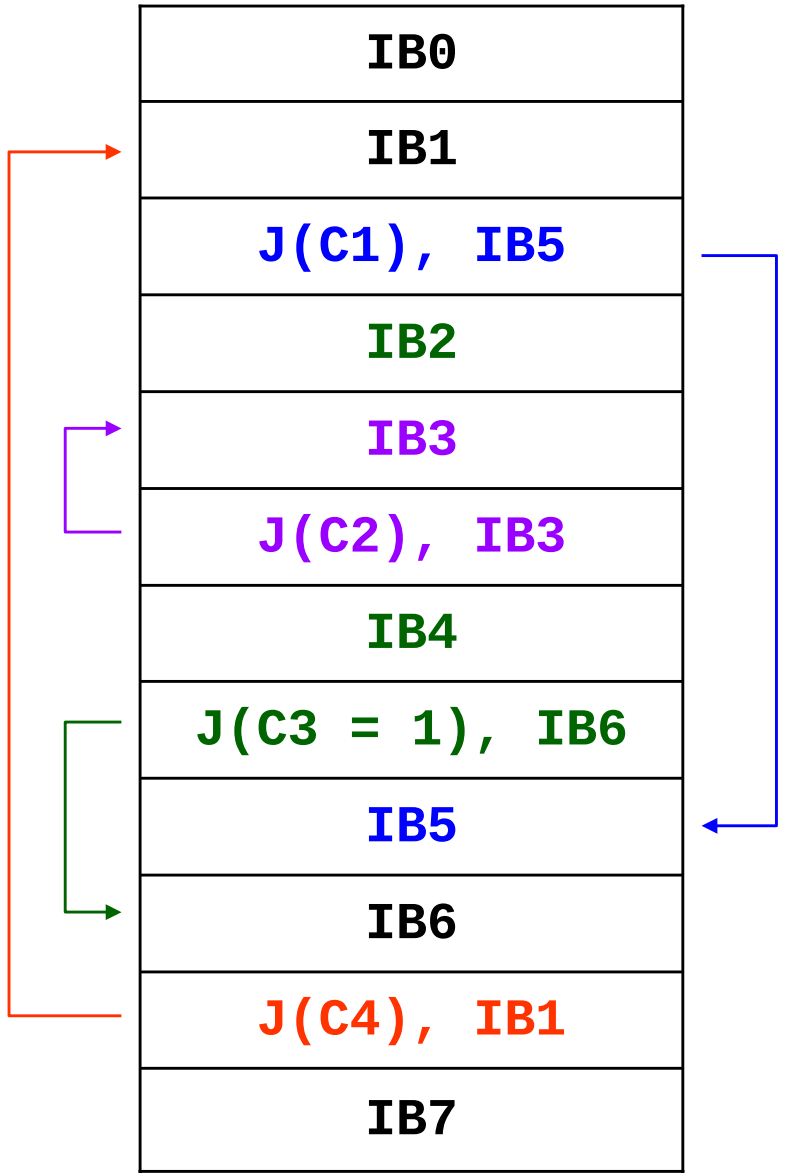
Trace cache entry ID

- PC of first instruction
- Sequence of taken / not-taken actions embedded in trace

Program Fragment

IB = instruction block

```
IB0(...);
while (C4) {
      IB1(...);
      if (C1)
         IB5(...);
      else {
         IB2(...);
         while (C2) IB3(...);
         IB4(...);
      }
      IB6(...);
}
IB7(...);
```



Trace Construction Example — 1

Program fragment

32 instructions

$I_0 - I_{31}$

3 conditional branches

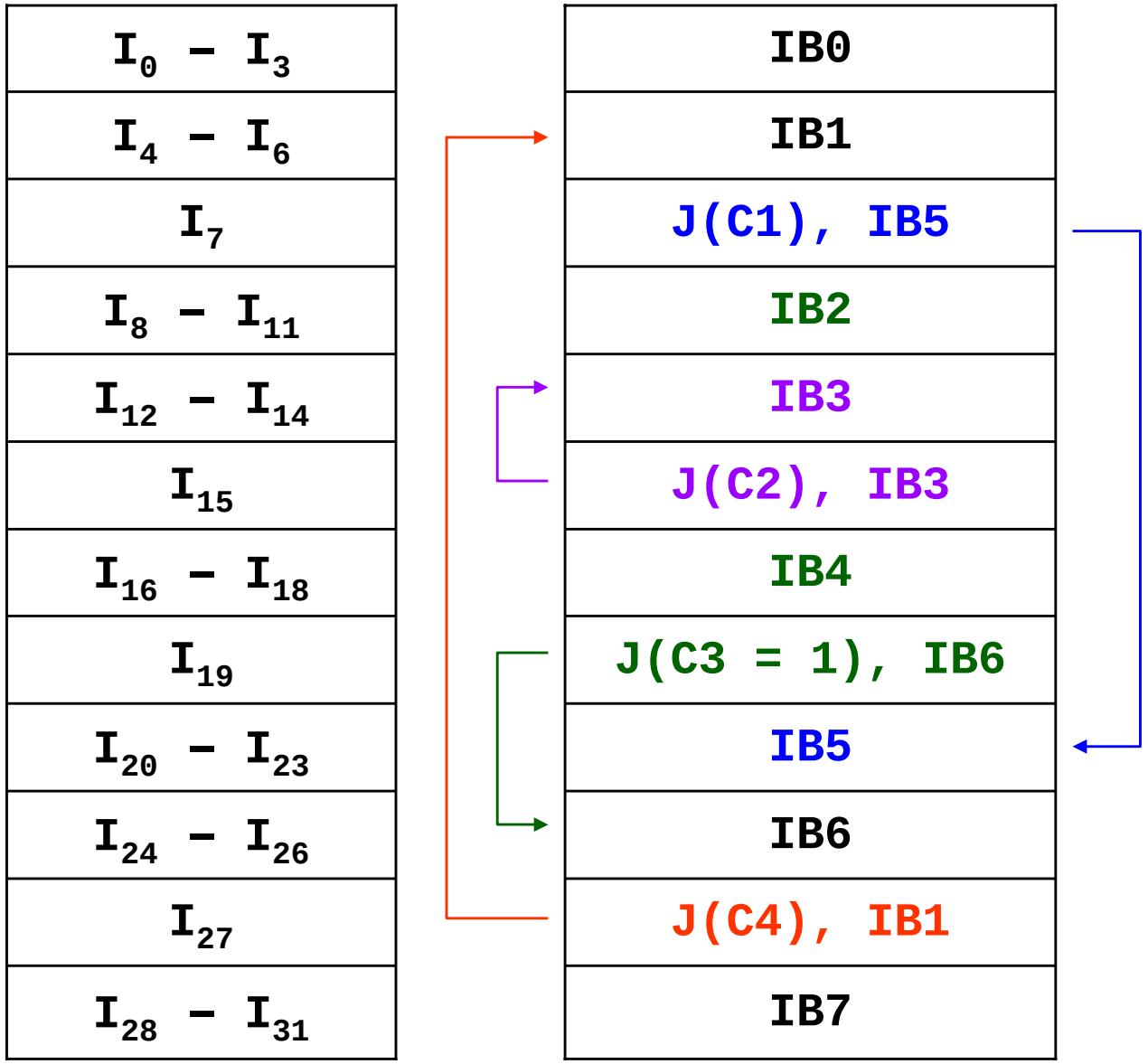
I_7 , I_{15} , I_{27}

1 absolute branch

I_{19}

Trace cache

8 instructions

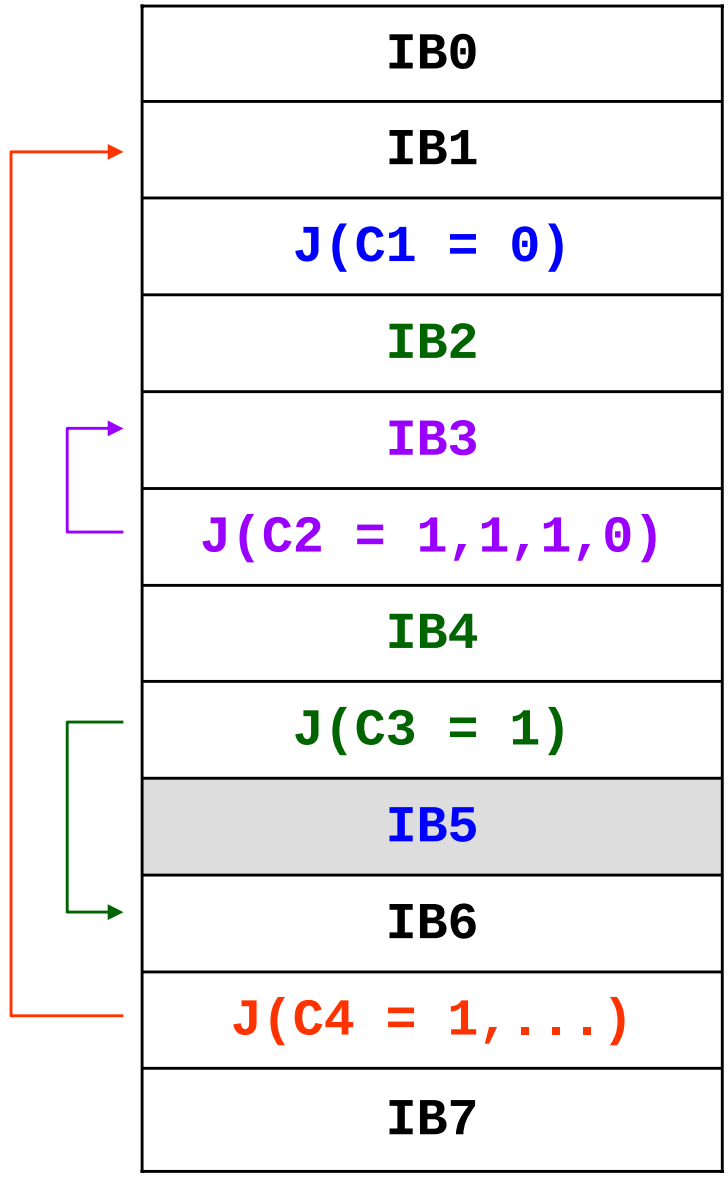


Trace Construction Example — 2

Runtime sequence

IB_0
IB_1
J(C1 = 0)
IB_2
IB_3
J(C2 = 1)
IB_3
J(C2 = 1)
IB_3
J(C2 = 1)
IB_3
J(C2 = 0)
IB_4
J(C3 = 1)
IB_6
J(C4 = 1)
IB_1 ...

I_0 – I_3
I_4 – I_6
I_7
I_8 – I_{11}
I_{12} – I_{14}
I_{15}
I_{16} – I_{18}
I_{19}
I_{20} – I_{23}
I_{24} – I_{26}
I_{27}
I_{28} – I_{31}



Trace Construction Example — 3

Runtime sequence

IB_0	I_0 – I_3	
IB_1	I_4 – I_6	T4
J(C1 = 0)	I_7	
IB_2	I_8 – I_{11}	T1
IB_3	I_{12} – I_{14}	
J(C2 = 1)	I_{15}	
IB_3	I_{12} – I_{14}	T2
J(C2 = 1)	I_{15}	
IB_3	I_{12} – I_{14}	
J(C2 = 1)	I_{15}	
IB_3	I_{12} – I_{14}	T3
J(C2 = 0)	I_{15}	
IB_4	I_{16} – I_{18}	
J(C3 = 1)	I_{19}	
IB_6	I_{24} – I_{26}	T4
J(C4 = 1)	I_{27}	

First Traces in Cache

(start after first branch)

T1	8	9	10	11	12	13	14	15
T2	12	13	14	15	12	13	14	15
T3	12	13	14	15	16	17	18	19
T4	24	25	26	27	4	5	6	7

Typical iteration of main loop

T1 T2 T3 T4

Initial misprediction

Fourth J(C2)

Predict C2 = 1 ⇒ T2

Validate C2 = 0

Cancel T2

Load T3

Trace Cache Header

header						trace instructions			
start PC	flag	mask	end type	fall through	target			…	
PC-1	0111	1111	conditional	PC-2	PC-3			…	
PC-2	110x	1110	absolute	—	PC-4			…	
PC-3	0xxx	1000	no branch	PC-4	—				

Start PC

Address of 1st instruction in trace

Low bits = index to trace cache

High bits = tag

Flag

Sequence of branch actions in trace

0/1 = not taken / not taken

x = not valid (don't care)

Mask

1 = valid flag bit in position

Sum of mask = branches in trace

End type

Type of last instruction in trace

Conditional branch

Absolute branch

No branch

Fall-through / target

PC address for next instruction fetch

Points to trace or instruction

Target for flag = 1 (taken)

Fall-through for flag = 0 (not-taken)

Trace Cache Header Example

header						trace instructions			
start PC	flag	mask	end type	fall through	target			…	
PC-1	0111	1111	Conditional (C)	PC-2	PC-3			…	
PC-2	110x	1110	Absolute (A)	—	PC-4			…	
PC-3	0xxx	1000	no branch (N)	PC-4	—				

	start PC	flag	mask	end type	fall through	target	trace instructions							
T1	**8**	**10**	**1x**	**C**	**16**	**12**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
T2	**12**	**11**	**11**	**C**	**16**	**12**	**12**	**13**	**14**	**15**	**12**	**13**	**14**	**15**
T3	**12**	**01**	**11**	**A**	**—**	**24**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
T4	**24**	**10**	**11**	**C**	**8**	**20**	**24**	**25**	**26**	**27**	**4**	**5**	**6**	**7**

Trace Cache Line Length

Trace length = instruction cache line length

Longer ⇒ multiple L1 accesses ⇒ higher miss rate

Shorter ⇒ blocks poorly with L1 ⇒ inefficient trace

Maximum branches per trace

Limit = branch predictor throughput

Predictions per CC

Terminate trace at subroutine returns

Return address varies ⇒ correct embedding difficult

Trace termination

Fill trace to end with NOPs (not scheduled in instruction pool)

Example

0	1	2	…	E	F
Trace length = cache line length = 16 bytes					

Instructions				Terminated			
0	1	…	B	NOP	NOP	NOP	NOP

Instruction Fetch Unit

Multiple Branch Predictor (MBP)

Predicts PC for traces

Access to trace cache and L1 instruction cache in parallel

PC misses in trace cache

Fetch+decode from L1 cache to instruction pool

Next PC logic predicts next branch action

Target predictor → next PC

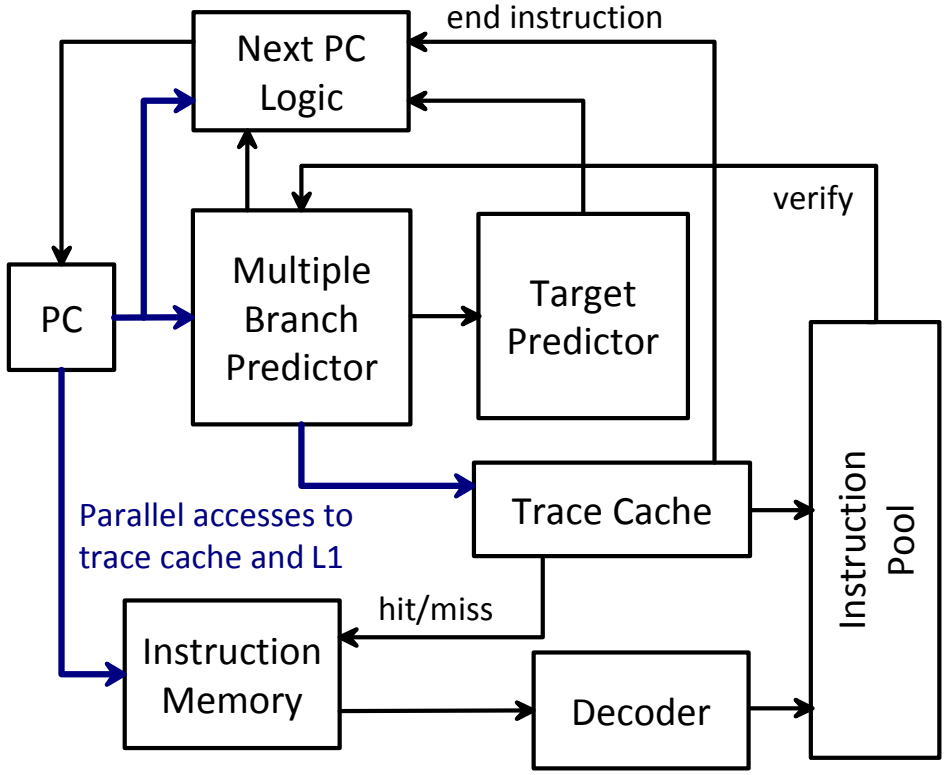
PC hits in trace cache

Cancel L1 access

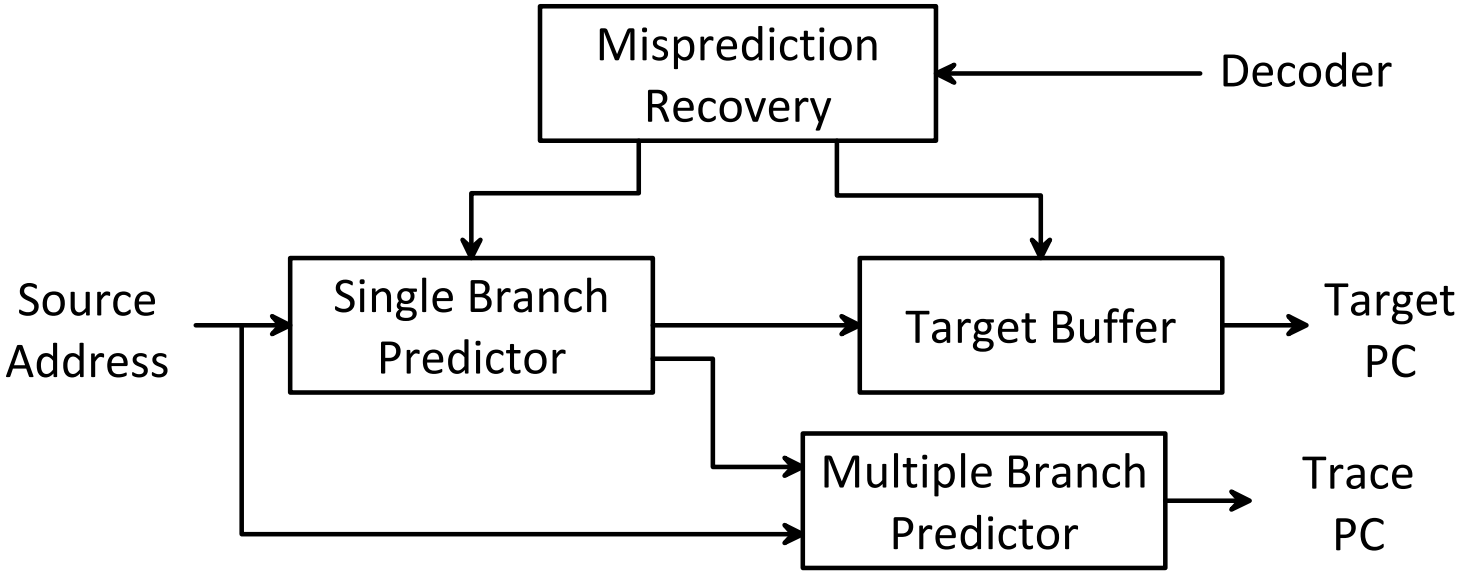
Fetch trace to instruction pool

MBP predicts embedded branches

Target predictor not used



Multiple Branch Predictor



Single branch predictor

Standard taken / not taken prediction (1-level or 2- level)

Multiple branch predictor

Predicts m embedded branches in trace

m-bit bitmap (0 = NT or 1 = T)

Bit = last prediction for branch from Single Branch Predictor

Multiple Branch Predictor Example

Instruction sequences

T1	8	9	10	11	12	13	14	15
T2	12	13	14	15	12	13	14	15
T3	12	13	14	15	16	17	18	19
T4	24	25	26	27	4	5	6	7

Single Branch History Register

	profile				target
I_7	0	0	0	0	I_{20}
I_{15}	1	1	1	0	I_{12}
I_{19}	1	1	1	1	I_{24}
I_{27}	1	1	1	1	I_4

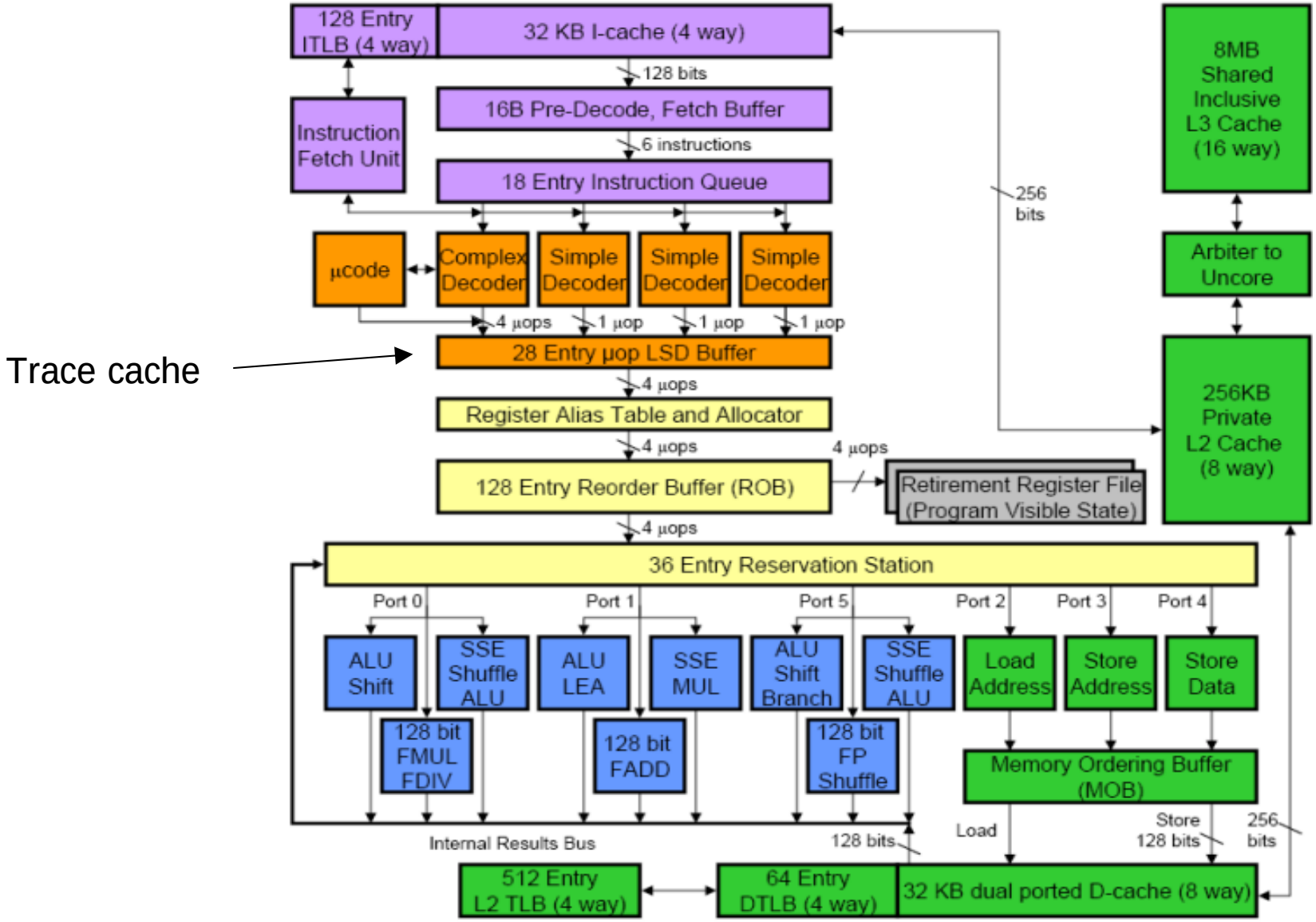
Single Branch Pattern Table

profile	pattern	
0000	0	0
1111	1	1
1110	1	1
1101	1	1
1011	1	1
0111	0	0

Multiple Branch Predictor (MBP)

Target	Trace	Predictions		Branch
I_8	T1	1	–	15
I_{12}	T2	1	1	15-15 (for profile != 0111)
	T3	0	1	15-19 (for profile == 0111)
I_{24}	T4	1	0	27-7

Intel Nehalem Micro-Architecture



David Kanter, "Inside Nehalem: Intel's Future Processor and System",
http://realworldtech.com/includes/templates/articles.cfm?ArticleID=RWT040208182719&mode=print

Branch Predication

Another Look at Branches

Two basic branch types

Branch type	loop	if-condition
Branch direction	Jump backward target PC < fall-through PC	Jump forward target PC > fall-through PC
Control type	code reuse on data streams	forward path of program flow
Predictability	high repetitive iteration pattern	low exotic data dependent statistics
Branch density	long loops ⇒ low density	high run 20% to 25% of instructions
Branch prediction and trace caching	good performance	poor performance expensive to implement 8 parallel instructions ⇒ 2 branches per clock cycle difficult to update branch history

Predication

Predication

Predication registers **`p0, p1, …, pN`**

`<p1> Instruction`

`p1 = 1` ⇒ CPU executes **`Instruction`**

`p1 = 0` ⇒ CPU executes **`NOP`**

Generalization of **`compare`** → **`conditional branch`** pairs

Alternative to if-condition branches

`Test_and_set` instructions for predication registers

Predicate standard instruction set (if-conversion)

Trade-off — eliminate branch stall at cost of canceling extra work

Example

```
    BNEZ R1, L1
    SUBI R1, R1, #1
L1: ADD R2, R2, R1
```

```
     p1 = CMP R1, 0
<p1> SUBI R1, R1, #1
     ADD R2, R2, R1
```

Predication in IA-64

IA-64 (Intel 64-bit architecture)

ISA for Intel Itanium series processors

Static predicate registers `p0` — `p15`

1-bit width written by compare instructions

`Regs[p0]` = **`1`**

Rotating predicate registers

48 renamed virtual copies of **`p0`** — **`p15`**

Allocated dynamically by hardware for parallel loop iterations

General instruction syntax

`[(qp)] mnemonic[.comp1][.comp2] dests = srcs`

`(qp)` — optional predication of instruction

`.comp` — optional comparison type

`dest` — destination

`dest = p1` ⇒ **`p1`** ← **`1`** on true and **`p1`** ← **`0`** on false

IA-64 Predication Example

```
switch (type)
{
  case 'a':
    type=type+10;
    break;
  case 'b':
    type=type+20;
    break;
  default:
    break;
}
```

Compile for IA-32

```
     MOV  EAX, [type]
     CMP EAX, 'a'
     JNE L1
     ADD EAX, 10
     JMP L2
L1:  CMP EAX, 'b'
     JNE L2
     ADD EAX, 20
L2:  MOV [type], EAX
```

Compile for IA-64

```
mov   r1 = type
ld4   r2 = [r1]
cmp.eq   p1 = 'a',r2
cmp.eq   p3 = 'b',r2
(p1)   add r2 = 10,r2
(p3)   add r2 = 20,r2
st4   [r1] = r2
```

Scheduling

CC1	mov r1 = type ld4 r2 = [r1]
CC2	cmp.eq p1 = 'a',r2 cmp.eq p3 = 'b',r2
CC3	(p1) add r2 = 10,r2 (p3) add r2 = 20,r2
CC4	st4 [r1] = r2

Ref: "Introducing the Intel Itanium Architecture"

Branching versus Predication

```
     MOV  EAX, [type]
     CMP EAX, 'a'
     JNE L1
     ADD EAX, 10
     JMP L2
L1:  CMP EAX, 'b'
     JNE L2
     ADD EAX, 20
L2:  MOV [type], EAX
```

CC1	`mov  r1 = type` `ld4  r2 = [r1]`
CC2	`cmp.eq  p1 = 'a',r2` `cmp.eq  p3 = 'b',r2`
CC3	`(p1)  add r2 = 10,r2` `(p3)  add r2 = 20,r2`
CC4	`st4  [r1] = r2`

Branching

- 9 instructions
- 3 run-once branch instructions
  - Branches predicted not-taken
  - Type = 'a' ⇒ 1 mispredictions
  - Type = 'b' ⇒ 1 mispredictions
  - Default ⇒ 2 mispredictions

Predication

- 7 instructions
- No branch instructions
  - No mispredictions
- Type = default ⇒ CC3 cancelled
  - Equivalent to stall
- Removes 1 stall for each type

Predication in ARM

Flag set suffix S

ALU instructions set status flags

N — Negative **Z — Zero**

C — Carry **V — oVerflow**

Conditional execution suffix

Execute instructions if flag combination is true

Example — operate-compare-if-else

Standard Execution	Conditional Execution
`SUB r3, r3, #1`	`SUBS r3, r3, #1`
`CMP r3, #0`	
`BEQ L1 ; branch on Z to L1`	`ADDNE r0, r1, r2`
`ADD r0, r1, r2 ; skip if r3 = 0`	`; NOP on Z`
`B L2 ; branch to L2`	`SUBEQ r0, r1, r2`
`L1: SUB r0, r1, r2 ; skip if r3 != 0`	`; NOP on !Z`
`L2: ADD r4, r5, r6`	`ADD r4, r5, r6`

Predication in PIC

Family of microcontrollers for embedded systems

Hundreds of versions — 8, 16, 32 bit data word

`BTFSC f, b`	`Cancel next instruction if bit b in word f = 0`	
`BTFSS f, b`	`Cancel next instruction if bit b in word f = 1`	
`INCFSZ f, d`	`d ← f + 1`	`Cancel next instruction if result = 0`
`DECFSZ f, d`	`d ← f – 1`	`Cancel next instruction one if result = 0`

Switch Implementation in PIC Assembly language

```
BTFSC PIR1, TMR1IF     ; Cancel next if (PIR1<TMR1IF> == 0)
GOTO T1_INT            ; branch to Timer1 ISR
BTFSC PIR1, ADIF       ; Cancel next if (PIR1<ADIF> == 0)
GOTO AD_INT            ; branch to A/D ISR
BTFSC PIR1, LCDIF      ; Cancel next if (PIR1<LCDIF> == 0)
GOTO LCD_INT           ; branch to LCD ISR
BTFSC PIR1, RBIF       ; Cancel next if (PIR1<RBIF> == 0)
GOTO PORTB_INT         ; branch to PortB ISR
GOTO INT_ERROR_LP1     ; default ISR
```

Generalized Predication Model

1-bit width predicate registers

p0 — pN

Regs[p0] = 1

Predication

Every standard instruction can be predicated (guarded)

Predicate comparison instructions

Standard integer, unsigned, float, double comparisons of ISA

Can take 1 or 2 predicate register destinations

Syntax

p_1 = CMP.type1 src_1,src_2 <p>

p_1,p_2 = CMP.type1.type2 src_1,src_2 <p>

p_1,p_2 — 1 or 2 destinations of specified types

C = CMP.type1.type2 — 1 or 2 compare conditions

<p> — instruction canceled if **p = 0**

Predicate Types

Predicate registers assigned according to deposit types

		unconditional		conditional		OR		AND		∨		∧	
p	C	T	F	T	F	T	F	T	F	T	F	T	F
0	0	0	0	-	-	-	-	-	-	-	1	0	0
0	1	0	0	-	-	-	-	-	-	1	-	0	0
1	0	0	1	0	1	-	1	0	-	1	1	0	-
1	1	1	0	1	0	1	-	-	0	1	1	-	0

	Assignment		Condition for Assignment
	T	F	
unconditional	(p && C)	(p && C')	—
conditional	C	C'	(p = 1)
OR	1	1	(p && C,C') = 1
AND	0	0	(p && !(C,C')) = 1
∨	1	1	(p || C,C') = 1
∧	0	0	(p && C,C') = 0

Direct Predicate Access

Predicate clearing and setting

`pclear` and **`pset`** clear (assign 0) and set (assign 1)

Act on a predicate register or group of predicate registers

Integer register to predicate register move

Copy 64-bit integer register to 64 predicate registers

Copy 64 predicate registers to 64-bit integer register

Predicate saving and restoring

Predicate registers are essential part of architectural state

Saved before procedure call, context switch, etc.

Restored after procedure call, context switch, etc.

`pspill` and **`pfill`** push and pop predicate registers to/from stack

Predicate Concatenation Examples

Sequence of predicate instructions

p1 = C1.UT	p1 ← condition C1	–

p1 = C2.OT	if C2 = 1
	  p1 ← 1
	else
	  no change in p1
		(1 && C2) = 1

p1 = C3.AT	if C3 = 0
	  p1 ← 0
	else
	  no change in p1
		(1 && !C3) = 1

p2 = 1	p2 ← 1	–

p2 = C4.∧T <p1>	if (C4 = 0 OR p1 = 0)
	  p2 ← 0
	else
	  no change in p2
		(p_1 && C4) = 0

Predication Example — 1

```
main()
{
    int x = 0, y, z;
    while (x < 10){
        y = x++;
        z = x + y;
    }
}
```

gcc output for IA-32 (Linux)

```
 mov [x], 0x0
     jmp L2
     ; miss on 1st
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jle L1
     ; miss on 1st and last
```

Predicated version

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     mov eax, [x]  <p1>
     mov [y], eax  <p1>
     add [x], 0x1  <p1>
     mov eax, [y]  <p1>
     add eax, [x]  <p1>
     mov [z], eax  <p1>
     jmp L1        <p1> ; miss on 1st
```

Predication Example — 2

```
main()
{
   int x = 0, y, z;
   while (x < 10 && y < 5){
      y = x++;
      z = x + y;
   }
}
```

gcc output for IA-32 (linux)

```
     mov [x], 0x0
     jmp L2
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jg end
     cmp [y], 0x4
     jle L1
end:
```

Predicated version

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.AT [y], 0x4
     mov eax, [x]  <p1>
     mov [y], eax  <p1>
     add [x], 0x1  <p1>
     mov eax, [y]  <p1>
     add eax, [x]  <p1>
     mov [z], eax  <p1>
     jmp L1        <p1>
```

```
; p1 = 0 if !C = 1
```

Predication Example — 3

```
main()
{
   int x = 0, y, z;
   while (x < 10 || y < 5){
      y = x++;
      z = x + y;
   }
}
```

gcc output for IA-32 (linux)

```
     mov [x], 0x0
     jmp L2
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jle L1
     cmp [y], 0x4
     jle L1
```

Predicated version

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.OT [y], 0x4
     mov eax, [x]  <p1>
     mov [y], eax  <p1>
     add [x], 0x1  <p1>
     mov eax, [y]  <p1>
     add eax, [x]  <p1>
     mov [z], eax  <p1>
     jmp L1        <p1>
```

```
; p1 = 1 if C = 1
```

Predication Example — 4

```
main()
{
   int x = 0, y, z;
   while (x < 10 && y < 5 || z < 20){
      y = x++;
      z = x + y;
   }
}
```

gcc →

```
     mov [x], 0x0
     jmp L2
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jg L3
     cmp [y], 0x4
     jle L1
L3:  cmp [z], 0x13
     jle L1
```

Predicated version ←

```
; p1 = 0 if !C = 1
; p1 = 1 if C = 1
```

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.AT [y], 0x4
     p1 = cmp.le.OT [z], 0x13
     mov eax, [x]   <p1>
     mov [y], eax   <p1>
     add [x], 0x1   <p1>
     mov eax, [y]   <p1>
     add eax, [x]   <p1>
     mov [z], eax   <p1>
     jmp L1         <p1>
```

Predication Example — 5a

```
main()
{
   unsigned x = 0, y, z;
   while (x < 10 && y < 5){
      y = x++;
      if (x % 2 == 0)
         z = x + y;
      else
         z = x - y;
   }
}
```

gcc →

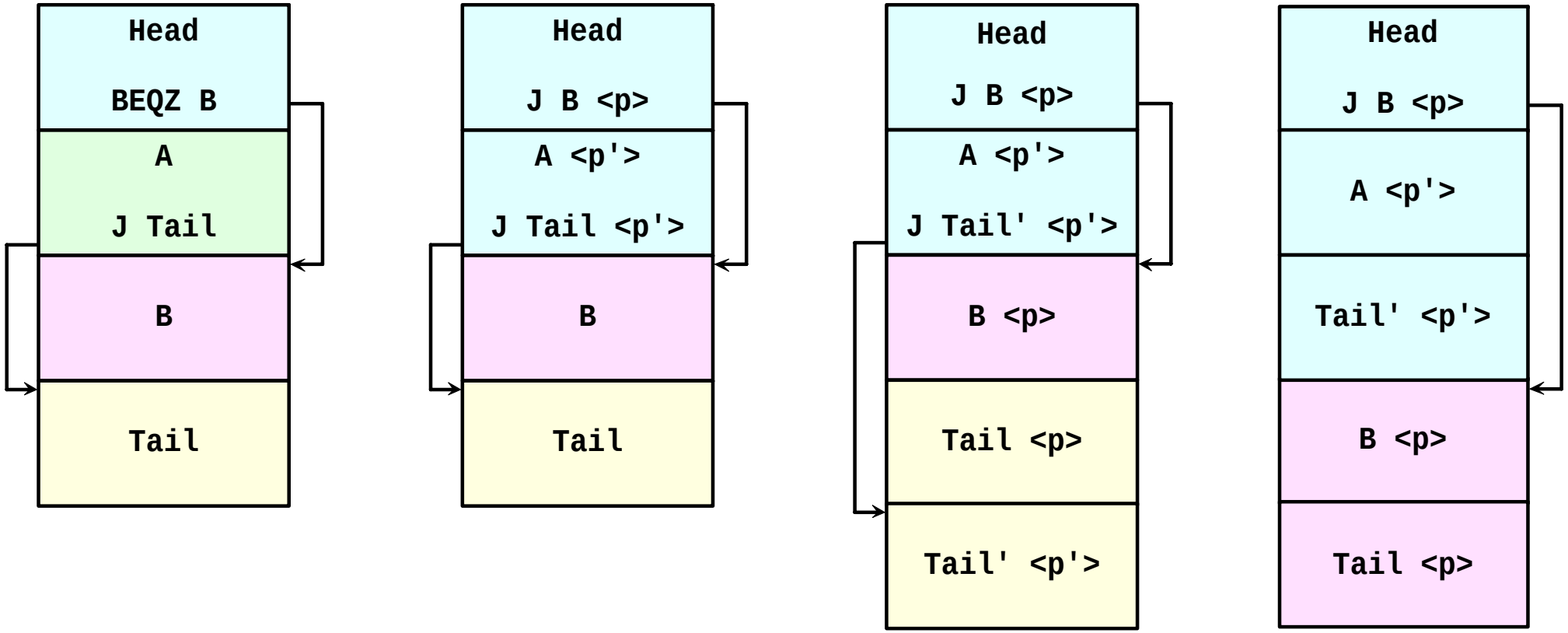
```
     mov [x], 0x0
     jmp L3
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [x]
     and eax, 0x1
     test eax, eax
     jne L2
     mov eax, [y]
     add eax, [x]
     mov [z], eax
     jmp L3
L2:  mov eax, [y]
     sub eax, [x]
     mov [z], eax
L3:  cmp [x], 0x9
     ja end
     cmp [y], 0x4
     jna L1
end
```

Predication Example — 5b

```
main()
{
   unsigned x = 0, y, z;
   while (x < 10 && y < 5){
      y = x++;
      if (x % 2 == 0)
         z = x + y;
      else
         z = x - y;
   }
}
```

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.AT [y], 0x4
     mov eax, [x]                          <p1>
     mov [y], eax                          <p1>
     add [x], 0x1                          <p1>
     mov eax, [x]                          <p1>
     and eax, 0x1                          <p1>
     p2,p3 = cmp.eq.UT.UF eax, 0x0         <p1>
     mov eax, [y]                          <p2>
     add eax, [x]                          <p2>
     mov [z], eax                          <p2>
     mov eax, [x]                          <p3>
     sub eax, [y]                          <p3>
     mov [z], eax                          <p3>
     jmp L1                                <p1>
```

Hyperblock Formation



General form of if-conversion

- Height reduction of control dependencies
  - Replace conditional branch (A or B) with predication
  - Fewer branch decisions ⇒ greater ILP and fewer distinct traces in trace cache
- Duplicate Tail code to avoid unconditional branch
- Compromise between branching and predication

Compiler Issues

Predication introduced as compiler optimization

- First pass — standard compilation with branches
- Optimization pass
  - if-conversion of branches to predication
  - Requires high level view of large sections of code
  - Generally start from innermost if-then-else sections

Predication not yet fulfilling its potential for improvement

- Complex issues in compiler theory
  - Exposure of inherent ILP
  - Branch removal versus excess fetching for cancelled instructions
  - Interactions with other compiler optimizations
    - Dead code removal, loop-invariant code motion, sub-expression elimination
- Small market
  - Not implemented in mass market PC processors
  - Important in smartphone, mainframe, and server sectors

Multipath Execution — Example 1

Predicated code

```
I0 p1,p2 = comp.eq.UT.UF r1,r2
I1 <p1>
I2 <p2>
I3 <p1>
I4 <p2>
I5 <p2>
I6 <p1>
```

Predicate comparison I_0 must be executed before I_1 – I_6

Either I_1, I_3, I_6 or I_2, I_4, I_5 must be cancelled

Multipath execution

Execute all of I_1 – I_6 without cancellations

Execute predicate comparison I_0 in any order

Cancel I_1, I_3, I_6 or I_2, I_4, I_5

- After I_0 is complete
- Before retirement of I_1 – I_6

Multipath versus Branch — Example 2

		1	2	3	4	5	6	7	8	9	10	11
	SEQ R3, R1, R2	F	D	X	M	W						
	BNEZ R3, I_2		F	D	X	M	W					
I_1				F	D	X	M	W	R1 = R2			
	J I_3			F	D	X	M	W				
I_2				F	F	D	X	M	W	R1 ≠ R2		
I_3					F	D	X	M	W			

8 clock cycles

	1	2	3	4	5	6	7	8	9	10
p1,p2 = comp.eq.UT.UF r1,r2	F	D	X	M	W					
<p1> I_1	F	D	X	M	W	Cancel if R1 ≠ R2				
<p2> I_2	F	D	X	M	W	Cancel if R1 = R2				
I_3	F	D	X	M	W					

5 clock cycles

$$S \sim \frac{8}{5} = 1.6$$