IA64 Architecture and Compilers

Dr. Allan Knies, Intel, IA64 Processor Division

Dr. Wei Li, Intel, Microcomputer Software Labs

Dr. Jesse Fang, Intel, Microprocessor Research Labs

IA-64 Application Architecture Tutorial

Allan D. Knies
IA-64 Architecture and Performance Group
Intel Corporation
allan.knies@intel.com

Objectives for This Tutorial

Provide background for some of the architectural decisions

Provide a description of the major features of the IA-64 application architecture

- Provide introduction and overview
- Describe software and performance usage models
- Mention relevant design issues

Show an example of IA-64 feature usage (C -> asm)

Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides (included, but not covered)

IA-64 Definition History

Two concurrent 64-bit architecture developments:

– IAX at Intel from 1991
  • Conventional 64-bit RISC
– Wideword at HP Labs from 1987
  • Unconventional 64-bit VLIW derivative

IA-64 definition started in 1994

– Extensive participation of Intel and HP architects, compiler writers, micro-architects, logic/circuit designers
– Several customers also participated as definition partners

Currently there are 3 generations of microprocessors in different stages of design

IA-64 Strategies

Extracting parallelism is difficult

- Existing architectures contain limitations that prevent sufficient parallelism on in-order implementations

Strategy

- Allow the compiler to exploit parallelism by removing static scheduling barriers (control and data speculation)
- Enable wider machines through large register files, static dependence specification, static resource allocation

IA-64 Strategies

Branches interrupt control flow/scheduling

- Mispredictions limit performance
- Even with perfect branch prediction, small basic blocks of code cannot fully utilize wide machines

Strategies

- Allow compiler to eliminate branches (and increase basic block size) with predication
- Reduce the number and duration of branch mispredicts by using compiler generated branch hints
- Allow compiler to schedule more than one branch per clock - multiway branch

IA-64 Strategies

Memory latency is difficult to hide

- Increasing relative to processor speed (larger cache miss penalties)

Strategy

- Allow the compile to schedule for longer latencies by using control and data speculation
- Explicit compiler control of data movement through an architecturally visible memory hierarchy

IA-64 Strategies

Procedure calls interrupt scheduling/control flow

- Software modularity is standard
- Call overhead from saving/restoring registers

Strategy

- Provide special support for software modularity
- Reduce procedure call/return overhead
  - Register Stack
  - Register Stack Engine (RSE)

IA-64 Strategies Summary

Move complexity of resource allocation, scheduling, and parallel execution to compiler

Provide features that enable the compiler to reschedule programs using advanced features (predication, speculation)

Enable wide execution by providing processor implementations that the compiler can take advantage of

Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides included (but not covered)

- *Loop Support*
- *Register Stack*
- *Memory Support*
- *Floating Point, Multi-media, 3D Graphics*

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Application State

Directly accessible CPU state

- 128 x 65-bit General registers (GR)
- 128 x 82-bit Floating-point registers (FR)
- 64 x 1-bit Predicate registers (PR)
- 8 x 64-bit Branch registers (BR)

Indirectly accessible CPU state

- Current Frame Marker (CFM)
- Instruction Pointer (IP)

Control and Status registers

- 19 Application registers (AR)
- User Mask (UM)
- CPU Identifiers (CPUID)
- Performance Monitors (PMC,PMD)

Memory

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

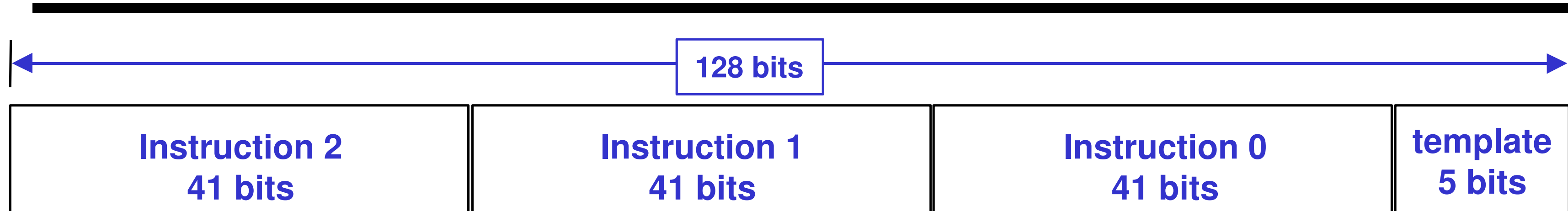
Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Instruction Formats: Bundles



Instruction Types

- M: Memory
- I: Shifts, MM
- A: ALU
- B: Branch
- F: Floating point
- L+X: Long

Template types

- Regular: MII, MLX, MMI, MFI, MMF
- Stop: MI_I M_MI
- Branch: MIB, MMB, MFB, MBB, BBB
- All come in two versions:
  - with stop at end
  - without stop at end

Instruction Formats: Instructions

major opc 4b	minor opcode or immediate 10 bits	register id 7 bits	register id 7 bits	register id 7 bits	qual. pred 6 bits

- Qualifying predicates (6 bits)
  - A few instructions do not have a QP
- Register operand identifiers (7 bits)
- Register result identifier(s) (6 or 7 bits)
- Immediate operands (8-22 bits)
- Minor opcode
- Major opcode (4 bits)

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Integer Instructions

Memory - load, store, semaphore, . . .

Arithmetic - add, subtract, shladd, . . .

Compare - lt, gt, eq, ne, . . ., tbit, tnat

Logical - and, or

Bitfields - deposit, extract

Shift Pair

Character

Shifts - left, right

32-bit support - cmp4, shladdp4

Move - various register files moves

No-ops

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Execution Semantics

Traditional architectures have sequential semantics

- The machine must always ***behave*** as if the instructions were executed in an unpipelined sequential fashion
- If a machine actually issues instructions in a different order or issues more than one instruction at a time, it must insure sequential execution semantics are obeyed

Case 1 - Dependent

```
add r1 = r2, r3
sub r4 = r1, r2
shl r2 = r4, r8
```

Case 2 - Independent

```
add r1 = r2, r3
sub r4 = r11, r21
shl r12 = r14, r8
```

Execution Semantics

IA-64 has parallel semantics

- The compiler uses templates with stops to indicate dependent operations
- Hardware does not have to check for dependent operations within instruction groups
  - WAR register dependences allowed
  - Memory operations still require sequential semantics
- Dependences disabled by predication dynamically

Case 1 - Dependent

```
add r1 = r2, r3 ;;
sub r4 = r1, r2 ;;
shl  r2 = r4, r8
```

Case 2 - Independent

```
add r1 = r2, r3
sub r4 = r11, r21
shl r12 = r14, r8 ;;
```

Case 3 - Predication

```
(p1) add r1 = r2, r3
(p2) sub r1 = r2, r3 ;;
shl r12 = r1, r8
```

IA-64 Application Architecture Tutorial

- Application State
- Instruction Format
- Integer Instructions
- Execution Semantics
- Control Speculation, Data Speculation
- Predication
- Parallel Compares
- Branch Architecture

Control and Data Speculation

Two kinds of instructions in IA-64 programs

- Non-Speculative Instructions -- known to be useful/needed
  - would have been executed in the original program
- Speculative instructions -- may or may not be used
  - Schedule operations before results are known to be needed
  - Usually boosts performance, but occasionally may degrade
  - Heuristics can guide compiler in aggressiveness
  - Need profile data for maximum benefit

Two kinds of speculation

- Control and Data

Moving loads up is a key to performance

- Hide increasing memory latency
- Computation chains frequently begin with loads

Speculation

Separates loads into 2 parts: speculative loading of data and detection of conflicts/faults.

Control Speculation

Original:

```
(p1)  br.cond
      ld8 r1 = [ r2 ]
```

Transformed:

```
      ld8.s r1 = [ r2 ]
      . . .
(p1)  br.cond
      . . .
      chk.s r1, recovery
```

Data Speculation

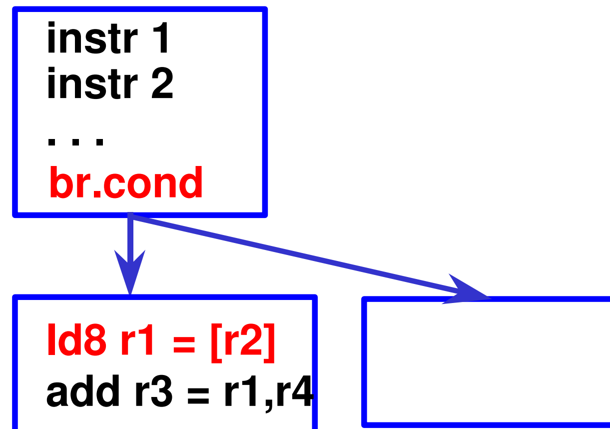
Original:

```
      st4 [ r3 ] = r7 ]
      ld8 r1 = [ r2 ]
```

Transformed:

```
      ld8.a r1 = [ r2 ]
      . . .
      st4 [ r3 ] = r7
      . . .
      chk.a r1, recovery
```

Control Speculation



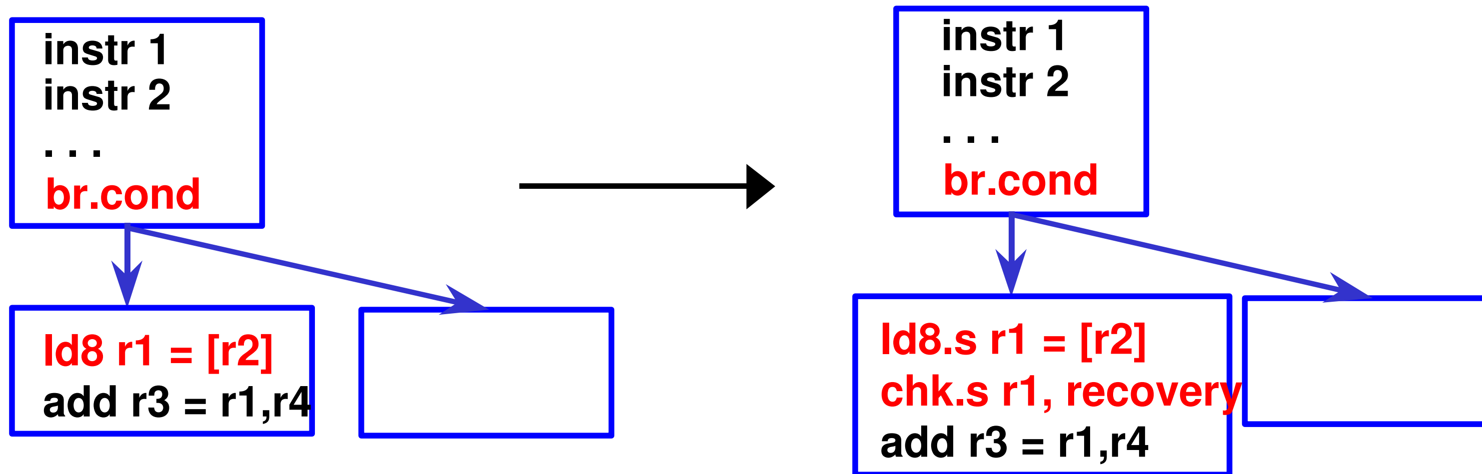
Example:

- Suppose br.cond is a check for a null pointer
- Suppose the load of r1 is dereferencing that pointer and then using it
- Normally, the compiler cannot reschedule the load before the branch because of potential fault

Control speculation is ...

- Moving loads (and possibly instructions that use the loaded values) above branches on which their execution is dependent

Control Speculation: Step 1



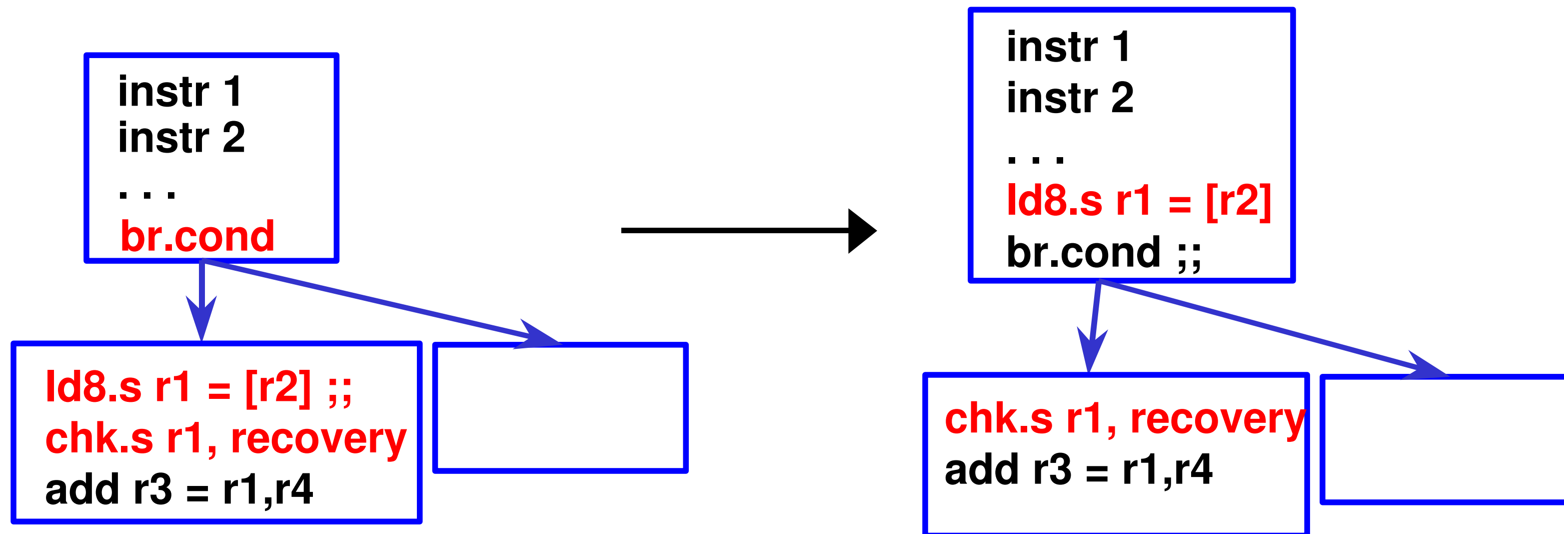
Separate load behavior from exception behavior

- ld.s which defers exceptions
- chk.s which checks for deferred exceptions

Exception token propagates from ld.s to chk.s

- NaT bits in General Registers, NaTVal (Special NaN value) in FP Registers

Control Speculation: Step 2



Reschedule ld8.s

– Now, ld8.s will defer a fault and set the NaT bit on r1

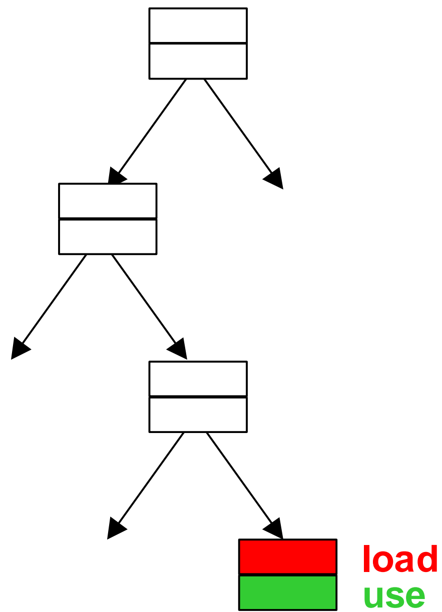
– chk.s checks r1's NaT bit and branches/faults if necessary

Allows faults to propagate

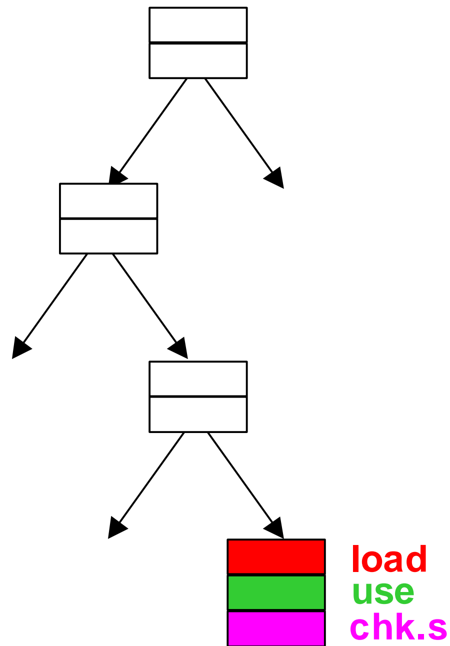
– NaT bits in General Registers, NaTVal (Special NaN value) in FP Registers

Control Speculation

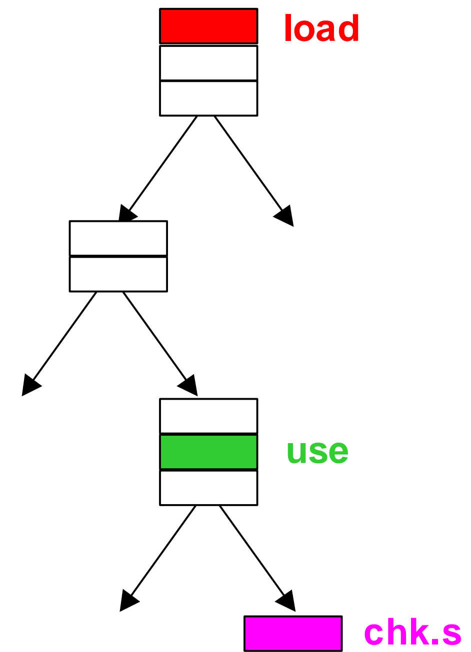
Original



Transform



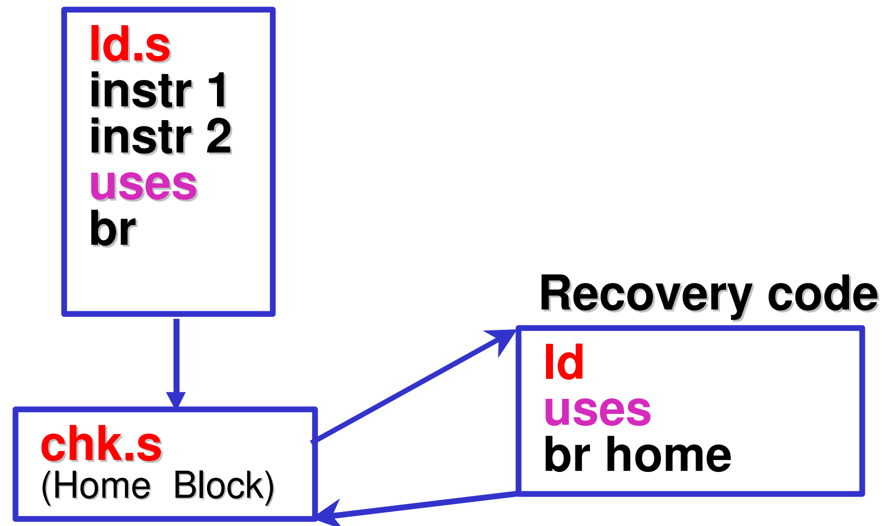
Reschedule



Hoisting Uses

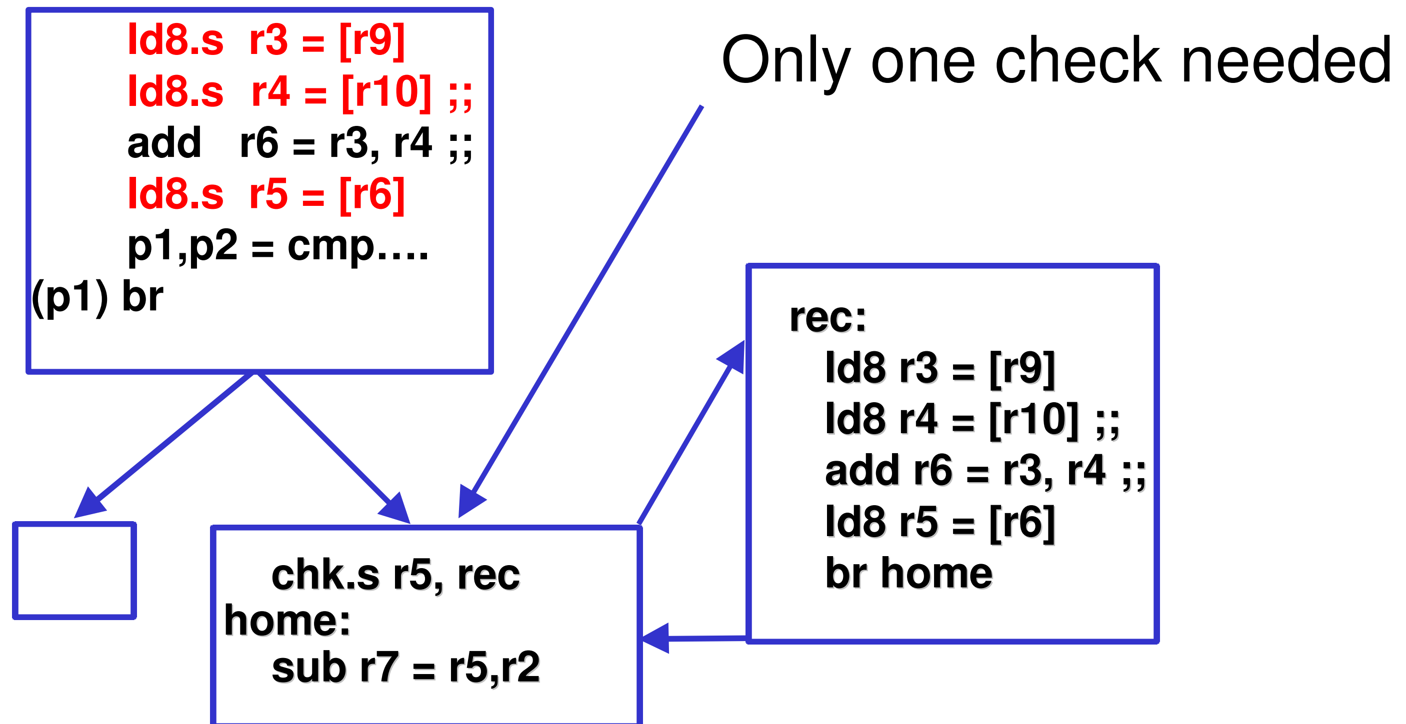
The uses of speculative data can also be executed speculatively

Requires extra recovery code and chk.s



NaT Propagation

All computation instructions propagate NaTs to reduce number of checks required



Exception Deferral

Deferral allows the efficient delay of costly exceptions

OS-controlled deferral of data-related faults

- Page faults
- Protection violations
- …

NaTs/Chks enable deferral with recovery

Architectural Support for Control Speculation

65th bit (NaT bit) on each GR indicates if an exception has occurred

Special speculative loads that set the NaT bit if a deferable exception occurs

Special chk.s instruction that checks the NaT bit and branches to recovery, if set

Computational instructions propagate NaTs like IEEE NaN's

Compare operations propagate "false" when writing predicates

instr2

st1 [r3] = r4

add r3 = r1,r4

Example:

– `st1` writes into memory

the

–

the store and the load addresses

load before the store

Such store to load dependences are

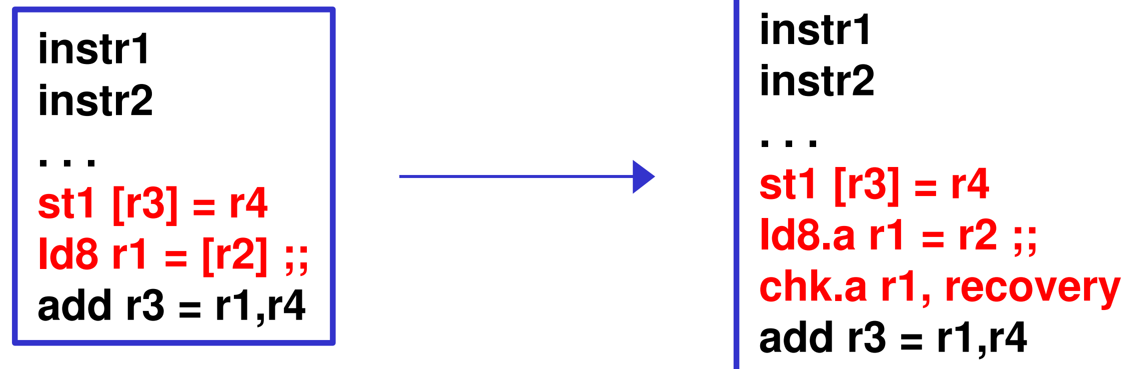
Data speculation is ...

–

instructions that use the loaded

stores

Data Speculation: Step 1



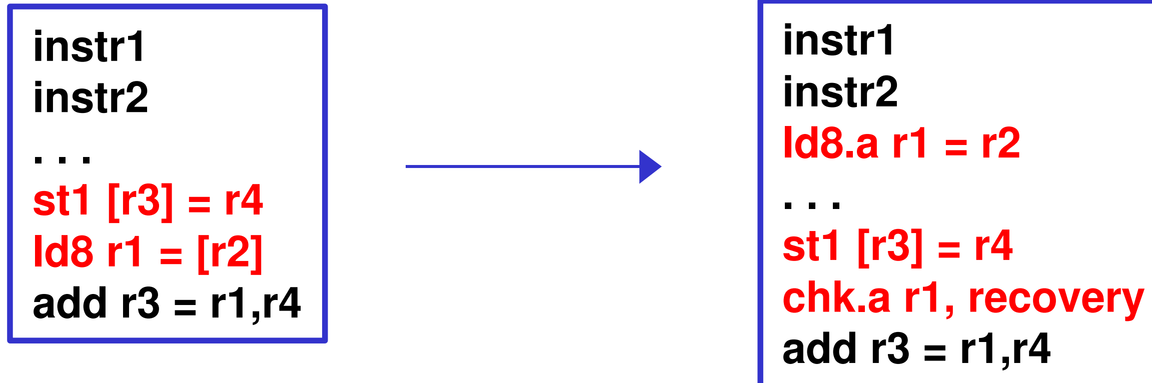
Separate load behavior from overlap detection

- ld8.a which performs normal loads and keeps bookkeeping (ALAT)
- chk.a which checks ALAT to see if conflicting store has occurred

Advanced load address table

- ld8.a puts information about advanced loads into table (address ranges accessed)
- stores and other memory writers ‘snoop’ ALAT and if overlapping loads are found, entries are deleted
- chk.a checks to see if a corresponding entry is in ALAT

Data Speculation: Step 2

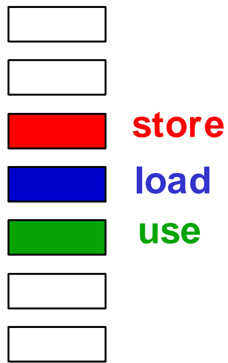


Reschedule ld8.a

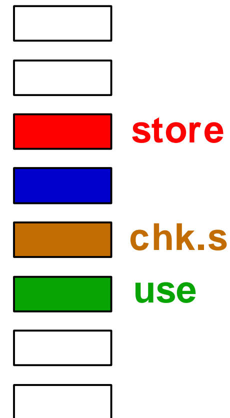
- Now, ld8.a will allocate an entry in the ALAT
- If the st1 instruction overlaps with the ld8.a address, then the ALAT entry will be removed
- chk.a checks for matching entry in ALAT -- if found, speculation was ok, if not found, need to re-execute

Data Speculation

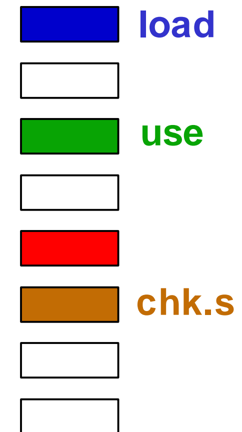
Original



Transform



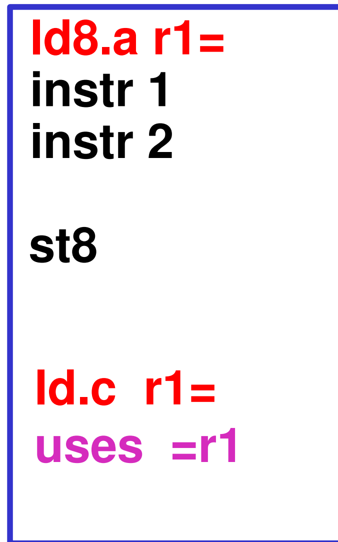
Reschedule



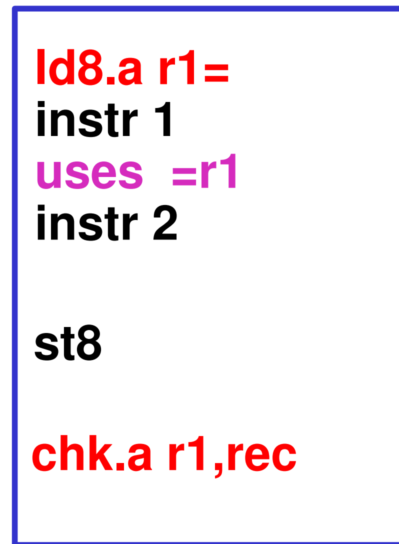
Hoisting Uses

Uses can be hoisted, but then chk.a needed for recovery

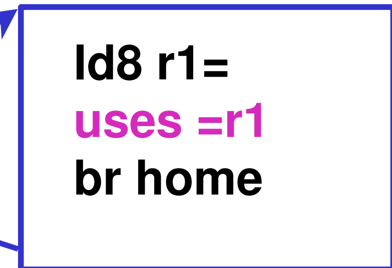
No hoisted uses



With hoisted uses



Recovery code



Architectural Support for Data Speculation

ALAT - HW structure containing information about outstanding advanced loads

Instructions

- ld.a - advanced loads
- ld.c - check loads
- chk.a - advance load checks

Speculative Advanced loads - ld.sa - is an control speculative advanced load with fault deferral (combines ld.a and ld.s)

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Predication Concepts

Branching causes difficult to handle effects

- Istream changes (reduces fetching efficiency)
- Requires branch prediction hardware
- Requires execution of branch instructions
- Potential branch mispredictions

IA-64 provides predication

- Allows some branches to be moved
- Allows some types of safe code motion beyond branches
- Basis for branch architecture and conditional execution

Predication

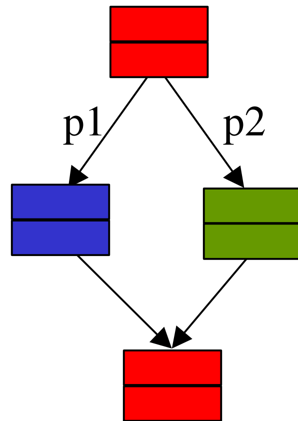
```
      cmp.eq
(p1) add     r7 = r2, r4
          sa
```

If p1 is performed, else it acts as a nop

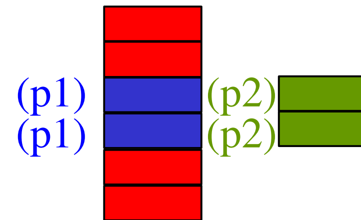
If p2 is performed, else it acts as a nop

Control Flow Simplification

Original



Predicated

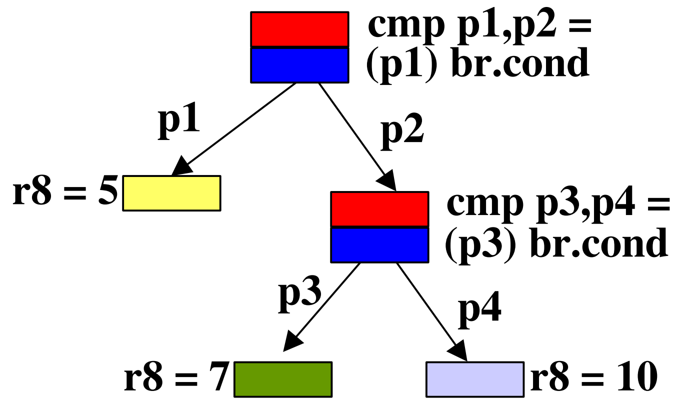


Predication

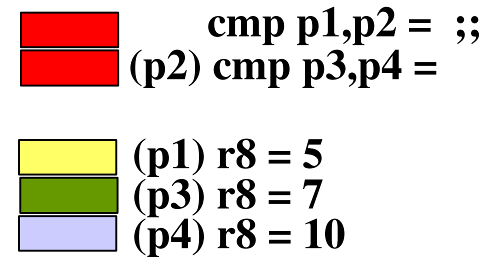
- Change control flow dependences into data dependences
- Removes branches
  - reduce/eliminate mispredictions and branch bubbles
  - instruction fetch efficiency
  - exposes parallelism

Multiple Selects

Original

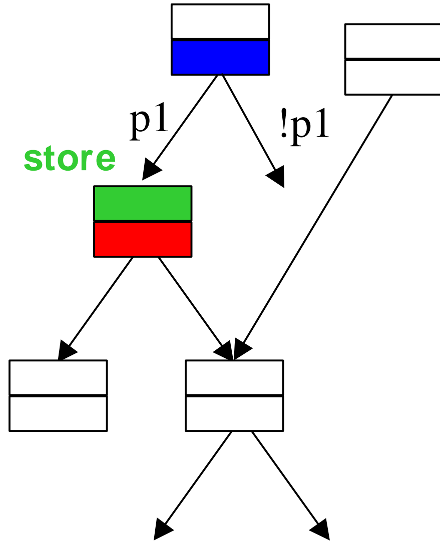


Transform/ Reschedule

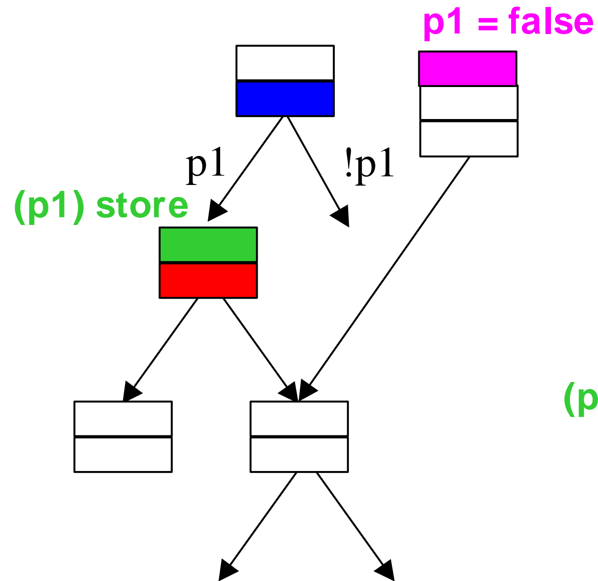


Downward Code Motion

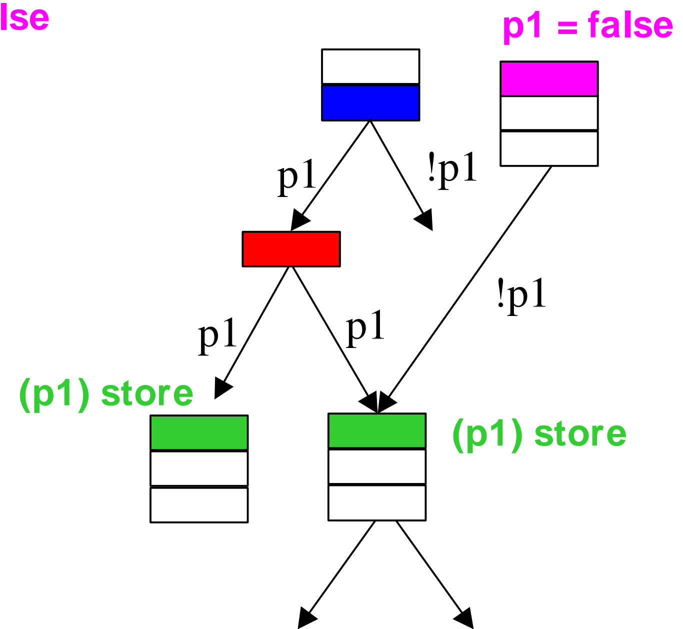
Original



Transform



Reschedule



Architectural Support

64 1-bit predicate registers (true/false)
- p0 - p63

Compare and test instructions write predicates with results of comparison/test
- most compare/test write result and complement
- Ex: cmp.eq p1,p2 = r1,0

Almost all instructions can have a qualifying predicate (qp)
- Ex: (p1) add r1 = r2, r3
- if qp is true, instruction executed normally
- if qp is false, instruction is squashed

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Parallel Compares

Parallel compares allow compound conditionals to be executed in a single instruction group.

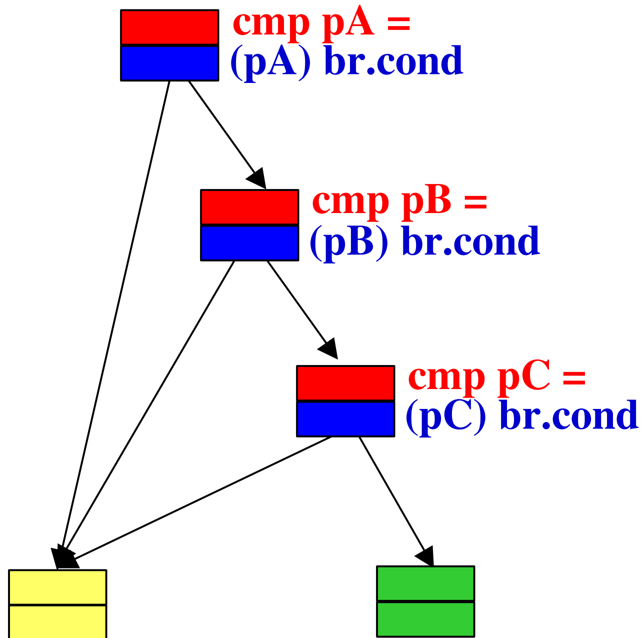
Example:

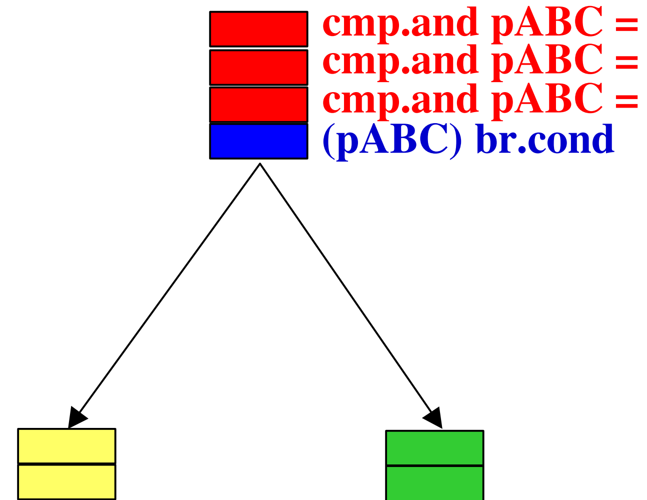
```
if ( a && b && c ) {  . . . }
```

Assembly:

```
cmp.eq p1 = r0,r0 ;; // init p1=1
cmp.ne.and p1 = rA,0
cmp.ne.and p1 = rB,0
cmp.ne.and p1 = rC,0
```

Height Reduction





Architectural Support

Compare

- equality: eq, ne
- relational: only against zero
- tbit and tnat

Allows for both ‘and’ and ‘or’ compares

- one side: and
- one side: or
- both sides of conditional: or.andcm, and.orcm

IA-64 Application Architecture Tutorial

Application State

Instruction Format

Integer Instructions

Execution Semantics

Control Speculation, Data Speculation

Predication

Parallel Compares

Branch Architecture

Branch Architecture

IP-offset branches (21-bit disp.)

Branch registers

- 8 registers for indirect jumps, call/ret link

Multi-way branches

- Bundle 1-3 branches in a bundle
- Allow multiple bundles to participate

Branch Execution

Unconditional branch

```
(p0) br   target;
```

Conditional branches

```
      cmp p1 = cond
(p1)  br    target;
```

Compare and branch can be in same instruction group

Compiler-directed static prediction w/dynamic prediction

- Reduced false mispredicts due to aliasing
- Frees space in H/W predictor
- Can give hint for dynamic predictor

Multiway Branches

Allow multiple branch targets to be selected in one instruction group

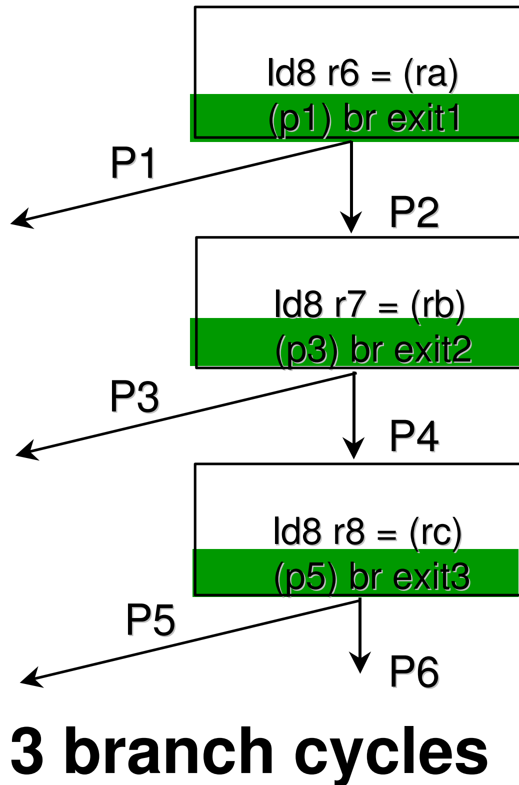
Example:

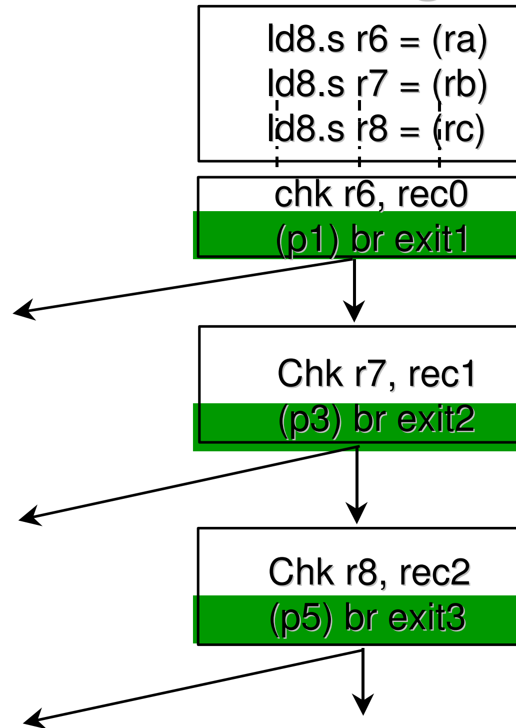
```
{ .bbb
(p1) br.cond target_1
(p2) br.cond target_2
(p3) br.call b1
}
```

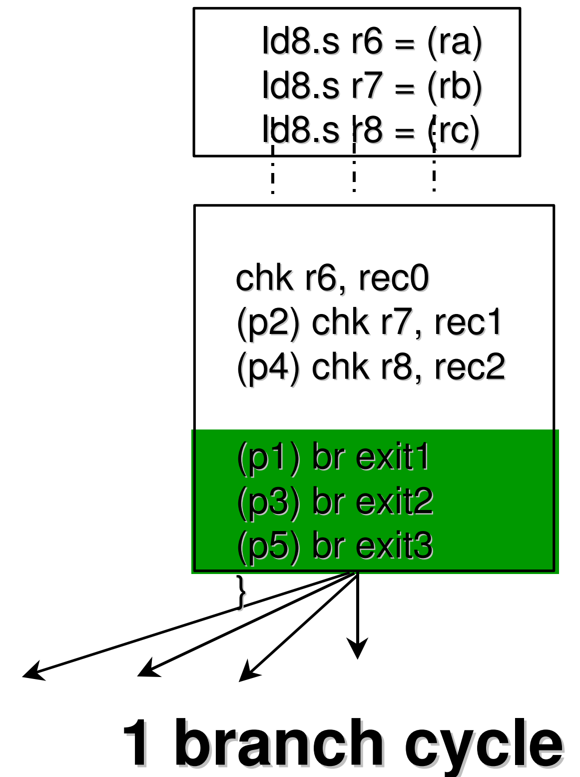
Four possible instructions executed next:

fall through, target_1, target_2, or address in b1

Control Height Reduction





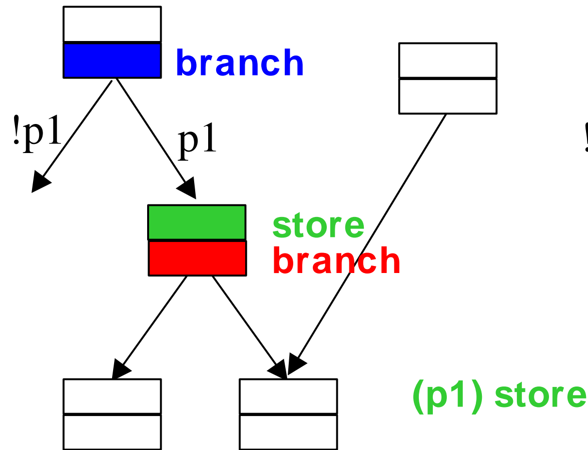


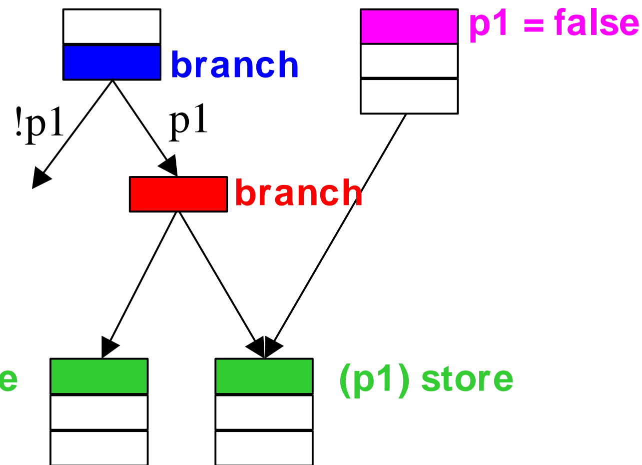
Control Height Reduction

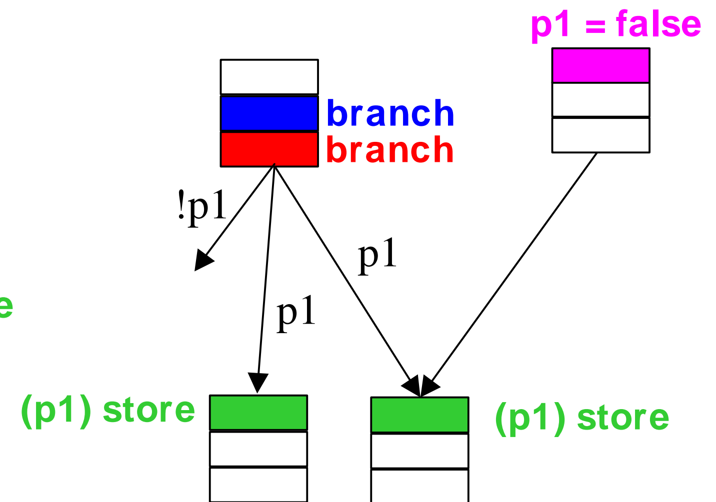
Original

Transform /Reschedule

Multiways Added







Branch Architecture

Notes:

- Multiple branches per clock is a natural side-effect of speculation
- Allows fast selection of multiple branch targets
- Branch prediction for both single and multiple branches is important for good performance
- Compiler profiling can help facilitate the use of hints
- Hints may reduce needed size/functionality of hardware predictors
- Works in conjunction with control speculation, data speculation, predication, and parallel compares

Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides included (but not covered)

Synthesis: ChkGetChunk()

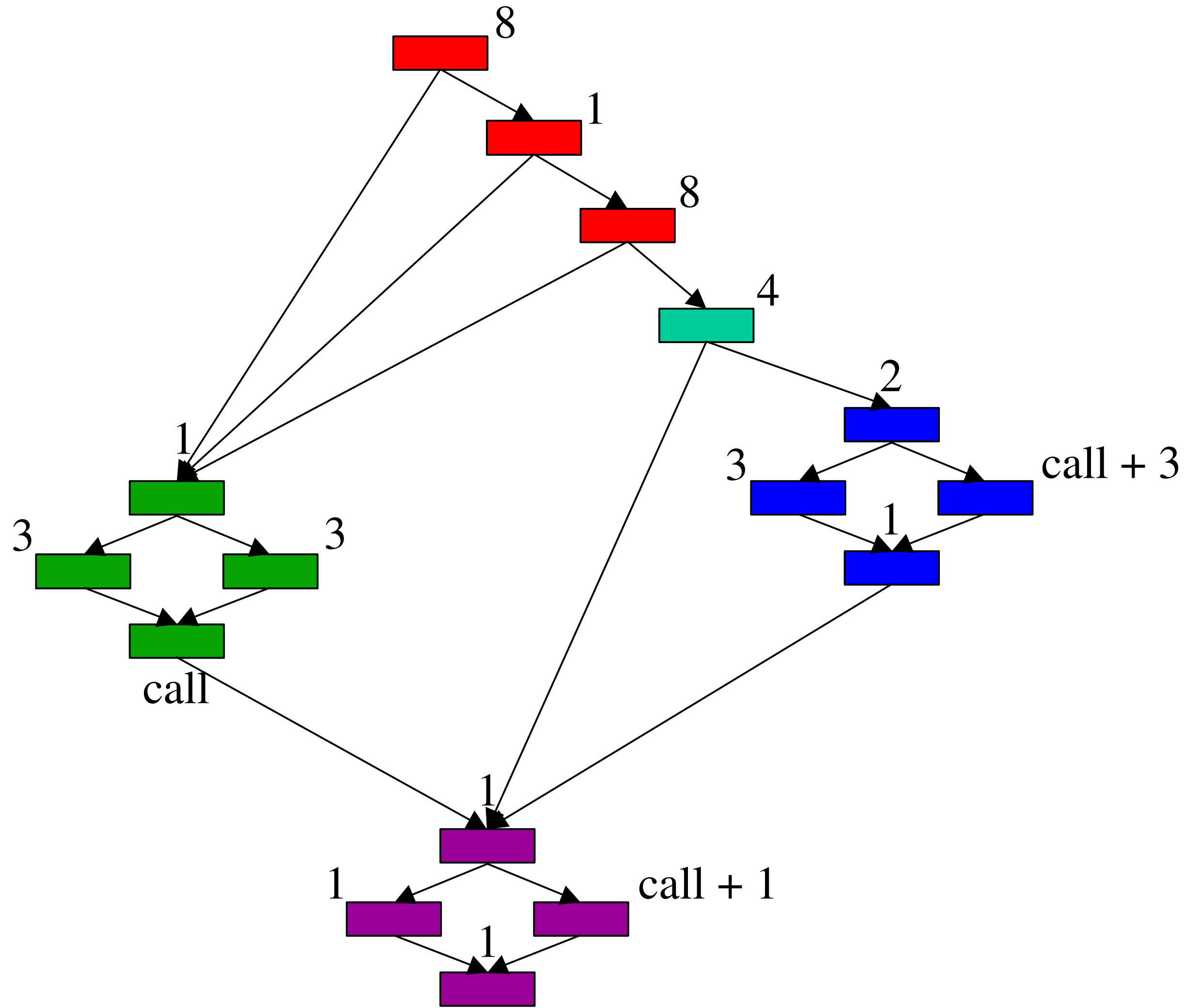
```
if (((Theory->Flags[ChunkNum] & 0x0008))
     && ((Theory->Flags[ChunkNum] & 0x0040))
     && (*(Theory->ChunkAddr[ChunkNum] - 28)) == SizeOfUnit) {
   StackPtr = (*(Theory->ChunkAddr[ChunkNum] - 20));
   if (Index >= StackPtr) {
      if (SetGetSwi)
         *Status    = -10009;
      else {
          Mem_DumpChunkChunk (0, ChunkNum);
          *Status    = 1005; } }
} else {
   if ((*(Theory->ChunkAddr[ChunkNum] - 28)) != SizeOfUnit) {
      *Status = 1003;
   } else {
      *Status = 1004; }
   Mem_DumpChunkChunk (0, ChunkNum);
   } }
return((Test= *Status==0 ? True:Ut_PrintErr (F,Z,*Status)));
```

Synthesis: ChkGetChunk()

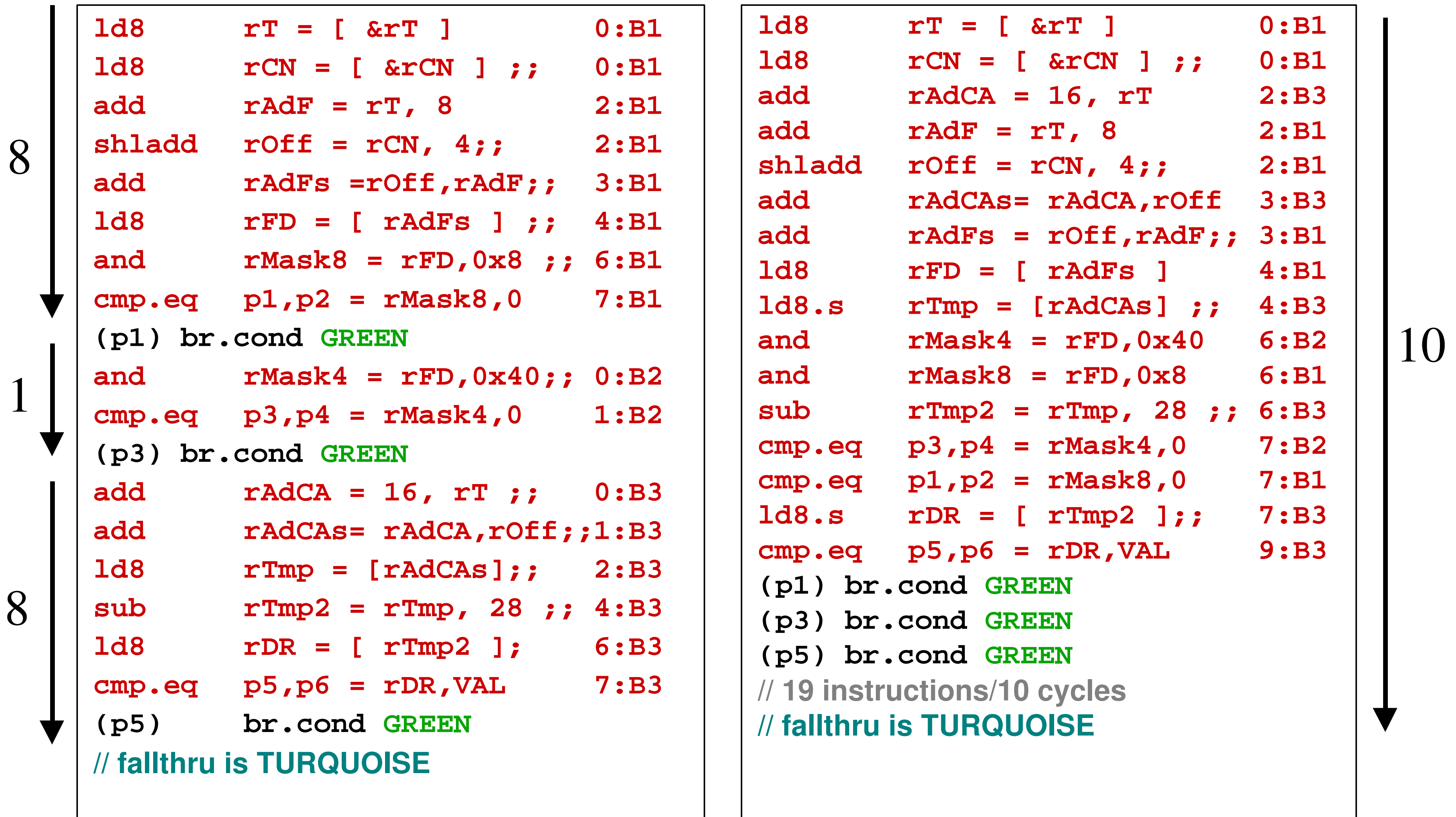
Assumptions for code examples

- Abstract machine model
- Unlimited instruction issue (execution) resources
- Loads have 2 cycle latency to first level cache
- All other instructions 1 cycle latency

Synthesis



Synthesis: ChkGetChunk()



Synthesis: ChkGetChunk()

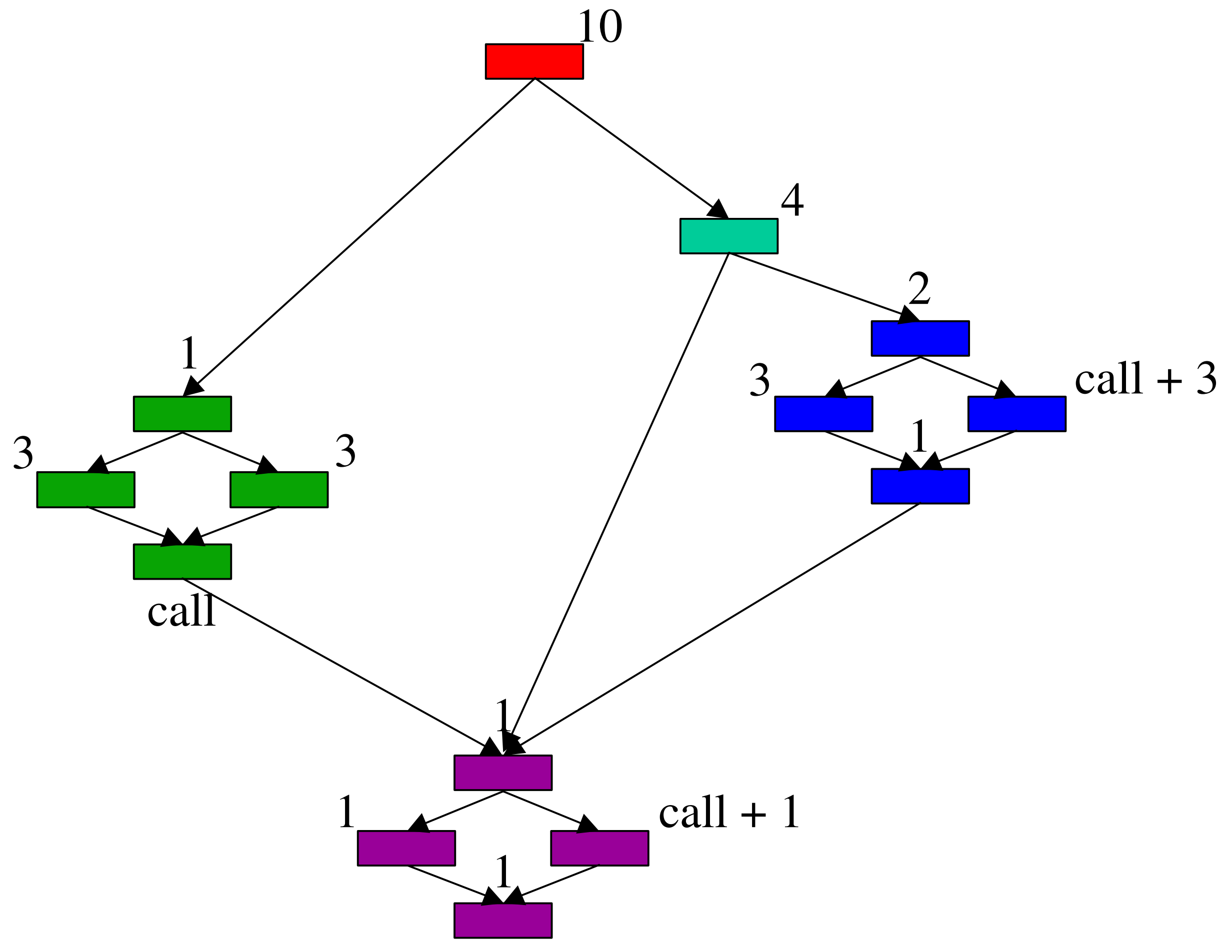
10

```
ld8       rT = [ &rT ]            0:B1
ld8       rCN = [ &rCN ] ;;       0:B1
add       rAdCA = 16, rT          2:B3
add       rAdF = rT, 8            2:B1
shladd    rOff = rCN, 4;;         2:B1
add       rAdCAs= rAdCA,rOff      3:B3
add       rAdFs = rOff,rAdF;;     3:B1
ld8       rFD = [ rAdFs ]         4:B1
ld8.s     rTmp = [rAdCAs] ;;      4:B3
and       rMask4 = rFD,0x40       6:B2
and       rMask8 = rFD,0x8        6:B1
sub       rTmp2 = rTmp, 28 ;;     6:B3
cmp.eq    p3,p4 = rMask4,0        7:B2
cmp.eq    p1,p2 = rMask8,0        7:B1
ld8.s     rDR = [ rTmp2 ];;       7:B3
cmp.eq    p5,p6 = rDR,VAL         9:B3
(p1) br.cond GREEN
(p3) br.cond GREEN
(p5) br.cond GREEN
// 19 instructions/10 cycles:  < 2 IPC
// fallthru is TURQUOISE
```

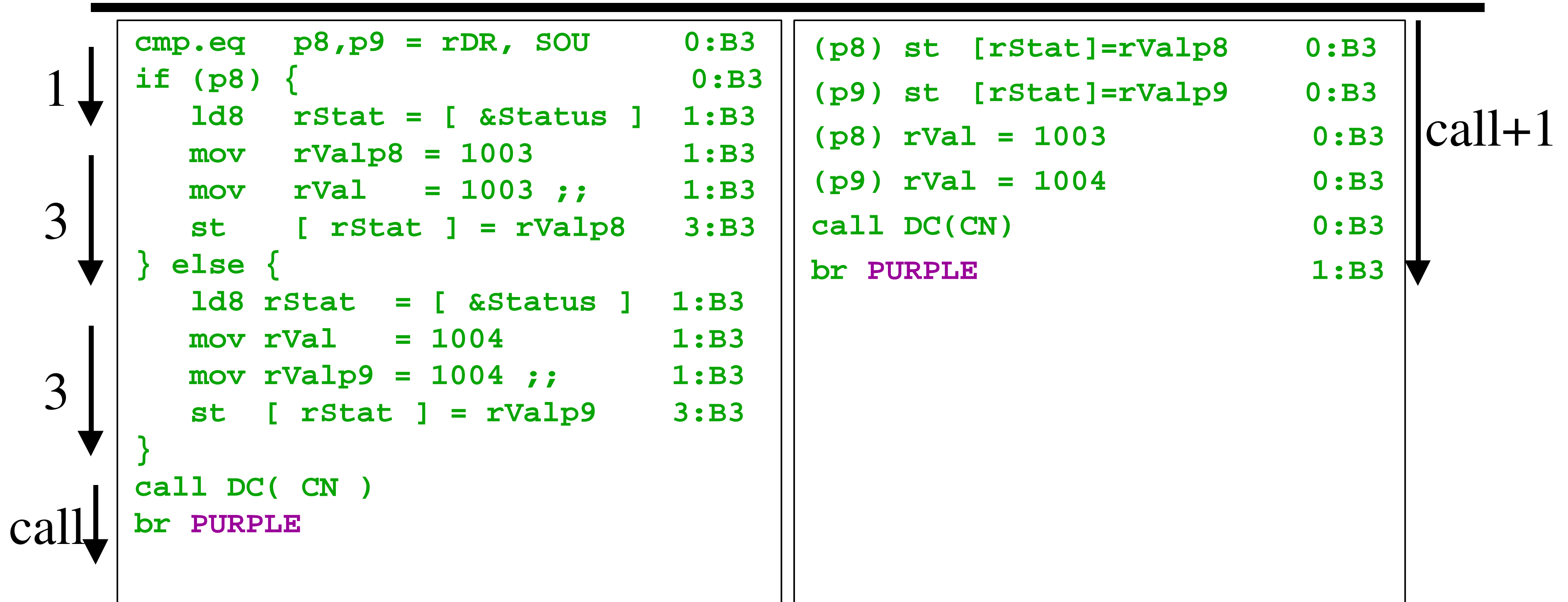
```
ld8       rT = [ &rT ]            0:B1
ld8       rCN = [ &rCN ] ;;       0:B1
                                  2:??
add       rAdCA = 16, rT          2:B3
add       rAdF = rT, 8            2:B1
shladd    rOff = rCN, 4;;         2:B1
add       rAdCAs= rAdCA,rOff      3:B3
add       rAdFs = rOff,rAdF;;     3:B1
ld8       rFD = [ rAdFs ]         4:B1
ld8.s     rTmp = [rAdCAs] ;;      4:B3
cmp.ne    p1,p2 = r0,r0           6:??
and       rMask4 = rFD,0x40       6:B2
and       rMask8 = rFD,0x8        6:B1
sub       rTmp2 = rTmp, 28 ;;     6:B3
cmp.eq.or.andcm   p1,p2=rMask4,0  7:B2
cmp.eq.or.andcm   p1,p2=rMask8,0  7:B1
ld8.s     rDR = [ rTmp2 ];;       7:B3
                                  9:??
cmp.eq.or.andcm   p1,p2=rDR,VAL   9:B3
(p1) br.cond      GREEN
// fallthru is TURQUOISE
```

10

Synthesis



Synthesis: ChkGetChunk()



Steps:

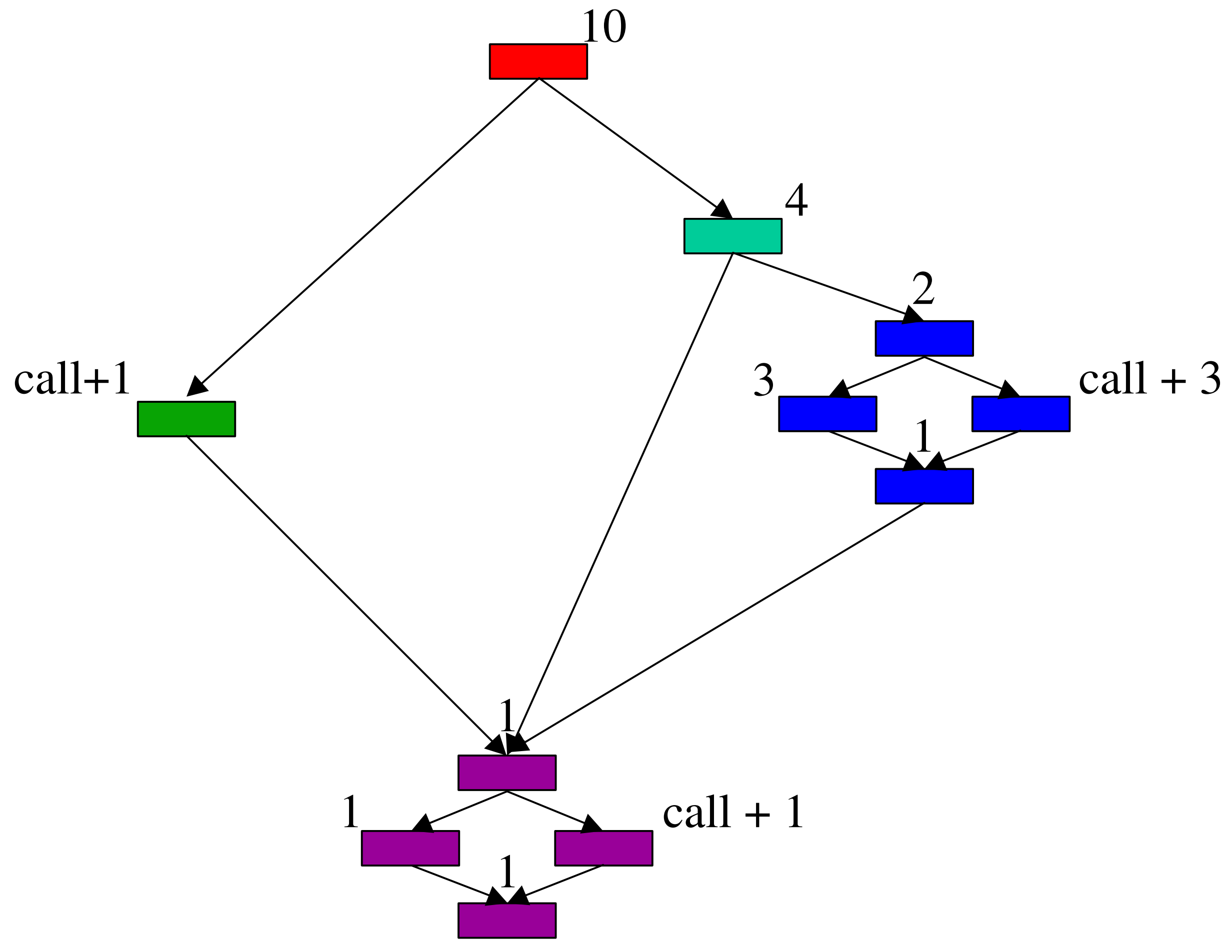
1) **Speculate `ld8` instructions into red block cycle 1**

2) **Copy and speculate `mov` instructions into red blocks cycle 0**

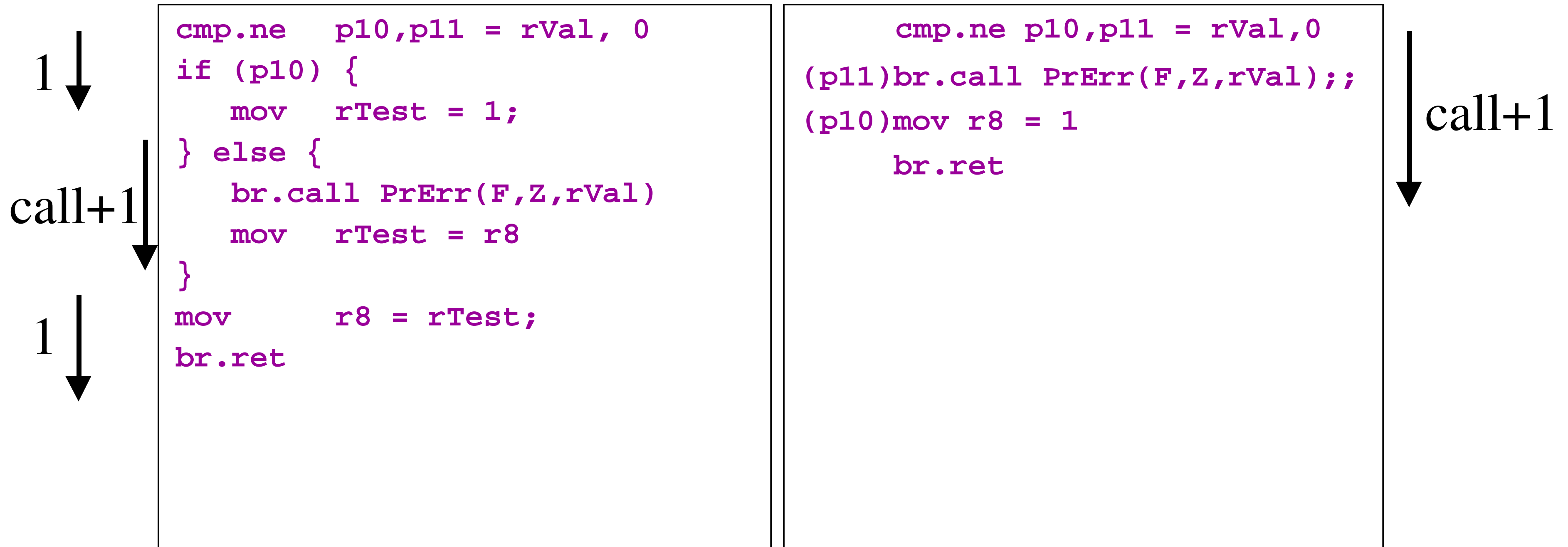
3) **Speculate `cmp` instruction into red block cycle 9**

4) **Predicate both sides of the conditional**

Synthesis



Synthesis: ChkGetChunk()

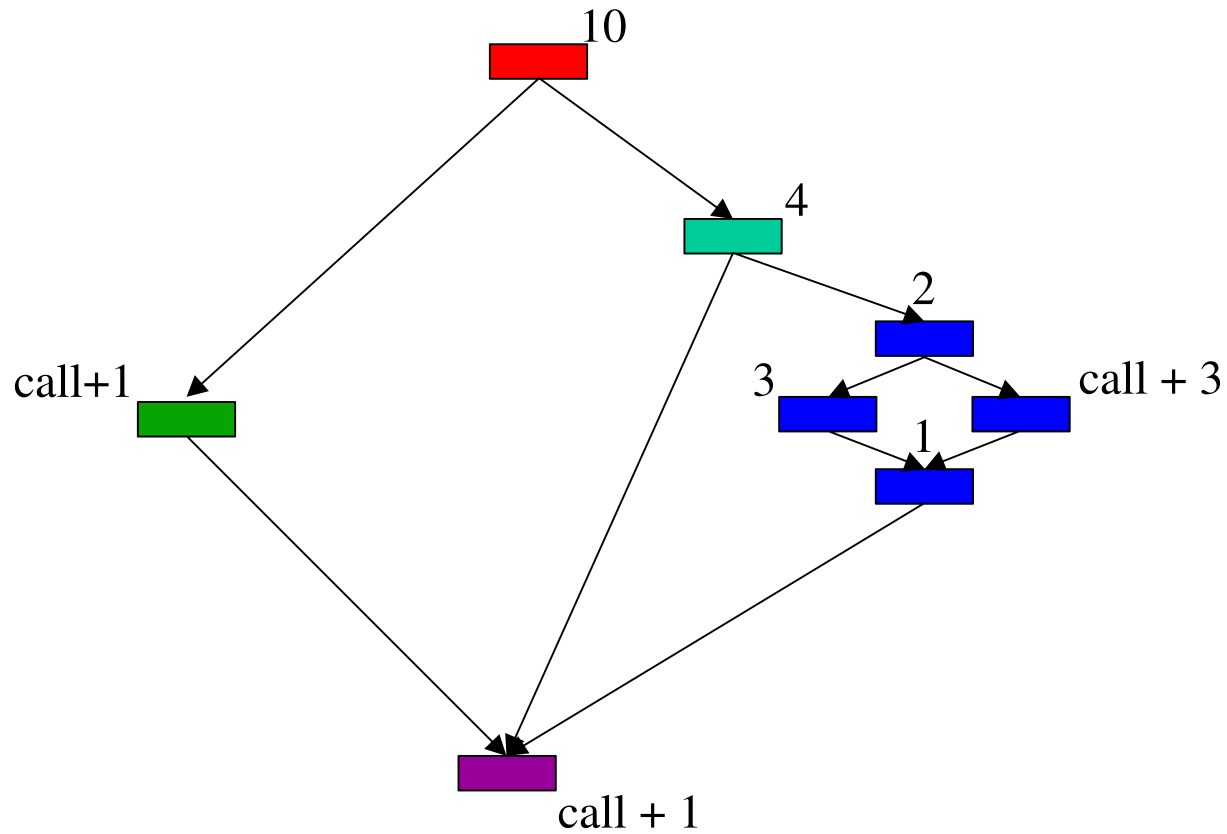


Steps:

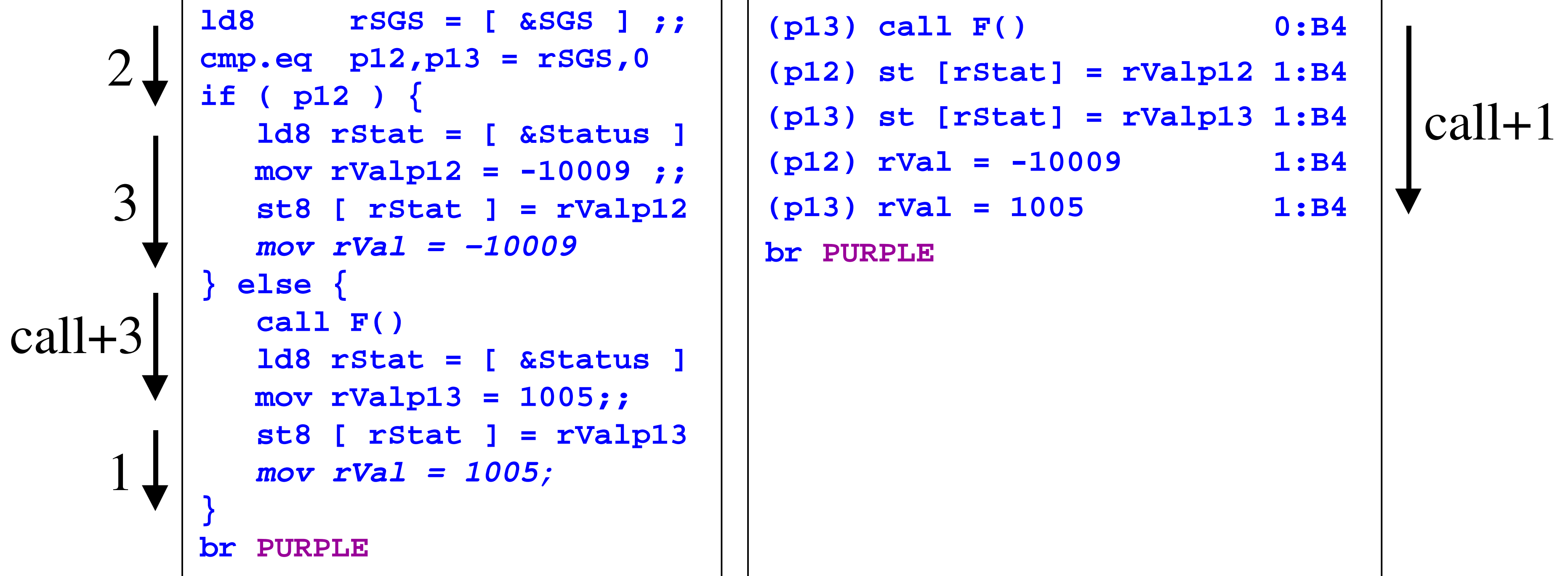
1) Replace `rTest` with `r8`

2) Predicate both sides of conditional

Synthesis



Synthesis: ChkGetChunk()



Steps:

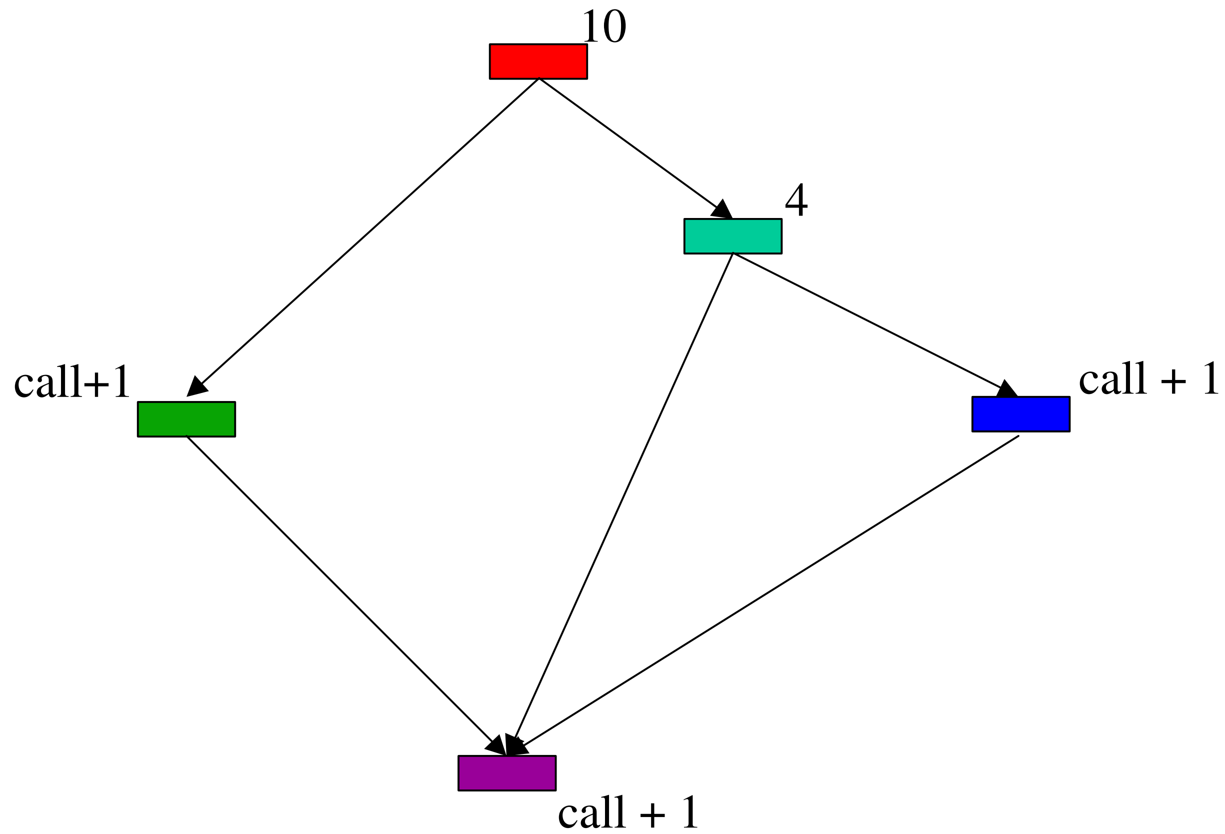
1) speculate all the `ld8`'s in to red block cycle 1

2) speculate the `cmp.eq` in to red block cycle 3

3) copy and speculate the `mov rVal` in to red block cycle 0

4) predicate both sides of conditional

Synthesis



Synthesis: ChkGetChunk()

4

```
sub       rTmp3 = rTmp2,20 ;;
ld8       rDR2 = [ rTmp3 ] ;;
st8       [ &StkPtr ] = rDR2
cmp.ge    p14,p15 = rDR2,rIdx
(p14) br.cond BLUE
br PURPLE
```

```
(p2) st8 [ &StkPtr ] = rDR2
(p14) br.cond BLUE
br PURPLE
(p13) call F()                    0:B4
(p12) st [ rStat ] = rValp12 1:B4
(p13) st [ rStat ] = rValp13 1:B4
(p12) rVal = -10009             1:B4
(p13) rVal = 1005               1:B4
br PURPLE
```

Steps:

1) speculate `sub`, `ld8`, and `cmp.ge` into the red block (cycles 6, 7, and 9)

2) predicate the `st8` with `(p2)`

3) concatenate with blue block

4) (cont. next page)

Synthesis: ChkGetChunk()

```
(p2) st8 [ &StkPtr ] = rDR2
(p14) br.cond BLUE
br PURPLE
(p13) call F()                0:B4
(p12) st [ rStat ] = rValp12 1:B4
(p13) st [ rStat ] = rValp13 1:B4
(p12) rVal = -10009          1:B4
(p13) rVal = 1005            1:B4
br PURPLE
```

```
(p2) st8 [&StkPtr] = rDR2   0:B4
(p13) call F()              0:B4
(p12) st [rStat] = rValp12 1:B4
(p13) st [rStat] = rValp13 1:B4
(p12) rVal = -10009        1:B4
(p13) rVal = 1005          1:B4
br PURPLE
```

call+1

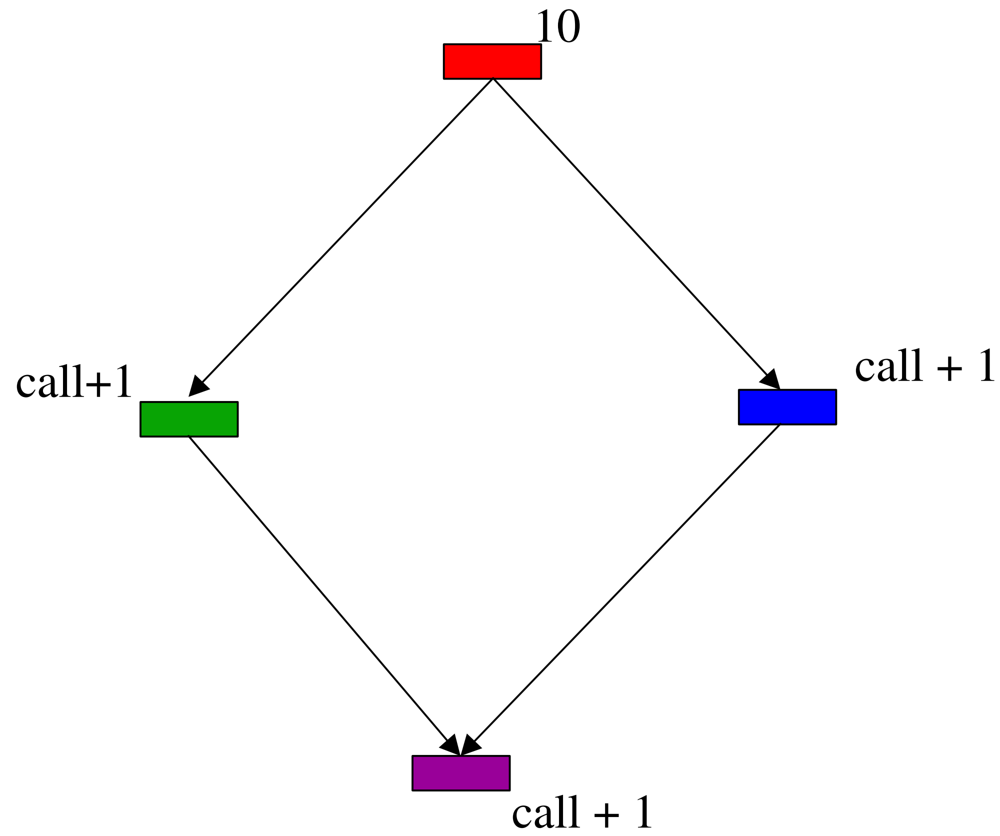
Steps:

1) qualify `p13` and `p12` (now in red block) so they can only be true when both `p2` and `p14` are true (use parallel and-compares) by either of the following:

- adding 4 parallel compares to red block
- lengthening red block by 1 cycle

2) now it is safe to remove the first PURPLE and BLUE branches

Synthesis



Synthesis: ChkGetChunk()

```
(p8) st  [rStat]=rValp8    0:B3
(p9) st  [rStat]=rValp9    0:B3
(p8) rVal = 1003           0:B3
(p9) rVal = 1004           0:B3
(p1) br.call DC(CN)        0:B3
br PURPLE                  1:B3
```

```
(p2) st8 [ &StkPtr ] = rDR2   0:B4
(p13) br.call DC(CN)          0:B4
(p12) st [ rStat ] = rValp12 1:B4
(p13) st [ rStat ] = rValp13 1:B4
(p12) rVal = -10009          1:B4
(p13) rVal = 1005            1:B4
br PURPLE
```

Steps:

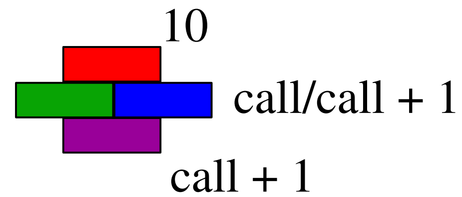
1) Note that `p2` is the 'blue/turquoise' branch predicate and that `p12` and `p13` are qualified with `p2` already

2) Predicate `br.call DC(CN)` with `p1`

3) If we further qualify `p8` and `p9` with `p1` (the 'green branch') in the red block, then the green and blue instructions are guaranteed to be independent! Can be done by either:

- adding 3 parallel compares to red block
- lengthening red block by 1 cycle

Synthesis



Cycles = 12 + 2 calls

Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides included (but not covered)

- *Loop Support*
- *Register Stack*
- *Memory Support*
- *Floating Point, Multi-media, 3D Graphics*

Key IA-64 Features

Loop Support*

Register Stack*

Memory Support

Floating Point, Multi-media, 3D Graphics

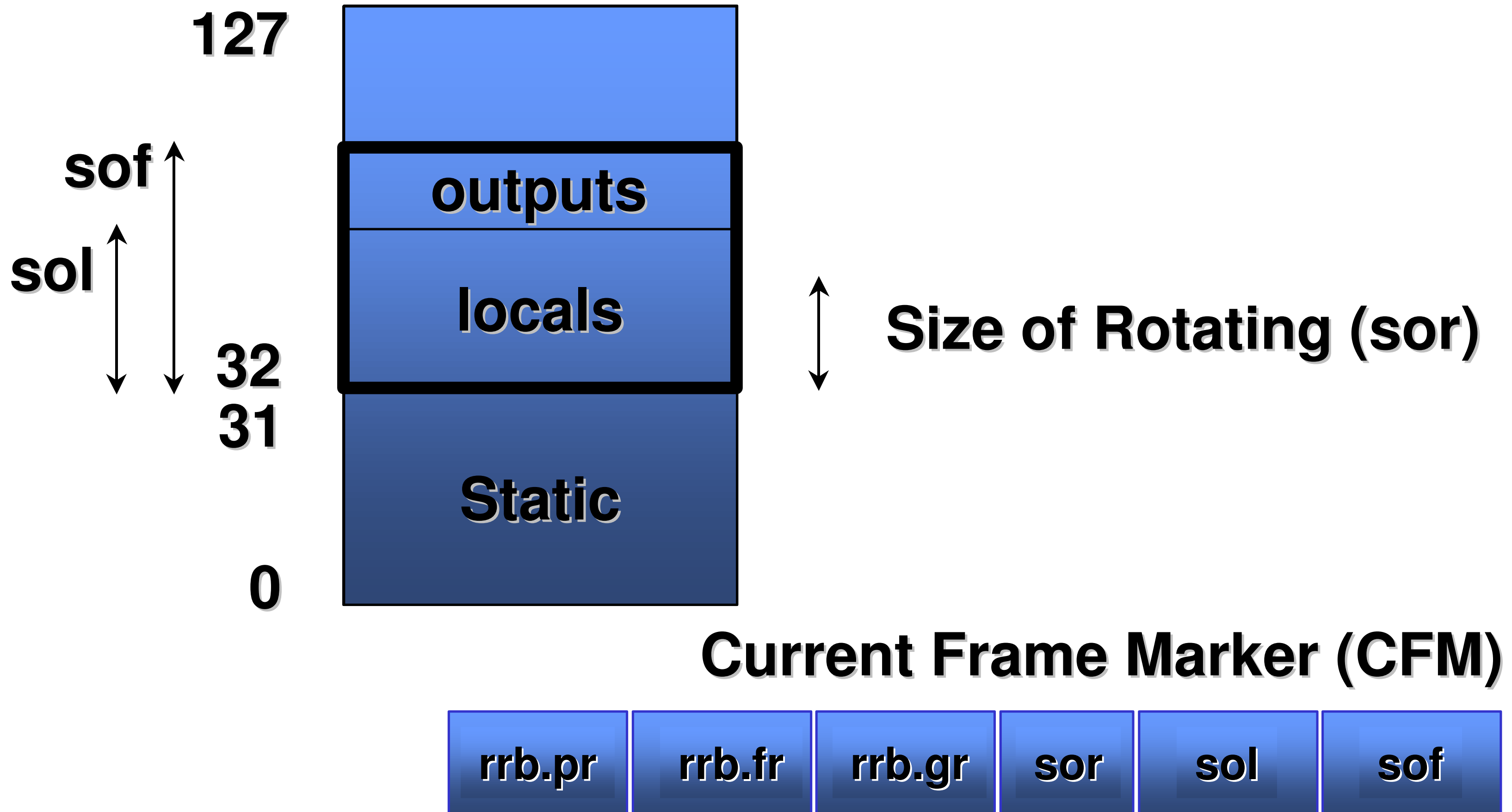
* Some slides provided by Dale Morris, HP Cupertino

Register Rotation

Motivation:

- pipeline-schedule loops onto HW
- remove extraneous work from loop
- minimize start-up overhead
- small code footprint
- maximum computational throughput with few instructions

GR Stack Frame w/ Rotation



GR Rotation

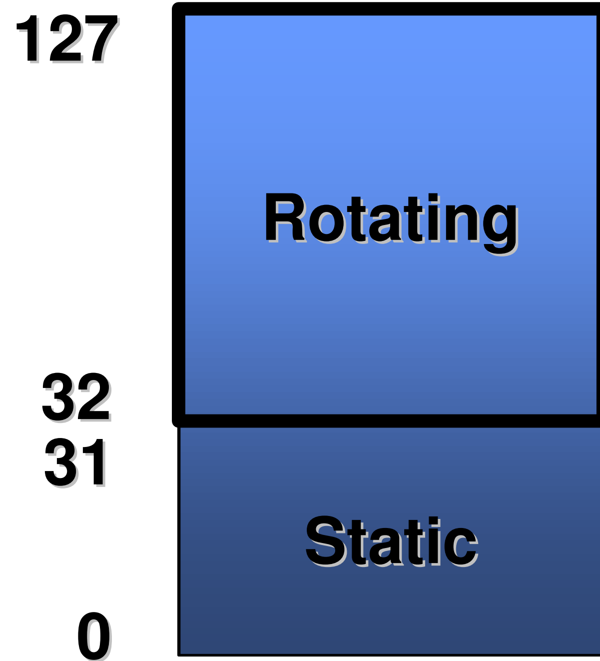
Size of rotating region multiple of 8

Rotating region overlays current frame

–

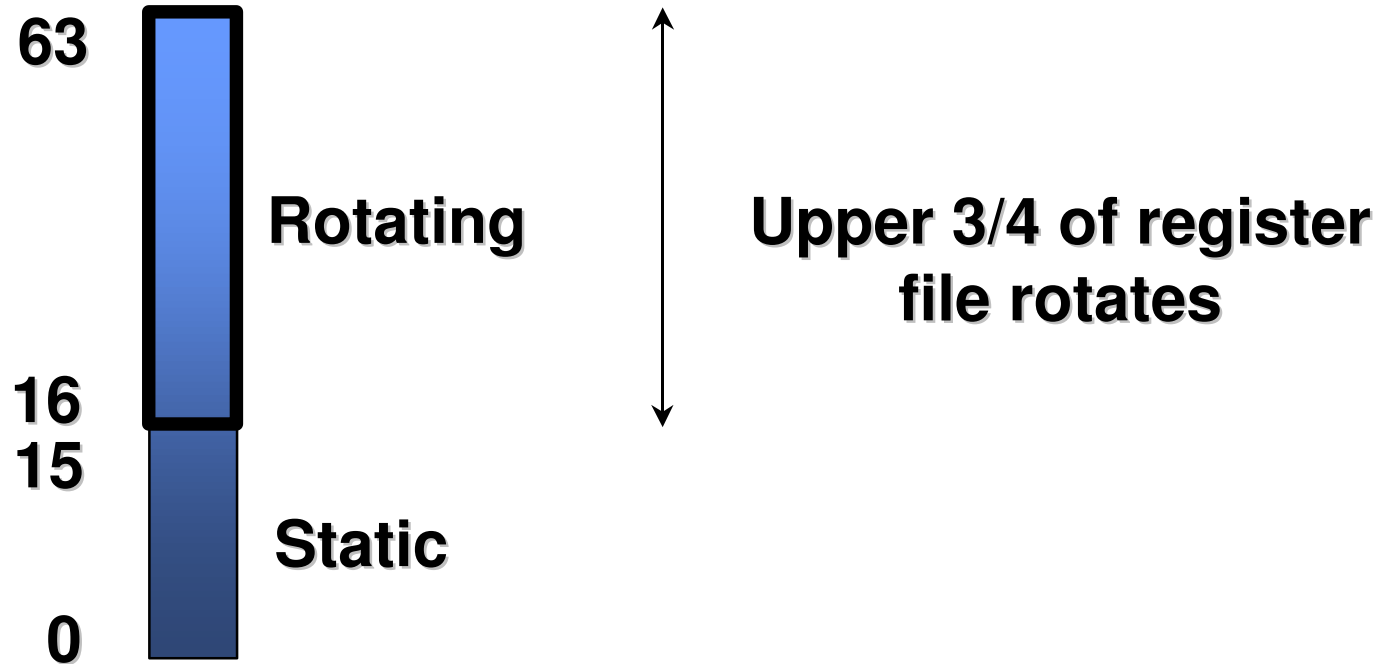
– Overlay allows rotation & stack renaming in a single level of

– Must copy input registers before loop



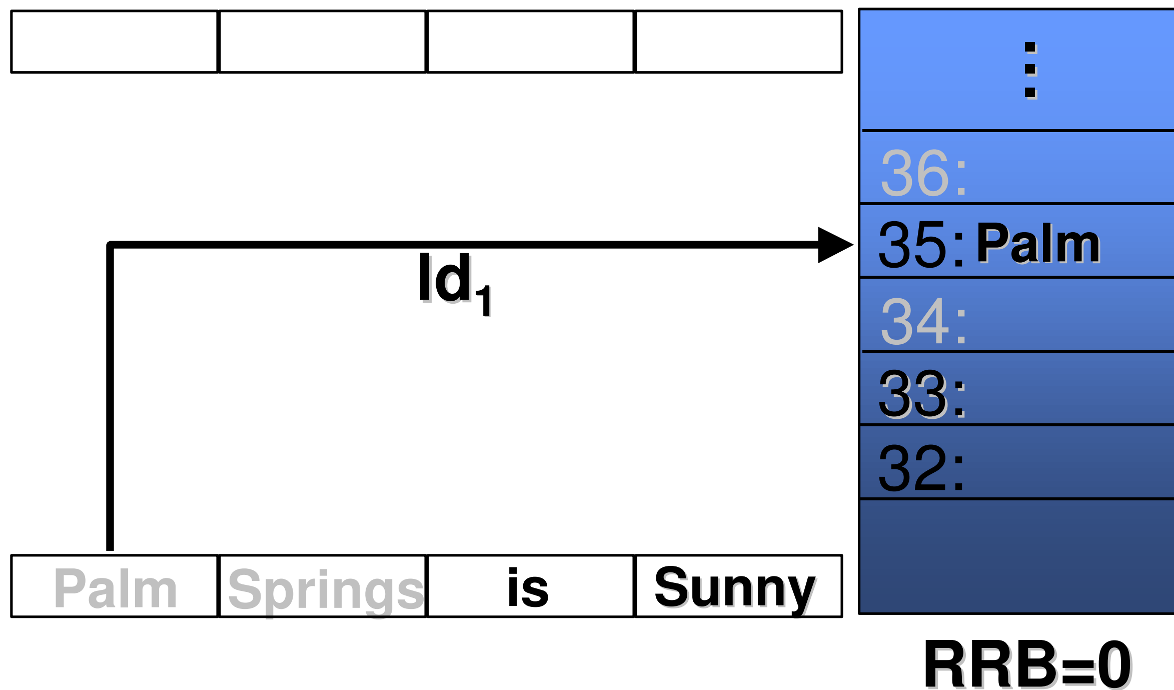
Upper 3/4 of register file rotates

Predicate Rotation



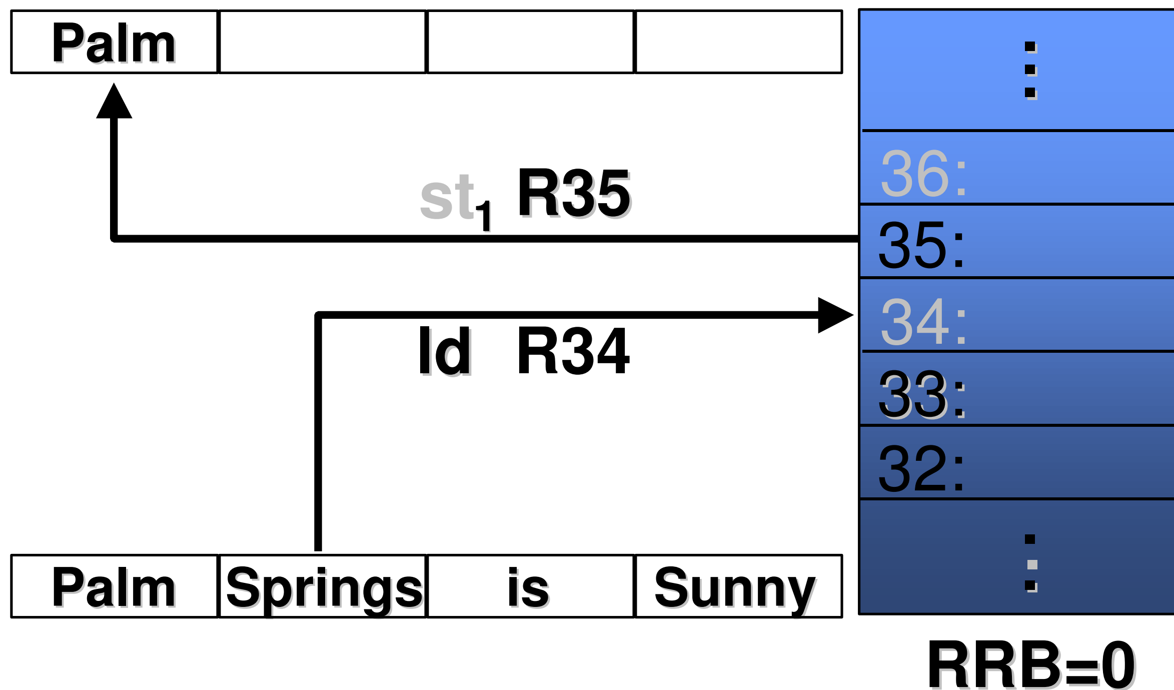
Register Rotation & RRB

Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a "virtual" register number
–



Register Rotation & RRB

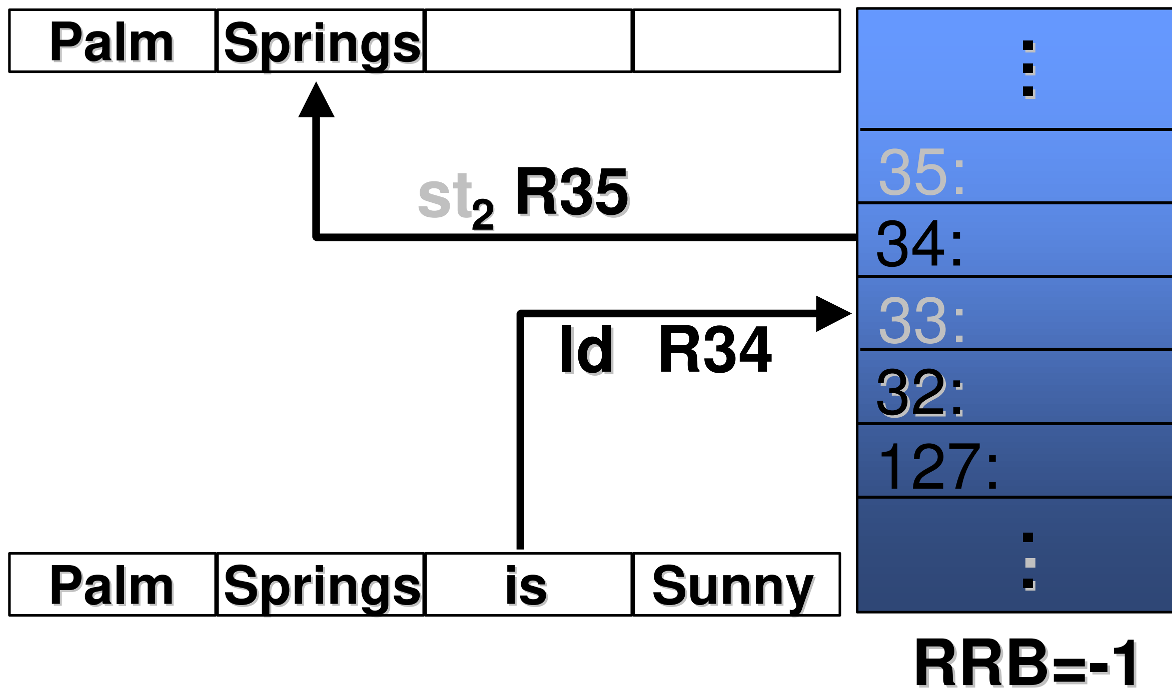
Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number
–



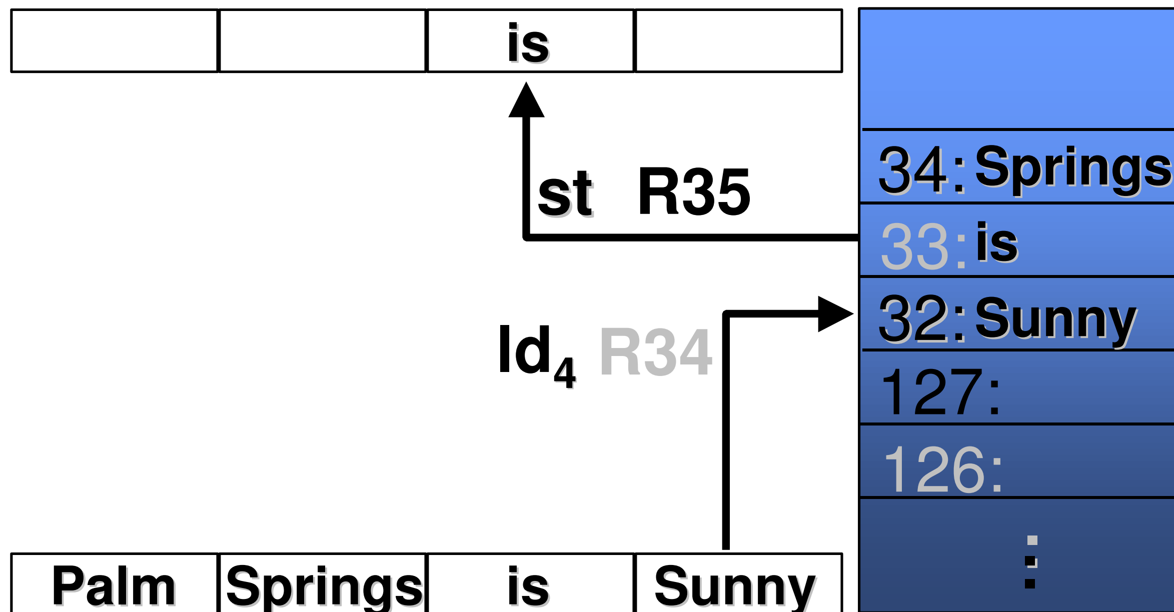
Register Rotation & RRB

Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number

–

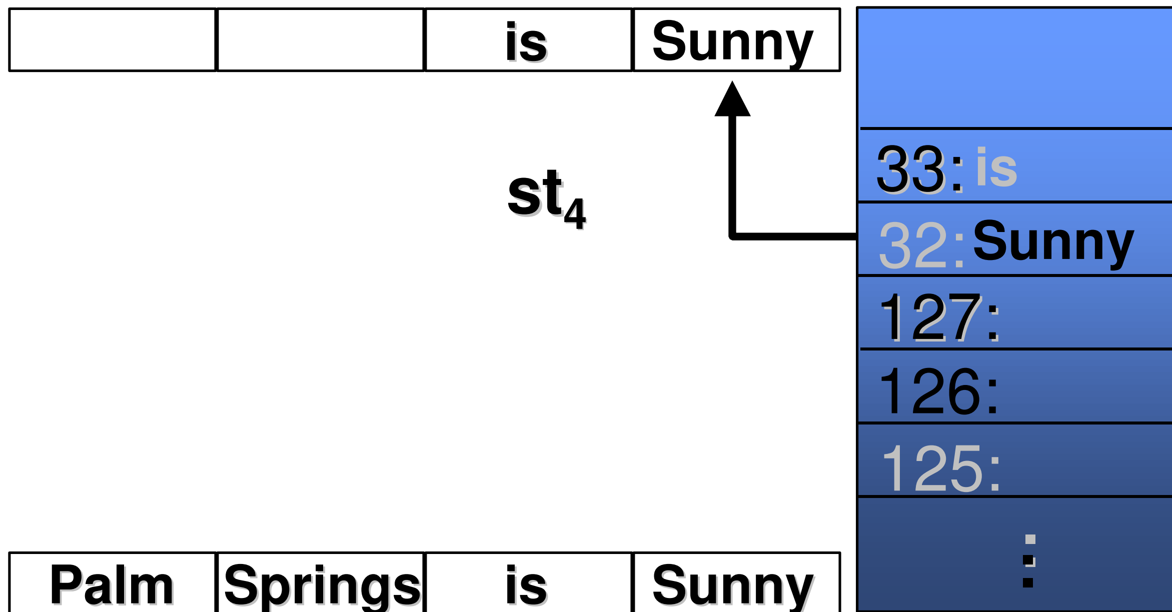


- Separate Rotating Register Base for each: GRs, FRs, PRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a “virtual” register number
  - RRB + virtual register number = physical register number.



Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number

- RRB + virtual register number = physical register number.



Loop Branches

br.cloop uses LC for simple, non-pipelined loops

- decrements LC and loops until LC is 0

br.ctop uses LC and EC for pipelined counted loops

br.wtop uses branch predicate and EC for pipelined "while" loops

br.cexit, br.wexit used for unrolled, pipelined loops

br.ctop

Function (simplified):

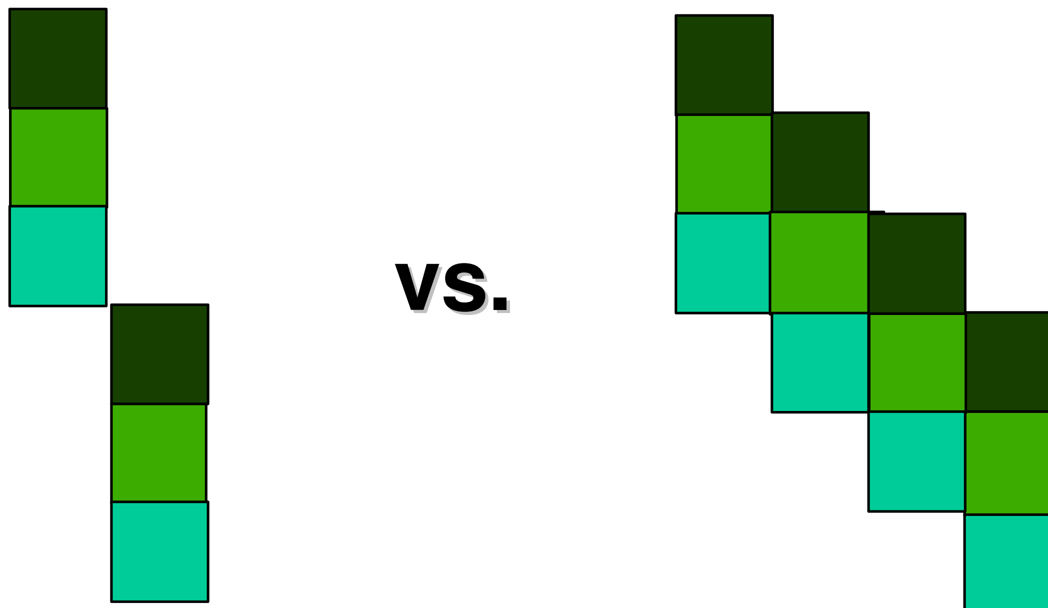
```
• if (LC>0) {
        LC--;    pr[63]=1;    rrb--;    loop;}
  else if (EC>0) {
        EC--;    pr[63]=0;    rrb--;    loop;}
  else
        fall_through;
```

LC counts main loop iterations

EC counts pipeline stages for drain

Software Pipelining

Overlapping execution of different loop iterations



More iterations in same amount of time

Software Pipelining

Synergistic use of IA-64 features:

- Full Predication
- Special branches
- Register rotation: removes loop copy overhead
- Predicate rotation: removes prologue & epilogue

Traditional architectures use loop unrolling

- High overhead: extra code for loop body, prologue, and epilogue

Especially Useful for Integer Code With Small Number of Loop Iterations

Pipelined Loop Example

DAXPY inner loop

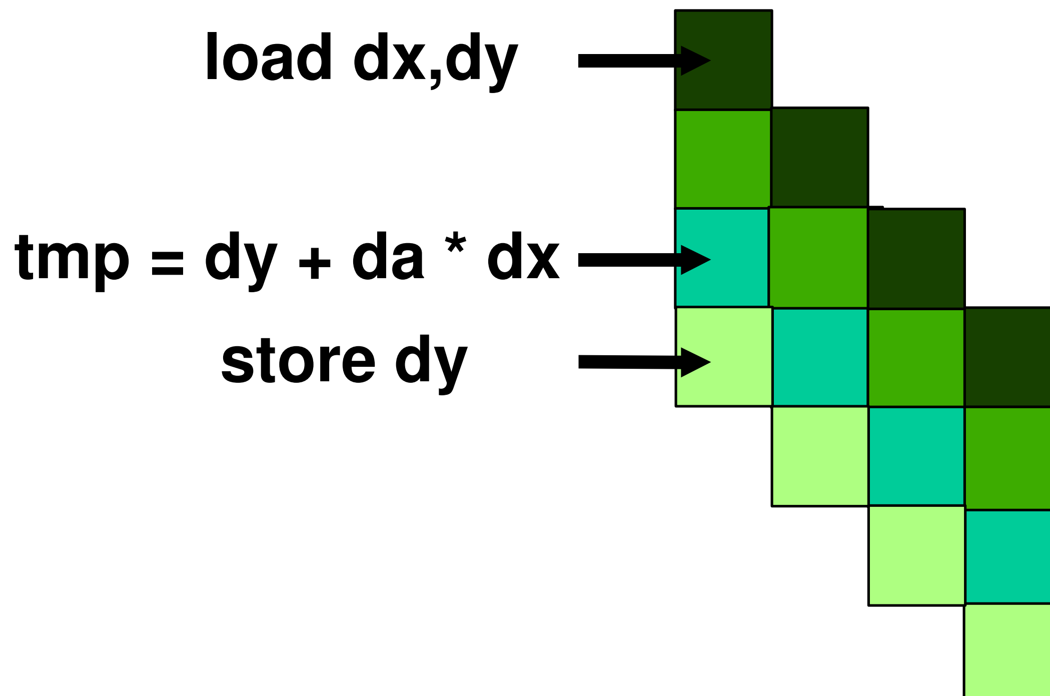
- dy[i] = dy[i] + (da * dx[i])
- 2 loads, 1 fma, 1 store / iteration

Machine assumptions

- can do 2 loads, 1 store, 1 fma, 1 br / cycle
- load latency of 2 clocks
- fma latency of 1 clocks (not realistic, but good for example)

Example: Pipeline

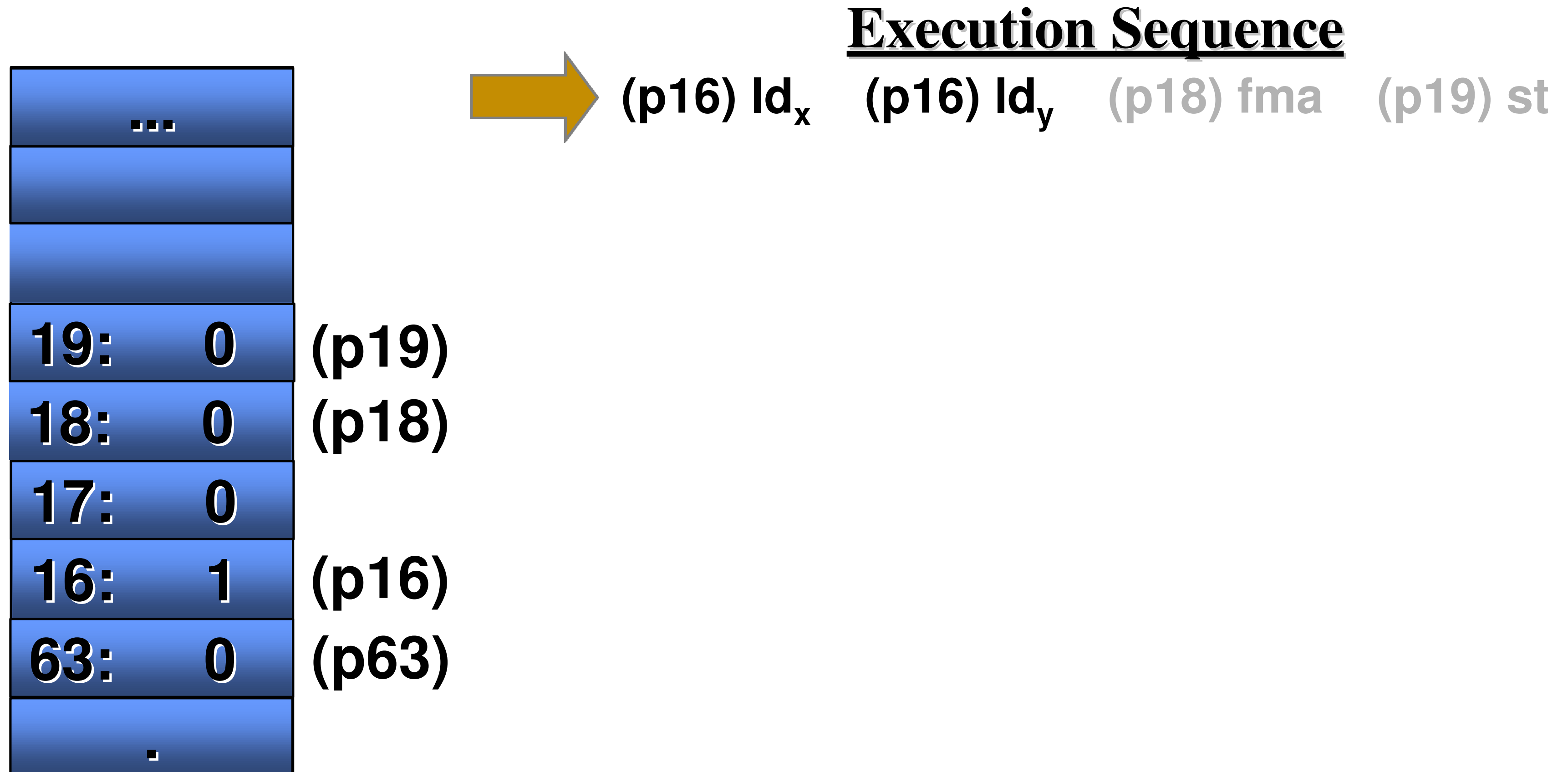
Each column represents 1 source iteration



Example Code

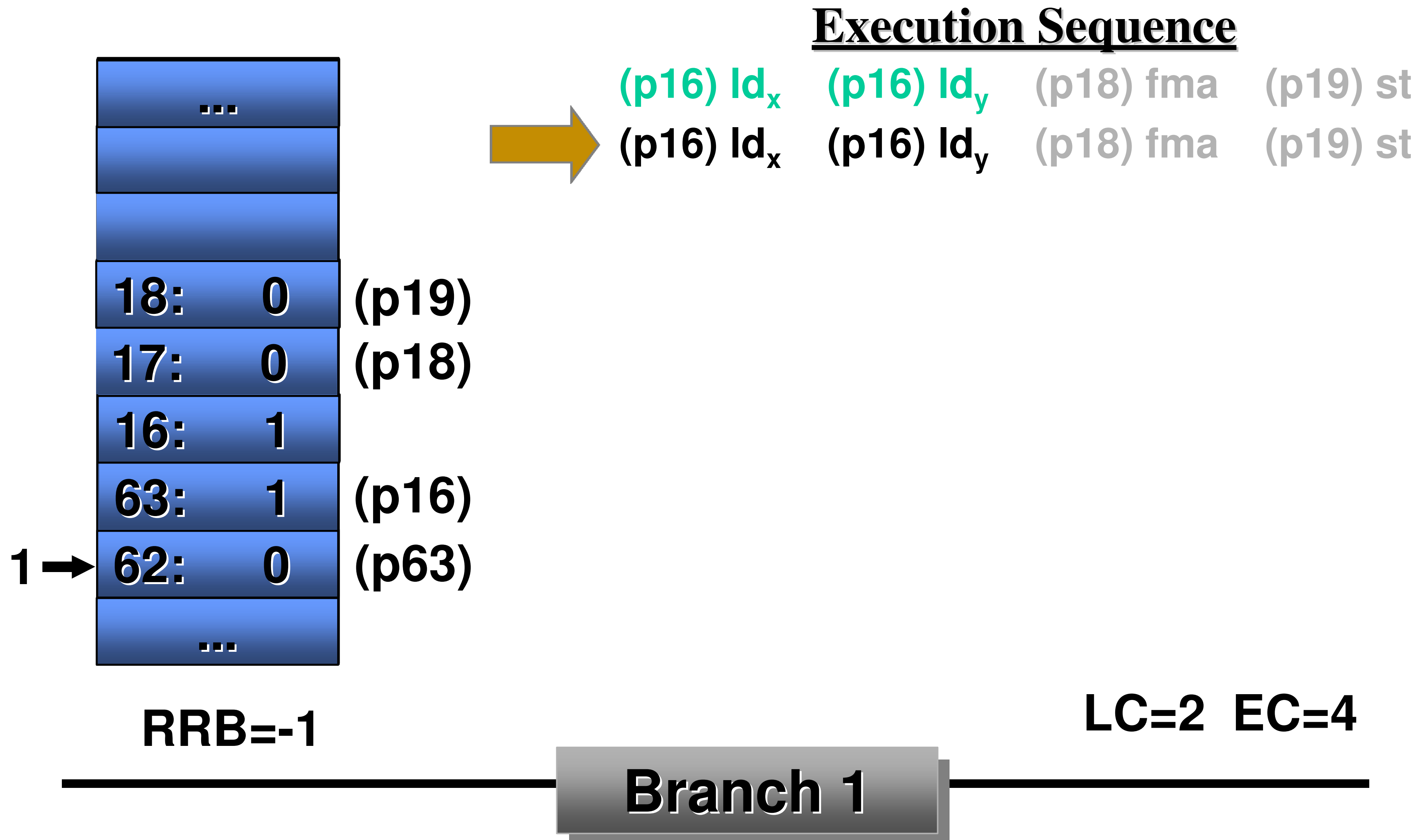
```
.rotf dx[3], dy[3], tmp[2]

        mov     ar.lc = 3               // #iterations-1
        mov     ar.ec = 4               // #stages
        mov     pr.rot = 0x10000
        ;;
looptop:
 (p16) ldfd   dx[0] = [dxsp],8
 (p16) ldfd   dy[0] = [dysp],8
 (p18) fma.d  tmp[0] = da, dx[2], dy[2]
 (p19) stfd  [dydp] = tmp[1],8
        br.ctop looptop
        ;;
```

Loop Execution

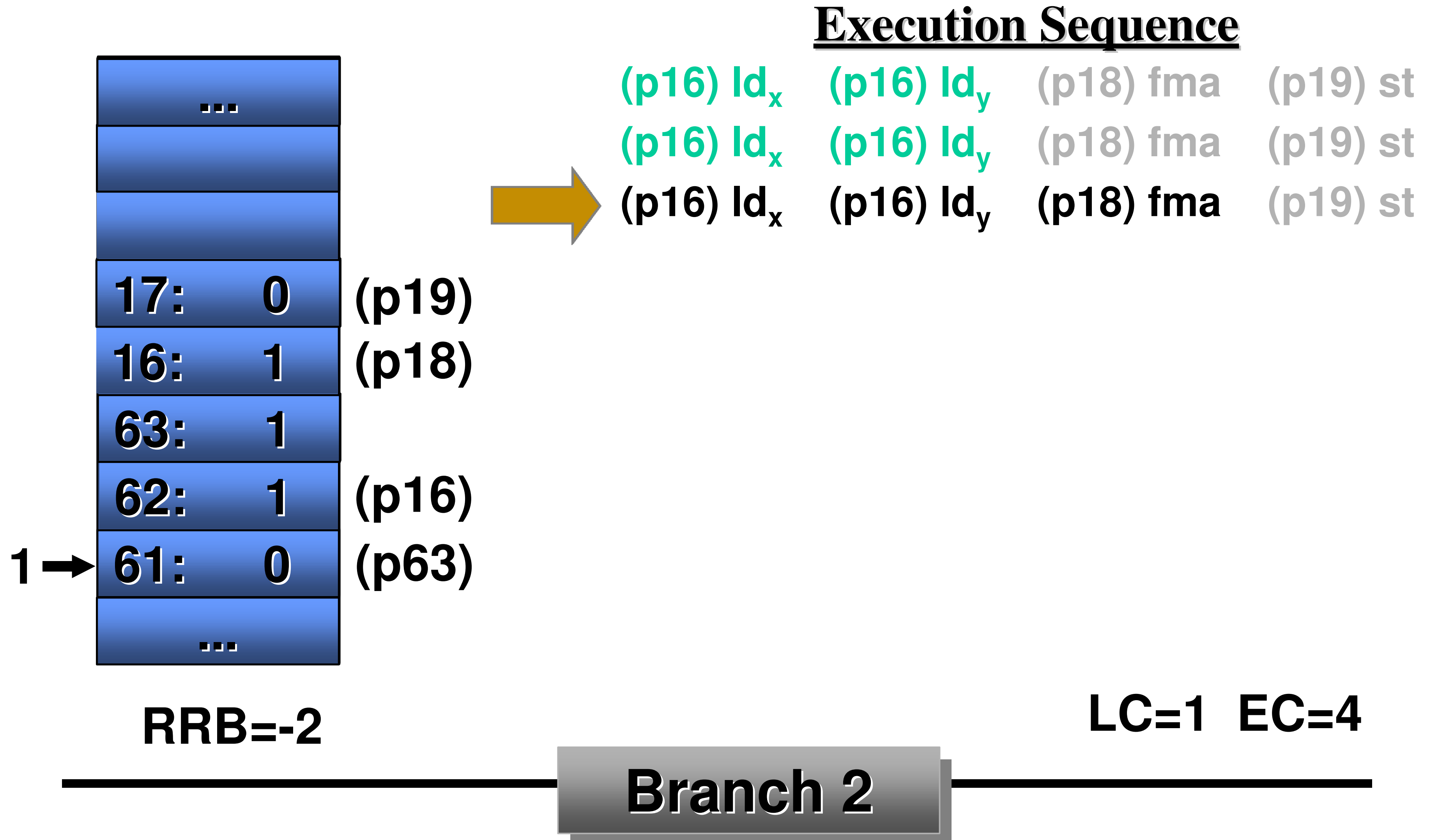


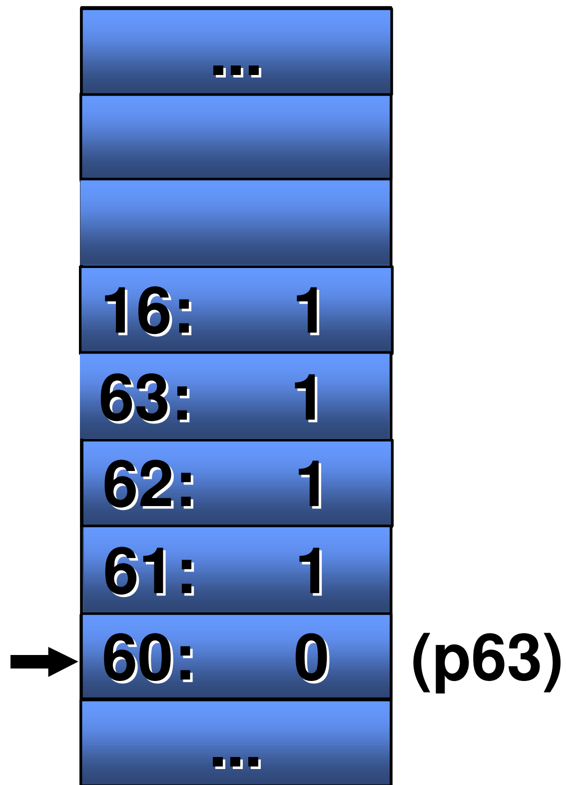
Initialization

Loop Execution



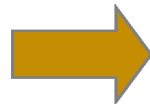
Loop Execution





(p63)

RRB=-3



Execution Sequence

(p16) ld ld$_y$ fma (p19)

(p16) ld ld$_y$ fma (p19)

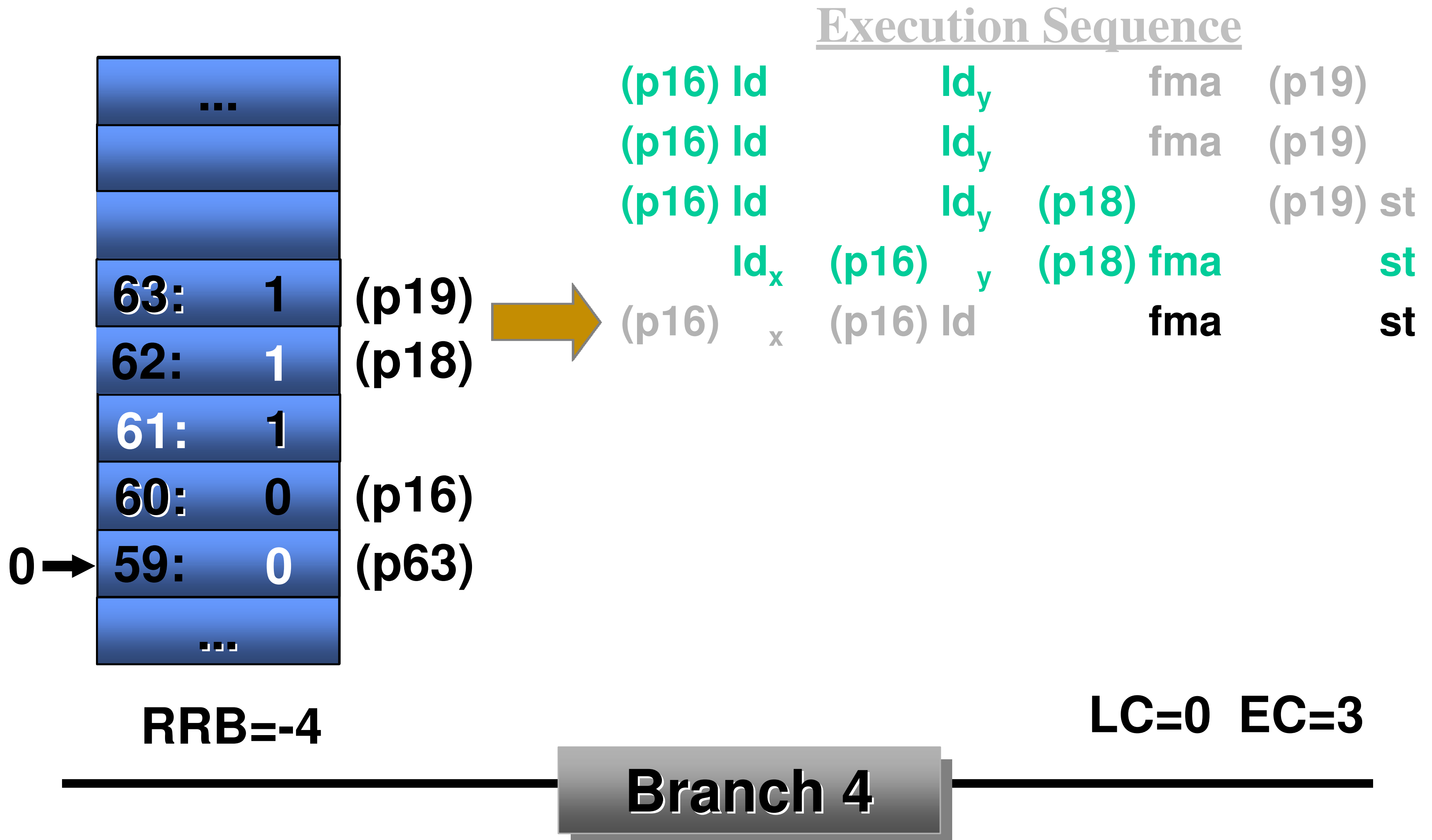
(p16) ld ld$_y$ (p18) (p19) st

ld$_x$ (p16) $_y$ (p18) fma (p19) st

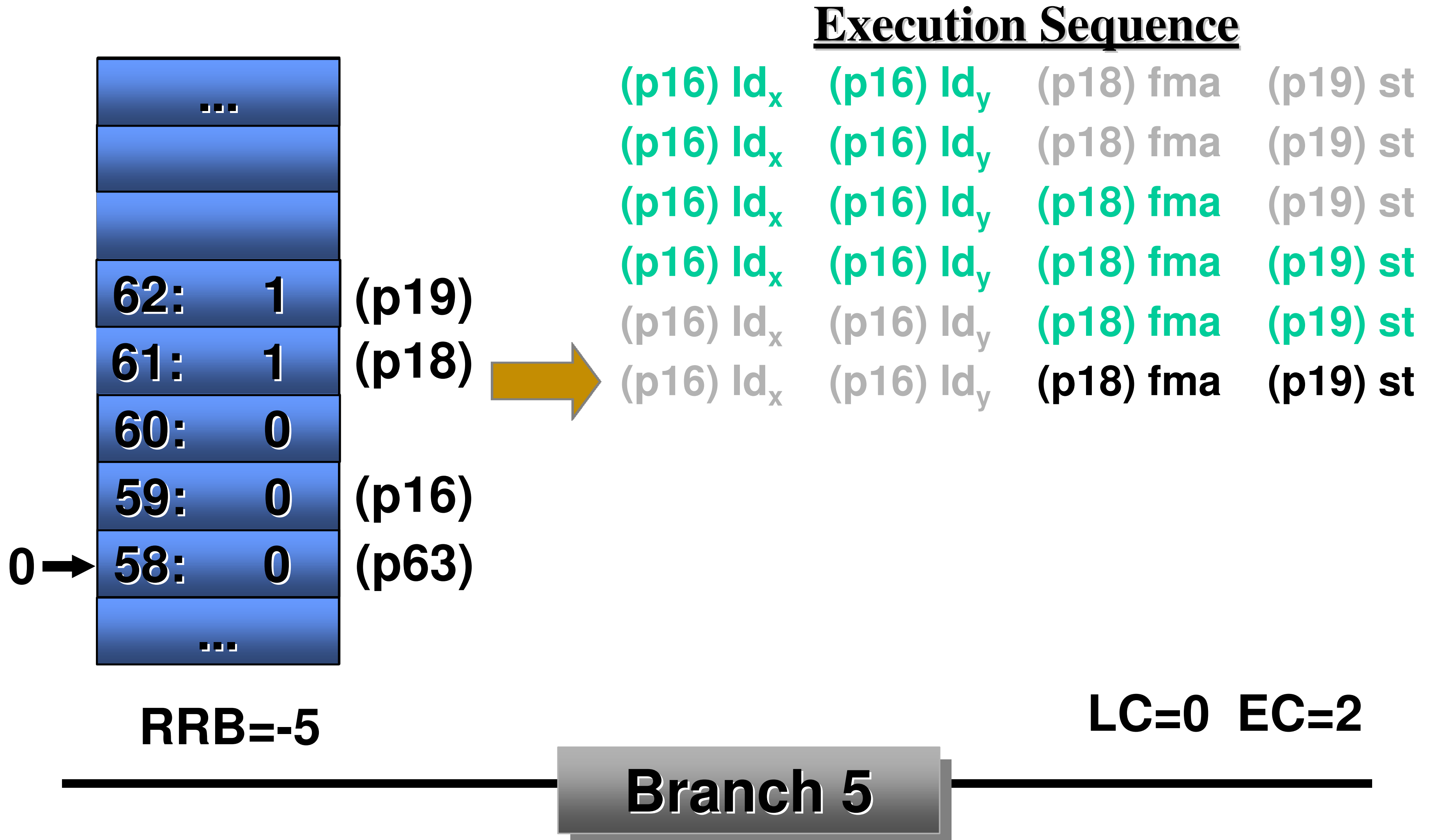
LC=0

Branch 3

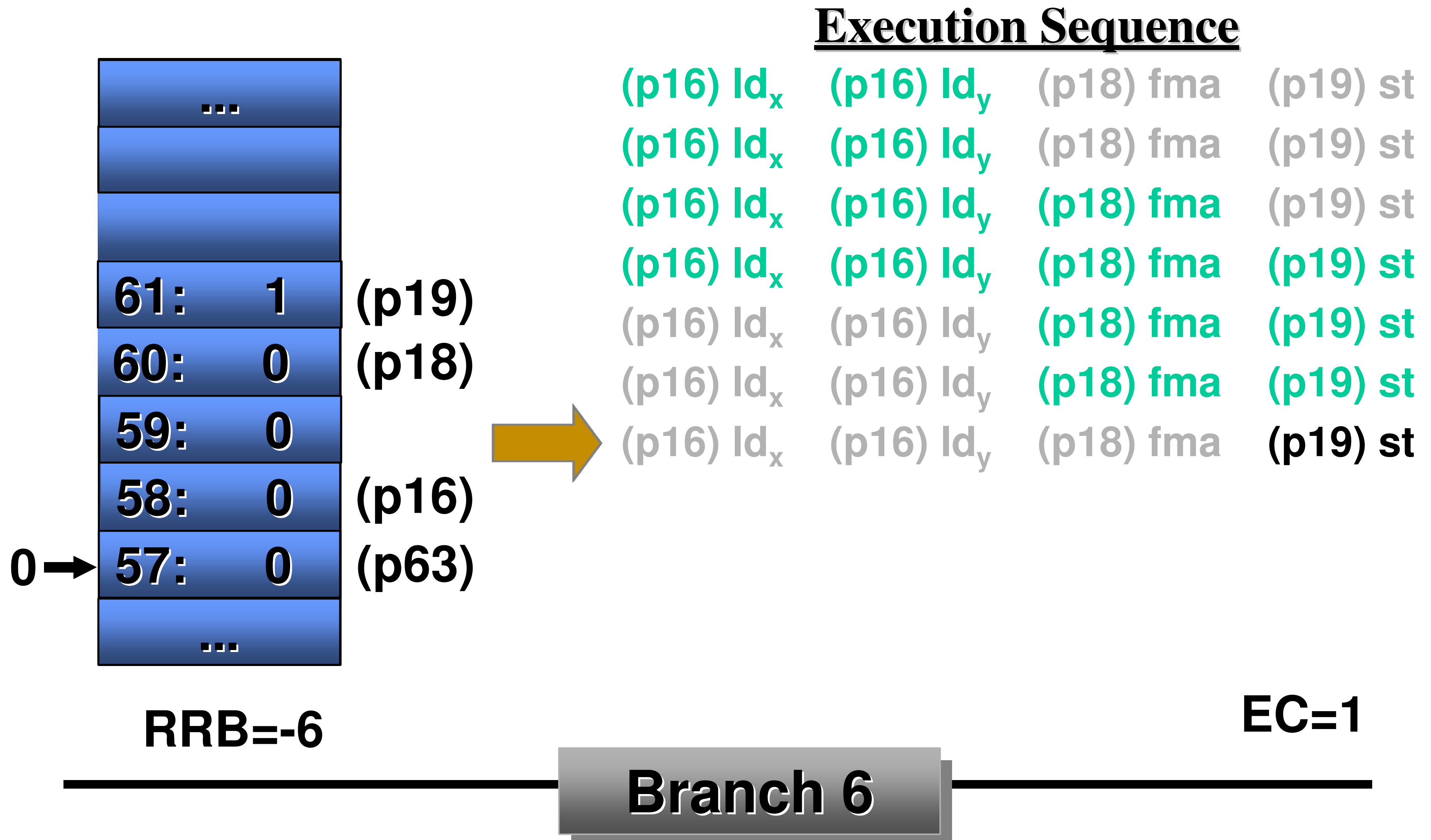
Loop Execution



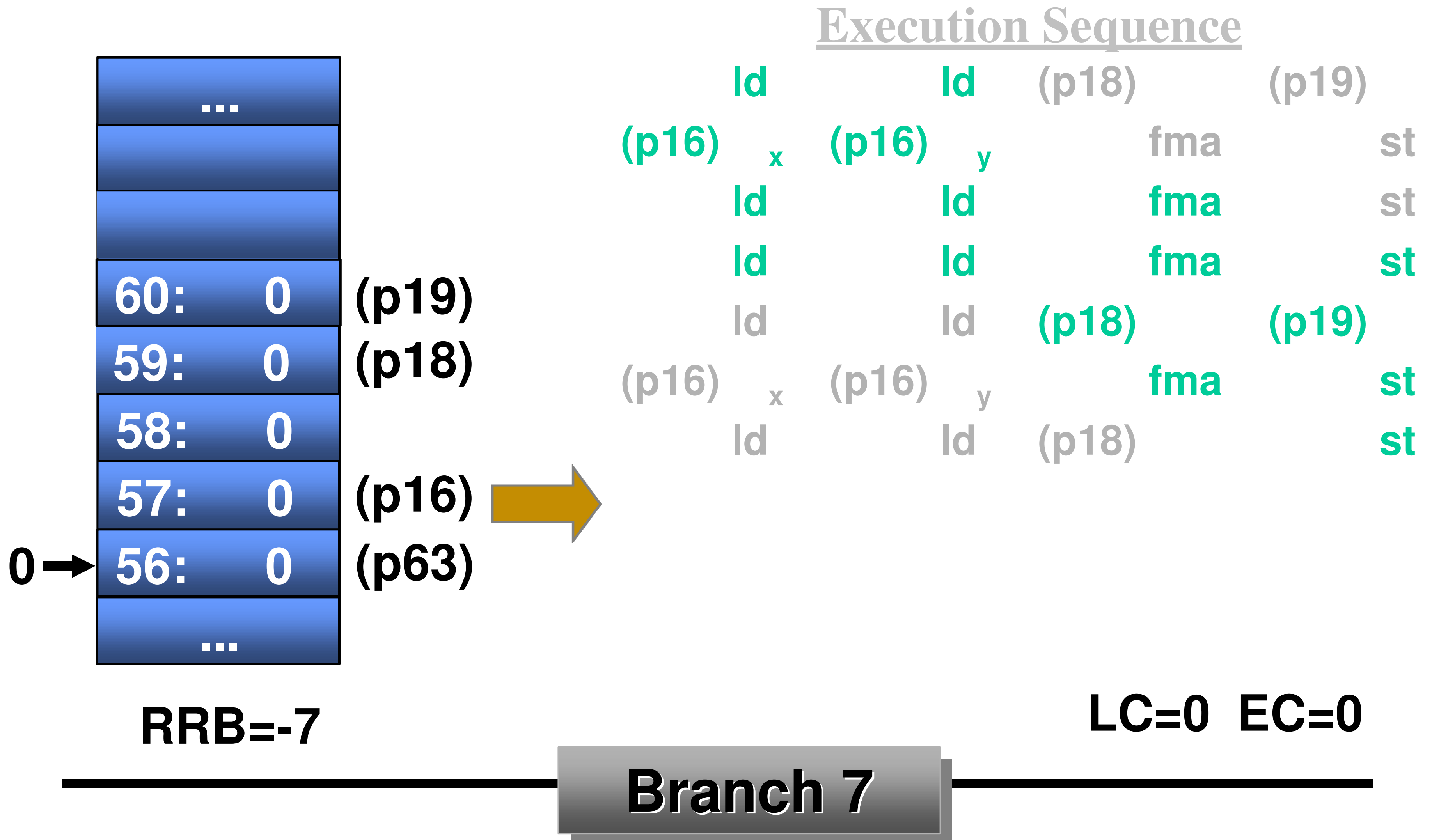
Loop Execution



Loop Execution



Loop Execution



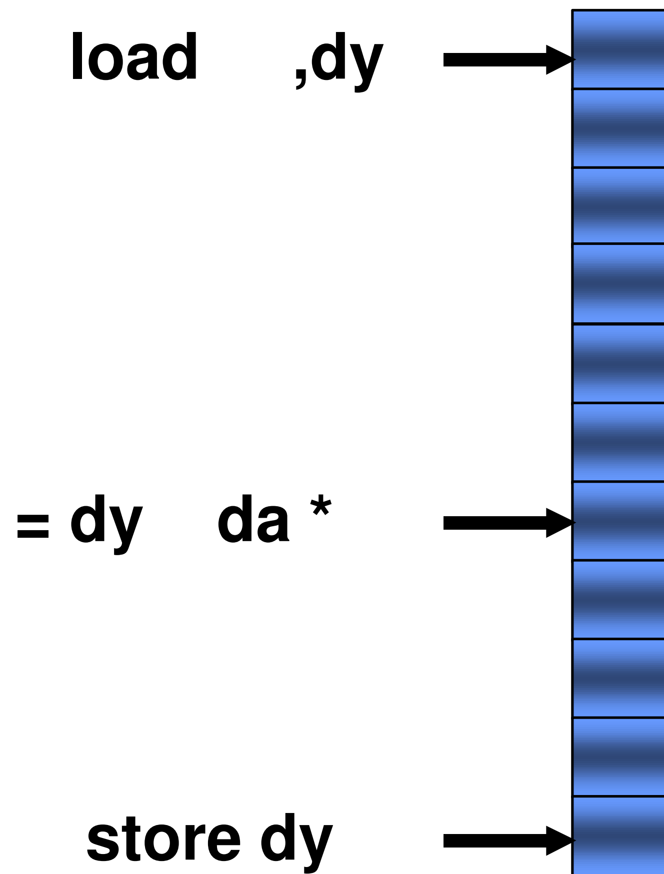
Pipelining & Latency

Suppose we change the latencies

- load latency of 6 clocks
- fma latency of 4 clocks

Example: New Pipeline

Each column represents 1 source iteration



Updated Loop

```
.rotf dx[7], dy[7], tmp[5]

        mov    ar.lc = 3              // #iterations-1
        mov    ar.ec = 11             // #stages
        mov    pr.rot = 0x10000
        ;;
looptop:
 (p16) ldfd   dx[0] = [dxsp],8
 (p16) ldfd   dy[0] = [dysp],8
 (p22) fma.d  tmp[0] = da, dx[6], dy[6]
 (p26) stfd  [dydp] = tmp[4],8
        br.ctop looptop
        ;;
```

Rotation: Summary

Loop pipelining maximizes performance; minimizes overhead

- Avoids code expansion of unrolling and code explosion of prologue and epilogue
- Smaller code means fewer cache misses
- Greater performance improvements in higher latency conditions

Reduced overhead allows S/W pipelining of small loops with unknown trip counts

- Typical of integer scalar codes

Key IA-64 Features

Loop Support*

Register Stack*

Memory Support

Floating Point, Multi-media, 3D Graphics

* Some slides provided by Dale Morris, HP Cupertino

IA-64 Register Model

Stack & Rotation support

SW-visible renaming resources

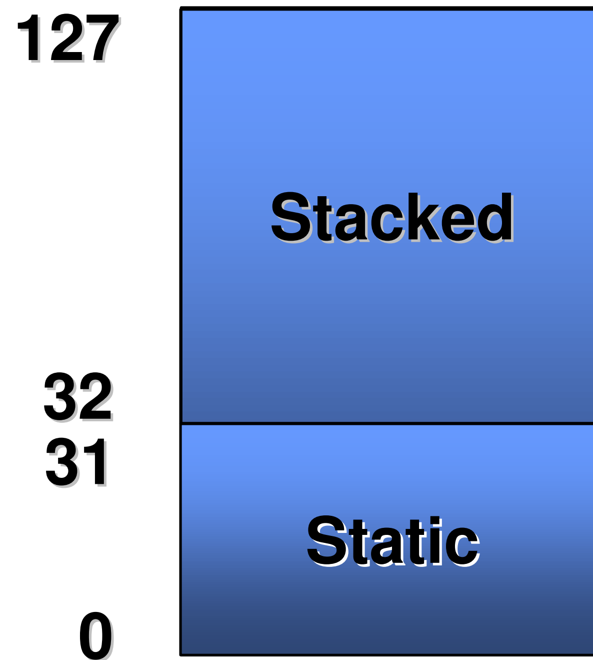
HW simplicity and explicit control

Register Stack

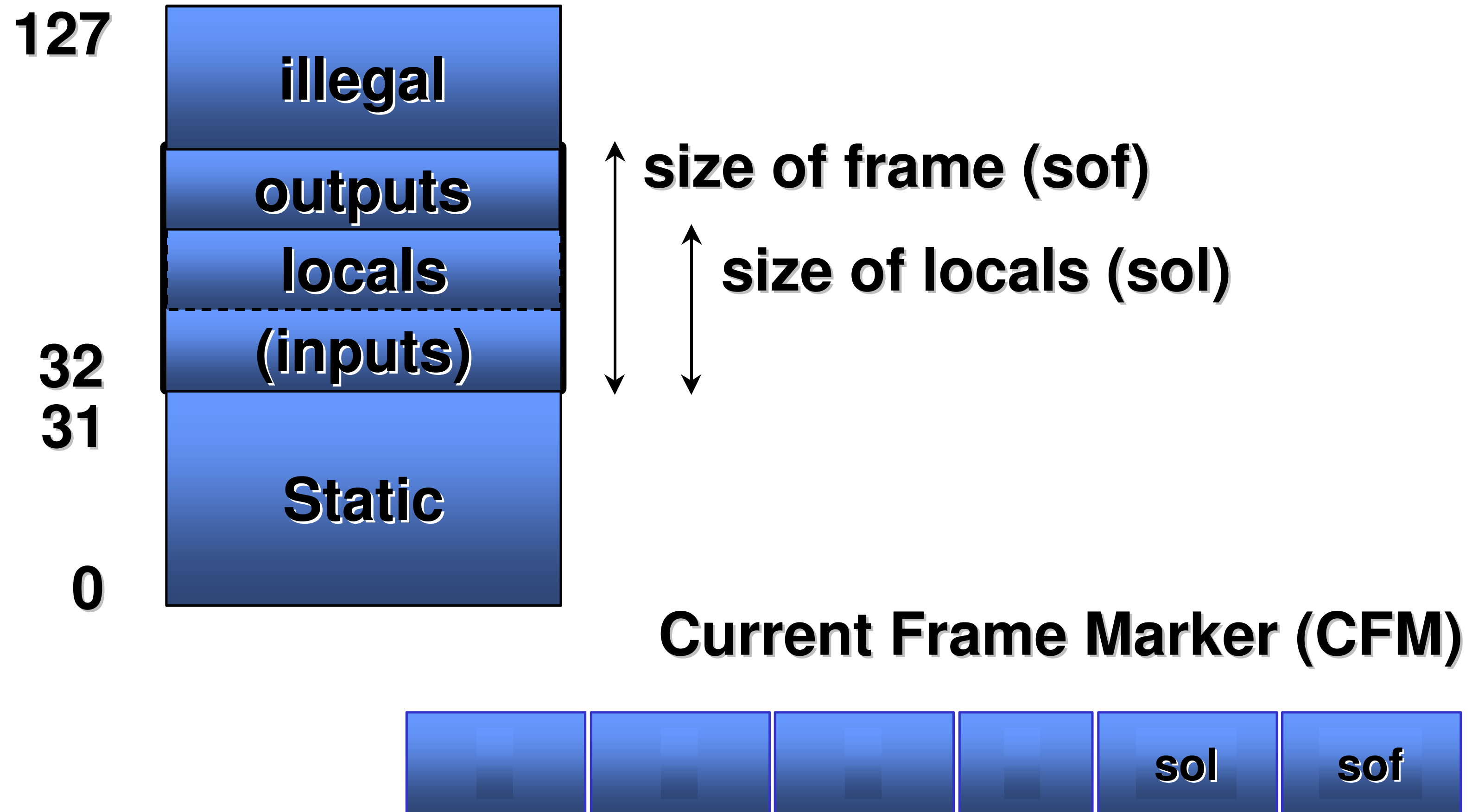
Motivation:

- Automatic save/restore of GRs on procedure call/return
- Cache traffic reduction
- Latency hiding of register spill/fill

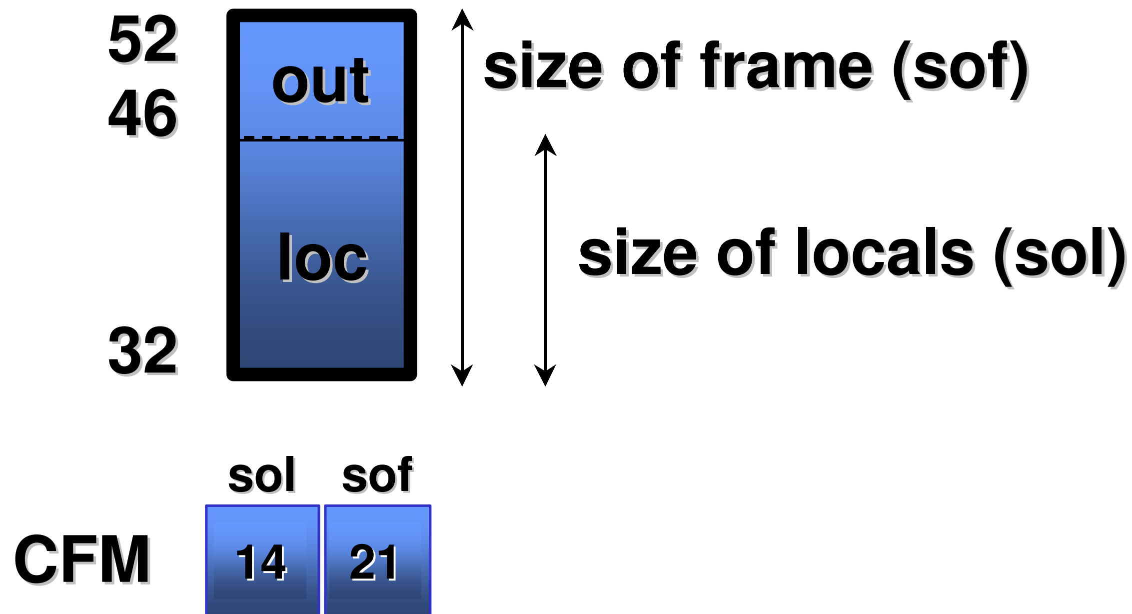
General Registers



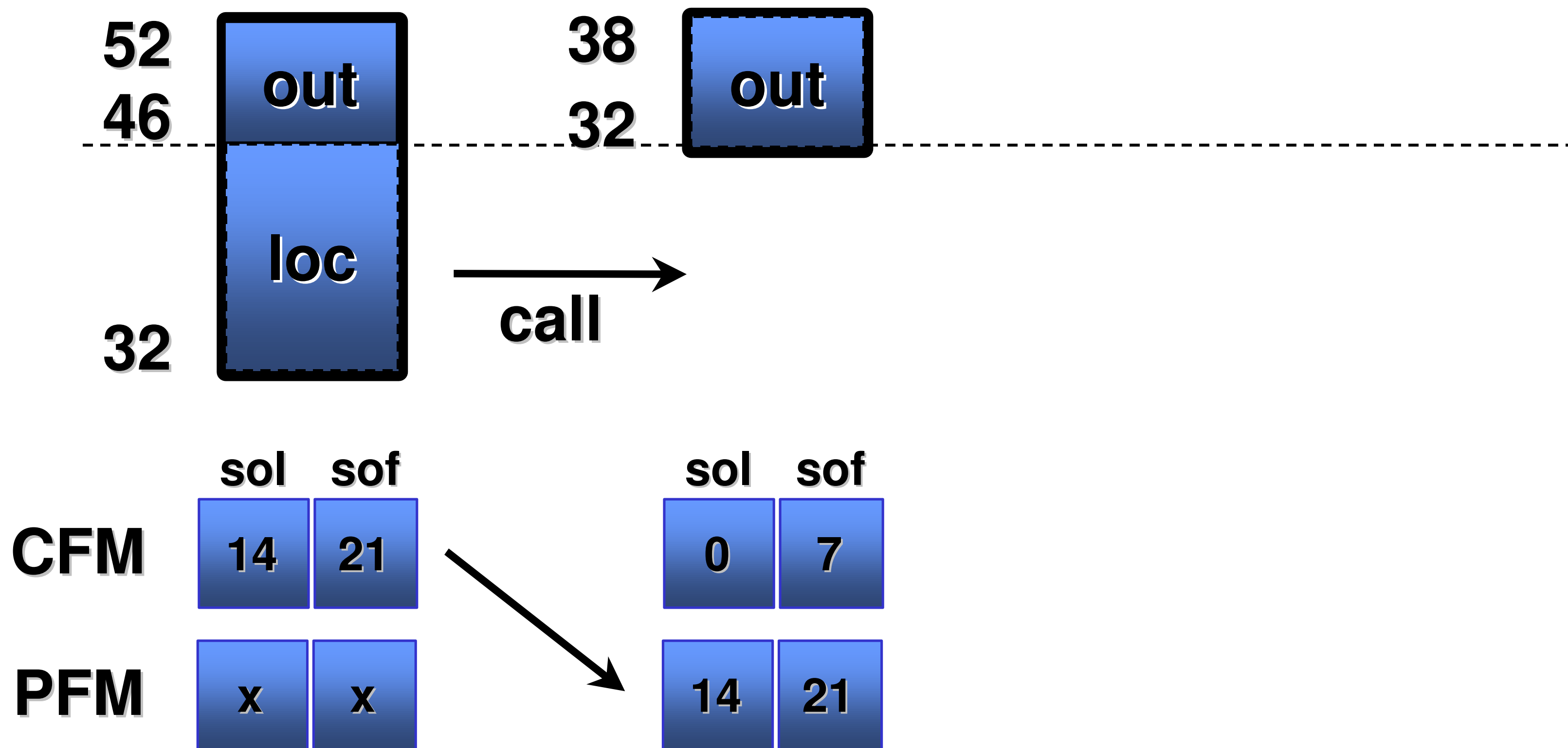
GR Stack Frame



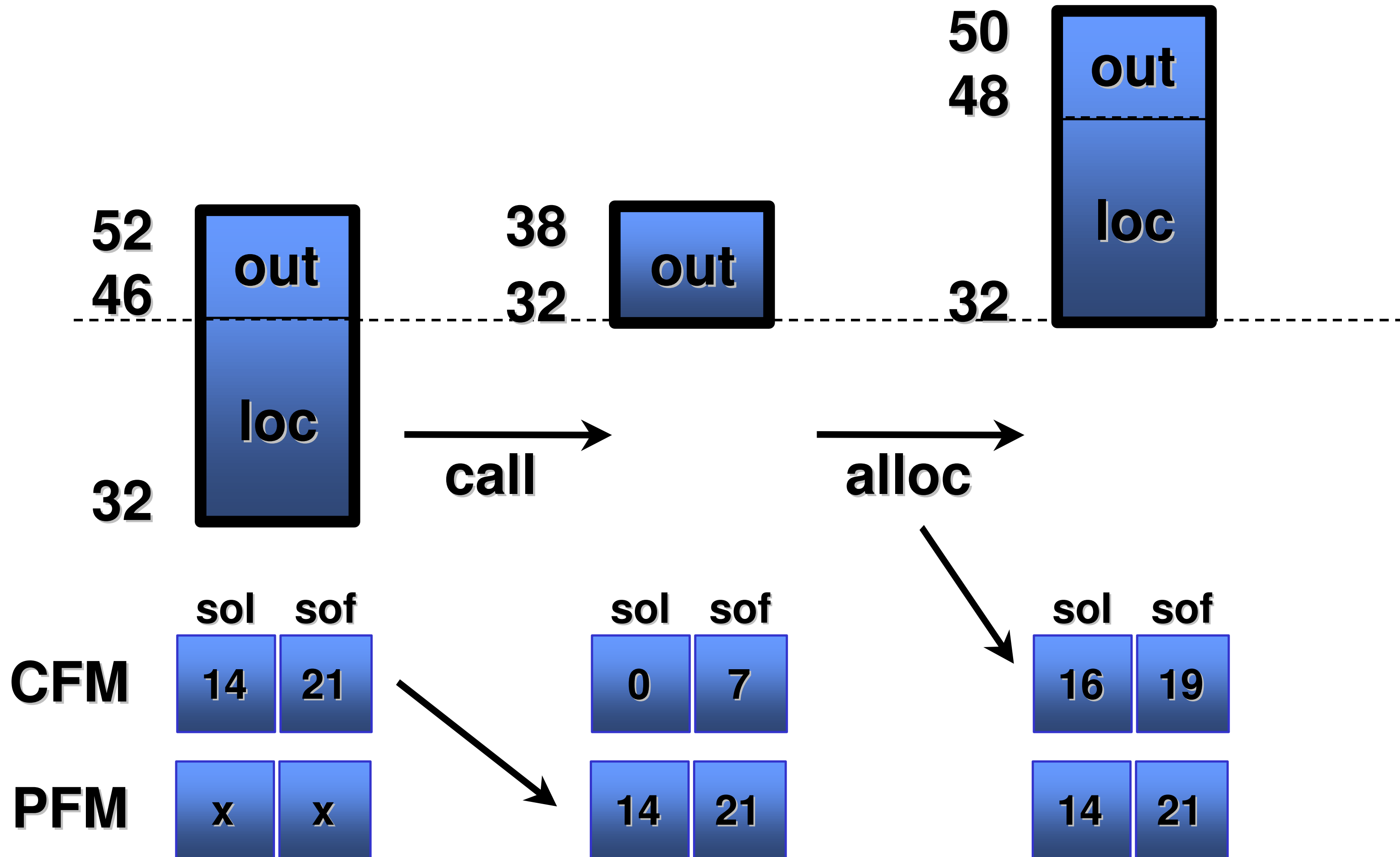
GR Stack Frame - Example



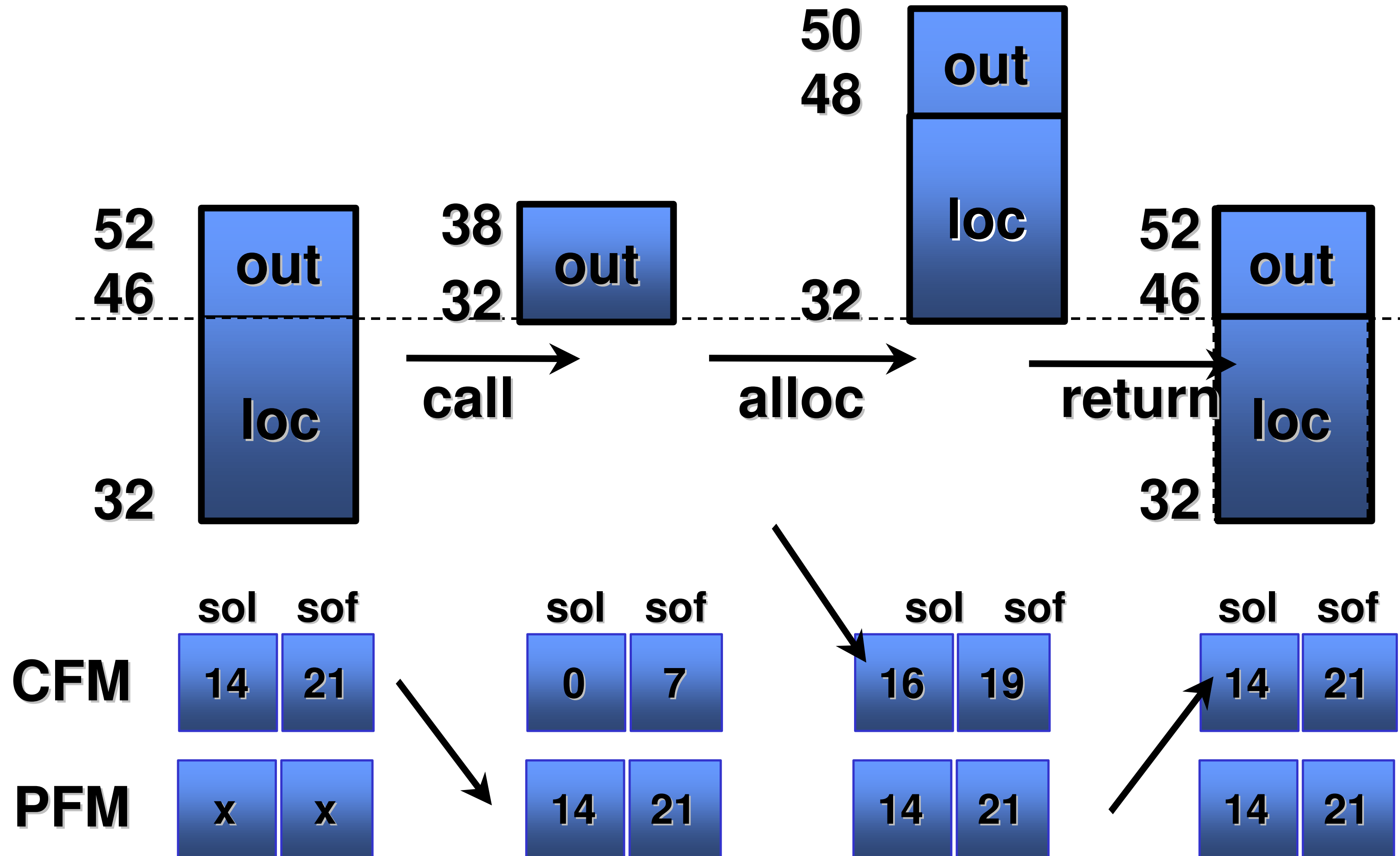
GR Stack Frame - Call



GR Stack Frame - Allocate



GR Stack Frame - Return



Instructions

br.call

- Copies CFM to PFM
- Creates new frame with only output regs
- Saves local regs from previous frame

alloc

- Resizes current frame
- Saves PFM to a GR

Instructions (cont.)

mov to PFS

- Restores PFM from a GR

br.ret

- Restores CFM from PFM
- Restores local regs for previous frame

Key IA-64 Features

Loop Support

Register Stack

Memory Support

Floating Point, Multi-media, 3D Graphics

Memory

- Byte addressable
- Accessed with 64-bit pointers
  - Upper 3-bits is segment id
  - Limited support for 32-bit pointers
- Access granularity and alignment
  - 1,2,4,8,10,16 bytes
  - Alignment on naturally aligned boundaries is recommended
    - Performance penalty may result if not
  - Instructions are always 16-byte aligned
- Accessed big or little endian byte order
- 32-bit virtual addressing support

Memory Hierarchy Control

Explicit control of cache allocation and deallocation

- Specify levels of the memory hierarchy affected by the access
- Allocation and Flush resolution is at least 32-bytes

Allocation

- Allocation hints indicate at which level allocation takes place
  - But always implies bringing the data close to the CPU
- Used in load, store, and explicit prefetch instructions

Deallocation and Flush

- Invalidates the addressed line in all levels of cache hierarchy
- Write data back to memory if necessary

Key IA-64 Features

Loop Support

Register Stack

Memory Support

Floating Point, Multi-media, 3D Graphics

Floating-point Architecture

- IEEE 754 compliant
- Single, double, double extended (80-bit)
- Canonical representation in 82-bit FP registers
- Multiply-add instruction
- 128 floating-point registers
  - Rotating, not stacking
- Load double/single pair
- Multiple FP status registers for speculation

Multimedia Support

Audio and video functions typically perform the same operation on arrays of data values

IA-64 defines a set of instructions to treat general register's as 8x8, 4x16, or 2x32 bit elements

- Three major types of instructions are defined:
  - Addition and subtraction (including special purpose forms)
  - Left shift, signed and unsigned right shift
  - Pack/Unpack; converts between different element sizes.

Semantically compatible with IA-32's MMX Technology

Parallel FP Support

- Enable Cost-effective 3D Graphics platforms
- Exploit data parallelism in applications using 32-bit floating-point data
  - Most applications and geometry calculations (transforms and lighting) are done with 32-bit floating-point numbers
  - Provides 2X increase in computation resources for 32-bit data parallel floating-point operations
- Floating-point Registers treated as 2x32 bit single precision elements
  - Full IEEE compliance
    - single, double, double-extended data types, *packed-64*
  - similar instructions as for scalar floating-point
  - availability of fast divide (non IEEE)