Lecture 17
Compiler-Architecture Interaction: EPIC Architecture

Outline

- Basic Compiler Structure
- Very Long Instruction Word Architectures
- Explicitly Parallel Instruction Computing

© W. W. Hwu and S. J. Patel, 2002

Objectives with Instruction Scheduling

- To reorder the code within a region of code such that it executes correctly in minimal time.
- … and with minimal support from the HW
- … but the HW is wide and deep.

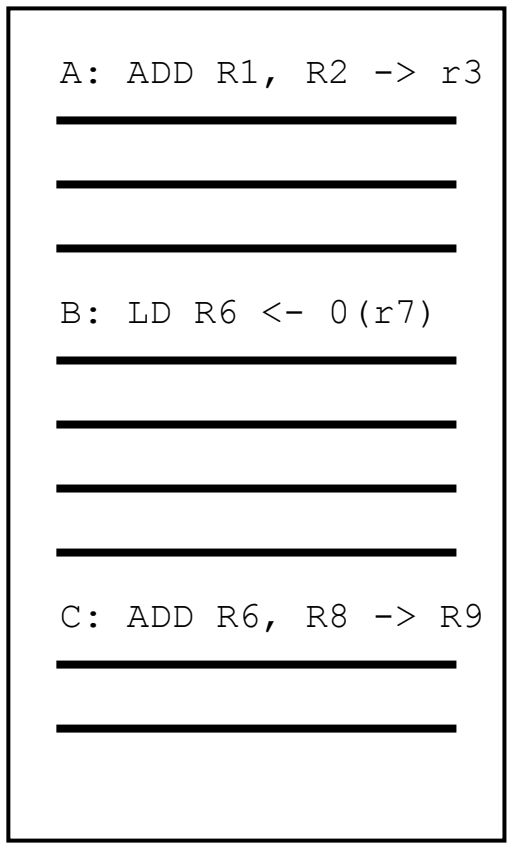
© W. W. Hwu and S. J. Patel, 2002

Requirements for effective static scheduling

- Obviously, a good compiler
- Good profiling mechanisms
  - Effective techniques for doing this are open
- Lots of architectural registers

© W. W. Hwu and S. J. Patel, 2002

Scheduling within a region

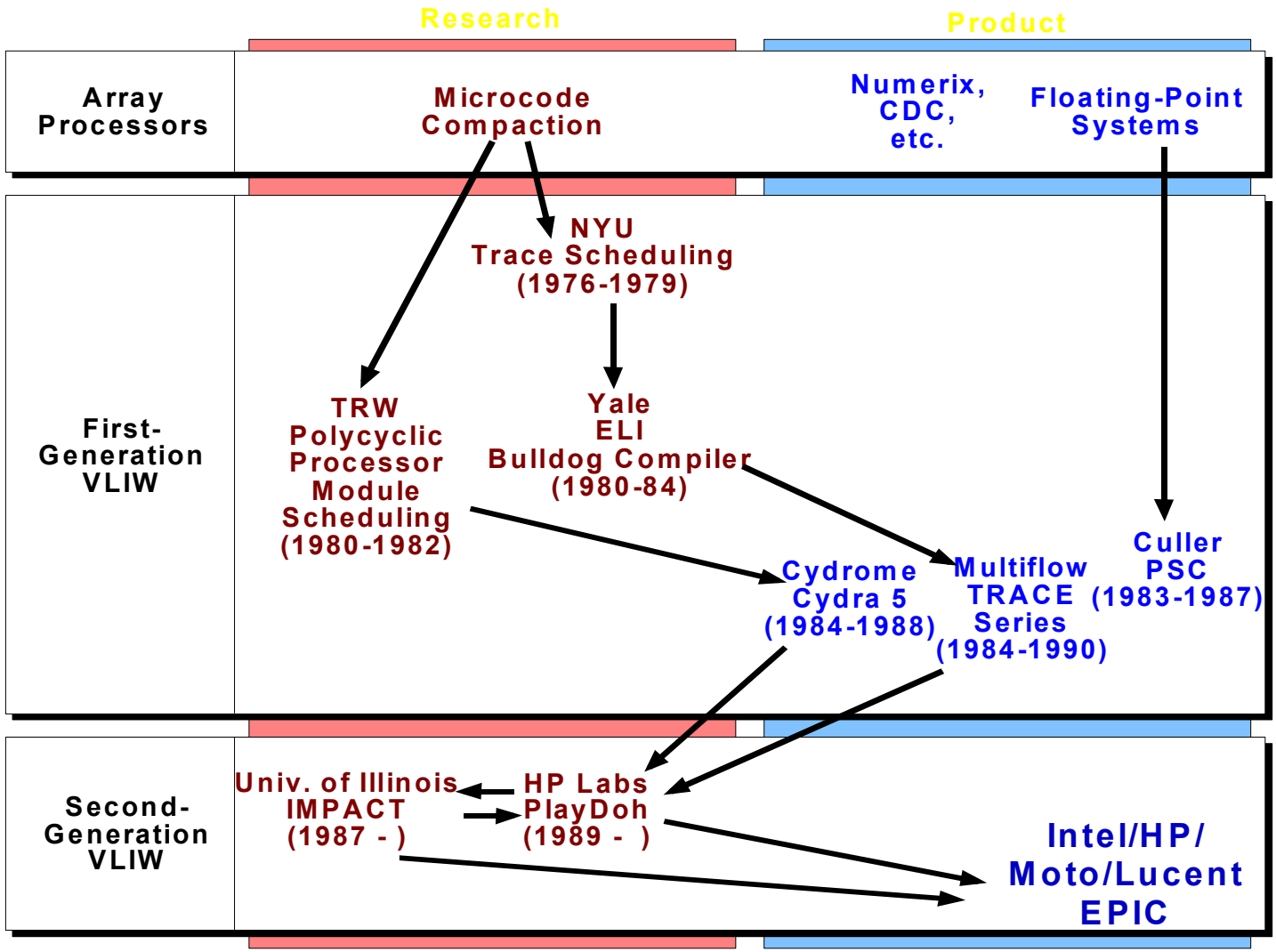


Question: When instruction A be scheduled?

Question: When can instruction C be scheduled?

© W. W. Hwu and S. J. Patel, 2002

Evolution of VLIW/EPIC



Source: Bob Rau

© W. W. Hwu and S. J. Patel, 2002
ECE 412, University of Illinois

What are the shortcomings of VLIW?

- Wasteful encoding with nops
- No compatibility across generations
- Even with block-coalescing techniques such as trace scheduling – moving instructions around can be impaired.
- Hard to create big regions
- Hard to move instructions across branches

© W. W. Hwu and S. J. Patel, 2002

Four Key Ideas of EPIC

- Bundles and templates
- (Unit assumed latency with wide words)
- Control Speculation
- Data Dependence Speculation
- Predication

© W. W. Hwu and S. J. Patel, 2002

VLIW Code Size Changes with Schedule

IALU	IALU	FPAdd	FPMul	Load	Store	Cmpp	Br
A	*nop*	B	*nop*	C	D	*nop*	*nop*
nop	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*
E	F	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*
G	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*	H

E, F dependent on C, C takes 2 cycles

256 bytes total

Load latency increases, one less IALU

IALU	FPAdd	FPMul	Load	Store	Cmpp	Br
A	B	*nop*	C	D	*nop*	*nop*
nop	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*
nop	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*
E	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*
F	*nop*	*nop*	*nop*	*nop*	*nop*	*nop*
G	*nop*	*nop*	*nop*	*nop*	*nop*	H

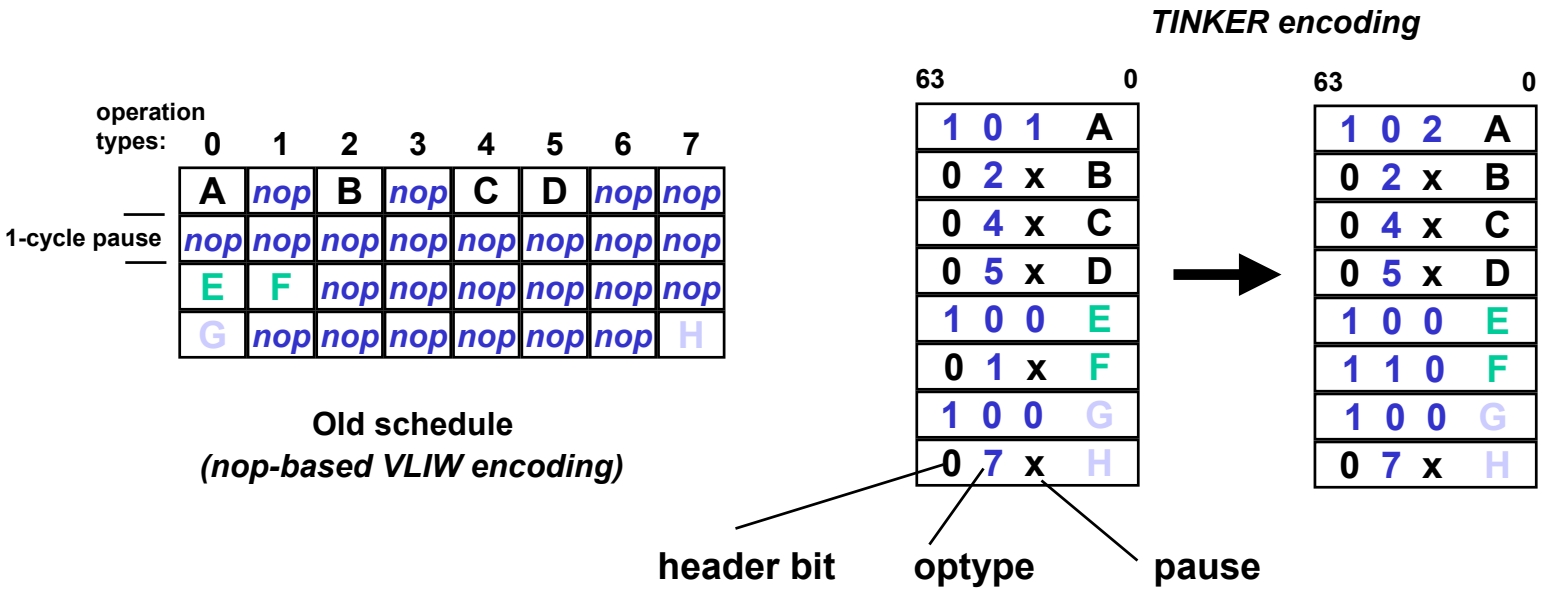
336 bytes total (10 extra *nop*s)

- code size changes are due to:
  - add/delete of nops
  - add/delete of empty cycles

© W. W. Hwu and S. J. Patel, 2002

TINKER Encoding (Conte)

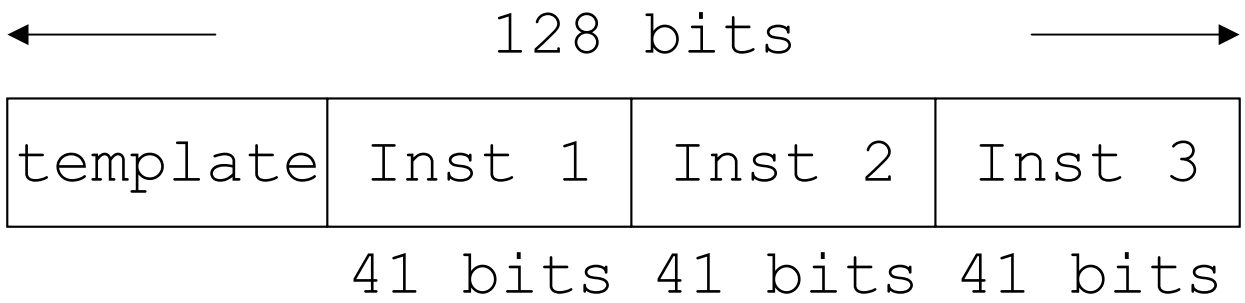
  - Variable-width MultiOp
- Fixed-width Ops
- Header bit, optype, pause specifier



© W. W. Hwu and S. J. Patel, 2002

IA-64 Encoding

- 5 bits of template specifier in every 128-bit bundle
- Each bundle contains 3 instructions
- 32 templates – some contain stops
- Implicit nop: after stop bits
  - Instructions after stop bits are for the next issue group
- Explicit nop: inserted due to lack of nop templates



© W. W. Hwu and S. J. Patel, 2002

IA-64 issue model

- Explicit parallelism
  - All instructions in a template are free of dependences with each other
- Multiple bundle issue
  - Itanium I and Itanium II both attempt to issue two bundles in each clock cycle
- Bundle rotation
  - If a stop occurs in the second issue bundle of a clock cycle, the remaining of the second bundle becomes the first bundle of the next clock cycle

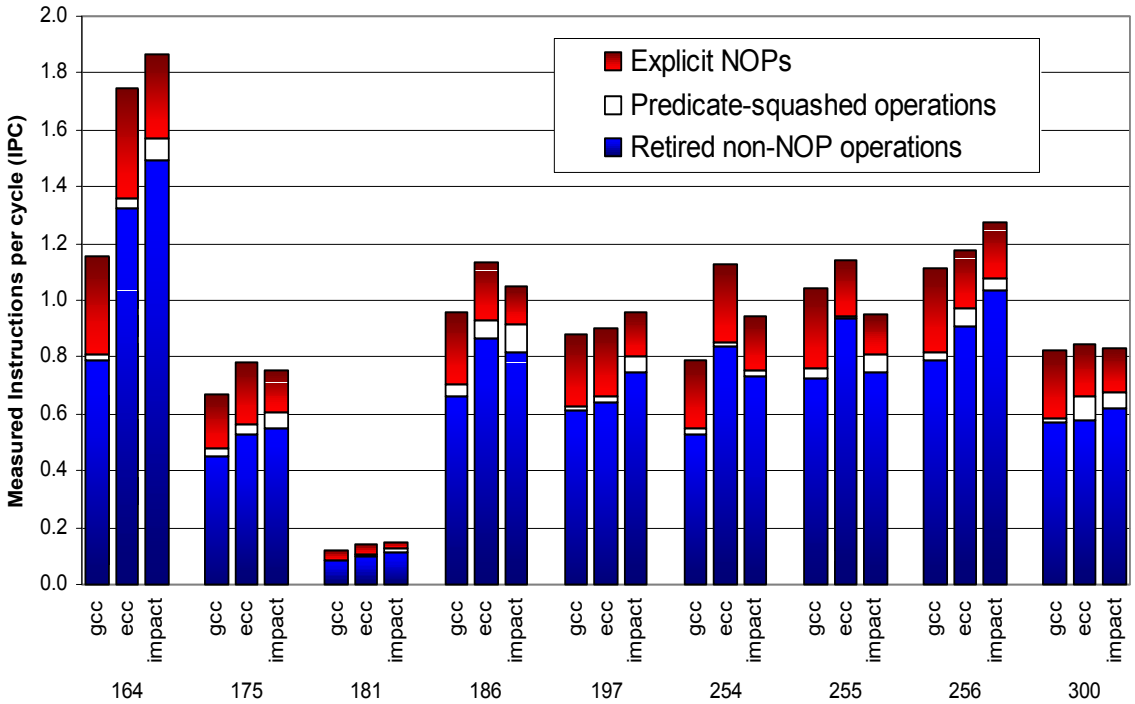
© W. W. Hwu and S. J. Patel, 2002

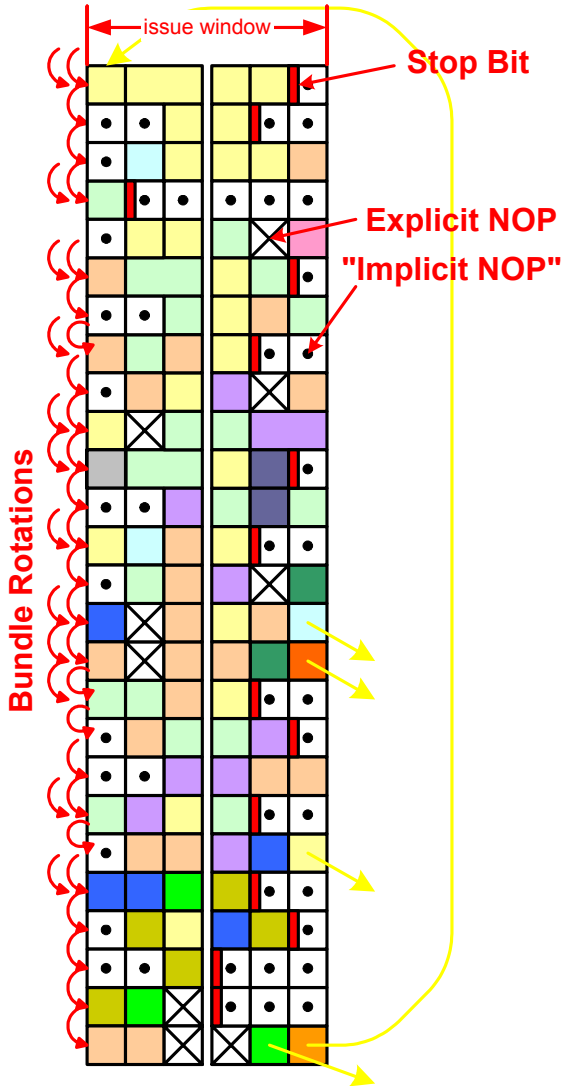
IA-64 Encoding Example

- Template example - MII*
  - The three instructions in the bundle are of memory type, integer type, and integer type, in that order
  - A stop is placed between the first and the second integer type instruction
  - In Itanium II, if the bundle is the second bundle of the current issue, bundle rotation occurs in the next clock cycle.

© W. W. Hwu and S. J. Patel, 2002

IPC and EPIC Instruction Issue





- Available ILP varies widely
- Bundling moderates between code size and efficient parallel issue

© W. W. Hwu and S. J. Patel, 2002

Control Speculation

- Scheduling an instruction before knowing that its execution is required

- Moving an instruction above a branch
  - Removes control dependency (increases ILP)
  - Win when branch predicted correctly

- Instruction sequence seen by hardware is changed!
  - Must ensure that execution result unaffected by such movement

© W. W. Hwu and S. J. Patel, 2002

Control Speculation Example

- A: ADD r6 <- r4 + 1
- **B: BRz (A), X**
- C: LDR r1 <- r2, 0
- D: LDR r3 <- r2, 4
- E: ADD r4 <- r3 + 1
- F: ADD r5 <- r1 + 1
- G: STR r4 -> r2, 4

- C: LDR r1 <- r2, 0
- D: LDR r3 <- r2, 4
- A: ADD r6 <- r4 + 1
- F: ADD r5 <- r1 + 1
- E: ADD r4 <- r3 + 1
- **B: BRZ (A), X**
- G: STQ r4 -> r2, 4

© W. W. Hwu and S. J. Patel, 2002

What is a sentinel?

- Each instruction has two parts:
  - Non-excepting part that performs actual operation
  - Sentinel part that flags an exception if necessary

- Non-excepting part of an instruction can be speculatively scheduled provided the sentinel part stays in its home block.

© W. W. Hwu and S. J. Patel, 2002

Control Speculation Example with Sentinels

- A: ADD r6 <- r4 + 1
- **B: BRz (A), X**
- C: LDR r1 <- r2, 0
- D: LDR r3 <- r2, 4
- E: ADD r4 <- r3 + 1
- F: ADD r5 <- r1 + 1
- G: STR r4 -> r2, 4

- C: (s) LDR r1 <- r2, 0
- D: (s) LDR r3 <- r2, 4
- A: ADD r6 <- r4 + 1
- F: (s) ADD r5 <- r1 + 1
- E: (s) ADD r4 <- r3 + 1
- **B: BRz (A), X**
- Sentinels for C,D,E,F
- G: STR r4 -> r2, 4

© W. W. Hwu and S. J. Patel, 2002

Control Speculation in IA-64

- Speculative load -- ld.s
  - Sets the NaT bit of the destination register if anything goes wrong.

- Speculation check -- chk.s
  - Placed in the *home block* of the original load.
  - Checks the NaT bit of a register and branches to a recovery block if it is set.

© W. W. Hwu and S. J. Patel, 2002

Data Dependence Speculation

- Often the compiler will not know for certain that a ST and a LD access the same memory location. Memory references can be *ambiguous*.

- The compiler can conservatively assume that all unknown LD and ST conflict.

© W. W. Hwu and S. J. Patel, 2002

Data Dependence Speculation Example

```
• A: ADD r6 <- r4 + 1
• B: ADD r4 <- r4 + 1
• C: STR  r1 -> r2, 0
     :
     :
• D: LDR  r3 <- r5, 0
```

```
• A: ADD r6 <- r4 + 1
• D: LDR  r3 <- r5, 0
• B: ADD r4 <- r4 + 1
• C: STR  r1 -> r2, 0
     :
     :
```

EPIC allows the compiler to be less conservative by using a special type of **check** instruction.

© W. W. Hwu and S. J. Patel, 2002

Data Dependence Speculation in IA-64

- Advanced load -- ld.a
  - Operates like a regular load, but adds an entry in the Advanced Load Address Table (ALAT).
- Check load -- ld.c
  - If ALAT hit, nothing done.
  - If ALAT miss, re-load data from memory.
- Advanced load check -- chk.a
  - If ALAT hit, nothing done.
  - If ALAT miss, branch to recovery block.

© W. W. Hwu and S. J. Patel, 2002

Predicated Execution

```
<p1> LDR r1,r2,0
```

- If p1 is TRUE, instruction executes normally
- If p1 is FALSE, instruction treated as NOP

© W. W. Hwu and S. J. Patel, 2002

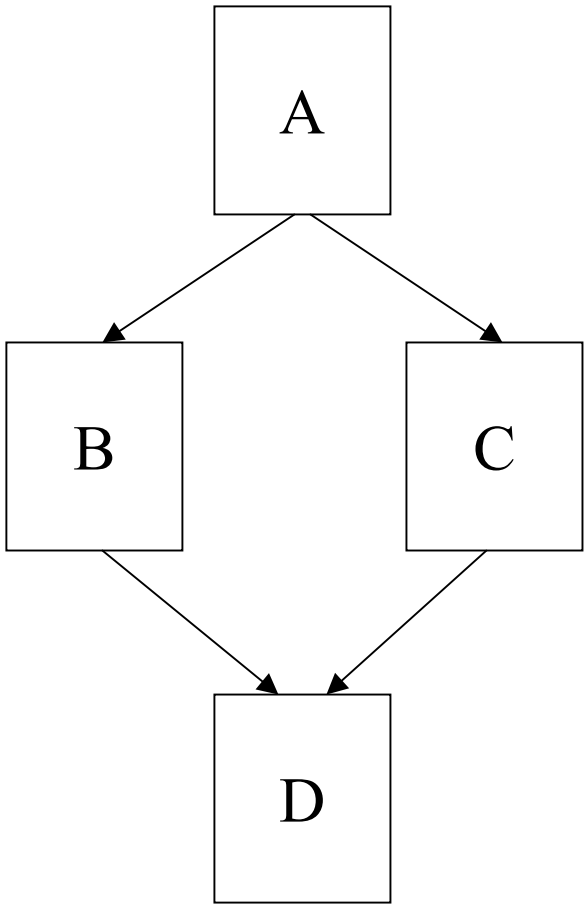
Predication Example

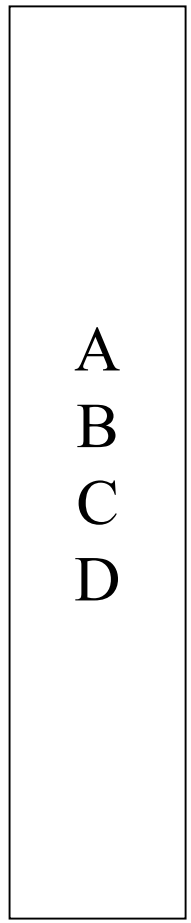
```
:
:
if (x == 10)
   c = c + 1;
:
:
```

```
        :
        :
        LDR r5, X
        p1 <- r5 eq 10
<p1>    LDR  r1 <- C
<p1>    ADD r1, r1, 1
<p1>    STR  r1 -> C
        :
        :
```

© W. W. Hwu and S. J. Patel, 2002

Predication very helpful for if-else





© W. W. Hwu and S. J. Patel, 2002
ECE 412, University of Illinois

If-else example

```
        :
        :
      p1,p2 <- r5 eq 10
<p1>  inst 1 from B
<p1>  inst 2 from B
<p1>    :
        :
<p2>  inst 1 from C
<p2>  inst 2 from C
        :
        :
```

© W. W. Hwu and S. J. Patel, 2002

EPIC : Full Predication Support

- Predicate defining instructions
- Full set of predicated instructions
- Separate predicate register file
- Best performance
- Cydra-5, IA-64, TI-C60, StarCore, ARM

- Almost all ISAs today have partial predication support : CMOV

© W. W. Hwu and S. J. Patel, 2002

Predication in IA-64

- Predicate Registers
  - pr0 – pr63, each contain 1 bit.

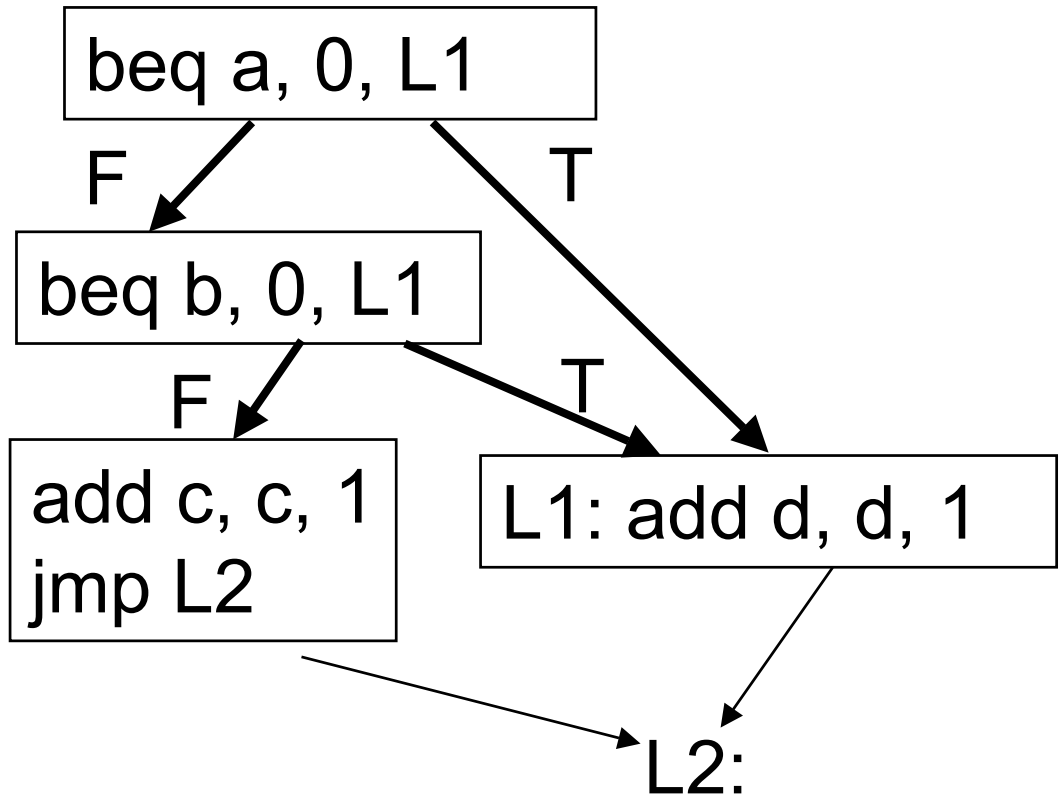
- Compare instructions -- cmp.<rel>.<comp>
  - Sets two predicate regs, based on relation.
  - <rel> can be eq, ne, lt, le, gt, ge,…
  - Optional completer (<comp>) allows for very sophisticated predication.

© W. W. Hwu and S. J. Patel, 2002

OR & AND Predicate Defines

For blocks reached on multiple conditions

```
If (a && b)
        c= c+1;
else
        d = d+1;
```



© W. W. Hwu and S. J. Patel, 2002

OR & AND Predicate Defines

```
        :
        :
        p1 <- 0
        p2 <- 1
        cmp.eq.or.andcm p1,p2,a,0
        cmp.eq.or.andcm p1,p2,b,0
<p2>    add c <- c, 1
<p1>    add d <- d, 1
        :
        :
```

© W. W. Hwu and S. J. Patel, 2002

Predication vs. Prediction

- What is the penalty associated with predication?
- When should a branch be predicated, when should it be predicted?
- On the Forefront: dynamic techniques

© W. W. Hwu and S. J. Patel, 2002