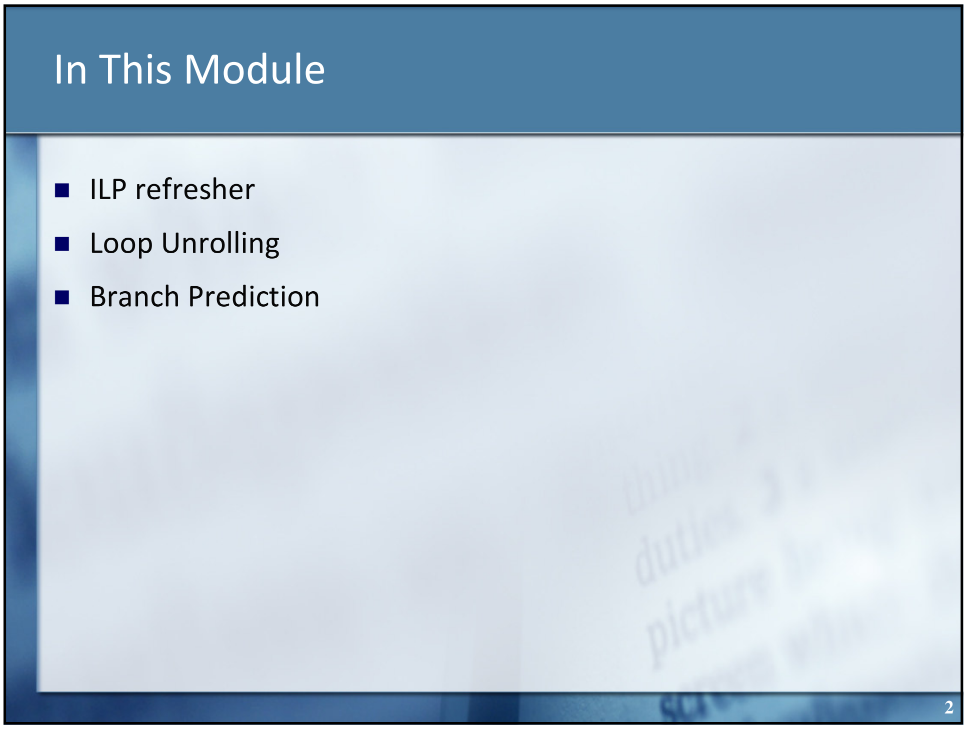
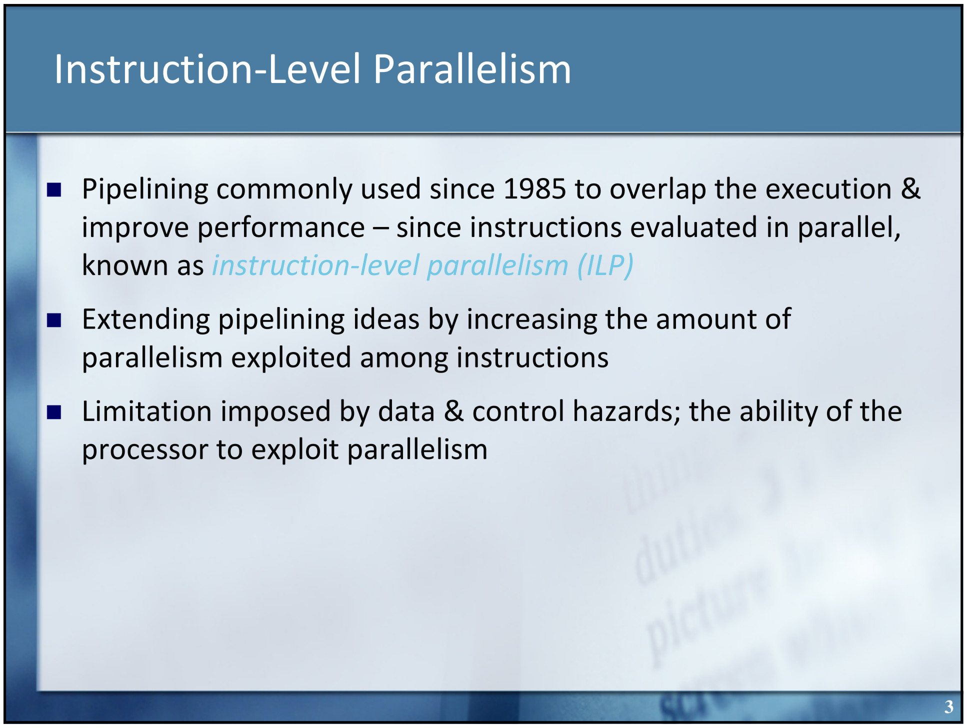
CDA 5106 Advanced Computer Architecture 1

Module 3b | Instruction-Level Parallelism Continued



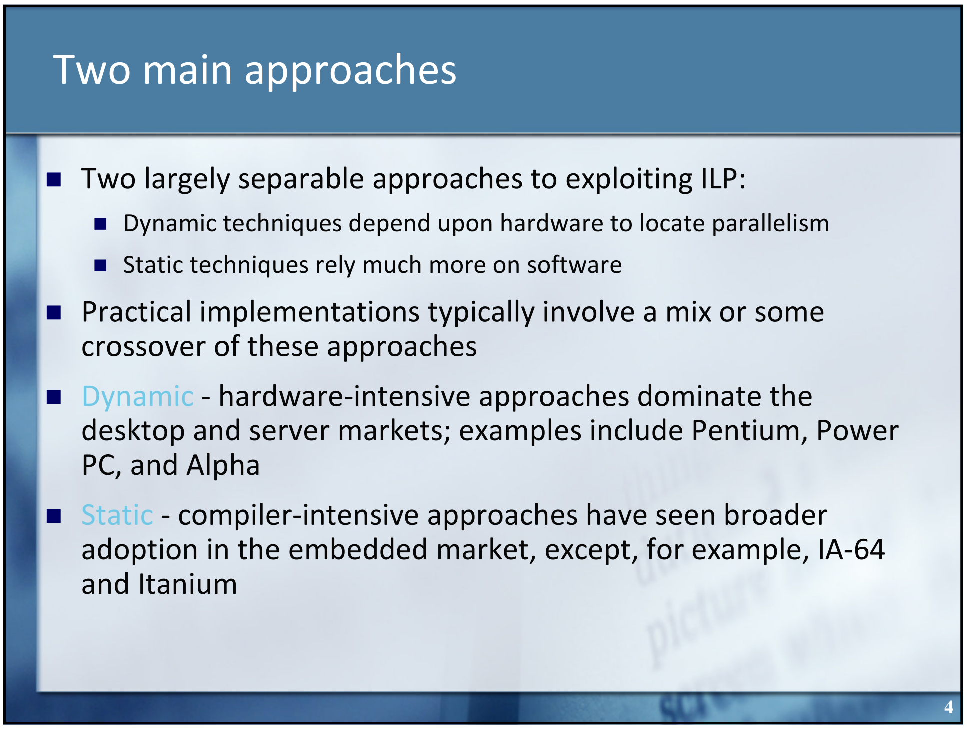
In This Module

- ILP refresher
- Loop Unrolling
- Branch Prediction



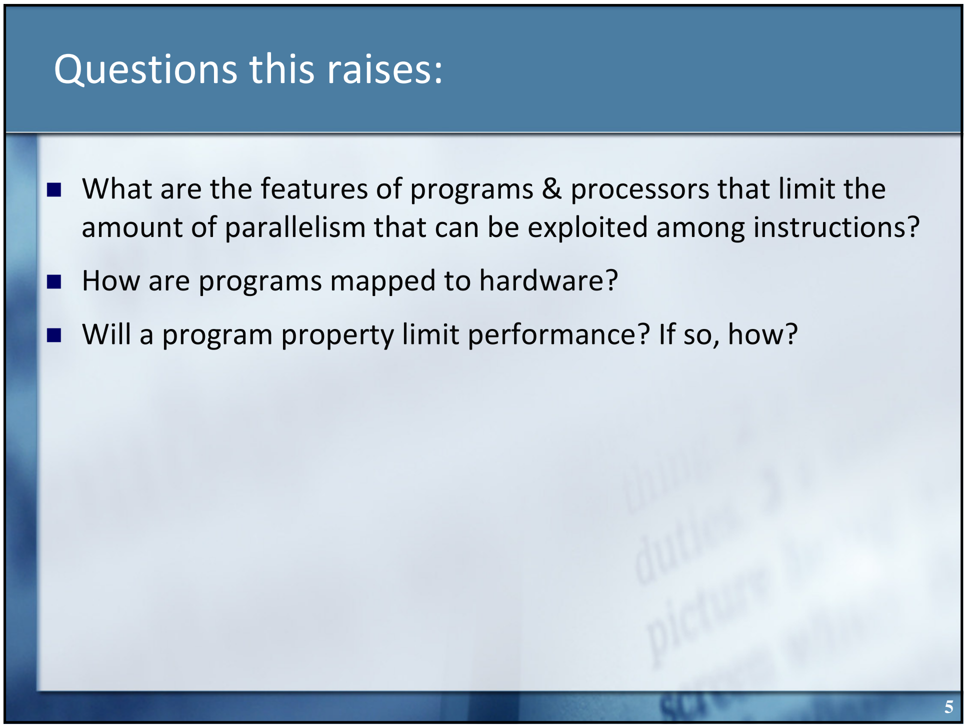
Instruction-Level Parallelism

- Pipelining commonly used since 1985 to overlap the execution & improve performance – since instructions evaluated in parallel, known as *instruction-level parallelism (ILP)*
- Extending pipelining ideas by increasing the amount of parallelism exploited among instructions
- Limitation imposed by data & control hazards; the ability of the processor to exploit parallelism



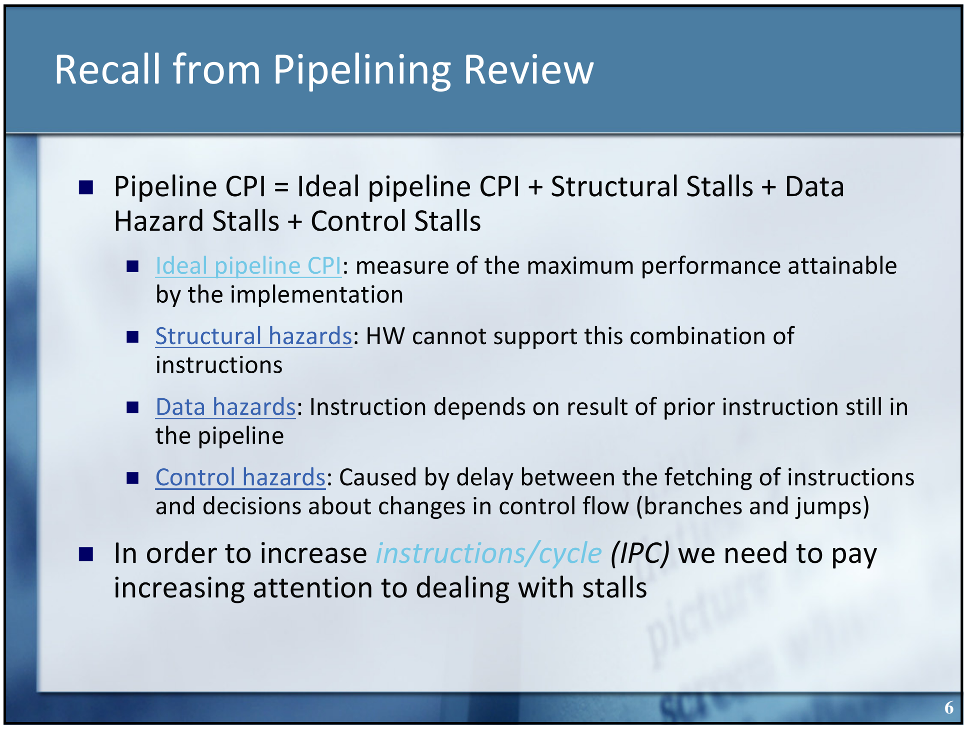
Two main approaches

- Two largely separable approaches to exploiting ILP:
  - Dynamic techniques depend upon hardware to locate parallelism
  - Static techniques rely much more on software
- Practical implementations typically involve a mix or some crossover of these approaches
- Dynamic - hardware-intensive approaches dominate the desktop and server markets; examples include Pentium, Power PC, and Alpha
- Static - compiler-intensive approaches have seen broader adoption in the embedded market, except, for example, IA-64 and Itanium



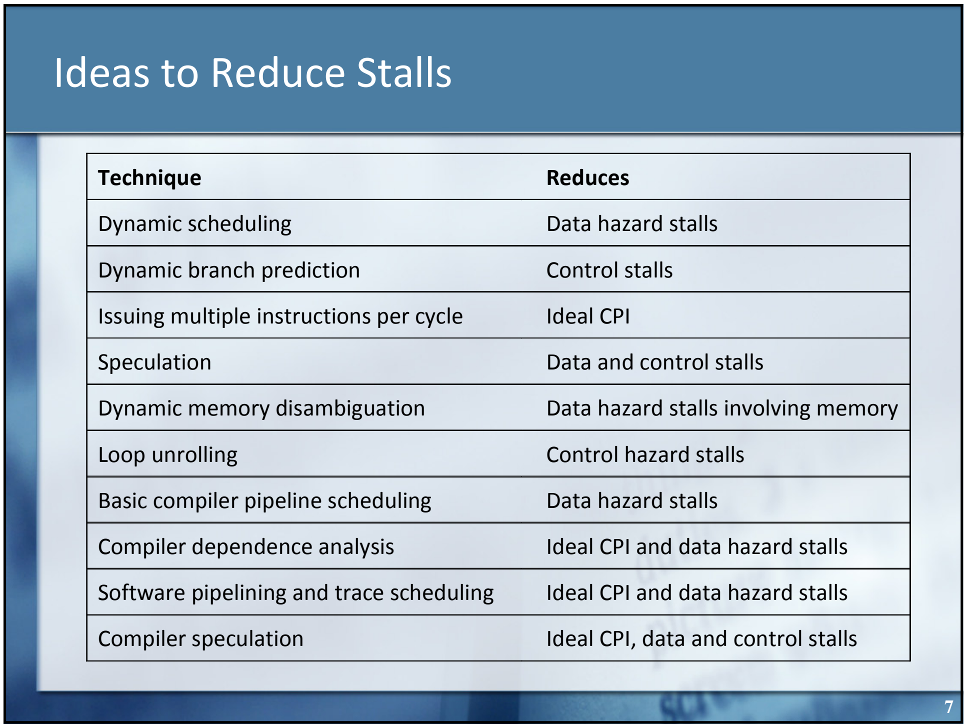
Questions this raises:

- What are the features of programs & processors that limit the amount of parallelism that can be exploited among instructions?
- How are programs mapped to hardware?
- Will a program property limit performance? If so, how?



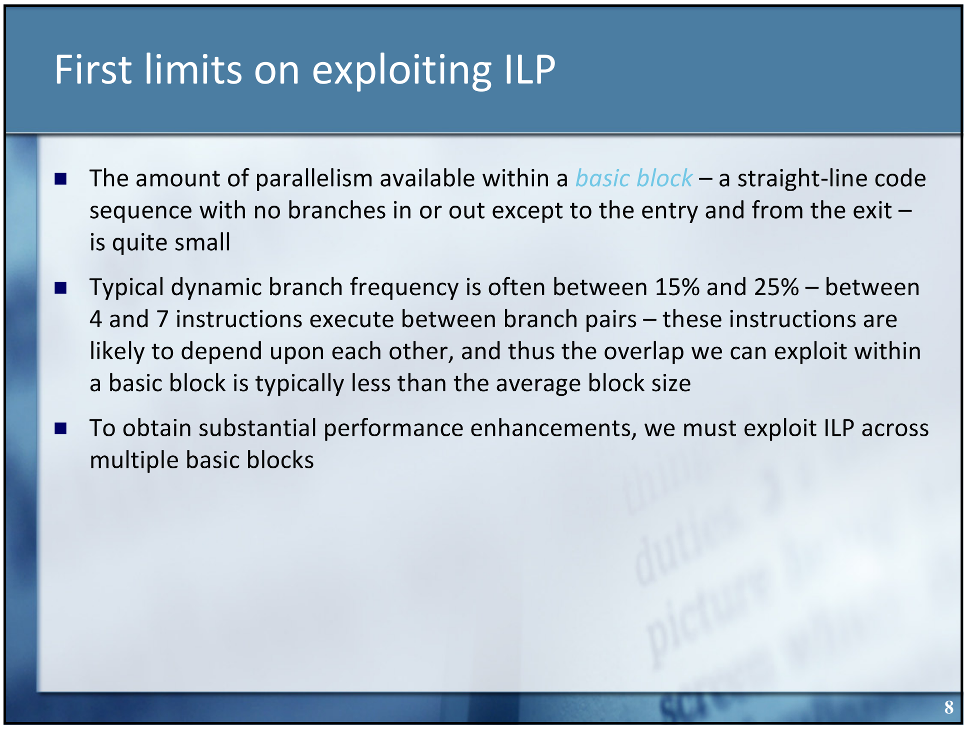
Recall from Pipelining Review

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)
- In order to increase *instructions/cycle (IPC)* we need to pay increasing attention to dealing with stalls



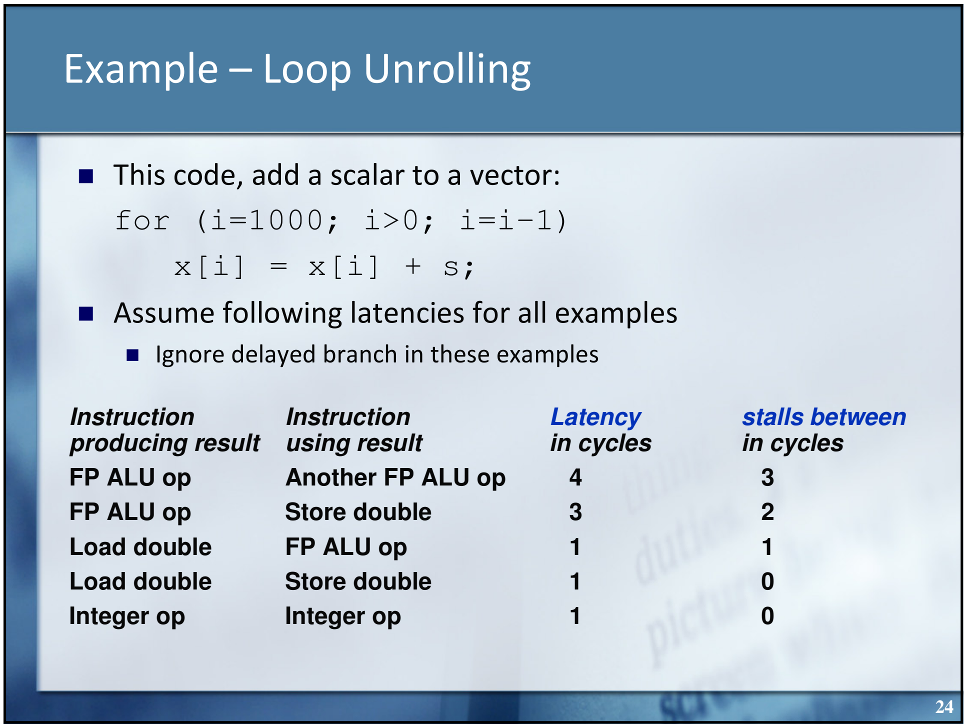
Ideas to Reduce Stalls

Technique	Reduces
Dynamic scheduling	Data hazard stalls
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI and data hazard stalls
Software pipelining and trace scheduling	Ideal CPI and data hazard stalls
Compiler speculation	Ideal CPI, data and control stalls



First limits on exploiting ILP

- The amount of parallelism available within a *basic block* – a straight-line code sequence with no branches in or out except to the entry and from the exit – is quite small
- Typical dynamic branch frequency is often between 15% and 25% – between 4 and 7 instructions execute between branch pairs – these instructions are likely to depend upon each other, and thus the overlap we can exploit within a basic block is typically less than the average block size
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks



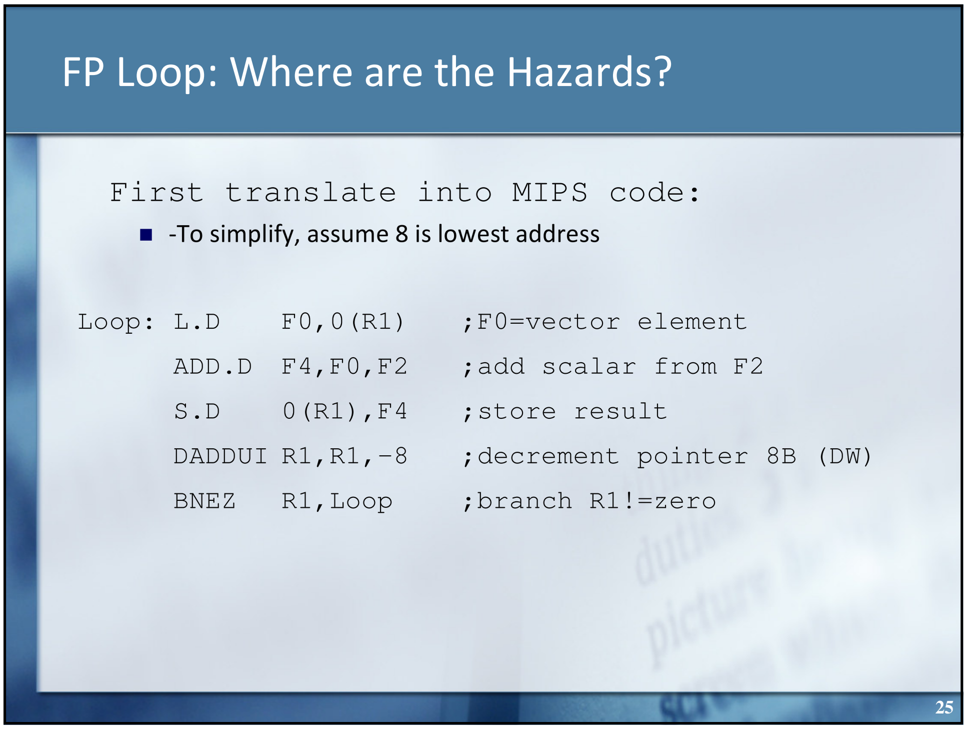
Example – Loop Unrolling

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

Instruction producing result	*Instruction using result*	*Latency in cycles*	*stalls between in cycles*
FP ALU op	**Another FP ALU op**	**4**	**3**
FP ALU op	**Store double**	**3**	**2**
Load double	**FP ALU op**	**1**	**1**
Load double	**Store double**	**1**	**0**
Integer op	**Integer op**	**1**	**0**

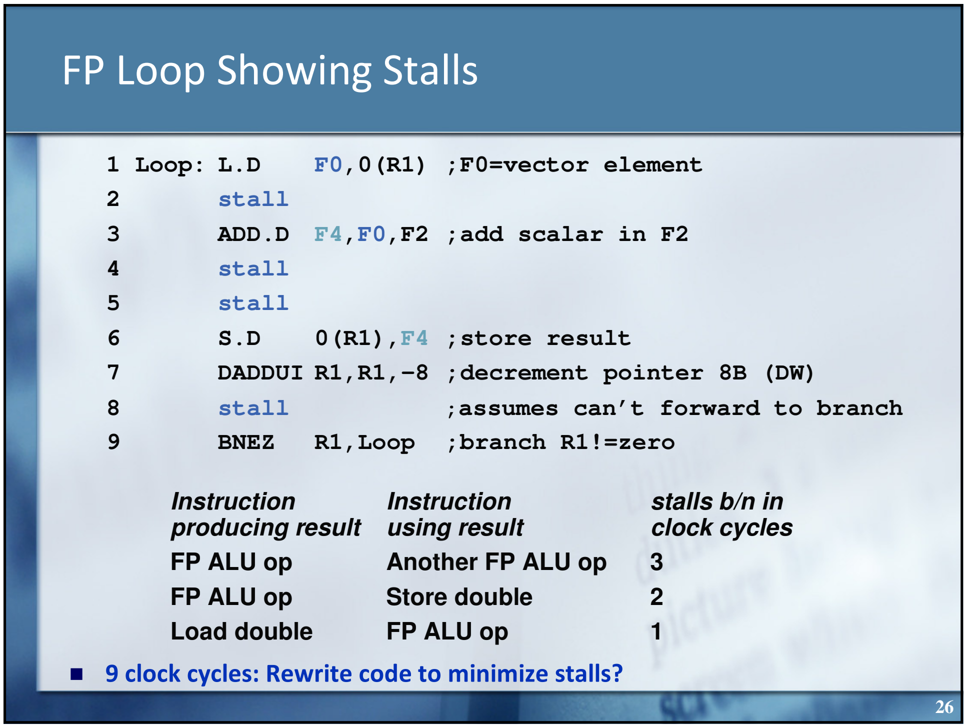


FP Loop: Where are the Hazards?

```
First translate into MIPS code:
```

- -To simplify, assume 8 is lowest address

```
Loop: L.D    F0,0(R1)   ;F0=vector element
      ADD.D  F4,F0,F2   ;add scalar from F2
      S.D    0(R1),F4   ;store result
      DADDUI R1,R1,-8   ;decrement pointer 8B (DW)
      BNEZ   R1,Loop    ;branch R1!=zero
```

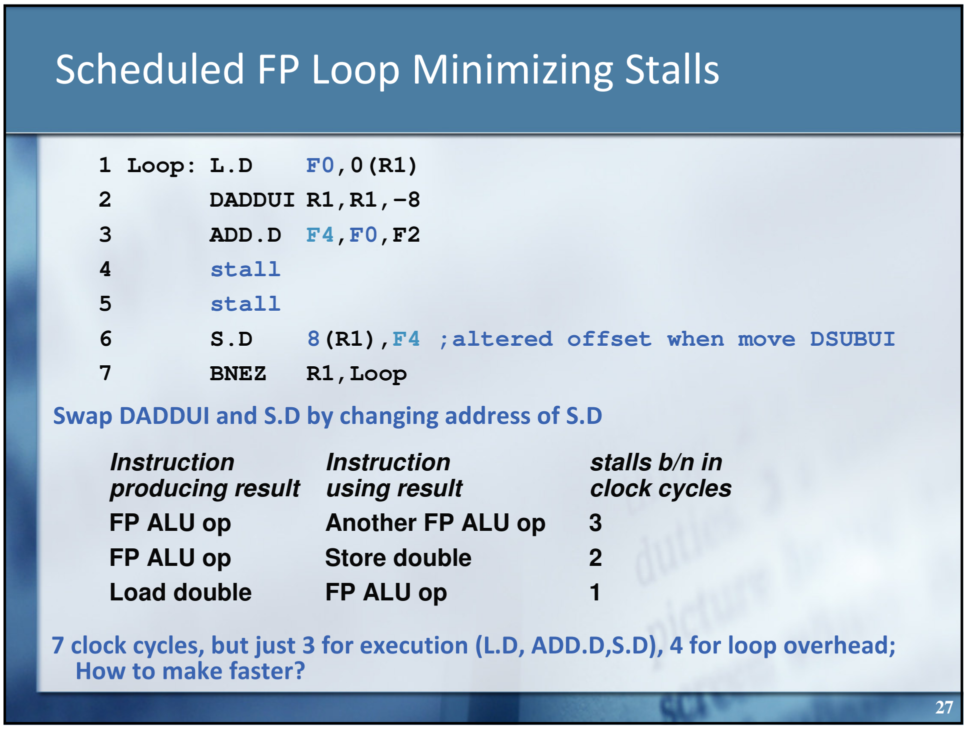


FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1) ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2 ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4 ;store result
7       DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
8       stall           ;assumes can't forward to branch
9       BNEZ   R1,Loop  ;branch R1!=zero
```

Instruction producing result	*Instruction using result*	*stalls b/n in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clock cycles: Rewrite code to minimize stalls?



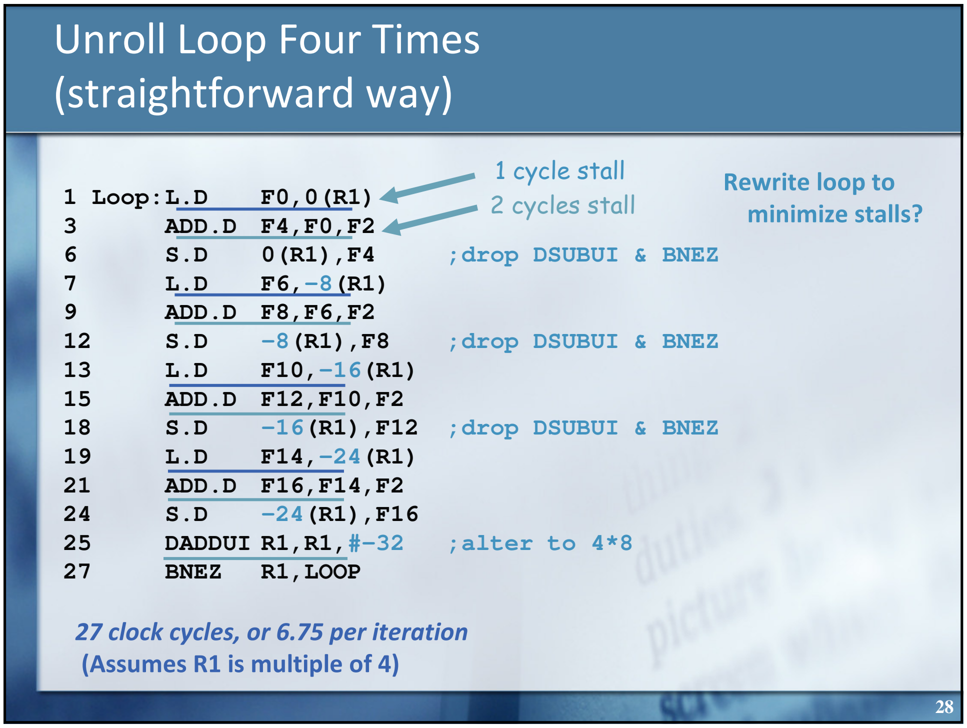
Scheduled FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4 ;altered offset when move DSUBUI
7       BNEZ   R1,Loop
```

Swap DADDUI and S.D by changing address of S.D

Instruction producing result	*Instruction using result*	*stalls b/n in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How to make faster?



Unroll Loop Four Times (straightforward way)

1 cycle stall

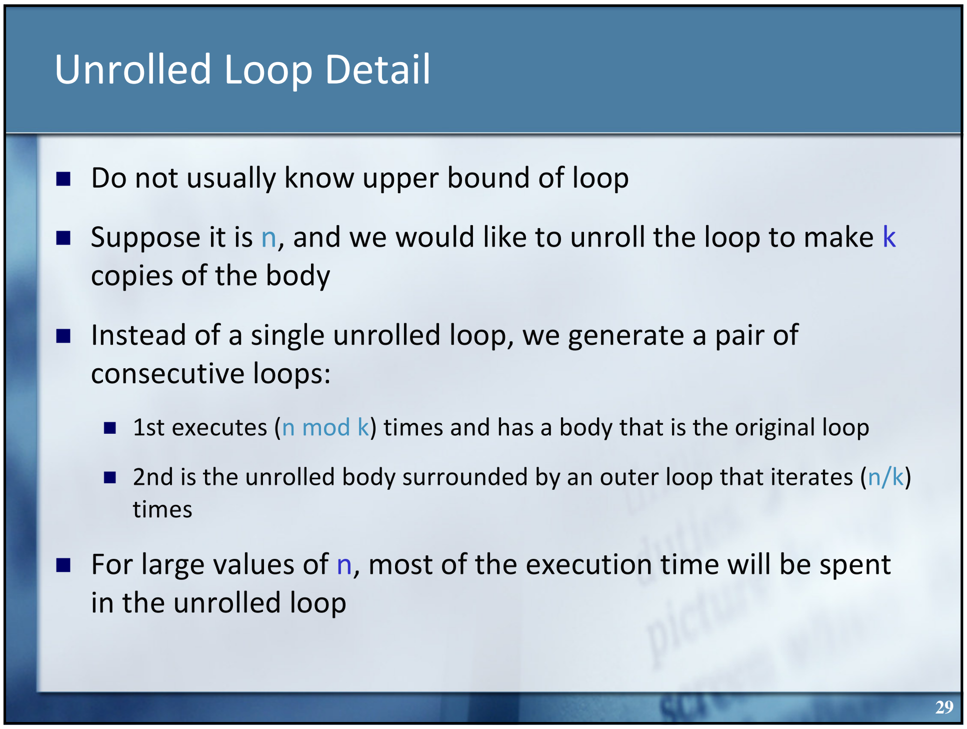
2 cycles stall

Rewrite loop to minimize stalls?

```
1  Loop:L.D    F0,0(R1)
3       ADD.D  F4,F0,F2
6       S.D    0(R1),F4     ;drop DSUBUI & BNEZ
7       L.D    F6,-8(R1)
9       ADD.D  F8,F6,F2
12      S.D    -8(R1),F8    ;drop DSUBUI & BNEZ
13      L.D    F10,-16(R1)
15      ADD.D  F12,F10,F2
18      S.D    -16(R1),F12  ;drop DSUBUI & BNEZ
19      L.D    F14,-24(R1)
21      ADD.D  F16,F14,F2
24      S.D    -24(R1),F16
25      DADDUI R1,R1,#-32   ;alter to 4*8
27      BNEZ   R1,LOOP
```

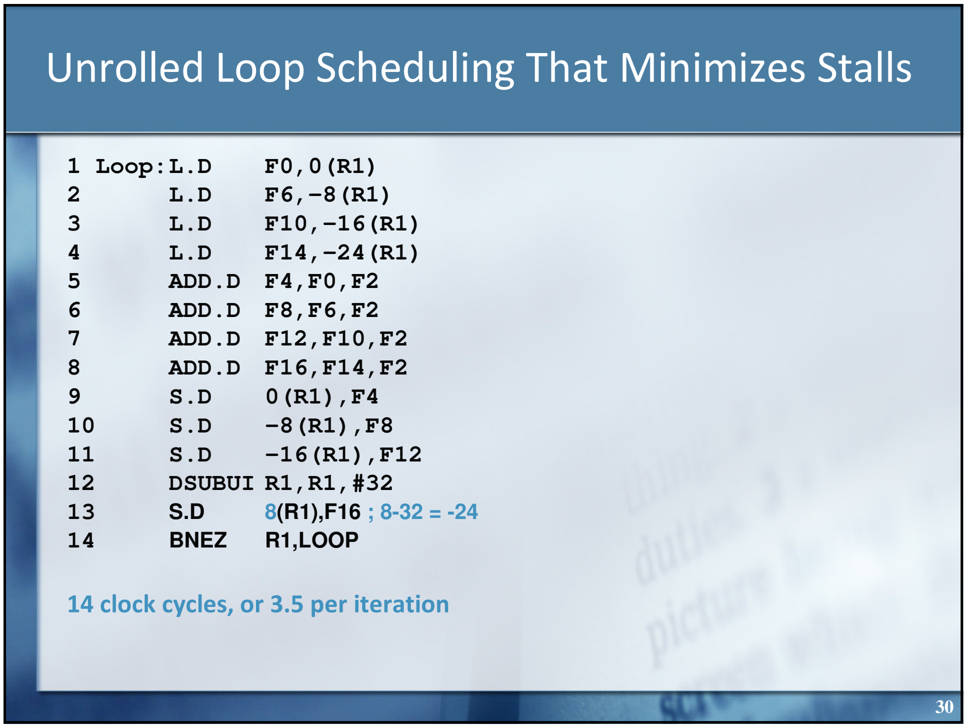
27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)



Unrolled Loop Detail

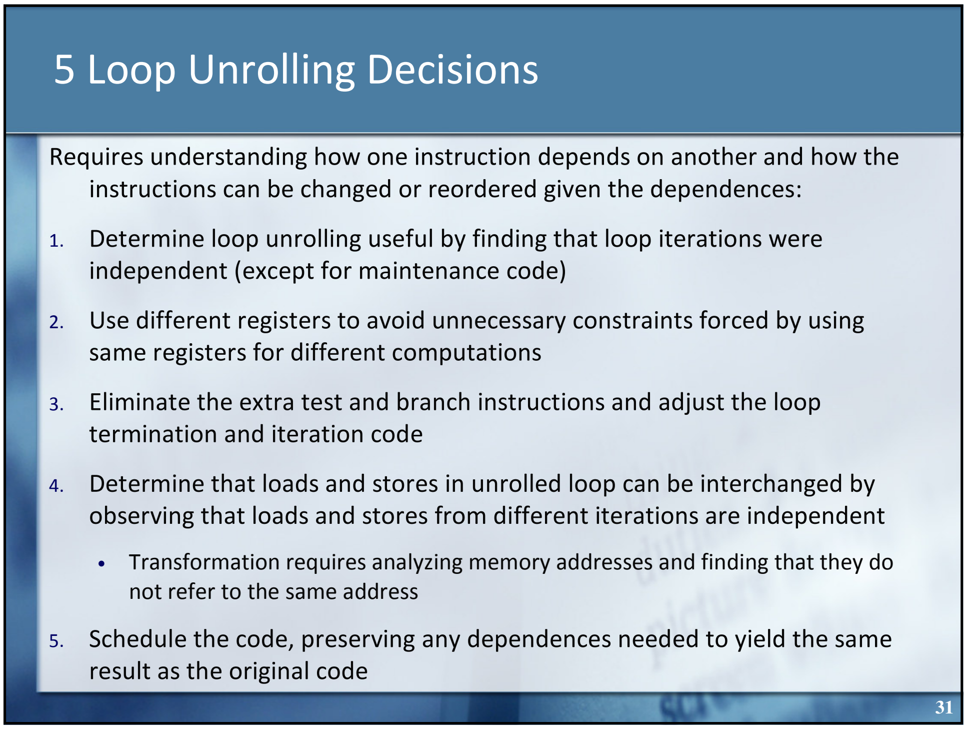
- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop



Unrolled Loop Scheduling That Minimizes Stalls

```
1  Loop:L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    0(R1),F4
10      S.D    -8(R1),F8
11      S.D    -16(R1),F12
12      DSUBUI R1,R1,#32
13      S.D    8(R1),F16 ; 8-32 = -24
14      BNEZ   R1,LOOP
```

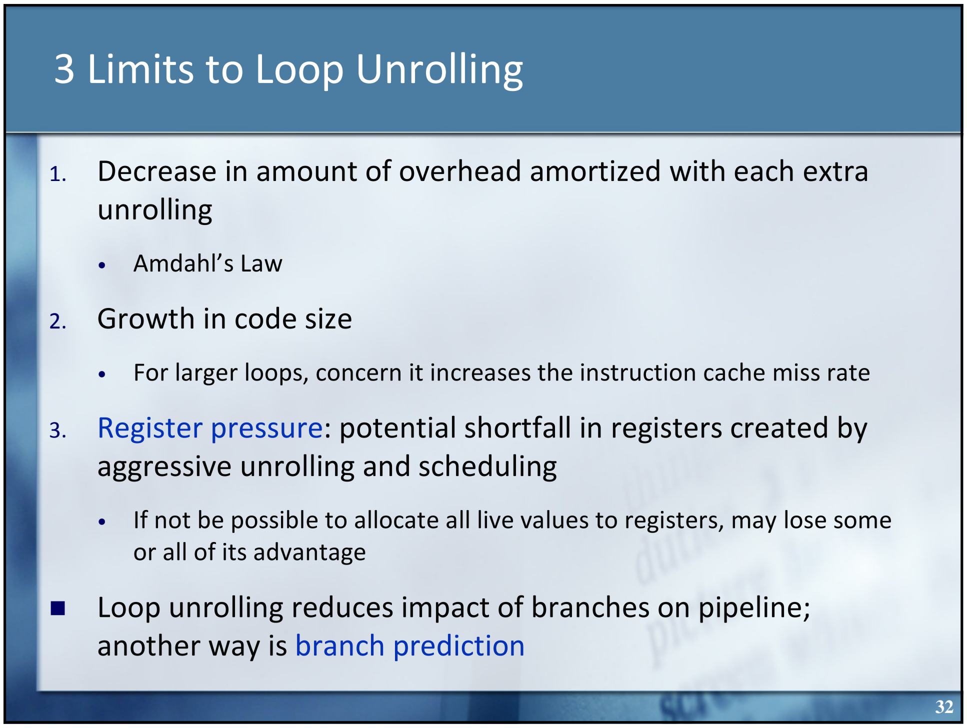
14 clock cycles, or 3.5 per iteration



5 Loop Unrolling Decisions

Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

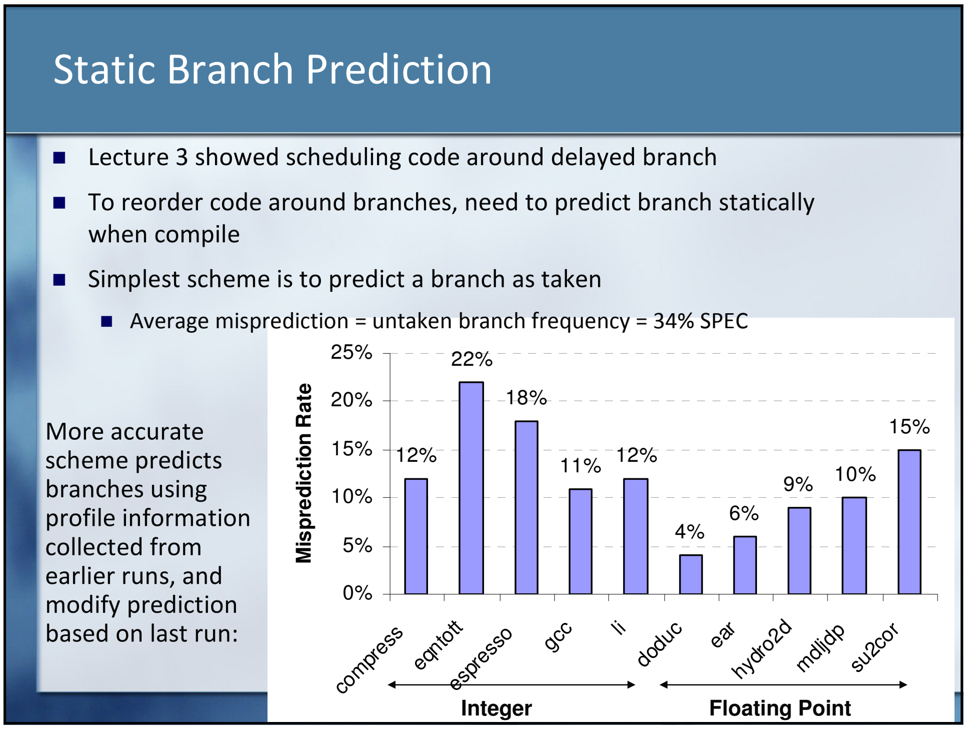
1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
   - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
5. Schedule the code, preserving any dependences needed to yield the same result as the original code



3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

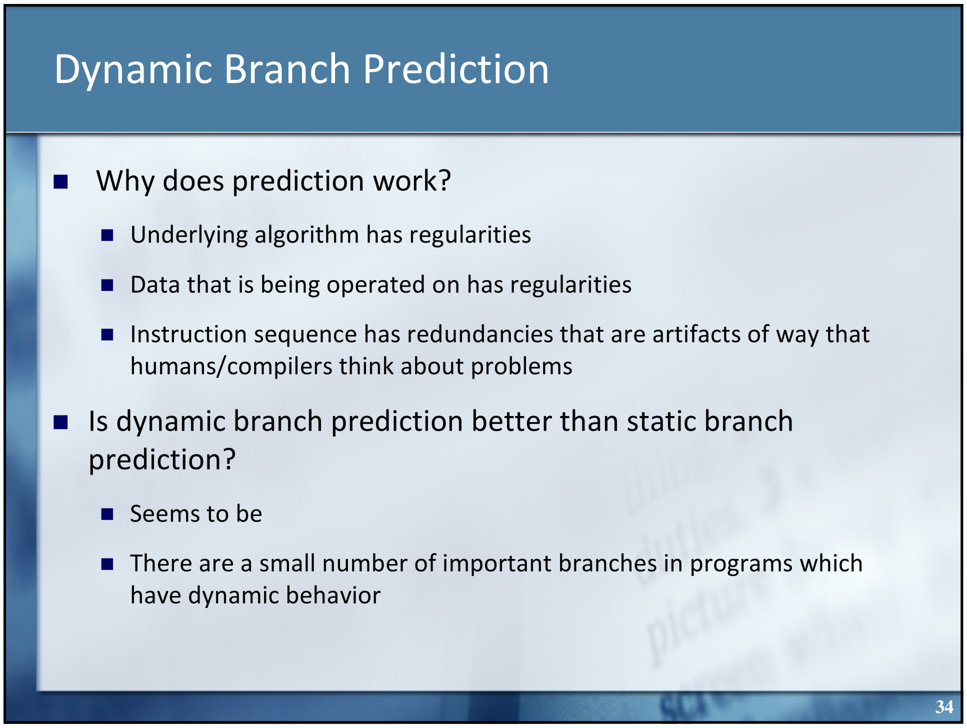
- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction



Static Branch Prediction

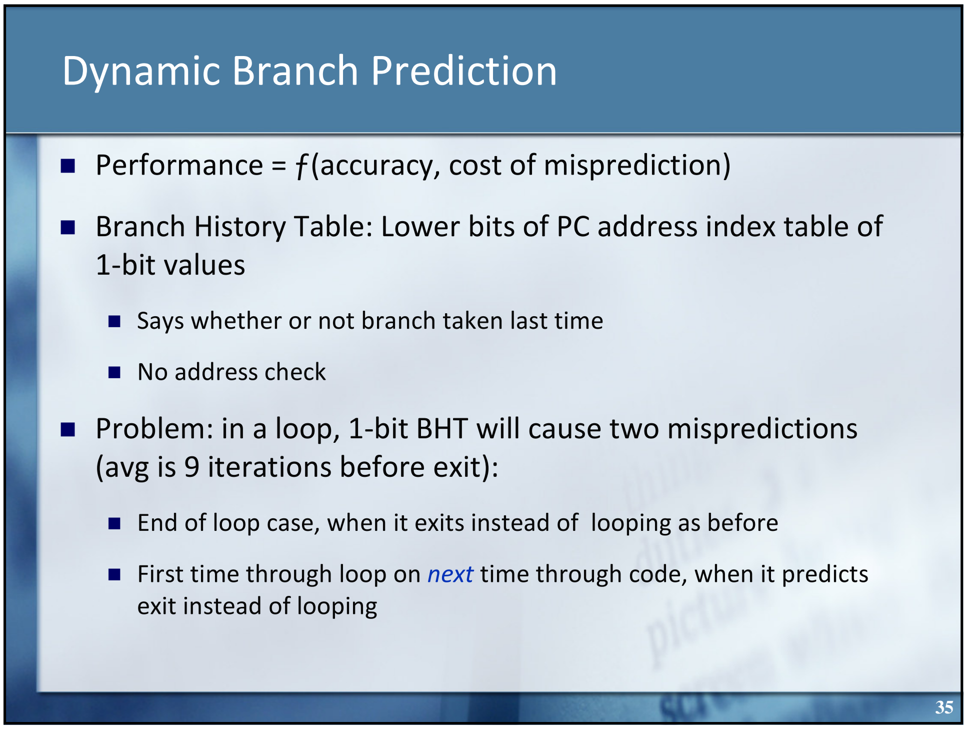
- Lecture 3 showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



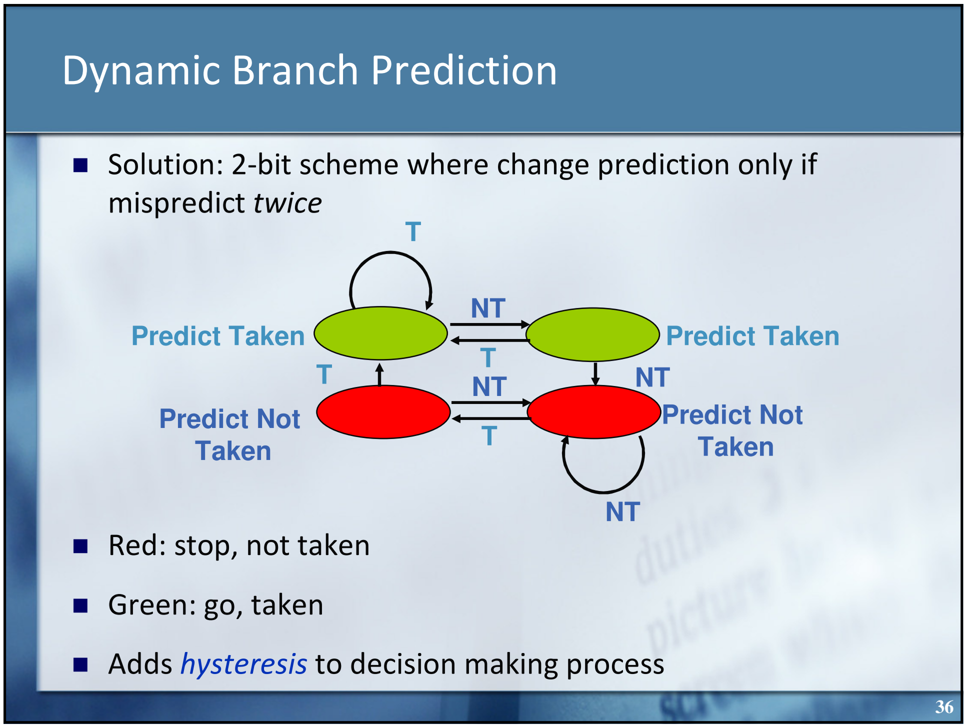
Dynamic Branch Prediction

- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior



Dynamic Branch Prediction

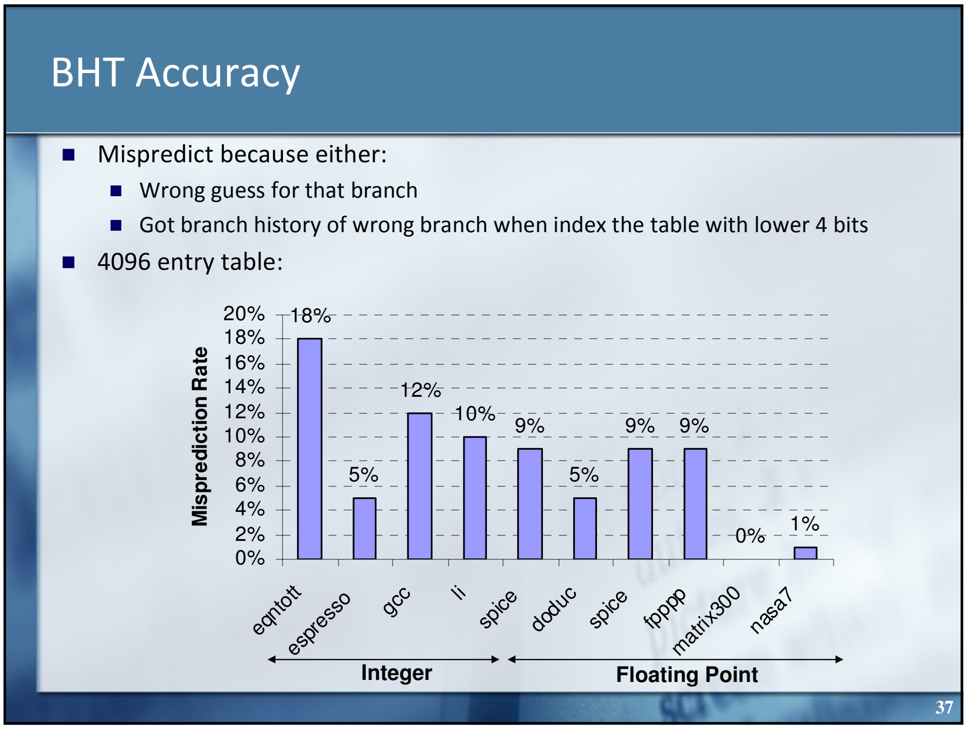
- Performance = *ƒ*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping



Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if mispredict *twice*

- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process



BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table with lower 4 bits
- 4096 entry table:

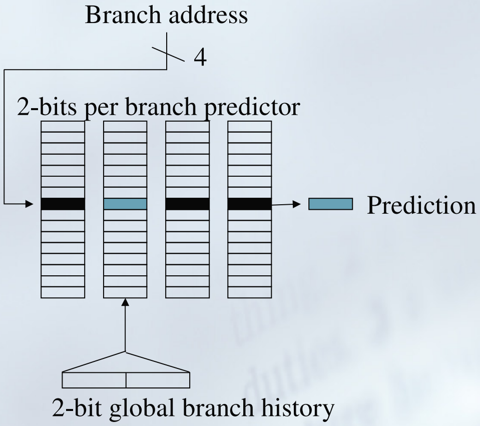
Correlated Branch Prediction

- Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table
- In general, (*m*,*n*) predictor means record last *m* branches to select between 2^m history tables, each with *n*-bit counters
  - Thus, old 2-bit BHT is a (0,2) predictor
- Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.
- Each entry in table has *m* *n*-bit predictors.

Correlating Branches

(2,2) predictor

– Behavior of recent branches selects between four predictions of next branch, updating just that prediction



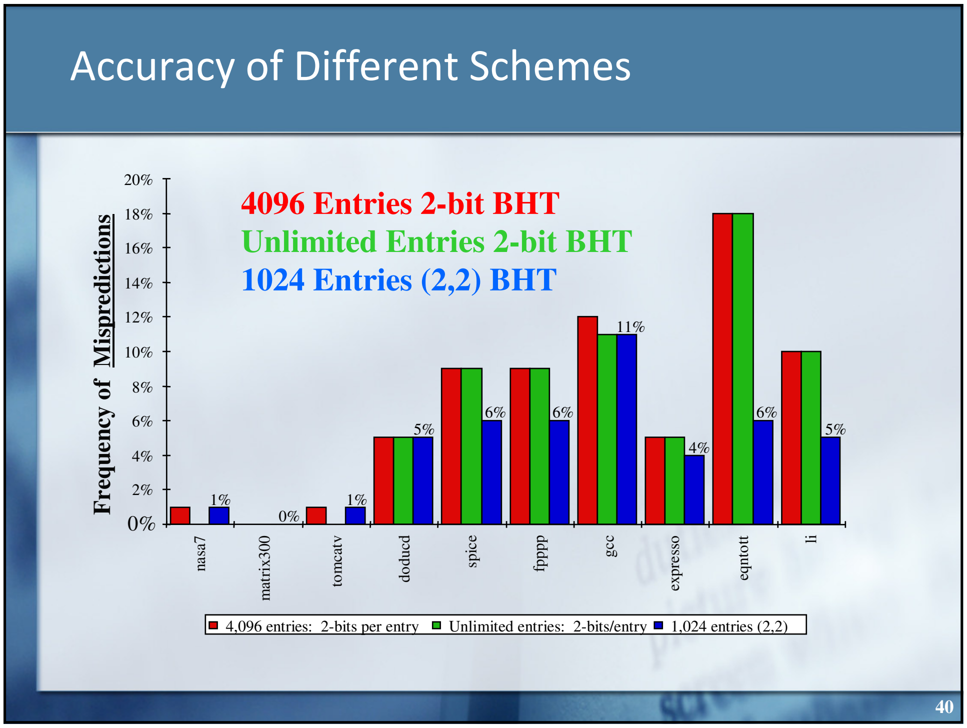
Branch address

4

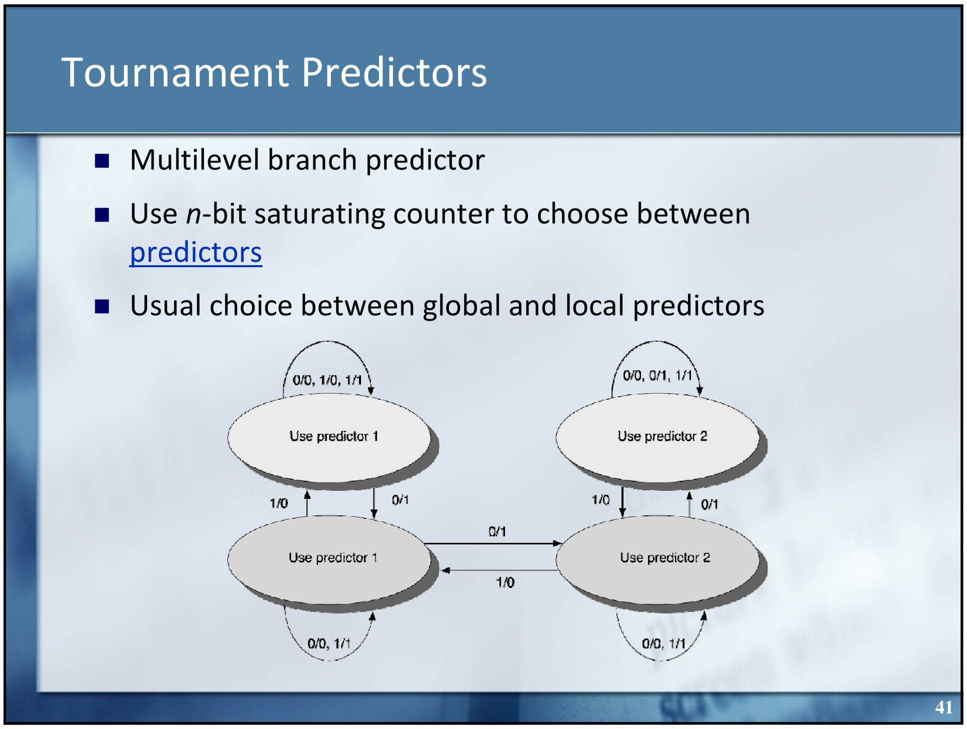
2-bits per branch predictor

Prediction

2-bit global branch history



Accuracy of Different Schemes



Tournament Predictors

- Multilevel branch predictor
- Use *n*-bit saturating counter to choose between predictors
- Usual choice between global and local predictors

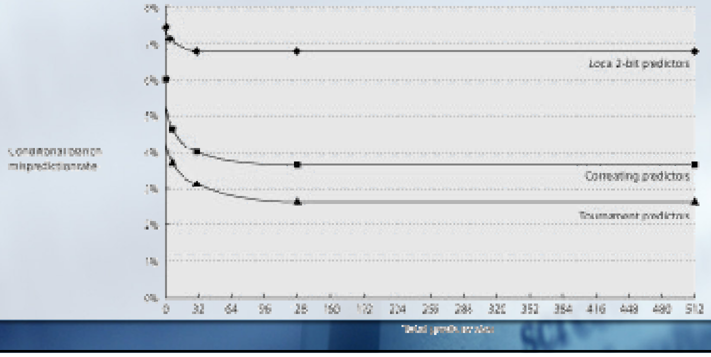
Tournament Predictors

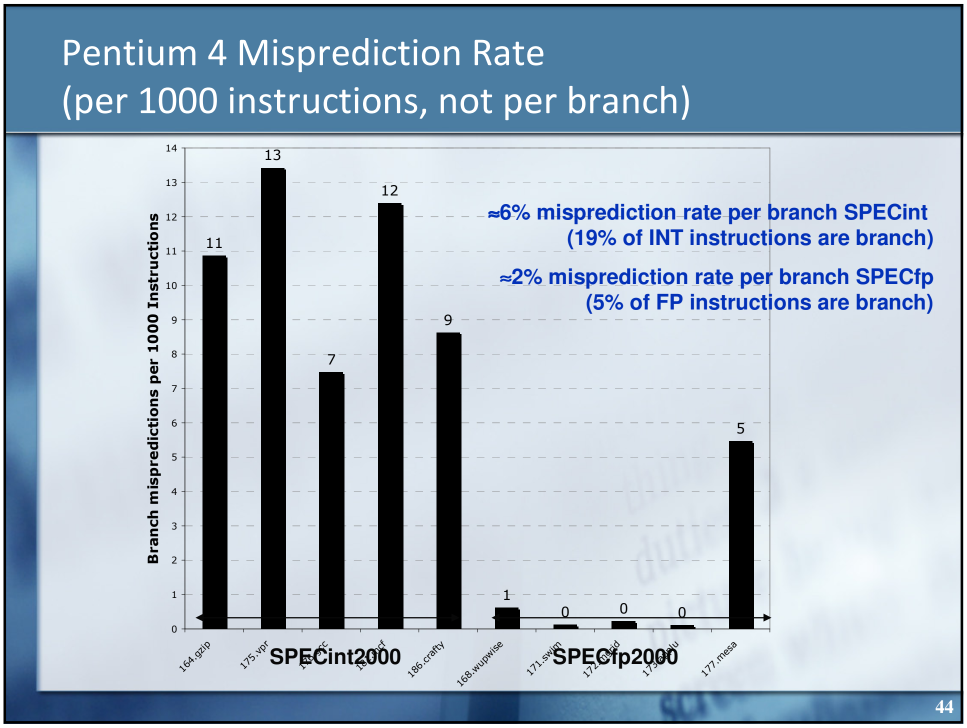
Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:

- Global predictor
  - 4K entries index by history of last 12 branches ($2^{12} = 4K$)
  - Each entry is a standard 2-bit predictor
- Local predictor
  - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
  - The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters

Comparing Predictors

- Advantage of tournament predictor is ability to select the right predictor for a particular branch
  - Particularly crucial for integer benchmarks.
  - A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

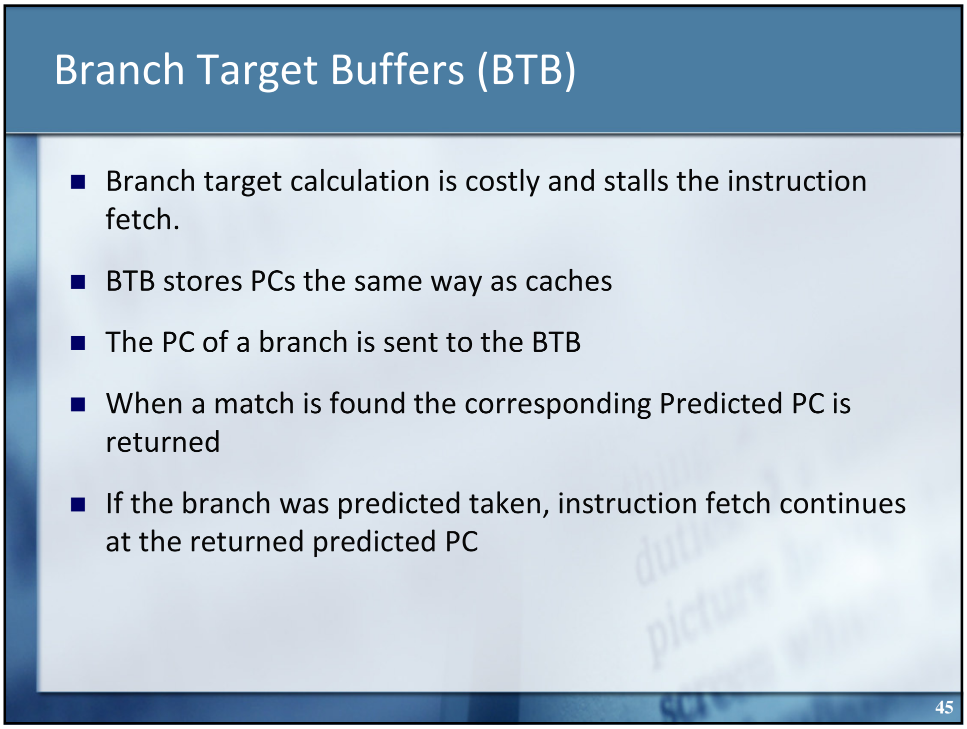




Pentium 4 Misprediction Rate
(per 1000 instructions, not per branch)

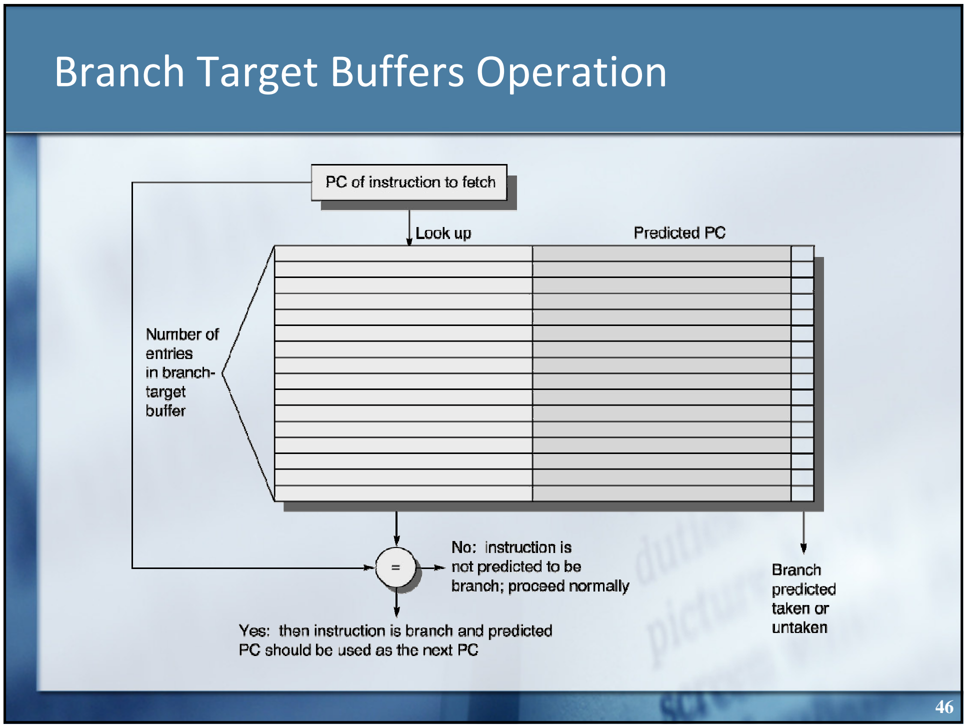
≈6% misprediction rate per branch SPECint
(19% of INT instructions are branch)

≈2% misprediction rate per branch SPECfp
(5% of FP instructions are branch)



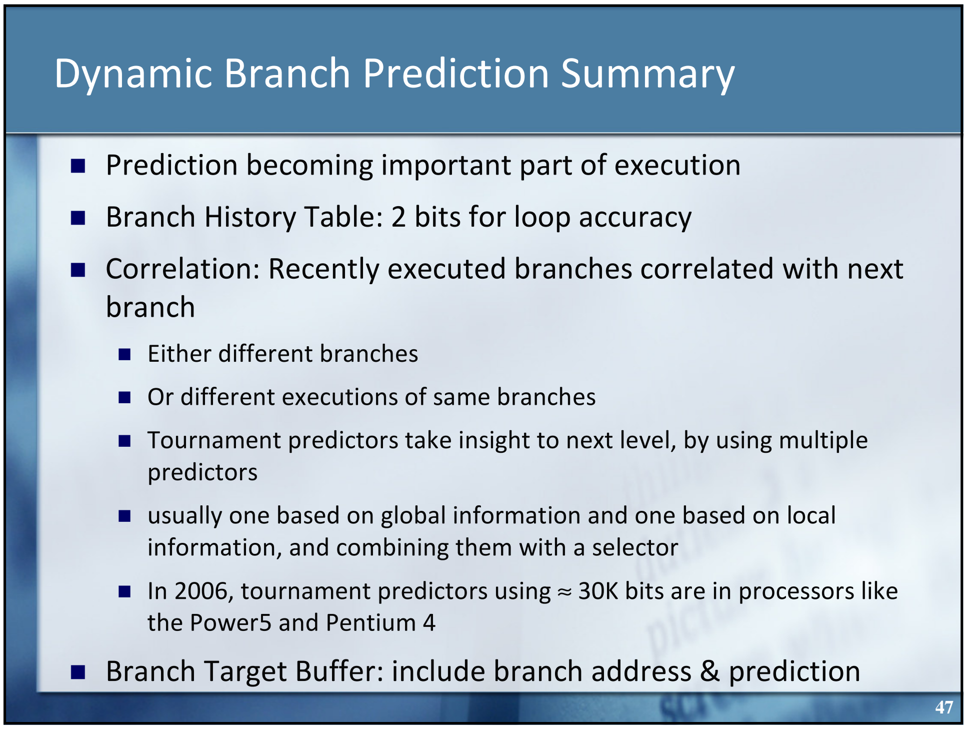
Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls the instruction fetch.
- BTB stores PCs the same way as caches
- The PC of a branch is sent to the BTB
- When a match is found the corresponding Predicted PC is returned
- If the branch was predicted taken, instruction fetch continues at the returned predicted PC



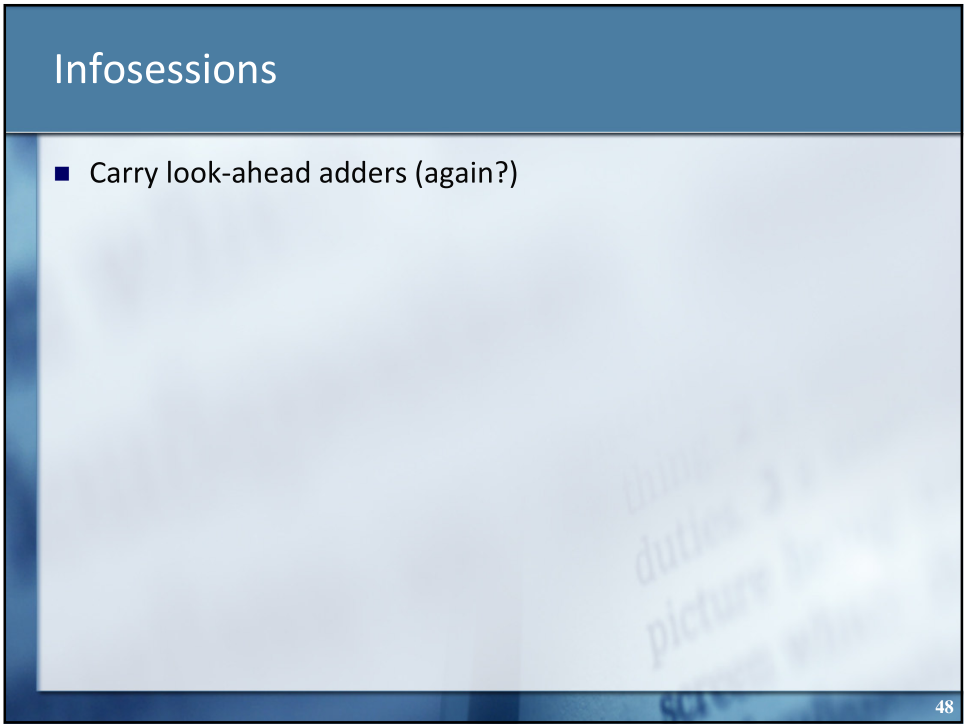
Branch Target Buffers Operation

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken



Dynamic Branch Prediction Summary

- Prediction becoming important part of execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
  - Either different branches
  - Or different executions of same branches
  - Tournament predictors take insight to next level, by using multiple predictors
  - usually one based on global information and one based on local information, and combining them with a selector
  - In 2006, tournament predictors using ≈ 30K bits are in processors like the Power5 and Pentium 4
- Branch Target Buffer: include branch address & prediction



Infosessions

- Carry look-ahead adders (again?)

48