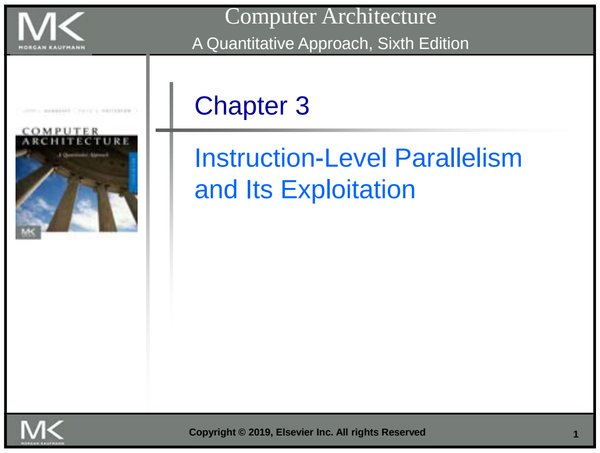


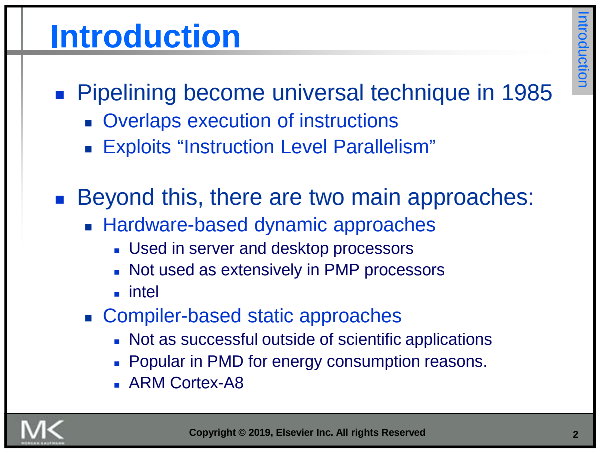
Computer Architecture

A Quantitative Approach, Sixth Edition

Chapter 3

Instruction-Level Parallelism and Its Exploitation

1

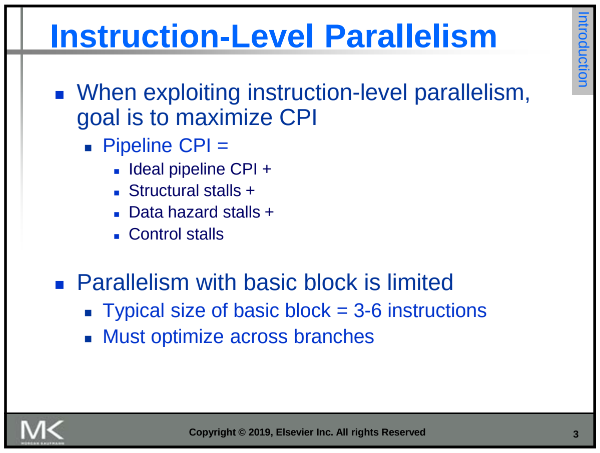


Introduction

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
    - intel
  - Compiler-based static approaches
    - Not as successful outside of scientific applications
    - Popular in PMD for energy consumption reasons.
    - ARM Cortex-A8

2

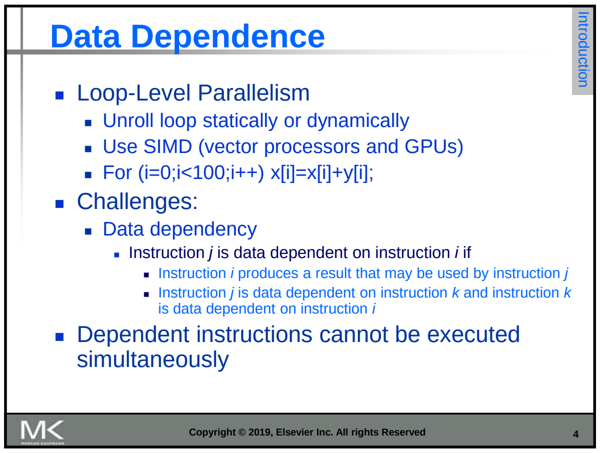


Instruction-Level Parallelism

Introduction

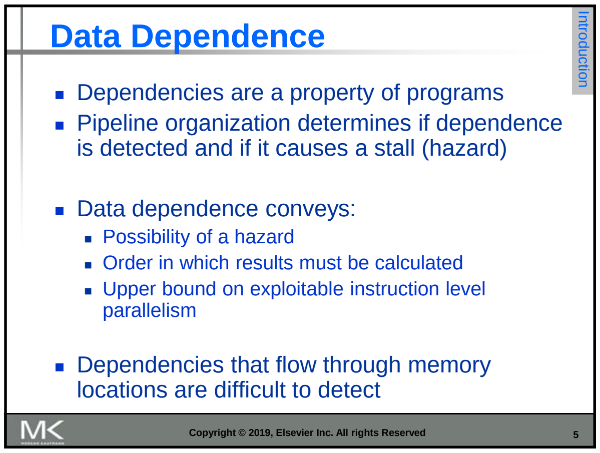
- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

3



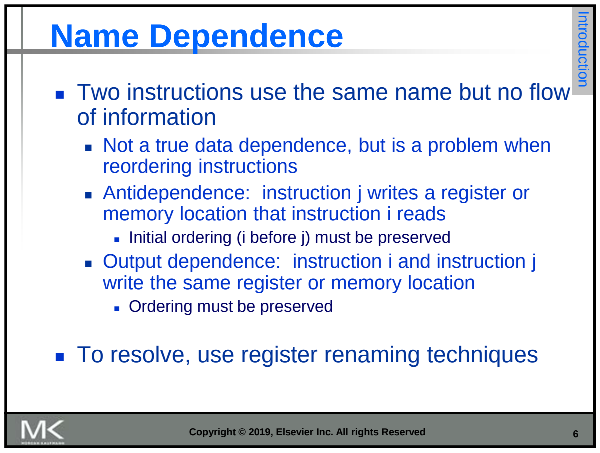
Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)
  - For (i=0;i<100;i++) x[i]=x[i]+y[i];
- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously



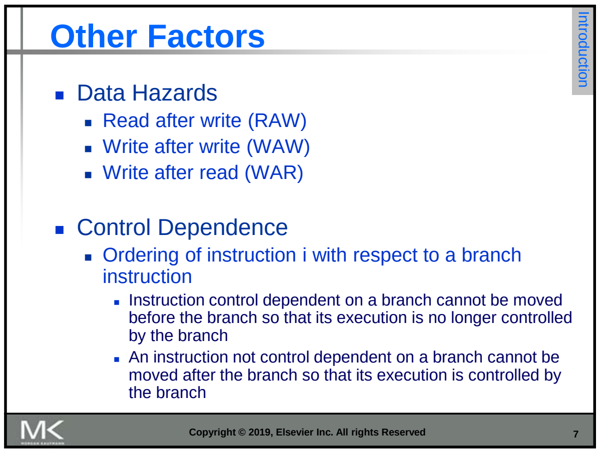
Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall (hazard)
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect



Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, but is a problem when reordering instructions
  - Antidependence: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - Output dependence: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use register renaming techniques

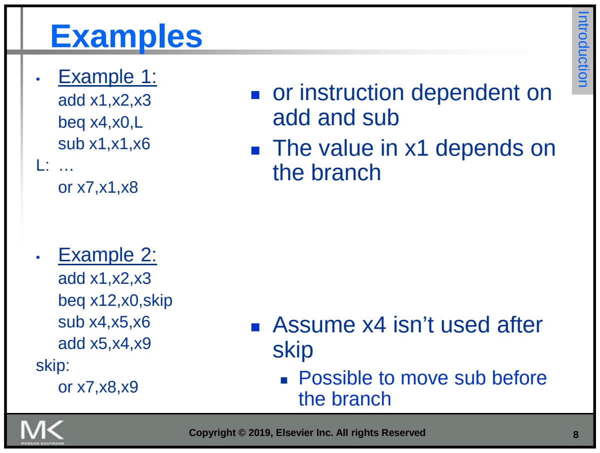


Other Factors

- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Introduction

7



Examples

- Example 1:

```
    add x1,x2,x3
    beq x4,x0,L
    sub x1,x1,x6
L:  …
    or x7,x1,x8
```

- or instruction dependent on add and sub
- The value in x1 depends on the branch

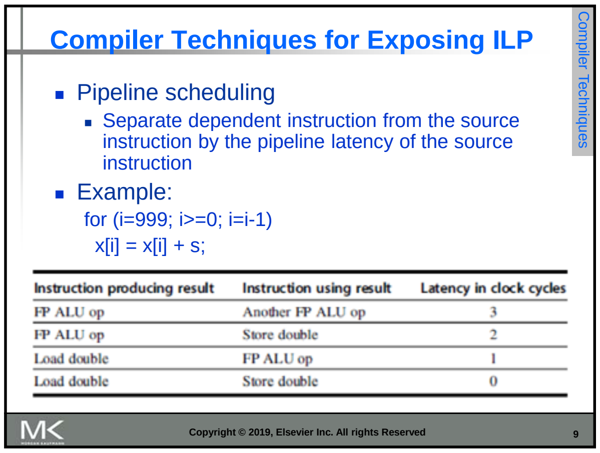
- Example 2:

```
    add x1,x2,x3
    beq x12,x0,skip
    sub x4,x5,x6
    add x5,x4,x9
skip:
    or x7,x8,x9
```

- Assume x4 isn't used after skip
  - Possible to move sub before the branch

Introduction

8



Compiler Techniques for Exposing ILP

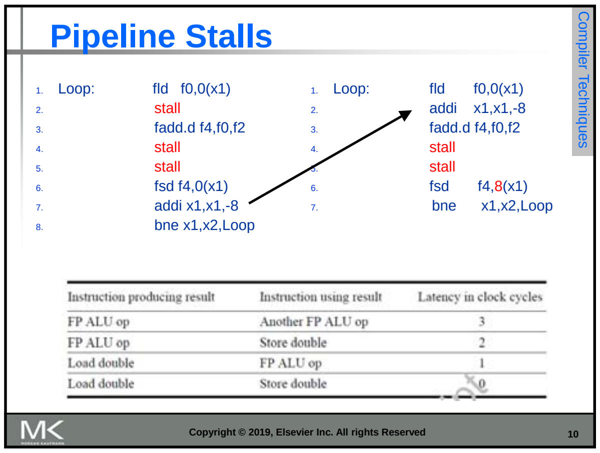
- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

  for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Compiler Techniques

9



Pipeline Stalls

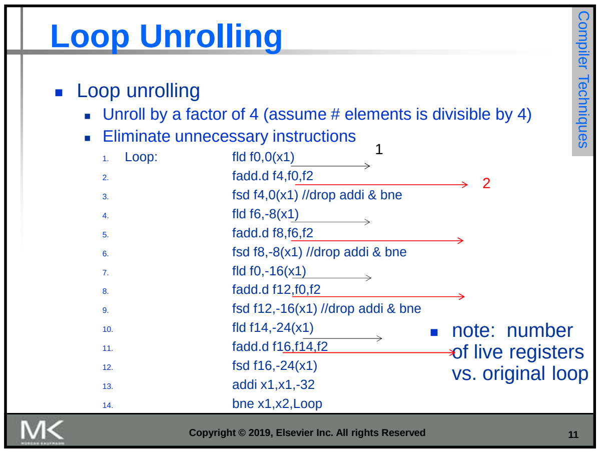
Compiler Techniques

```
Loop:   fld   f0,0(x1)
        stall
        fadd.d f4,f0,f2
        stall
        stall
        fsd f4,0(x1)
        addi x1,x1,-8
        bne x1,x2,Loop
```

```
Loop:   fld     f0,0(x1)
        addi    x1,x1,-8
        fadd.d f4,f0,f2
        stall
        stall
        fsd     f4,8(x1)
        bne     x1,x2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

10



Loop Unrolling

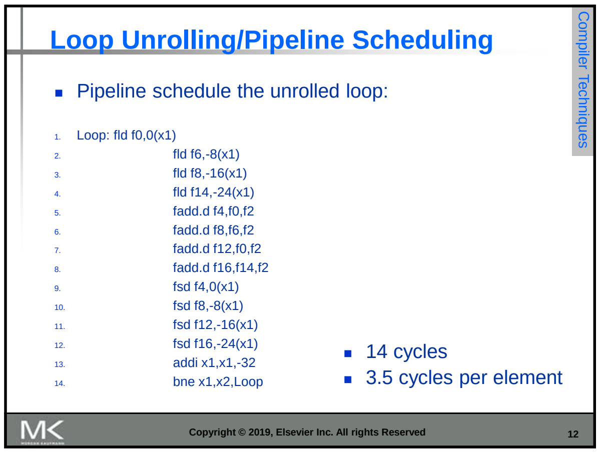
Compiler Techniques

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   fld f0,0(x1)                      1
        fadd.d f4,f0,f2                   2
        fsd f4,0(x1) //drop addi & bne
        fld f6,-8(x1)
        fadd.d f8,f6,f2
        fsd f8,-8(x1) //drop addi & bne
        fld f0,-16(x1)
        fadd.d f12,f0,f2
        fsd f12,-16(x1) //drop addi & bne
        fld f14,-24(x1)
        fadd.d f16,f14,f2
        fsd f16,-24(x1)
        addi x1,x1,-32
        bne x1,x2,Loop
```

- note: number of live registers vs. original loop

11



Loop Unrolling/Pipeline Scheduling

Compiler Techniques

- Pipeline schedule the unrolled loop:

```
Loop: fld f0,0(x1)
      fld f6,-8(x1)
      fld f8,-16(x1)
      fld f14,-24(x1)
      fadd.d f4,f0,f2
      fadd.d f8,f6,f2
      fadd.d f12,f0,f2
      fadd.d f16,f14,f2
      fsd f4,0(x1)
      fsd f8,-8(x1)
      fsd f12,-16(x1)
      fsd f16,-24(x1)
      addi x1,x1,-32
      bne x1,x2,Loop
```

- 14 cycles
- 3.5 cycles per element

12

Strip Mining

Compiler Techniques

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
    - “Strip mining”

13

Branch Prediction

Branch Prediction

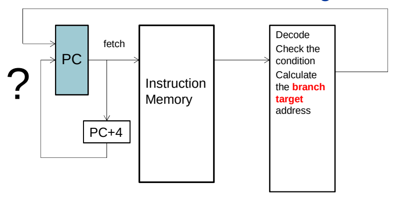
- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
    - (*m*,*n*) predictor: behavior from last *m* branches to choose from 2^m *n*-bit predictors
- Tournament predictor:
  - Combine correlating predictor with local predictor

14

Branch Prediction

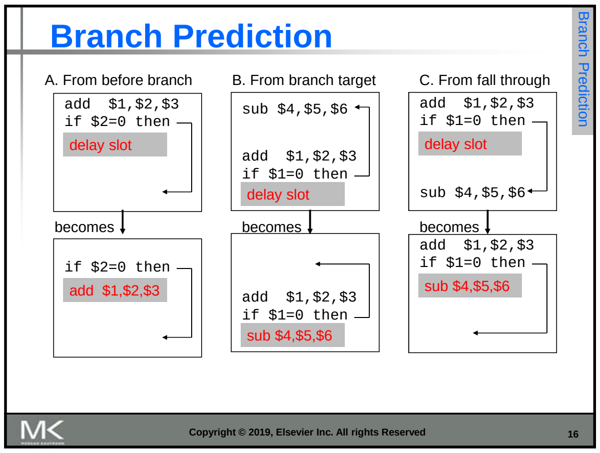
Branch Prediction

- Branching completes in 2 cycles – We know the target address after the second stage



- 1 Cycle delay

15



Branch Prediction

A. From before branch

```
add  $1,$2,$3
if $2=0 then
delay slot
```

becomes

```
if $2=0 then
add  $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
delay slot
```

becomes

```
add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

C. From fall through

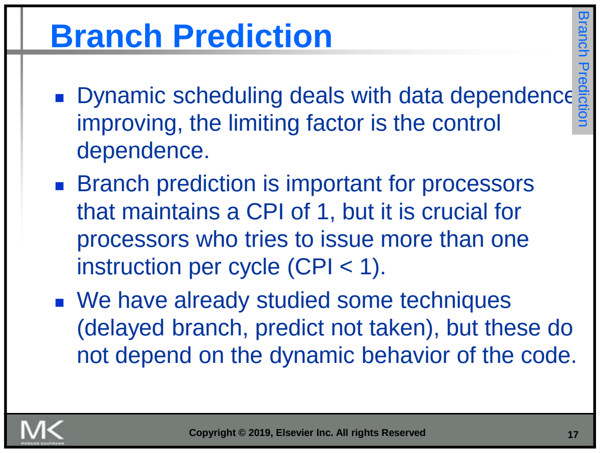
```
add  $1,$2,$3
if $1=0 then
delay slot

sub $4,$5,$6
```

becomes

```
add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

Branch Prediction

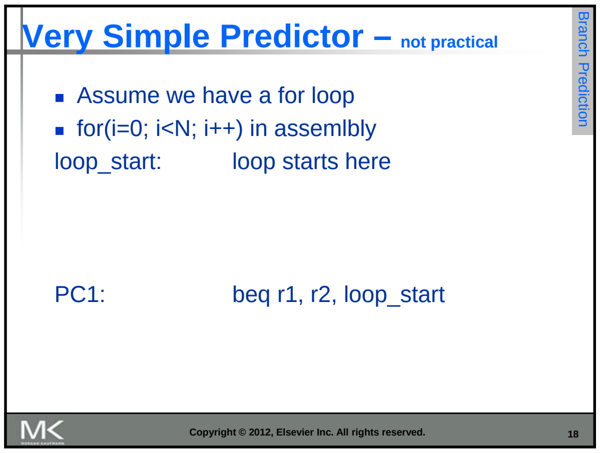
16



Branch Prediction

- Dynamic scheduling deals with data dependence improving, the limiting factor is the control dependence.
- Branch prediction is important for processors that maintains a CPI of 1, but it is crucial for processors who tries to issue more than one instruction per cycle (CPI < 1).
- We have already studied some techniques (delayed branch, predict not taken), but these do not depend on the dynamic behavior of the code.

17



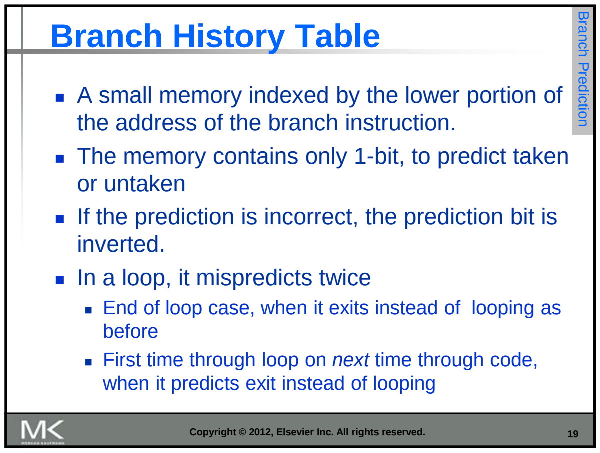
Very Simple Predictor – not practical

- Assume we have a for loop
- for(i=0; i<N; i++) in assemlbly

```
loop_start:     loop starts here



PC1:            beq r1, r2, loop_start
```

Branch Prediction

18

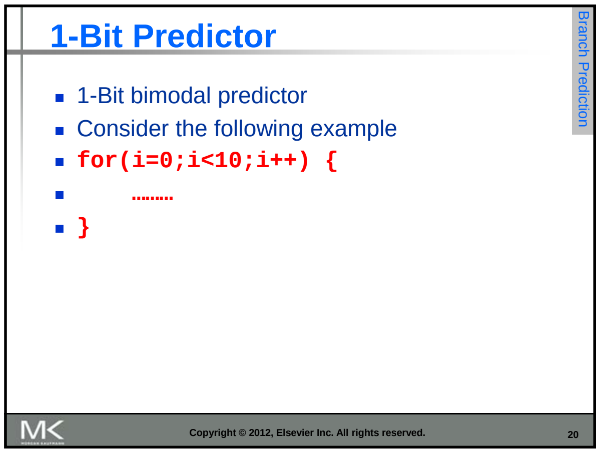


Branch History Table

- A small memory indexed by the lower portion of the address of the branch instruction.
- The memory contains only 1-bit, to predict taken or untaken
- If the prediction is incorrect, the prediction bit is inverted.
- In a loop, it mispredicts twice
  - 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

Branch Prediction

19

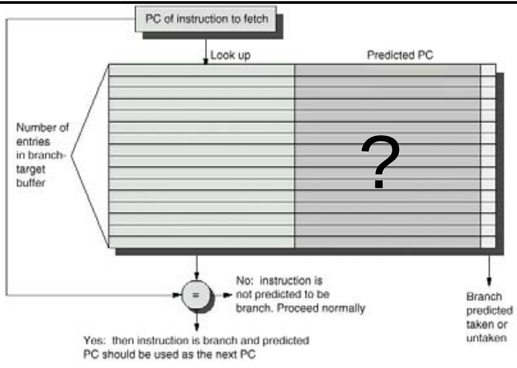


1-Bit Predictor

- 1-Bit bimodal predictor
- Consider the following example
- `for(i=0;i<10;i++) {`
- ………
- `}`

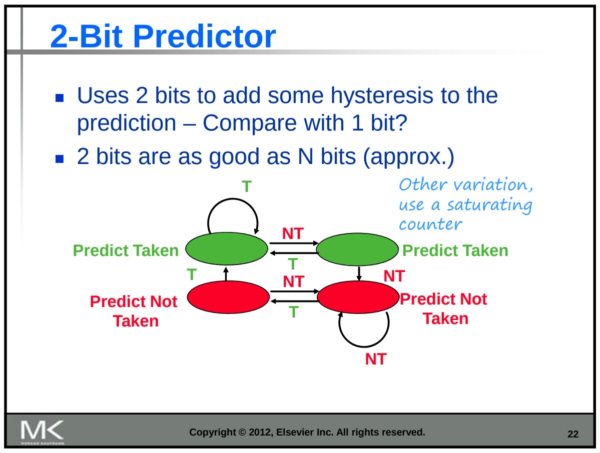
Branch Prediction

20


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

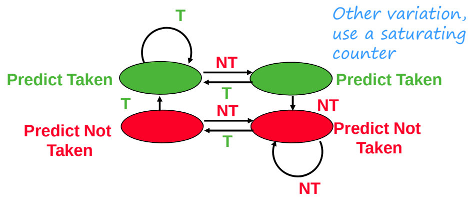
Consider a loop that is taken 9 times in a row then not taken

21

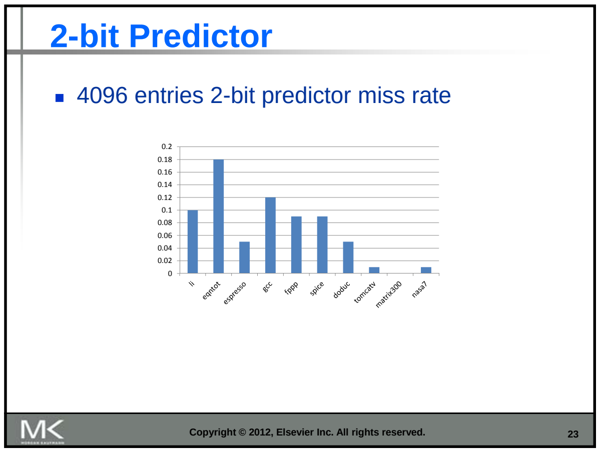


2-Bit Predictor

- Uses 2 bits to add some hysteresis to the prediction – Compare with 1 bit?
- 2 bits are as good as N bits (approx.)

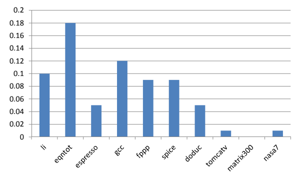


22



2-bit Predictor

- 4096 entries 2-bit predictor miss rate



23