Computer Architecture

Superpipelining and Branch Prediction

Ting-Jung Chang

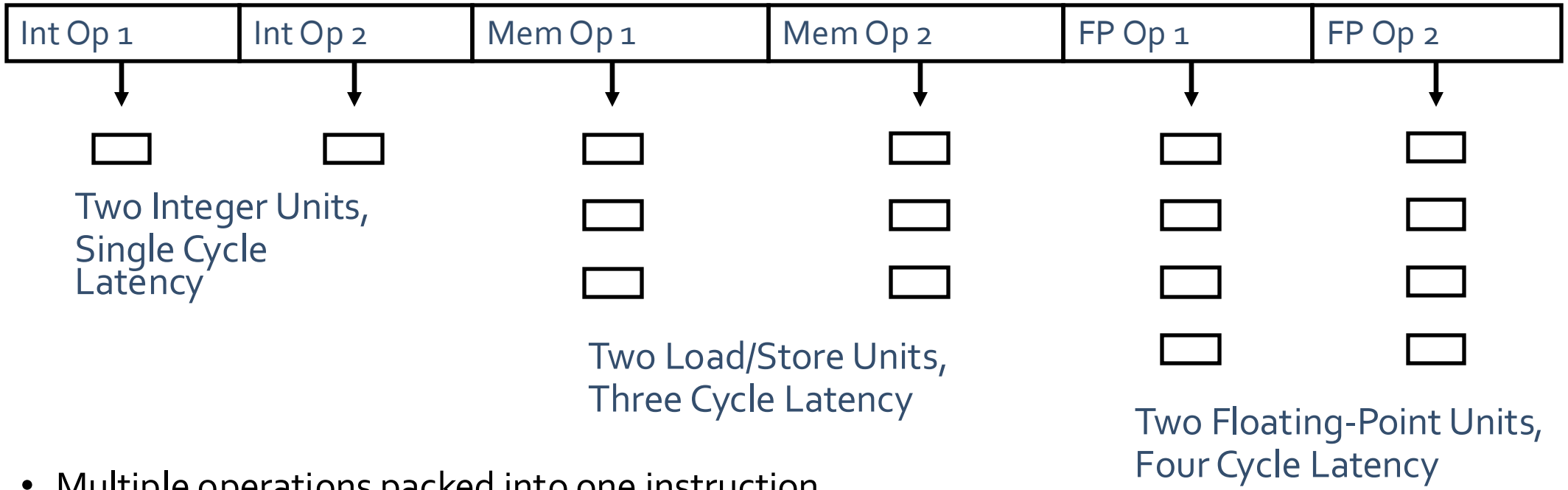
NYCU CS

Recap: Exploiting ILP (Instruction-Level Parallelism)

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

H&P 6, Fig. 3.19

Recap: VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks

Recap: Limits of Static Scheduling

- Statically unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena.
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.
  - Google's Tensor Processing Unit is a VLIW machine.

Course Administration

- PS2 solution released
- PS3 is out
- Midterm 10/21
  - Lecture 1-6/7
  - Ps1 – Ps3
  - 90 minutes closed-book exam (13:20~)

Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Agenda

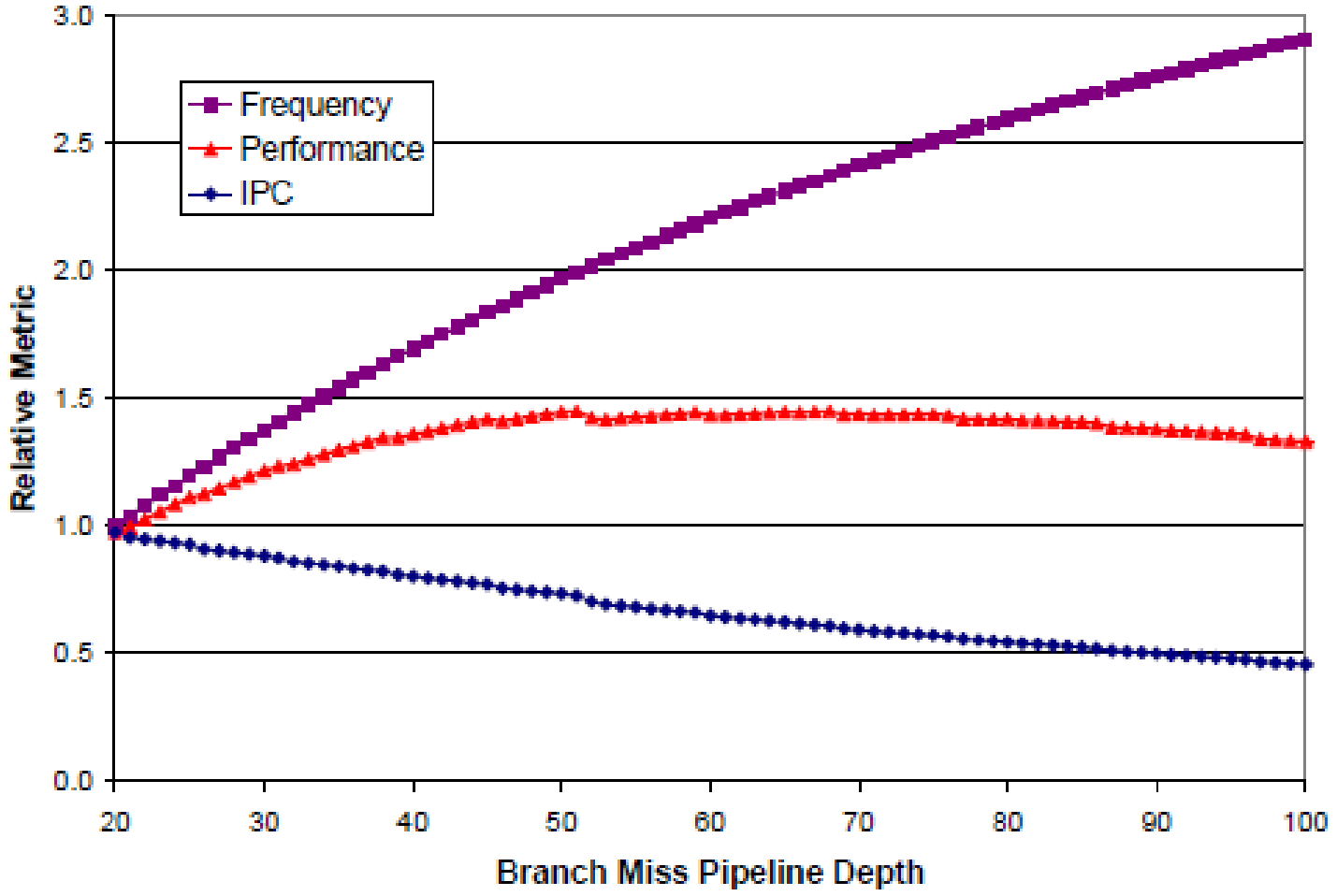
- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Raising Clock Frequency

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Reducing cycle-time reduces program time
  - Increase pipeline depth
- Reducing CPI reduces program time
  - Wider issue/execute processors
- Reducing number of instructions reduces program time
  - Compiler optimization and complex instructions

Increasing Frequency Increases CPI



[Sprangle and Carmean ISCA 2002]

Workload Influences Optimal Pipeline Depth

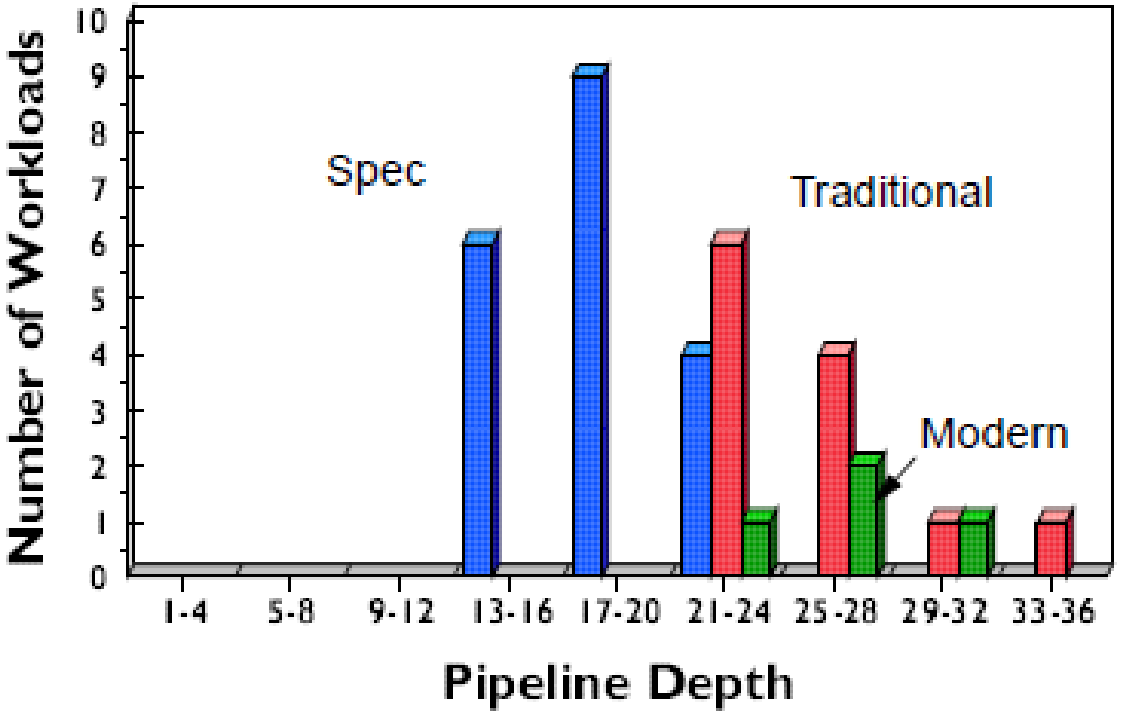
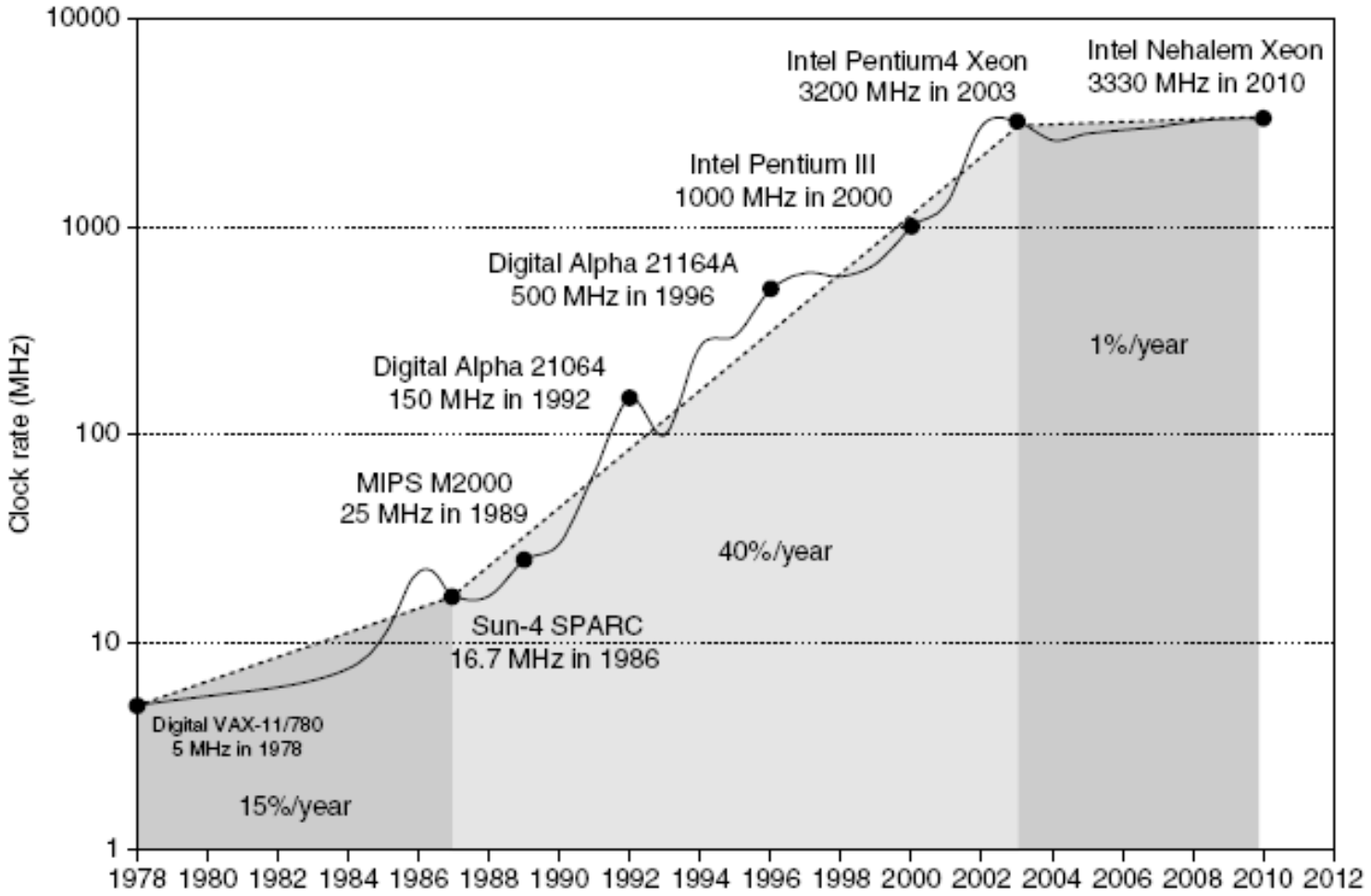


Fig. 9 shows the distribution of pipeline depth optima for different workload classes. Blue are the Spec (C) workloads, red are the traditional workloads, and green are the modern (C++ and Java) workloads.

[Harstein and Puzak ISCA 2002]

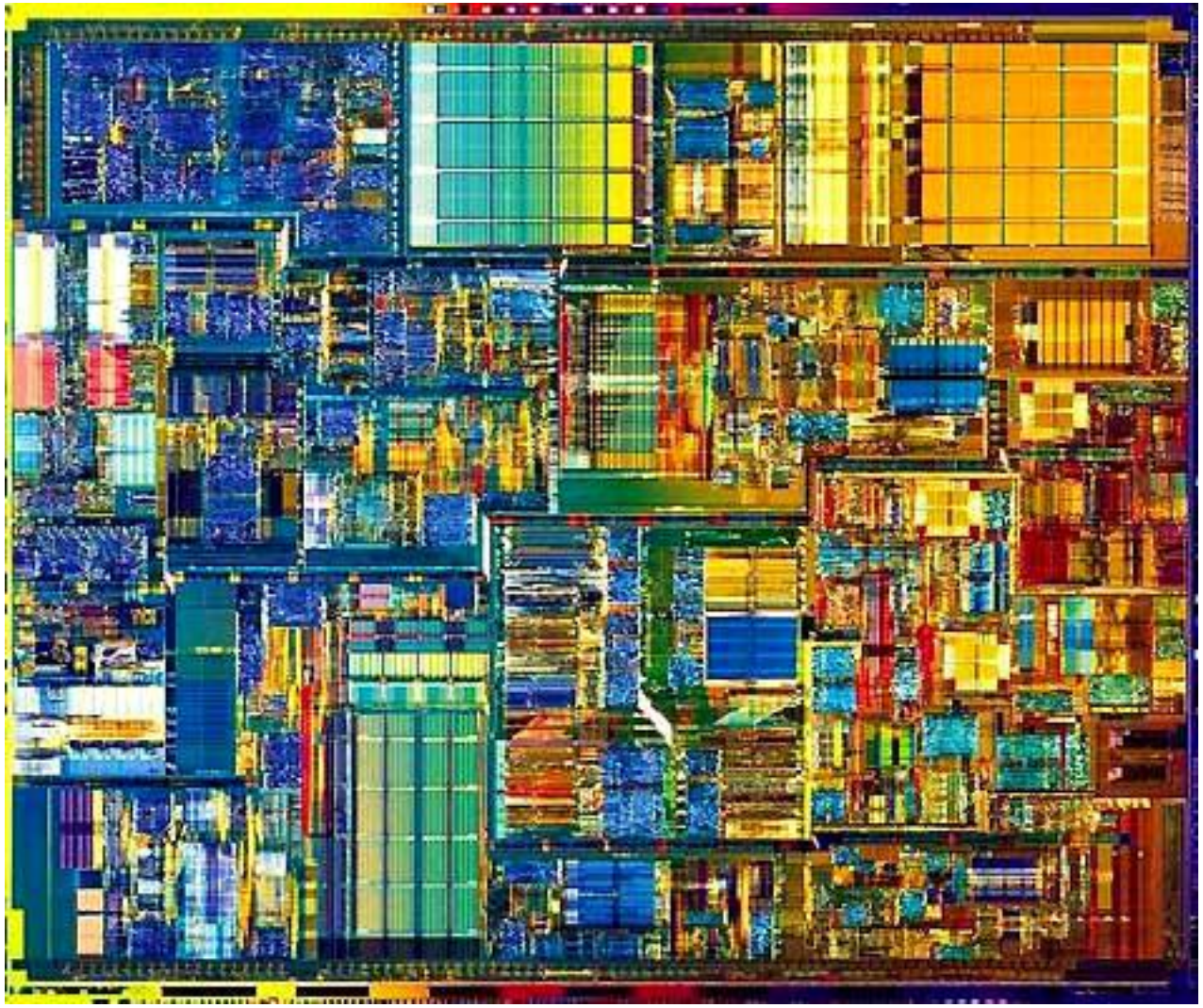
Clock Frequency Improvement



Agenda

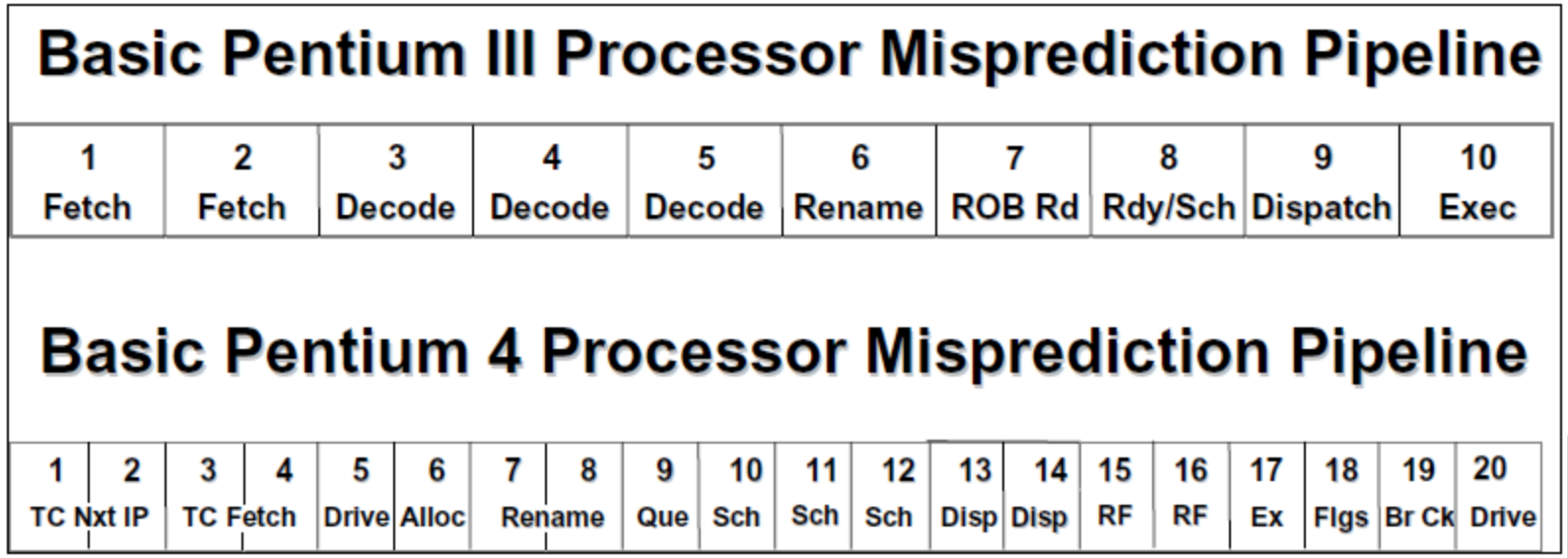
- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Pentium 4



Intel

Pentium III vs. Pentium 4 Pipeline



Many more stages…

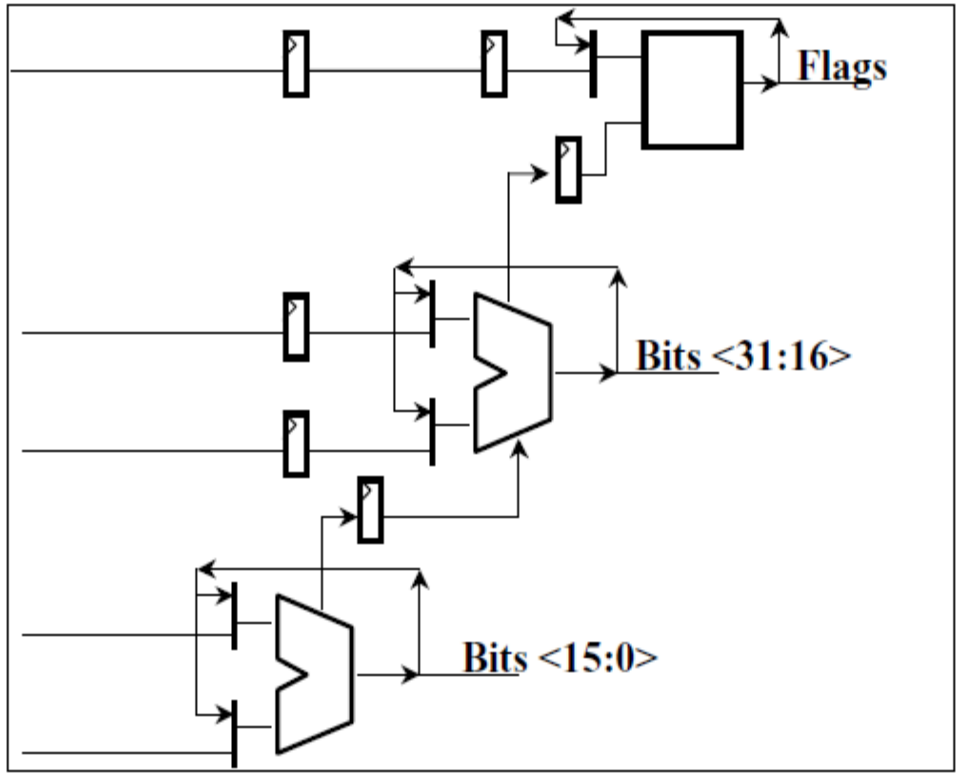
- Drive Stage needed just to drive signals between pipelines

[Hilton et al, Intel Technology Journal]

Pentium 4 Pipeline Stages

- It is a 5-issue superscalar processor
- Increasing the number of pipeline stages increases the clock frequency
  - It took the industry 28 years to hit 1 GHz and only 18 months to reach 2 GHz.
  - The price paid for deeper pipelines is that it is very difficult to ovoid stalls (That is why when Pentium 4 was introduced its performance was worse than Pentium 3.)

Double Pumped ALU



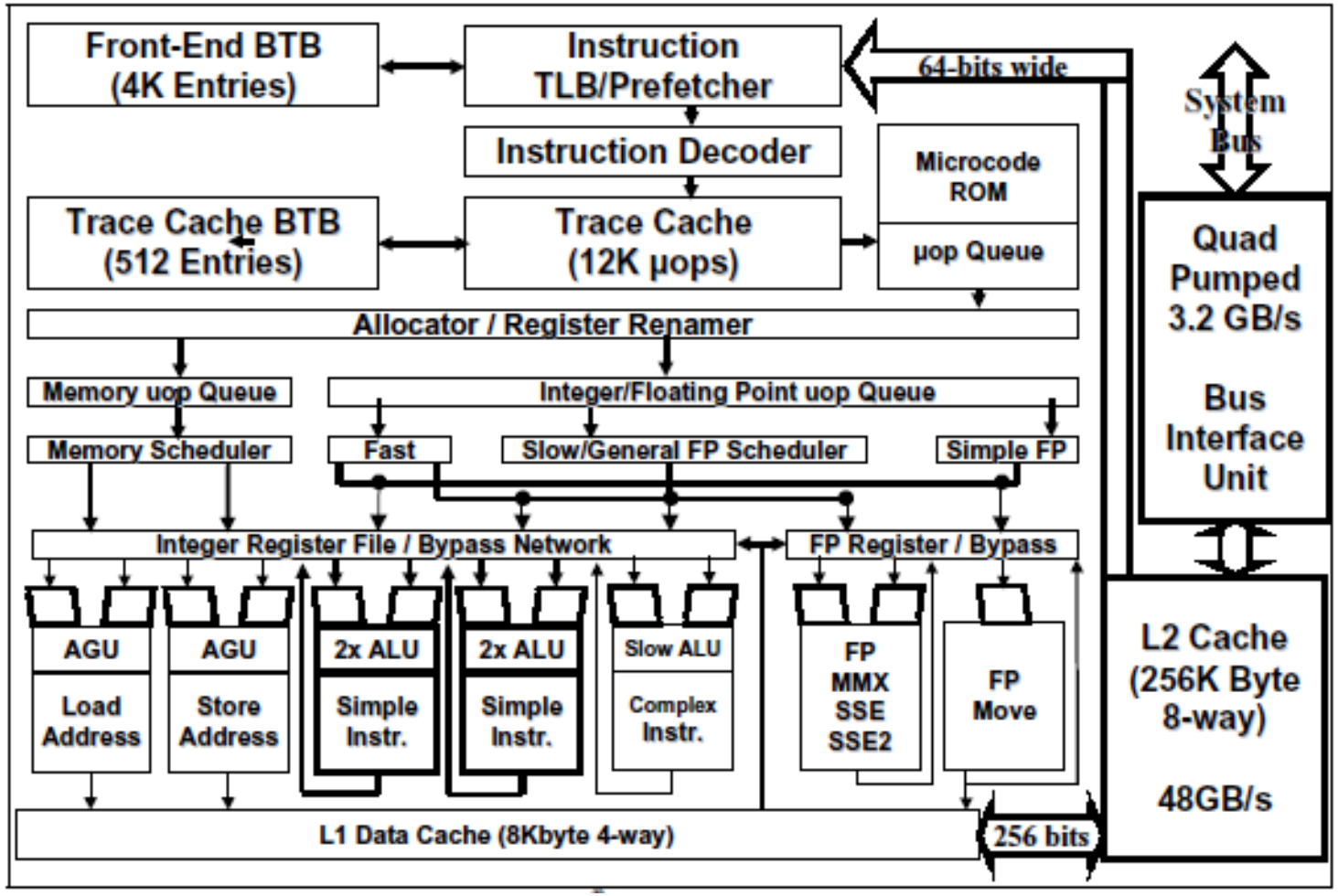
- Executes at 2x core frequency!
- Bypass low bits early within pipeline
- Cut carry of 32-bit add
- Calculate Flags in 3rd cycle

[Hilton et al, Intel Technology Journal]

Small L1 Cache for Low Latency

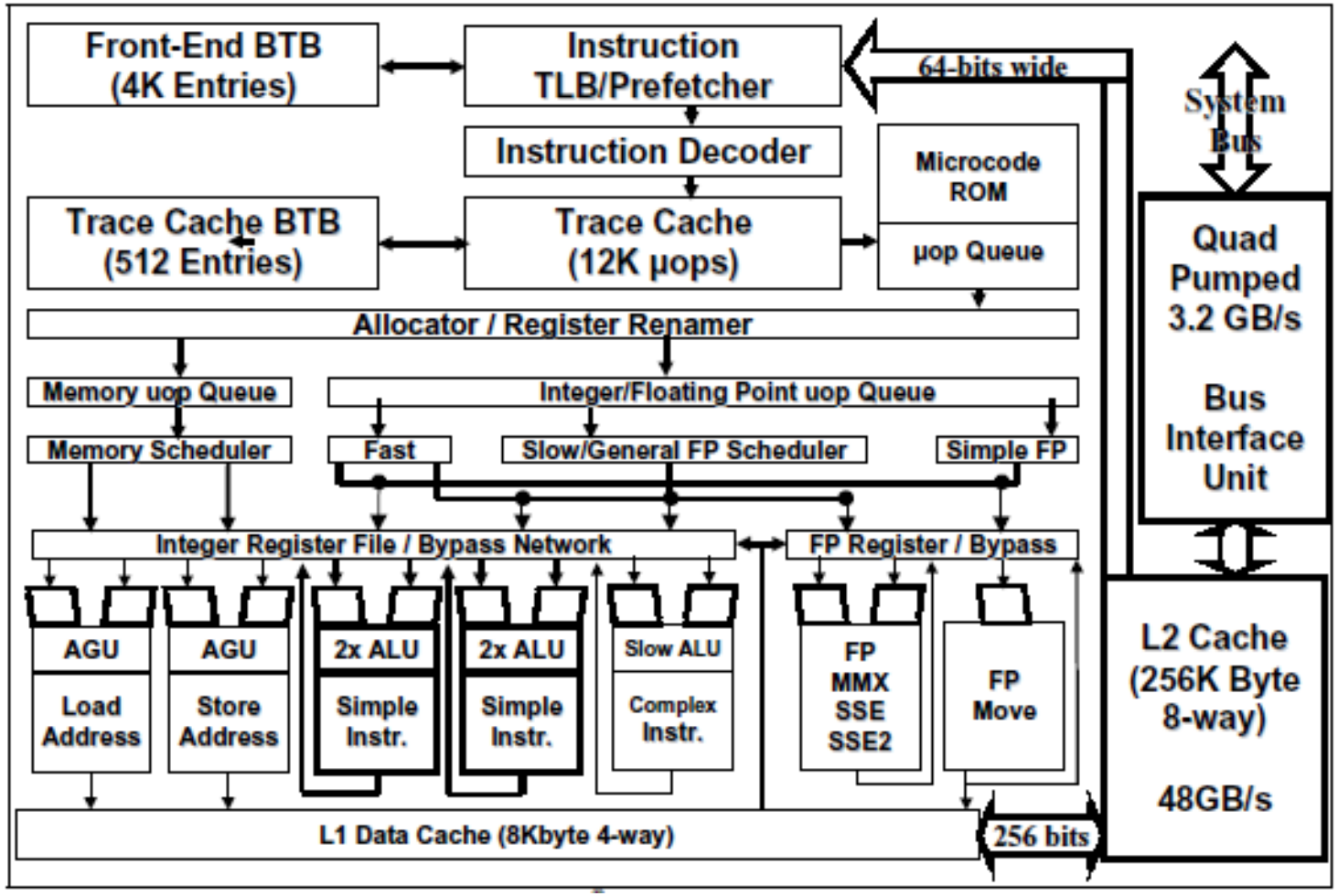
- 8KB L1 Cache
- Much smaller than contemporary L1 caches
- Done to allow low latency of 2-cycles.
- Write-through to L2
- Low latency cache especially important to x86 ISA because so few registers requires going to stack often

Trace Cache



[Hilton et al, Intel Technology Journal]

Pentium 4 Architecture



[Hilton et al, Intel Technology Journal]

Other Deeply Pipelined Processor

Really Deep:

- POWER 7 / POWER 7+ / POWER 8

Deep:

- Intel Core
- Intel Core 2
- Intel Core i3, i5, i7 (Nehalem etc.)
- Intel Core i3, i5, i7 2nd Gen. (Sandy/Ivy Bridge)
- POWER 9

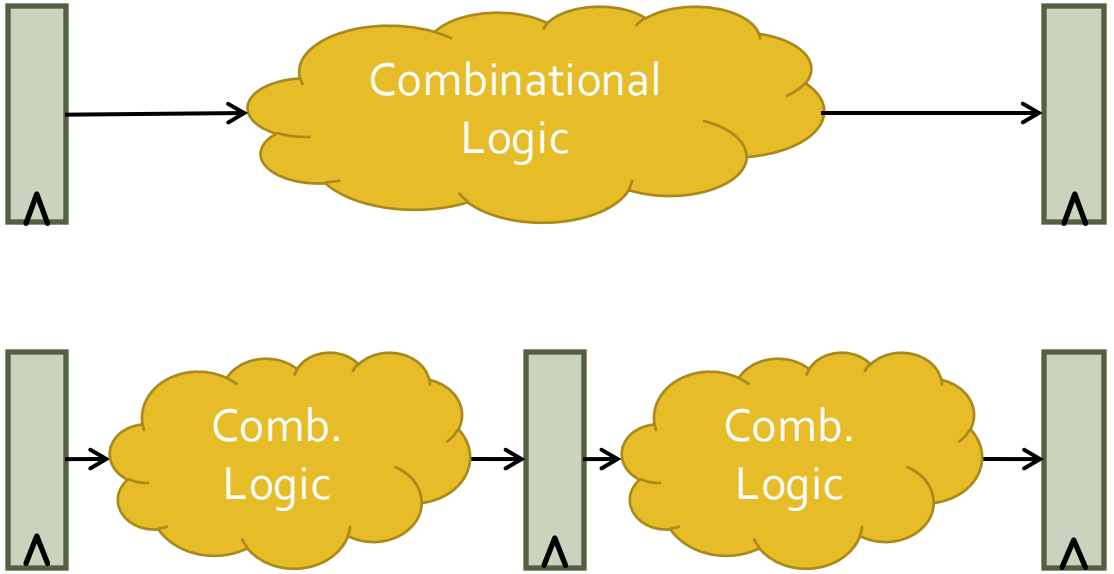
Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Problems with Superpipelining

- Hard to build
- Hard to cut critical paths
- More bypassing needed
- Flip-flop overheads become larger part of cycle
- Large pipelining requires more “pipeline” parallelism in program
  - Requires better branch prediction
- Power!!!

Flip Flops/Latches Dominating Critical Path

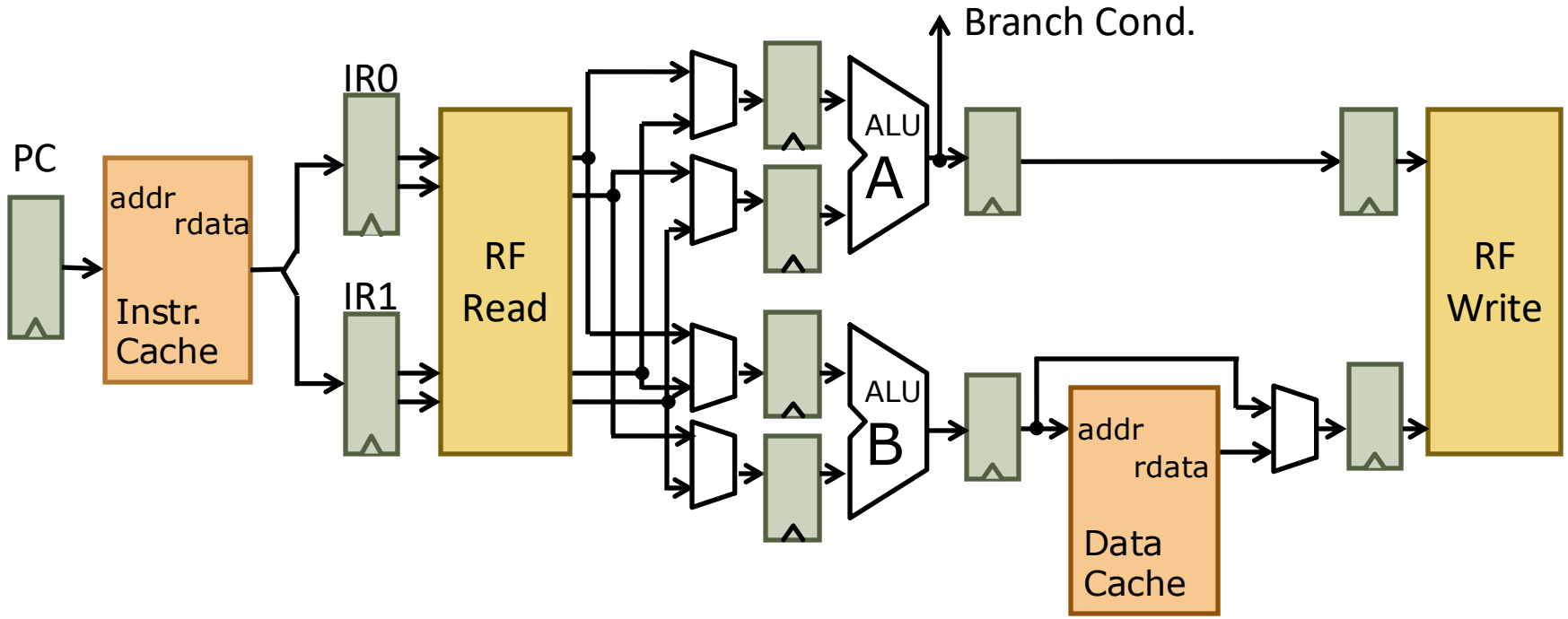


- Flip Flop/Latch delay becomes larger portion of cycle
- Setup Time dominates with short cycles
- Can also have Hold Time problems

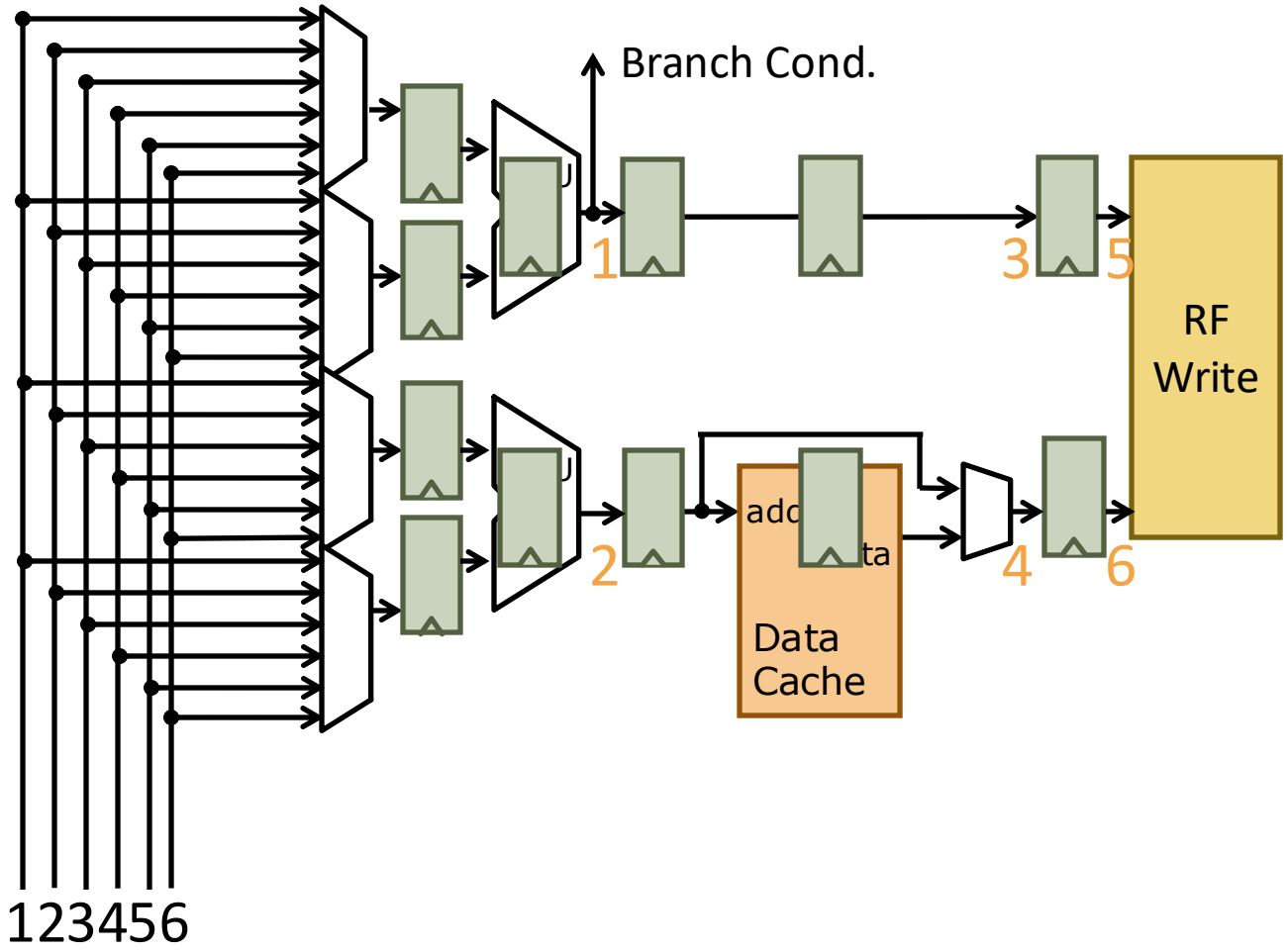
Partitioning/Clustering of Structures

- Faster clock frequency makes global communication challenging
- Faster clock frequency makes centralizes structures harder to meet timing (Ex: Issue Queue, ROB)
- Solution, Partition/Cluster resources and add extra latency to cross between clusters. (Ex: 21264)

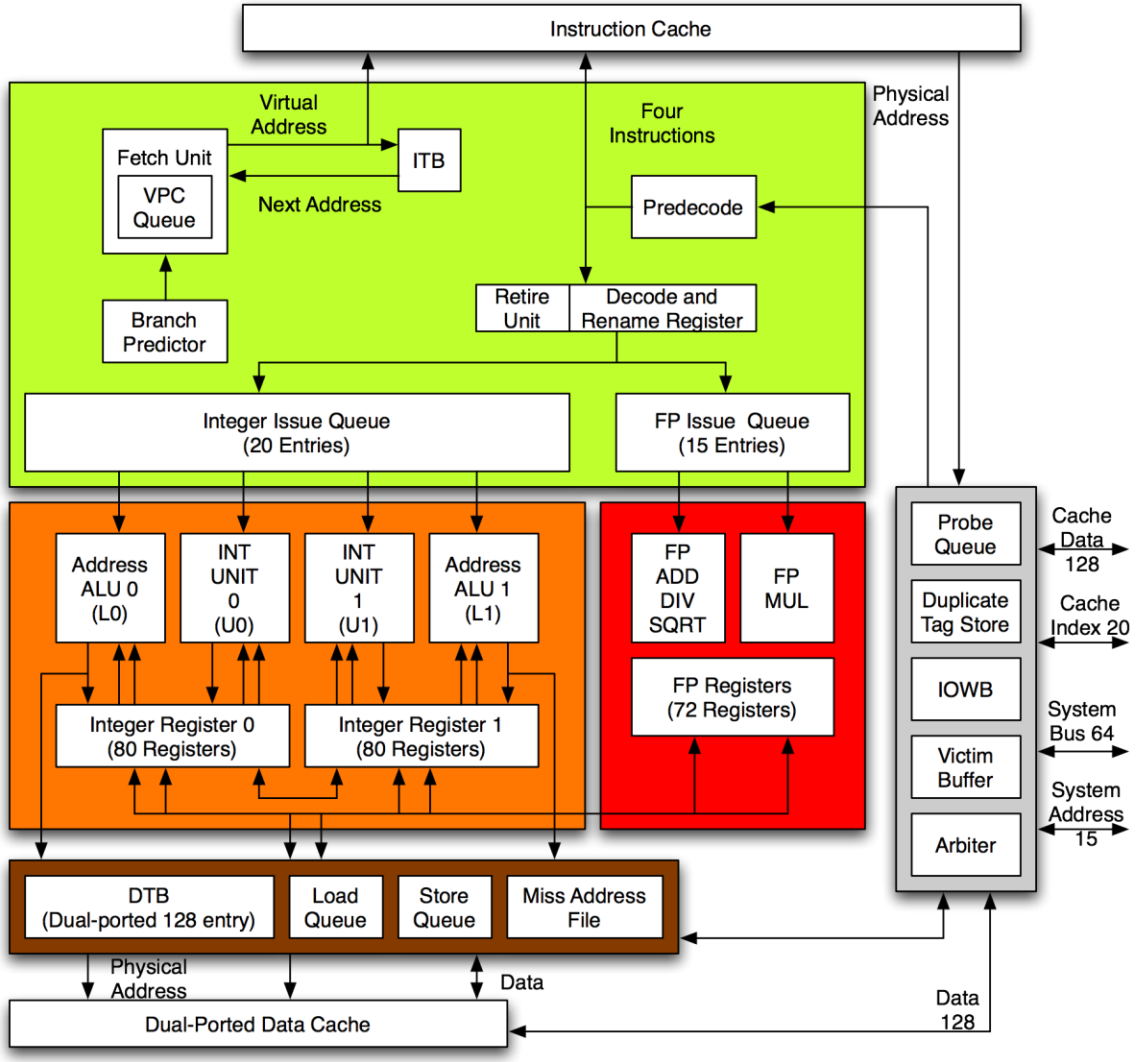
Bypassing in Superscalar Pipelines



Bypassing in Superscalar Pipelines

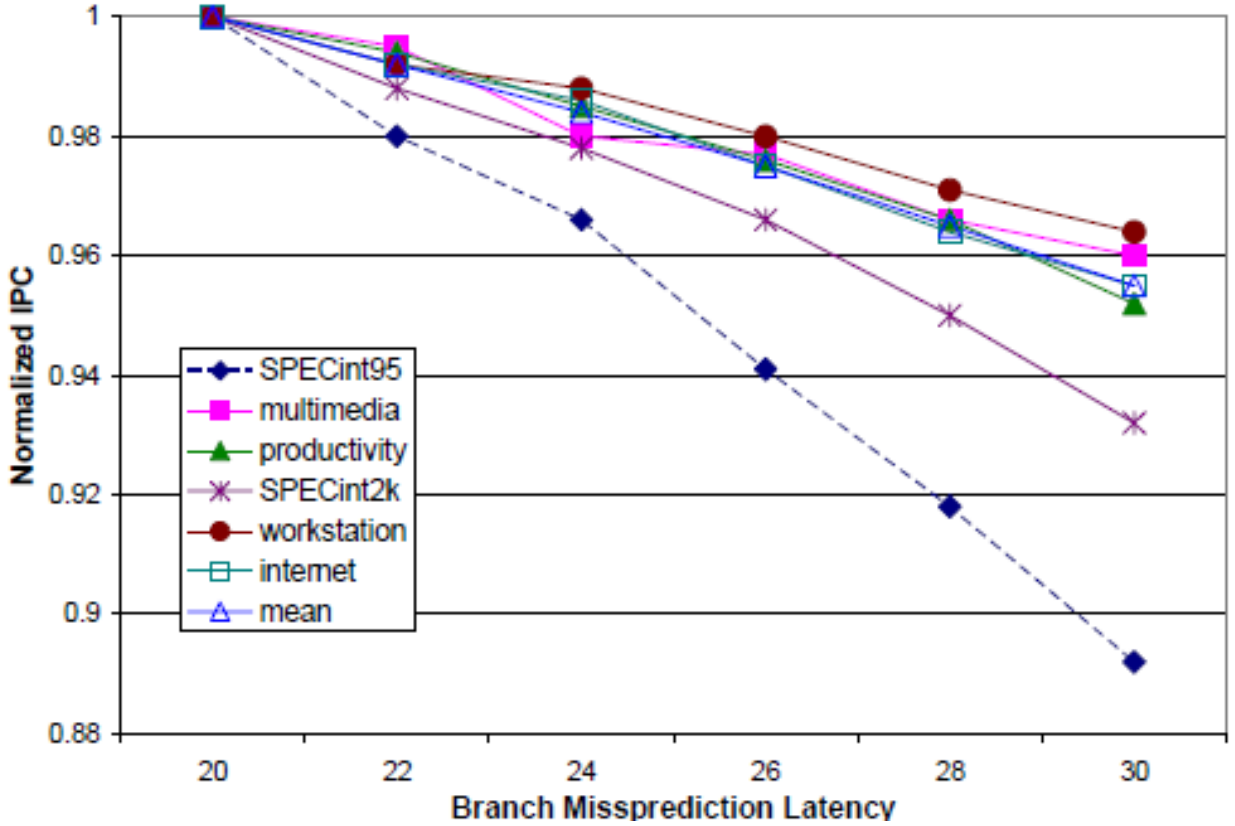


Alpha 21264 Is Clustered



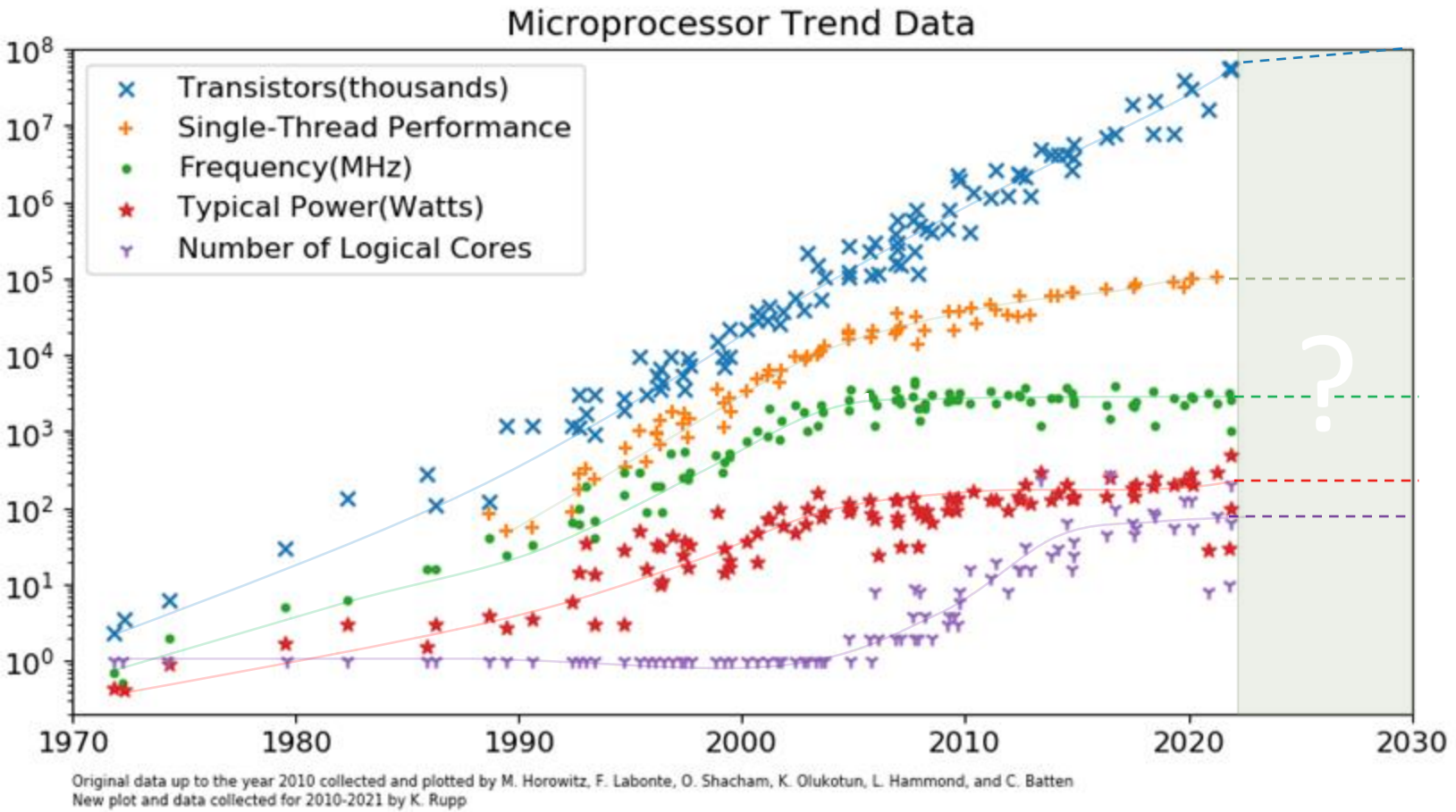
Branch Prediction

- Long Pipes demand spectacular branch predictors



[Sprangle and Carmean ISCA 2002]

Power Limits Performance



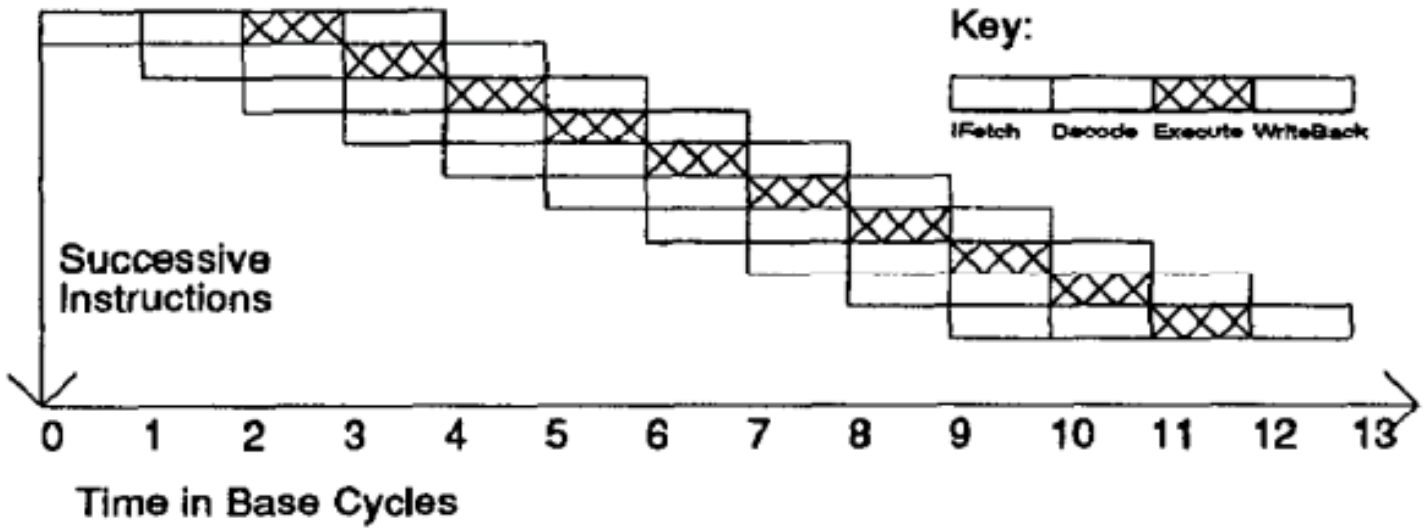
Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Classifying ILP Machines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Classifying ILP Machines

Baseline scalar RISC

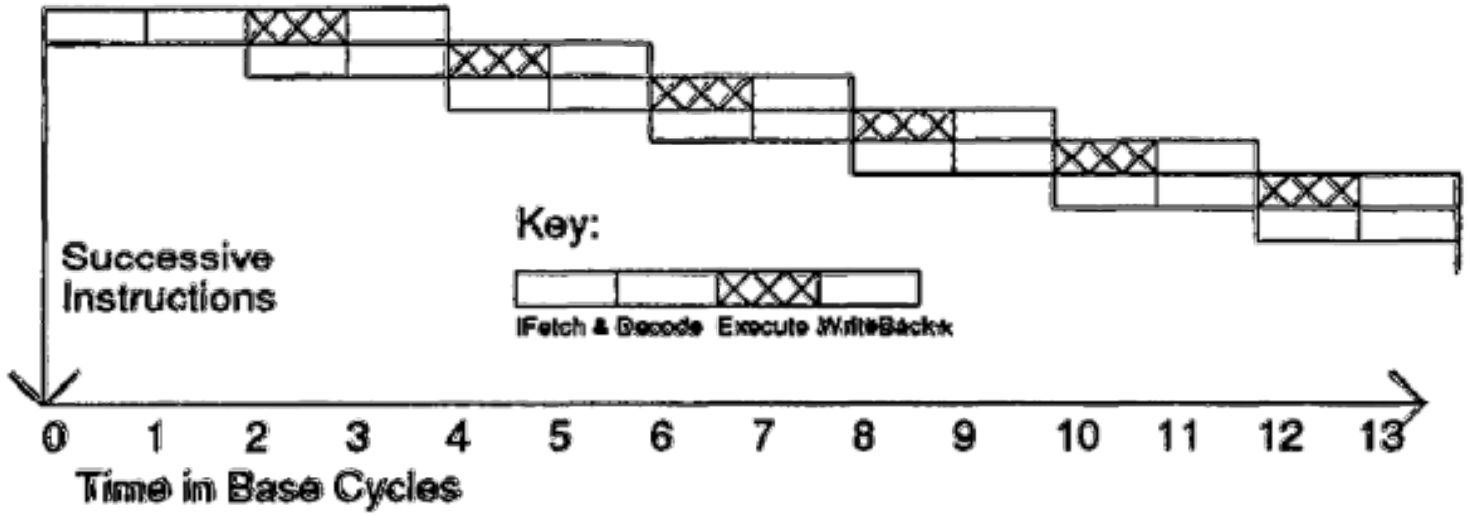
- Issue parallelism = IP = 1
- Operation latency = OP = 1
- Peak IPC = 1



Classifying ILP Machines

Underpipelined Machines

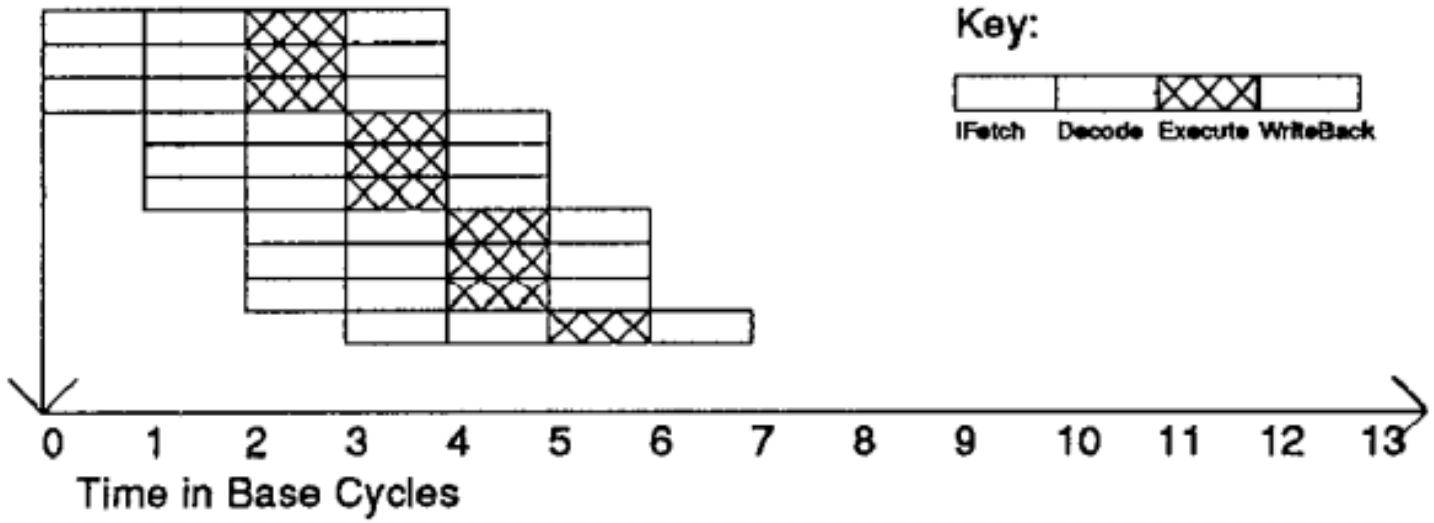
- cycle > operation latency
- issues < 1 instr. per cycle



Classifying ILP Machines

Superscalar Machines

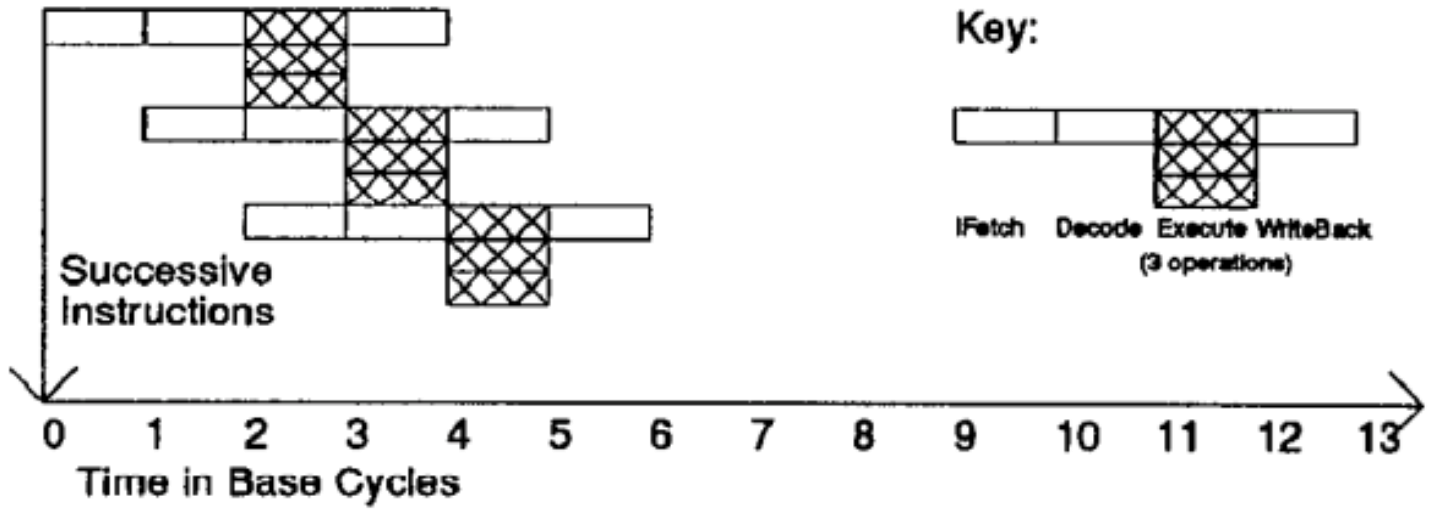
- Issue parallelism = IP = n inst / cycle
- Operation latency = OP = 1 cycle
- Peak IPC = n inst / cycle (n x speedup?)



Classifying ILP Machines

VLIW Machines (Very Long Instruction Word)

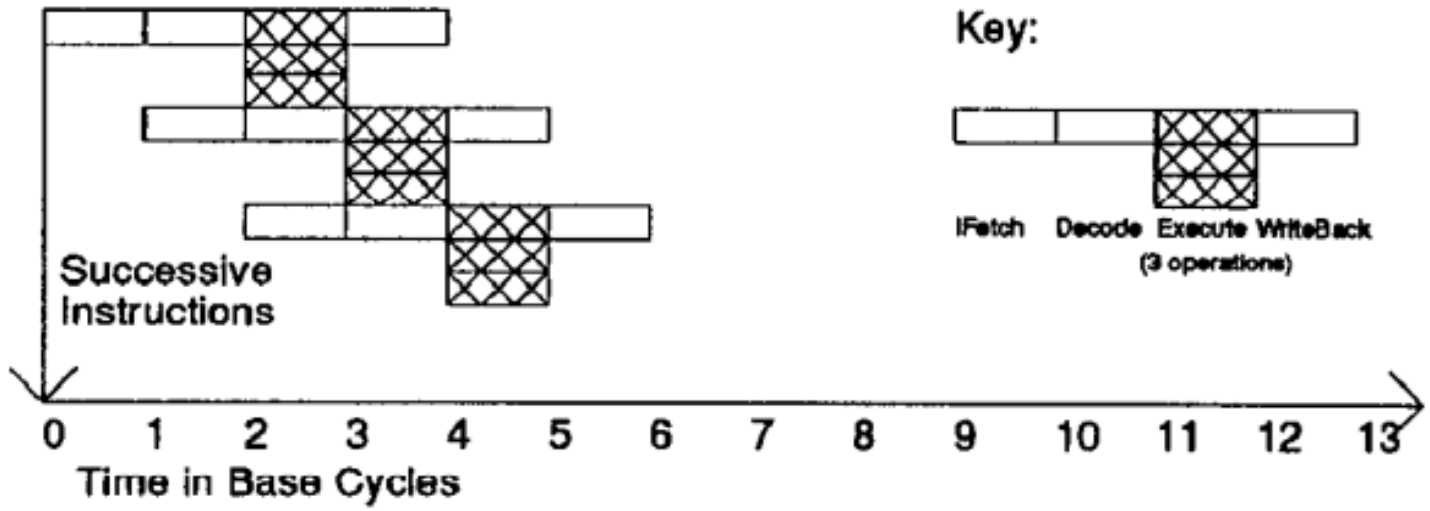
- Issue parallelism = IP = n inst / cycle
- Operation latency = OP = 1 cycle
- Peak IPC = n inst / cycle = 1 VLIW / cycle



Classifying ILP Machines

VLIW Machines (Very Long Instruction Word)

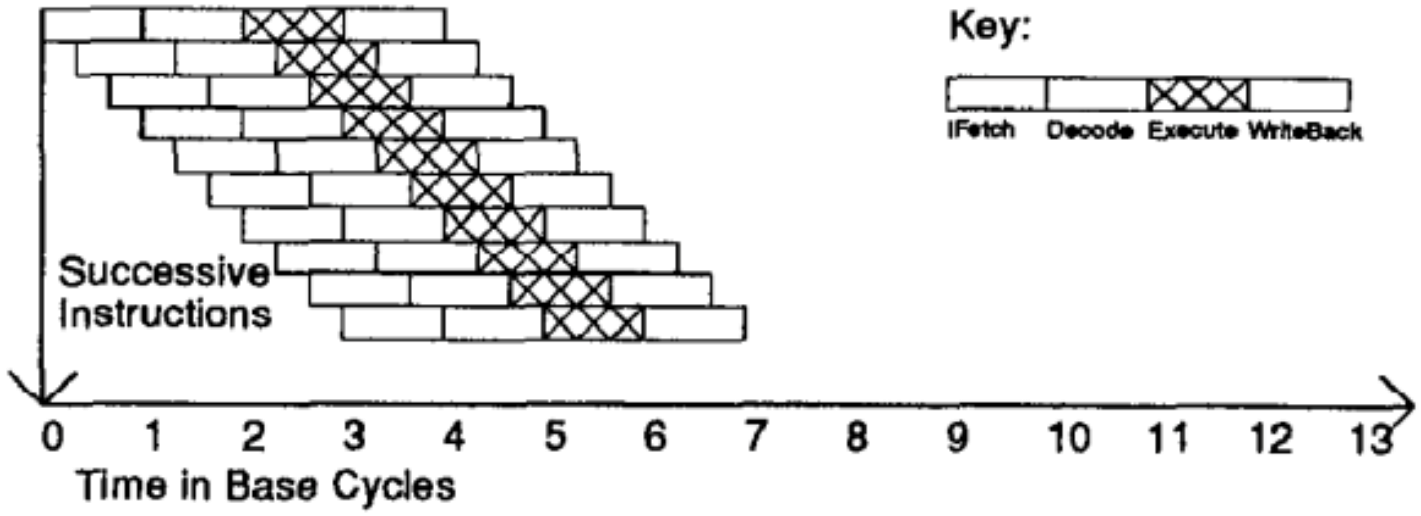
- Issue parallelism = IP = n inst / cycle
- Operation latency = OP = 1 cycle
- Peak IPC = n inst / cycle = 1 VLIW / cycle



Classifying ILP Machines

Superpipelined

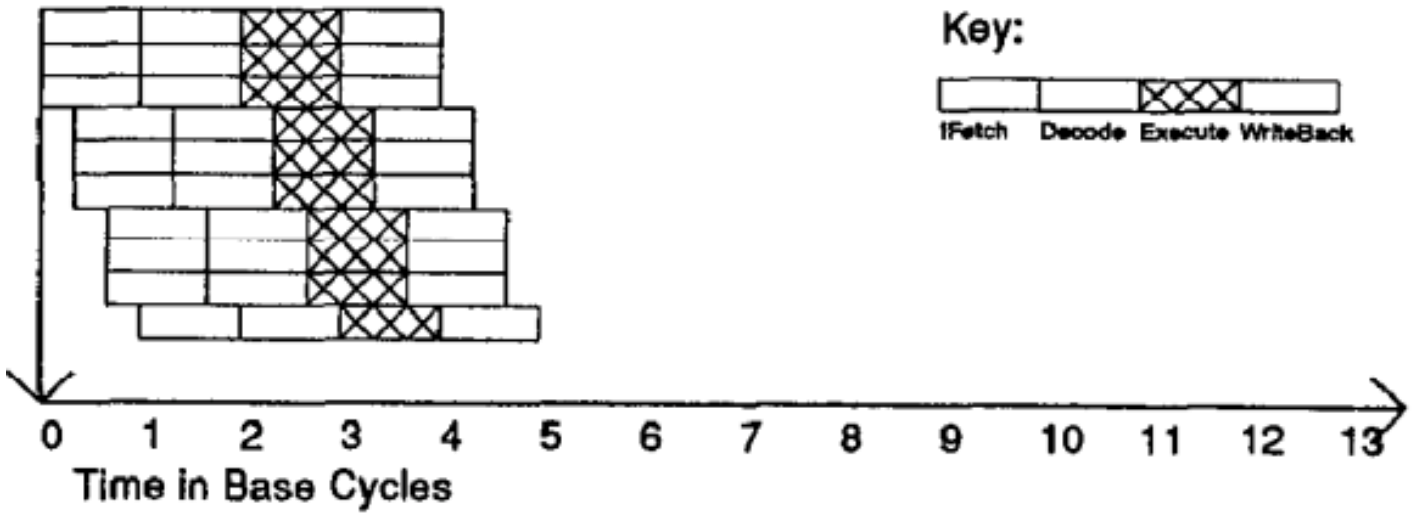
- Issue parallelism = IP = 1 inst / minor cycle
- Operation latency = OP = m minor cycles
- Peak IPC = m inst / major cycle (m x speedup?)



Classifying ILP Machines

Superpipelined-Superscalar

- Issue parallelism = IP = n inst / minor cycle
- Operation latency = OP = m minor cycles
- Peak IPC = n x m instr / major cycle



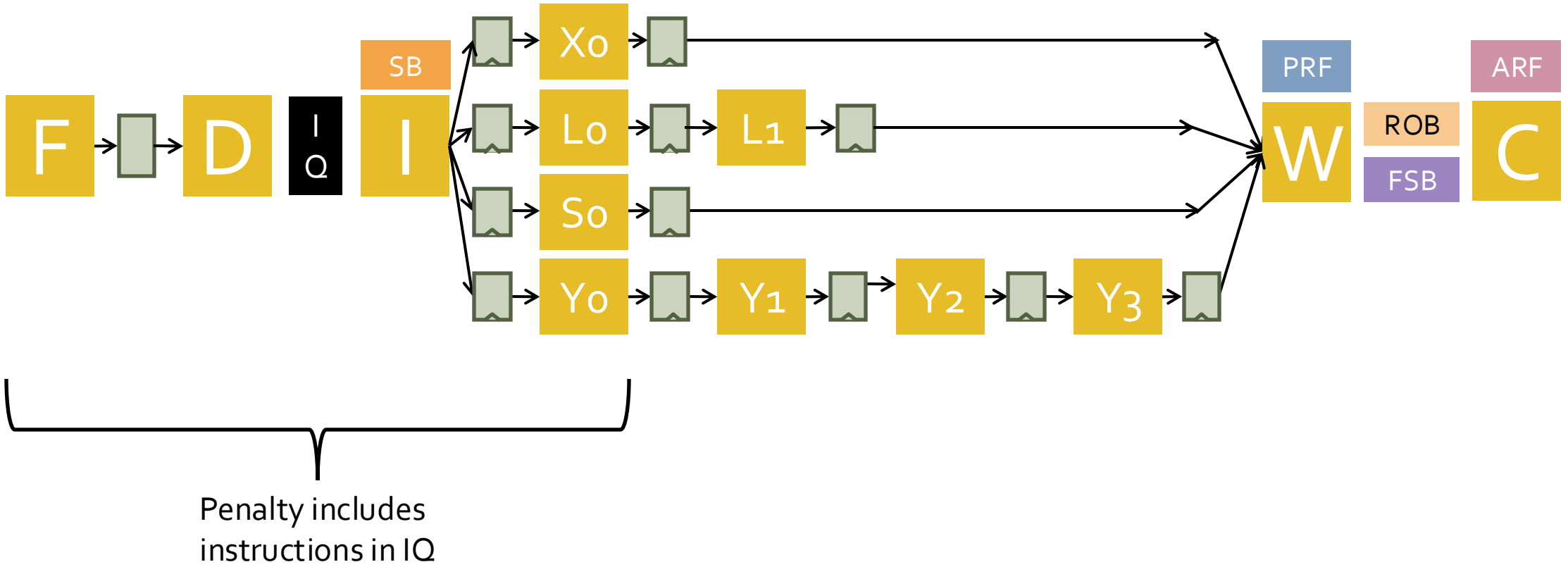
Superscalar vs. Superpipelined

- Roughly equivalent performance
  - If n = m then both have about the same IPC
  - Parallelism exposed in space vs. time

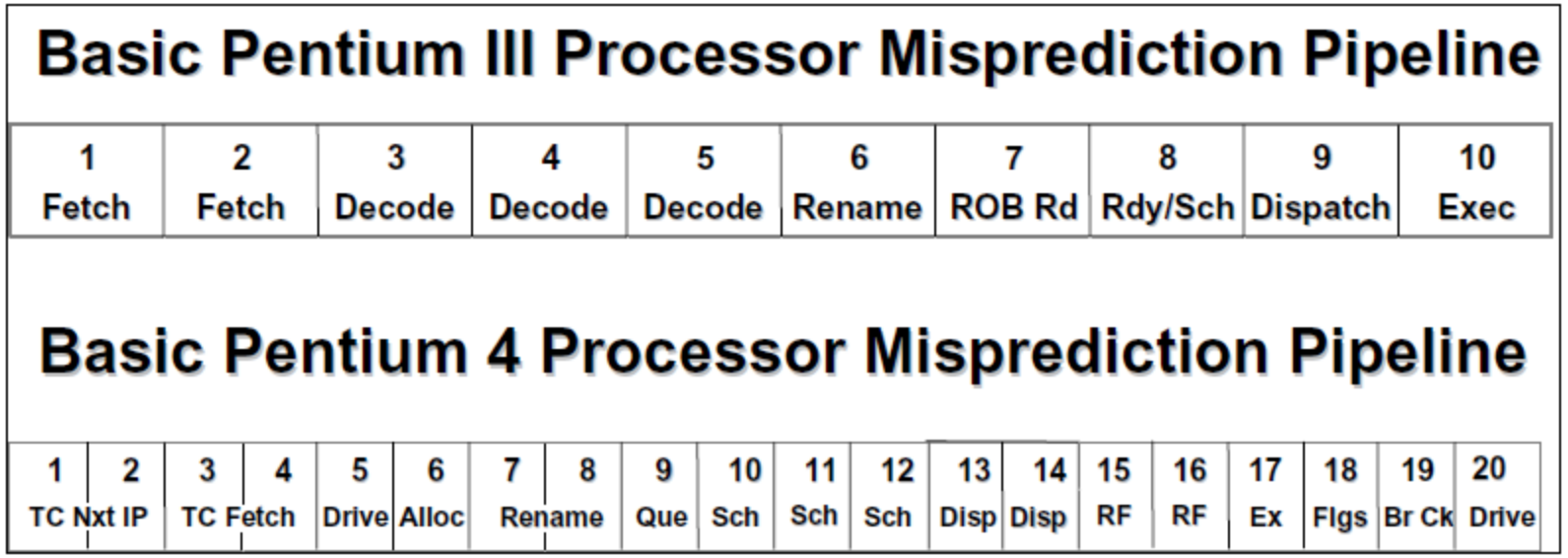
Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Longer Frontends Means More Control Flow Penalty



Longer Pipeline Frontends Amplify Branch Cost

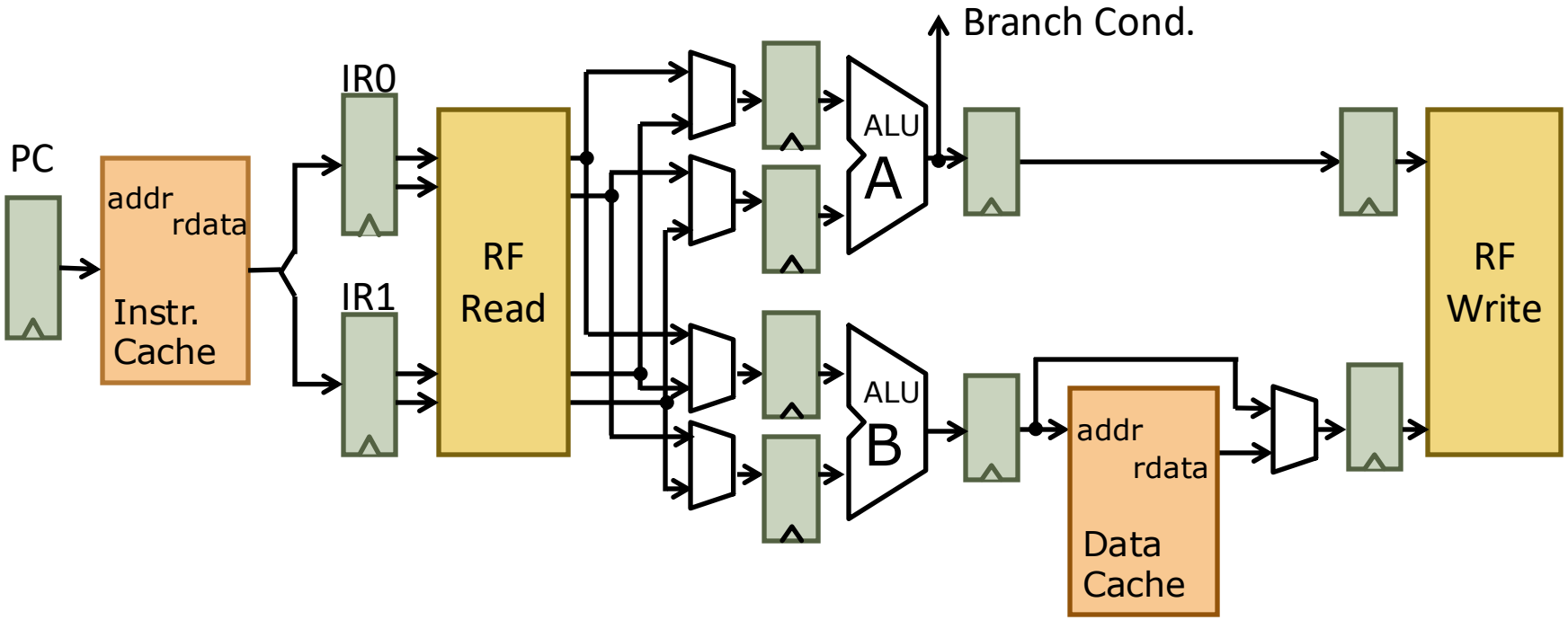


Pentium 3: 10 cycle branch penalty

Pentium 4: 20 cycle branch penalty

Image from: The Microarchitecture of the Pentium 4 Processor by Glenn Hinton et al. Appeared in Intel Technology Journal Q1, 2001. Image courtesy of Intel

Dual Issue and Branch Cost



Superscalars Multiply Branch Cost

```
BEQZ   F   D   I   A0  A1  W
OpA    F   D   I   B0  -   -
OpB        F   D   I   -   -   -
OpC        F   D   I   -   -   -
OpD            F   D   -   -   -   -
OpE            F   D   -   -   -   -
OpF                F   -   -   -   -   -
OpG                F   -   -   -   -   -
OpH                    F   D   I   A0  A1  W
OpI                    F   D   I   B0  B1  W
```

Dual-issue Processor has twice the mispredict penalty

How much work is lost if pipeline doesn't follow correct instruction flow?

~pipeline width x branch penalty

Average Run-Length between Branches

- Average dynamic instruction mix of SPEC CPU 2017 :

	SPECint	SPECfp
Branches	19%	11%
Loads	24%	26%
Stores	10%	7%
Other	47%	56%

[Limaye et al, ISPASS'18]

- SPECint17: perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz
- SPECfp17: bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms

What is the average run length between branches?

Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Branch Prediction

- Essential in modern processors to mitigate branch delay latencies

Two types of Prediction

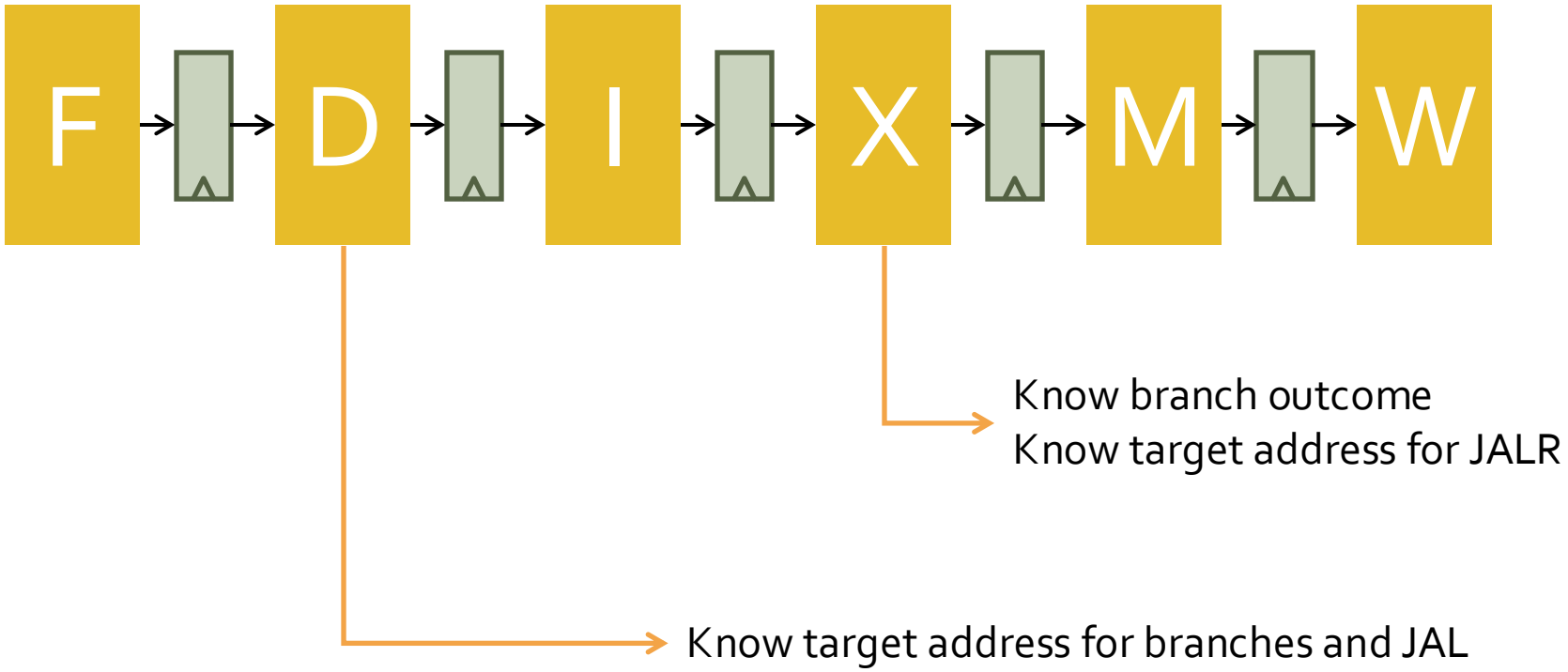
1. Predict Branch Outcome
2. Predict Branch/Jump Address

RISC-V Branches and Jumps

- Each instruction fetch depends on one or two pieces of information from the preceding instruction:
  1. Is the preceding instruction a taken branch?
  2. If so, what is the target address?

Instruction	Taken known?	Target known?
JAL		
JALR		
BRANCH		

Where is the Branch Information Known?



Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches
    - loop unrolling Increases run length between branches
  - Reduce resolution time – instruction scheduling
    - Compute the branch condition as early as possible (of limited value)

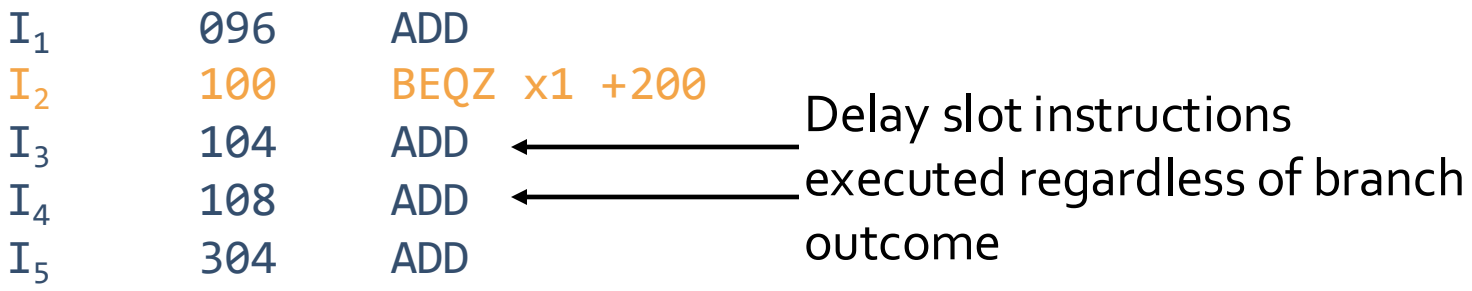
Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches
    - loop unrolling Increases run length between branches
  - Reduce resolution time – instruction scheduling
    - Compute the branch condition as early as possible (of limited value)
- Hardware solutions
  - Bypass – usually results are used immediately
  - Change architecture – find something else to do Delay slots
    - replace pipeline bubbles with useful work (requires software cooperation)
  - Speculate (accurately) – branch prediction
    - Speculative execution of instructions beyond the branch

Branch Delay Slots

(expose control hazard to software)

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.



Branch Prediction

- Motivation:
  - Branch penalties limit performance of deeply pipelined processors
  - Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly
- Required hardware support:
  - Prediction structures:
    - Branch history tables, branch target buffers, etc.
  - Mispredict recovery mechanisms:
    - Keep result computation separate from commit
    - Kill instructions following branch in pipeline
    - Restore state to state following branch

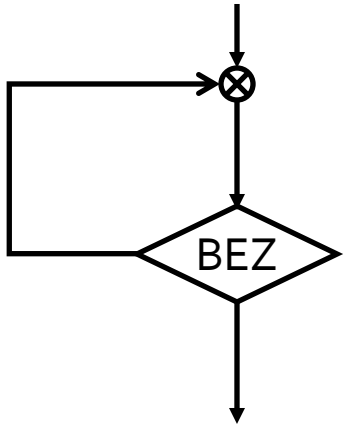
Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

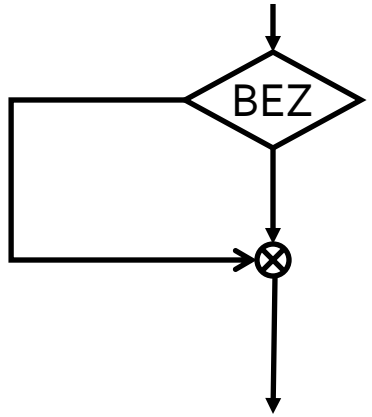
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%



Static Software Branch Prediction

- Extend ISA to enable compiler to tell microarchitecture if branch is likely to be taken or not (Can be up to 80% accurate)

```
BR.T F D X M W
OpA    F - - - -
Targ     F D X M W
BR.NT      F D X M W
OpB          F D X M W
OpC            F D X M W

What if hint is wrong?
BR.T F D X M W
OpA    F - - - -
Targ     F - - - -
OpA        F D X M W
```

Static Hardware Branch Prediction

1. Always Predict Not-Taken
   - What we have been assuming
   - Simple to implement
   - Know fall-through PC in Fetch
   - Poor Accuracy, especially on backward branches
2. Always Predict Taken
   - Difficult to implement because don't know target until Decode
   - Poor accuracy on if-then-else
3. Backward Branch Taken, Forward Branch Not Taken
   - Better Accuracy
   - Difficult to implement because don't know target until Decode

Agenda

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Dynamic Branch Prediction

Learning based on past behavior

- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution

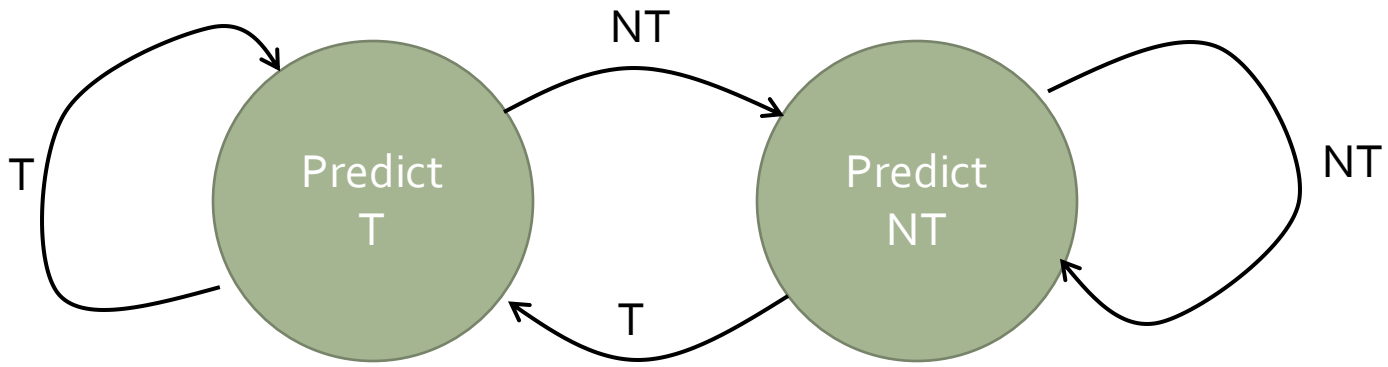
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

Dynamic Hardware Branch Prediction:

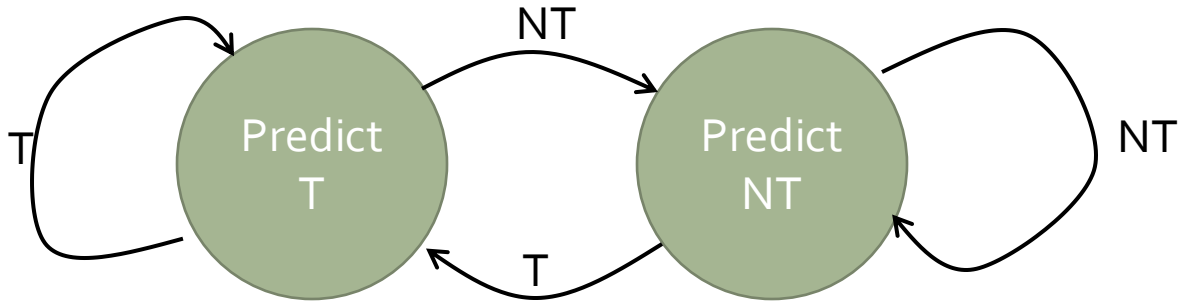
Exploiting Temporal Correlation

- Exploit structure in program: The way a branch resolves may be a good indicator of the way it will resolve the next time it executes (Temporal Correlation)

1-bit Saturating Counter



1-bit Saturating Counter



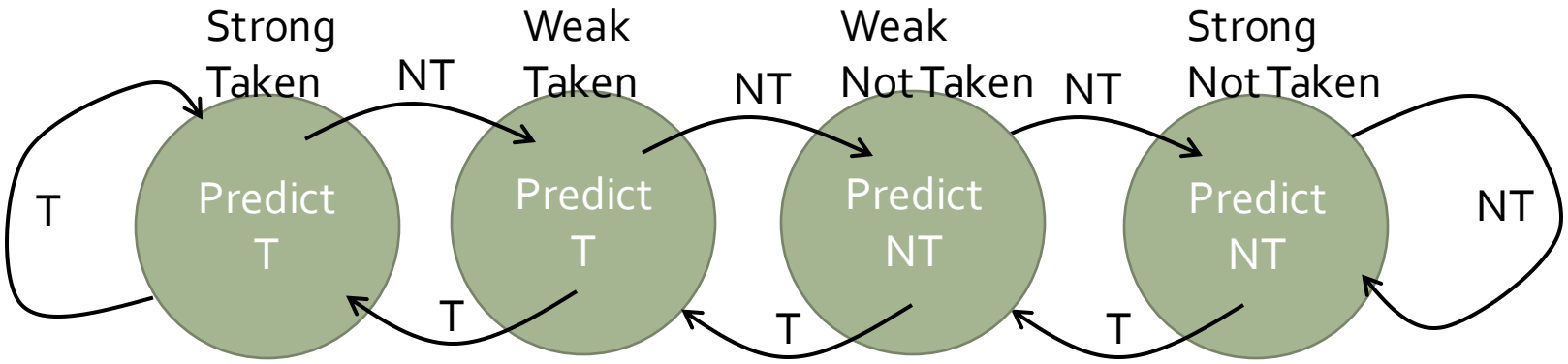
Iteration	Prediction	Actual	Mispredict?
1	NT	T	Y
2	T	T	
3	T	T	
4	T	NT	Y
…			
1	NT	T	Y
2	T	T	
3	T	T	
4	T	NT	Y

For Backward branch in loop

- Assume 4 Iterations
- Assume is executed multiple times

Always 2 Mispredicts

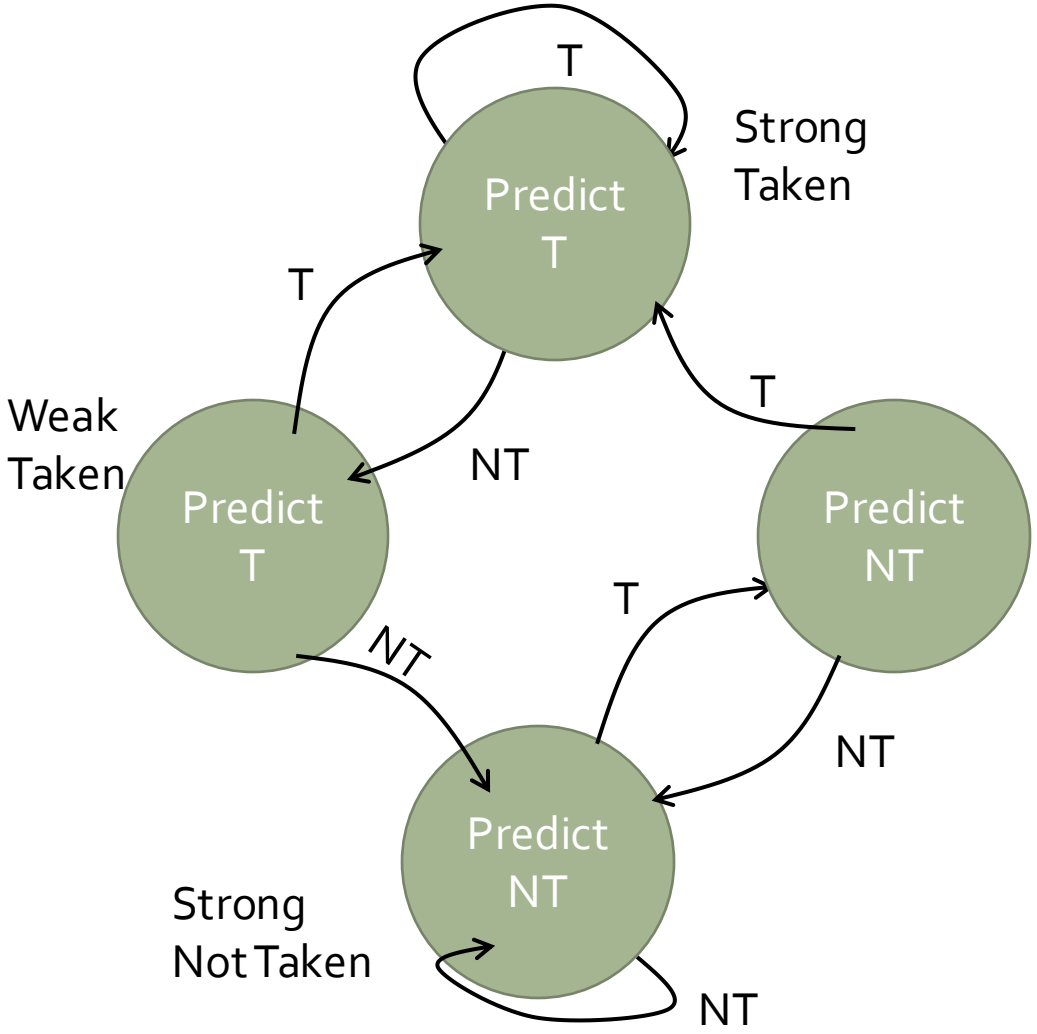
2-bit Saturating Counter



Iteration	Prediction	Actual	Mispredict?	State
1	NT	T	Y	Strong NT
2	NT	T	Y	Weak NT
3	T	T		Weak T
4	T	NT	Y	Strong T
…				
1	T	T		Weak T
2	T	T		Strong T
3	T	T		Strong T
4	T	NT	Y	Strong T

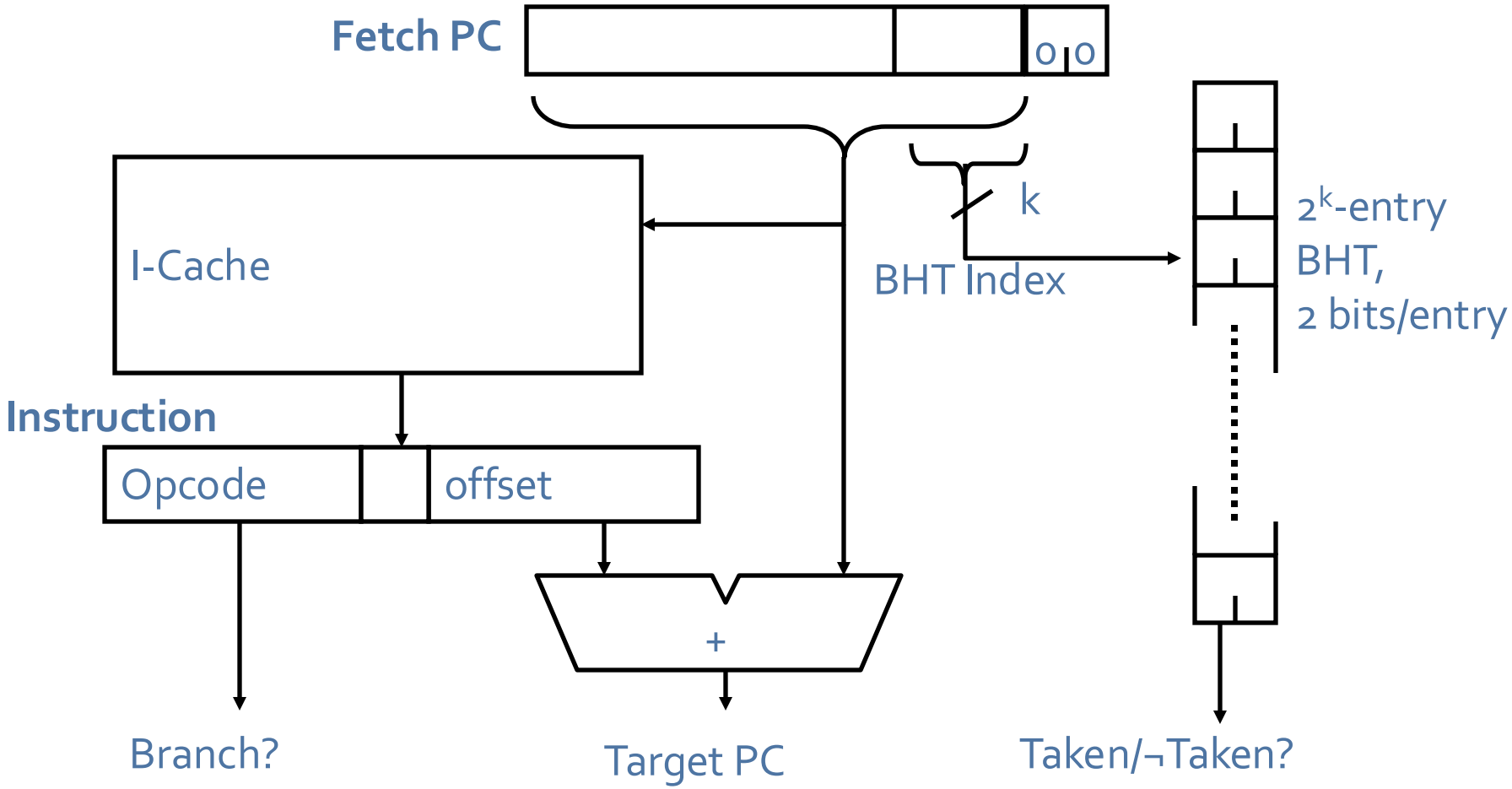
Only 1 Mispredict at end of loop

Other 2-bit FSM Branch Predictors



Jump directly to strong from weak

Branch History Table (BHT)



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

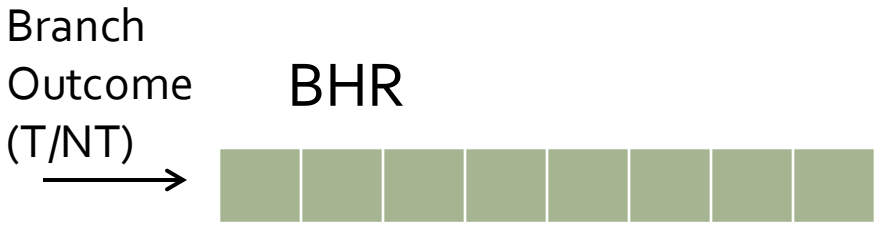
Exploiting Spatial Correlation

[Yeh and Patt, 1992]

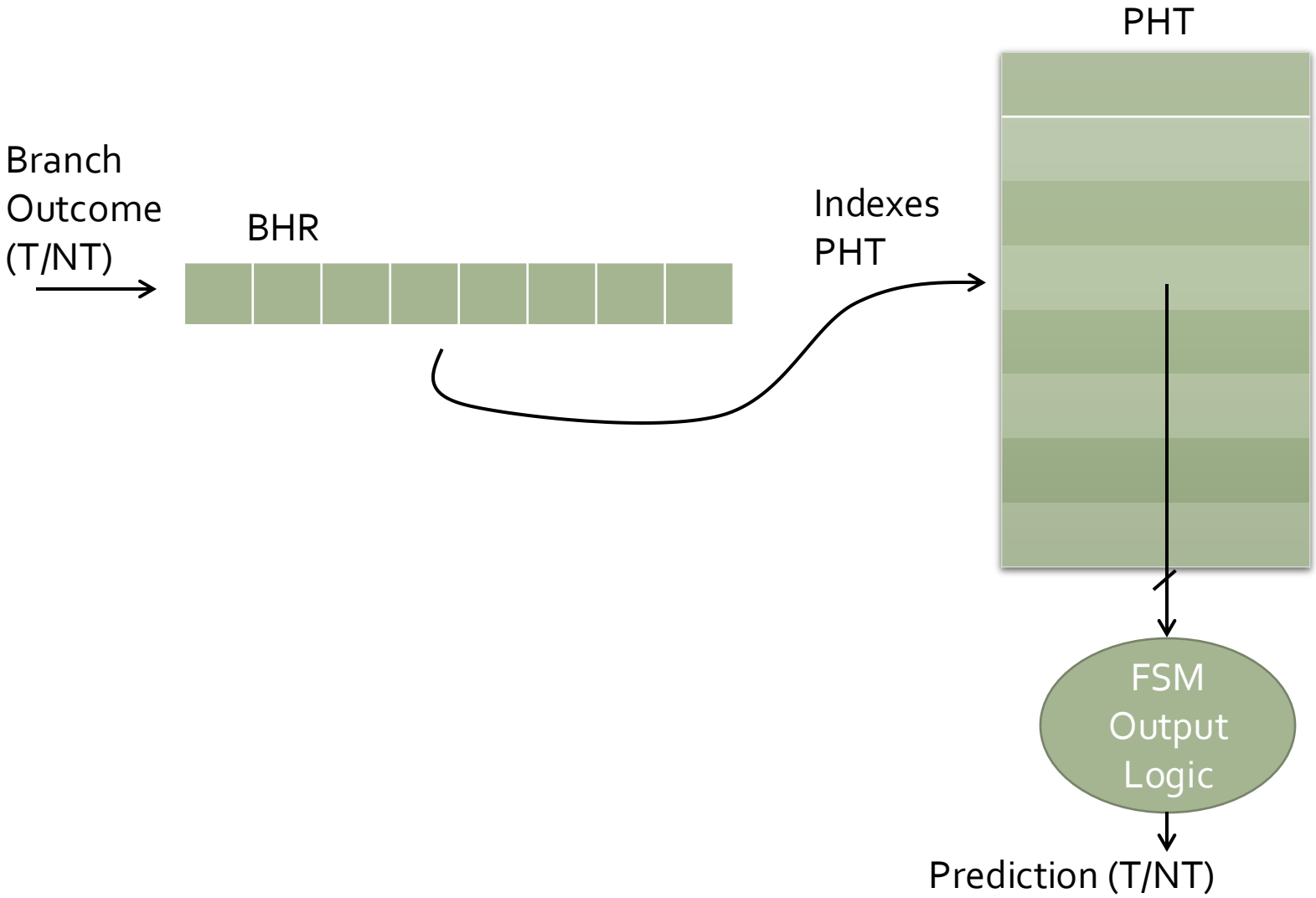
```
if (x[i] < 7) then
        y += 1;
if (x[i] < 5) then
        c -= 4;
```

If first condition false, second condition also false

Branch History Register, **BHR**, records the direction of the last N branches executed by the processor (Shift Register)



Pattern History Table (PHT)



Two-Level Branch Predictor

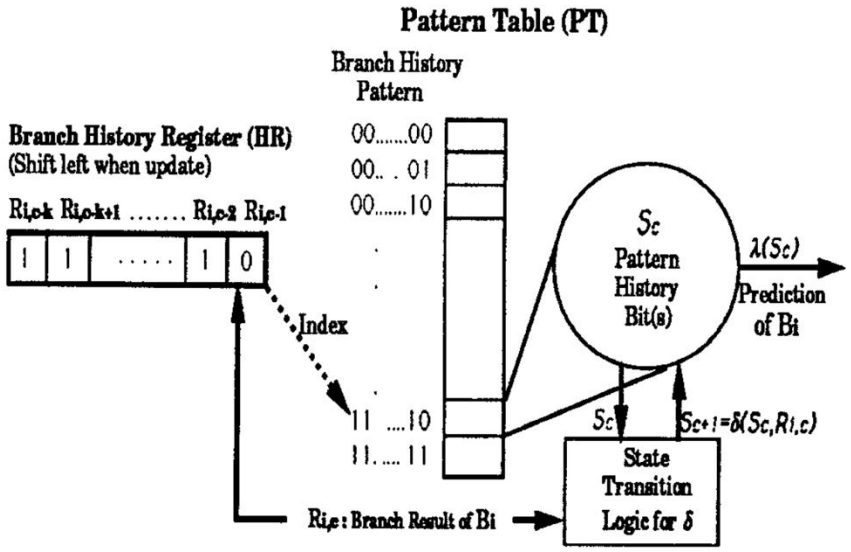


Figure 1: The structure of the Two-Level Adaptive Training scheme.

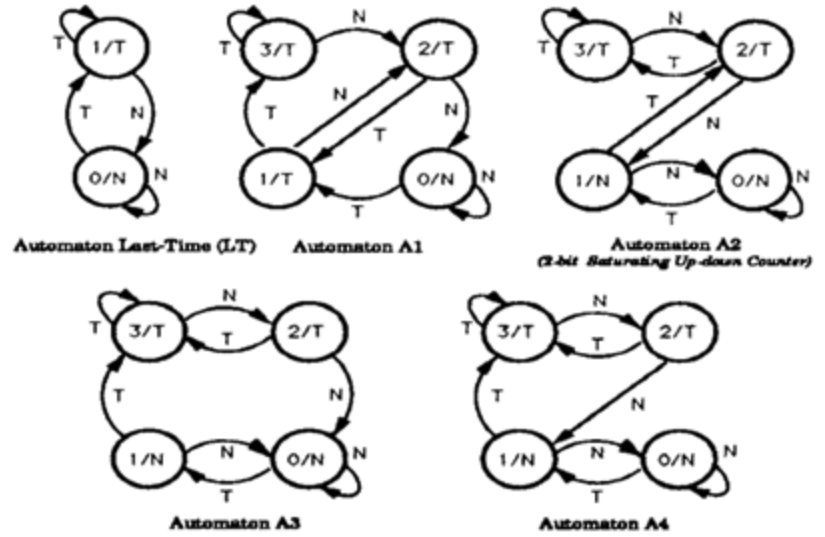
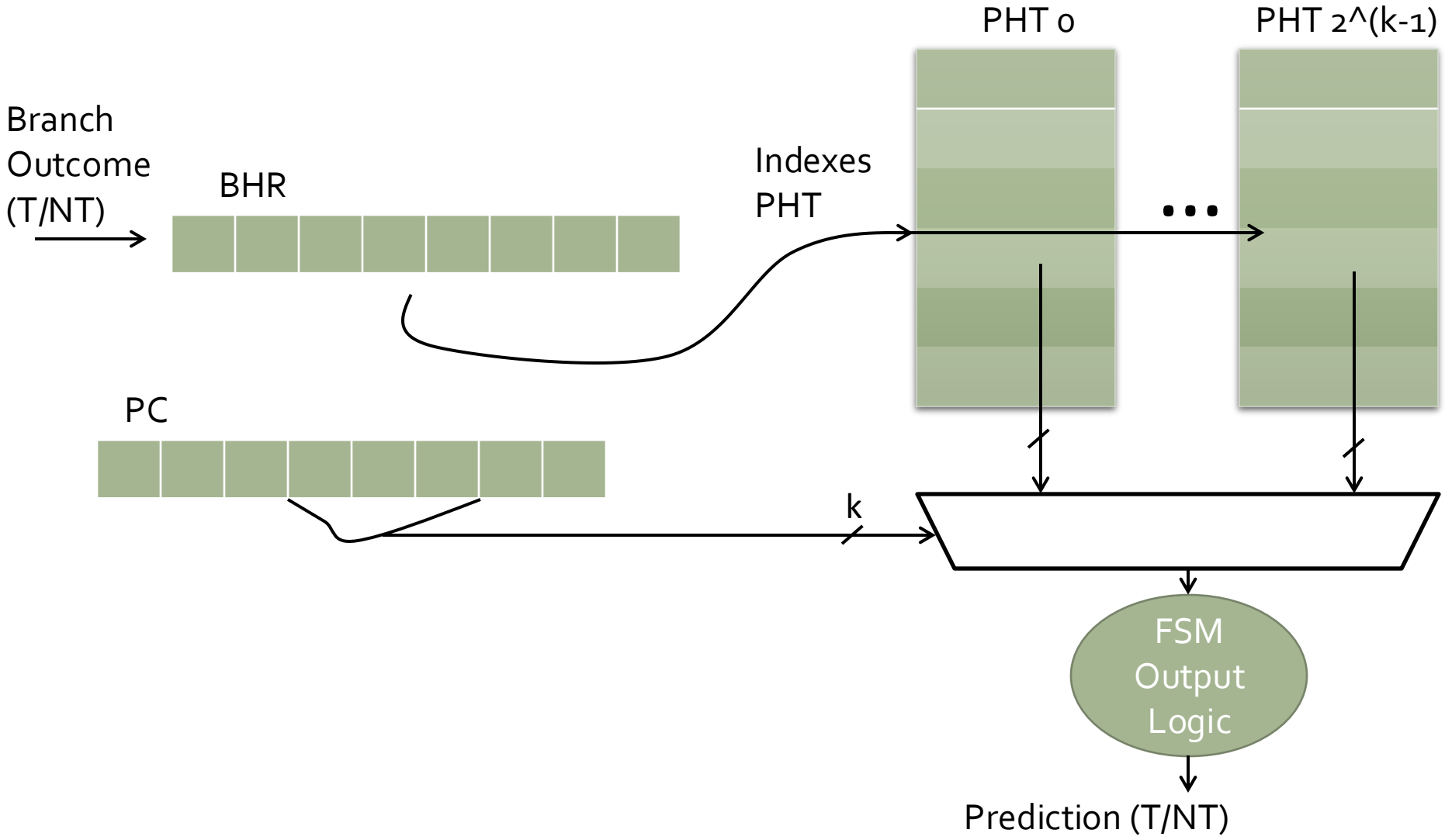


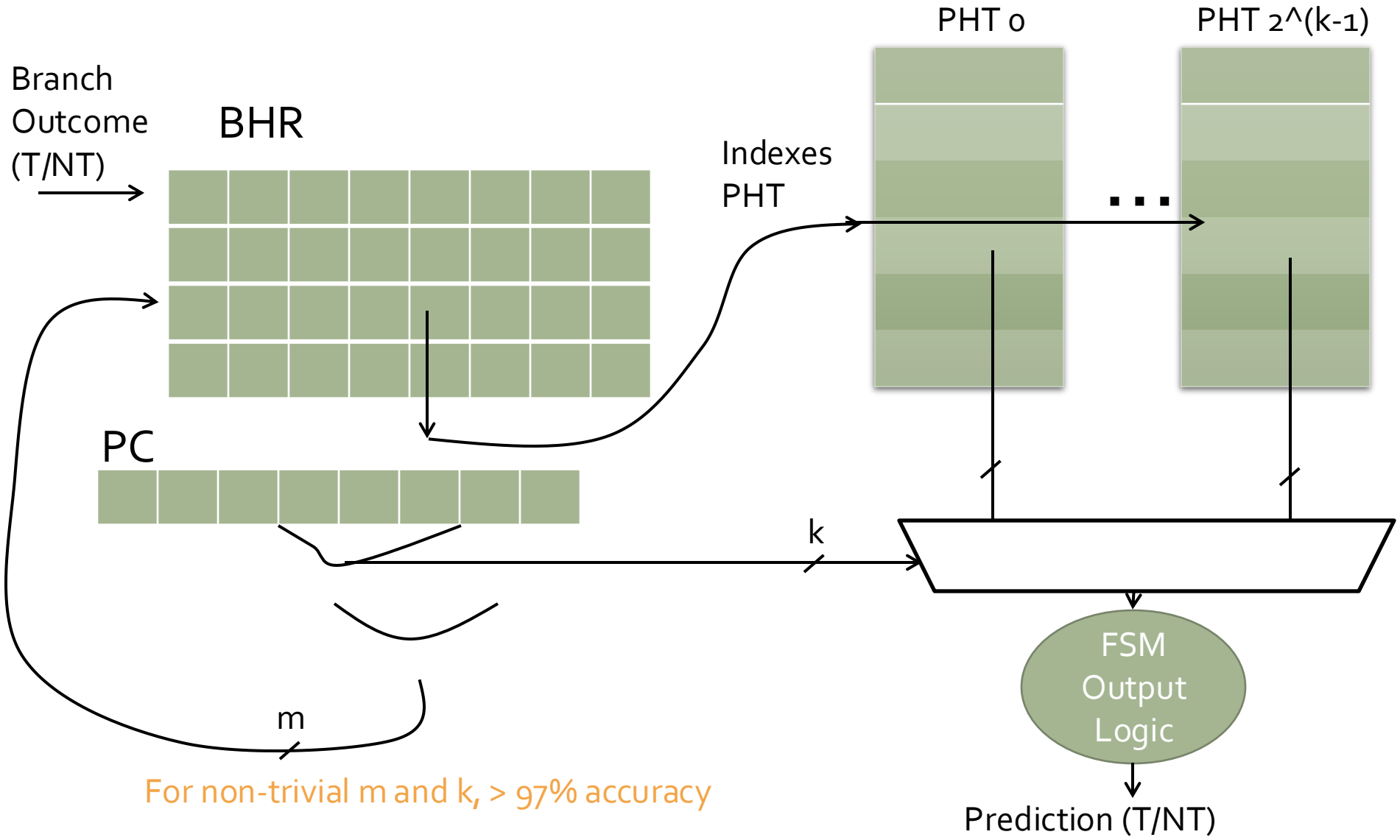
Figure 2: The state transition diagrams of the finite-state machines used for updating the pattern history in the pattern table entry.

Tse-Yu Yeh and Yale N. Patt. 1991. Two-level adaptive training branch prediction. In Proceedings of the 24th annual international symposium on Microarchitecture (MICRO 24).

Two-Level Branch Predictor



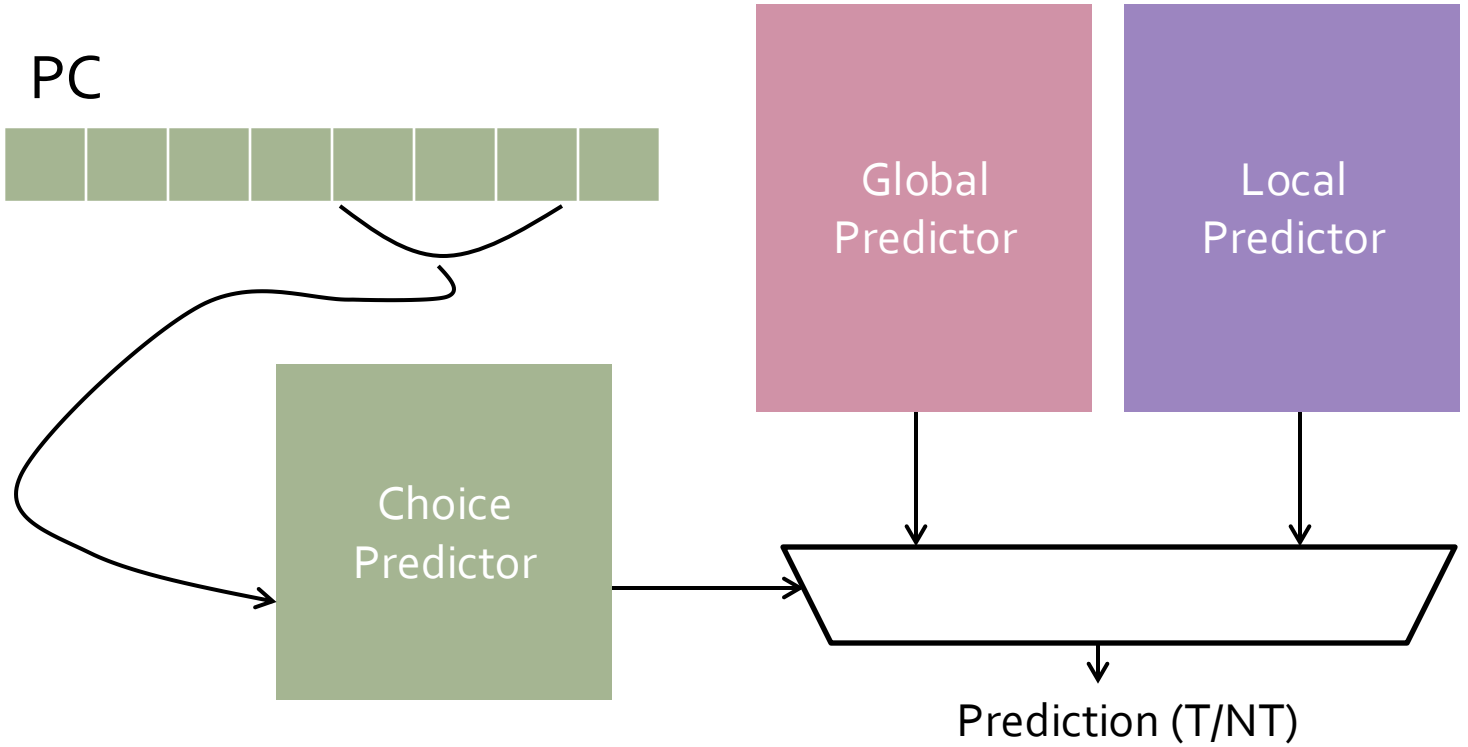
Generalized Two-Level Branch Predictor



For non-trivial m and k, > 97% accuracy

Tournament Branch Predictor

[Alpha 21264, 1996]



- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

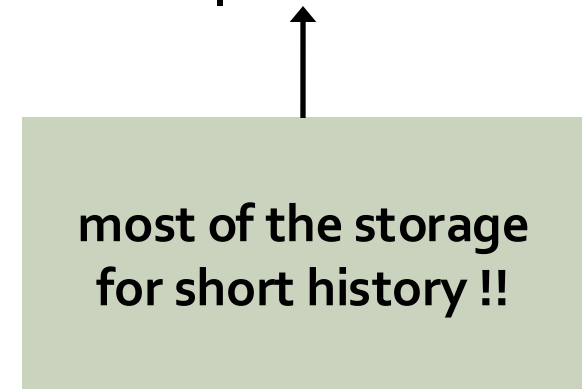
TAGE Predictor

[Seznec & Michaud, 2006]

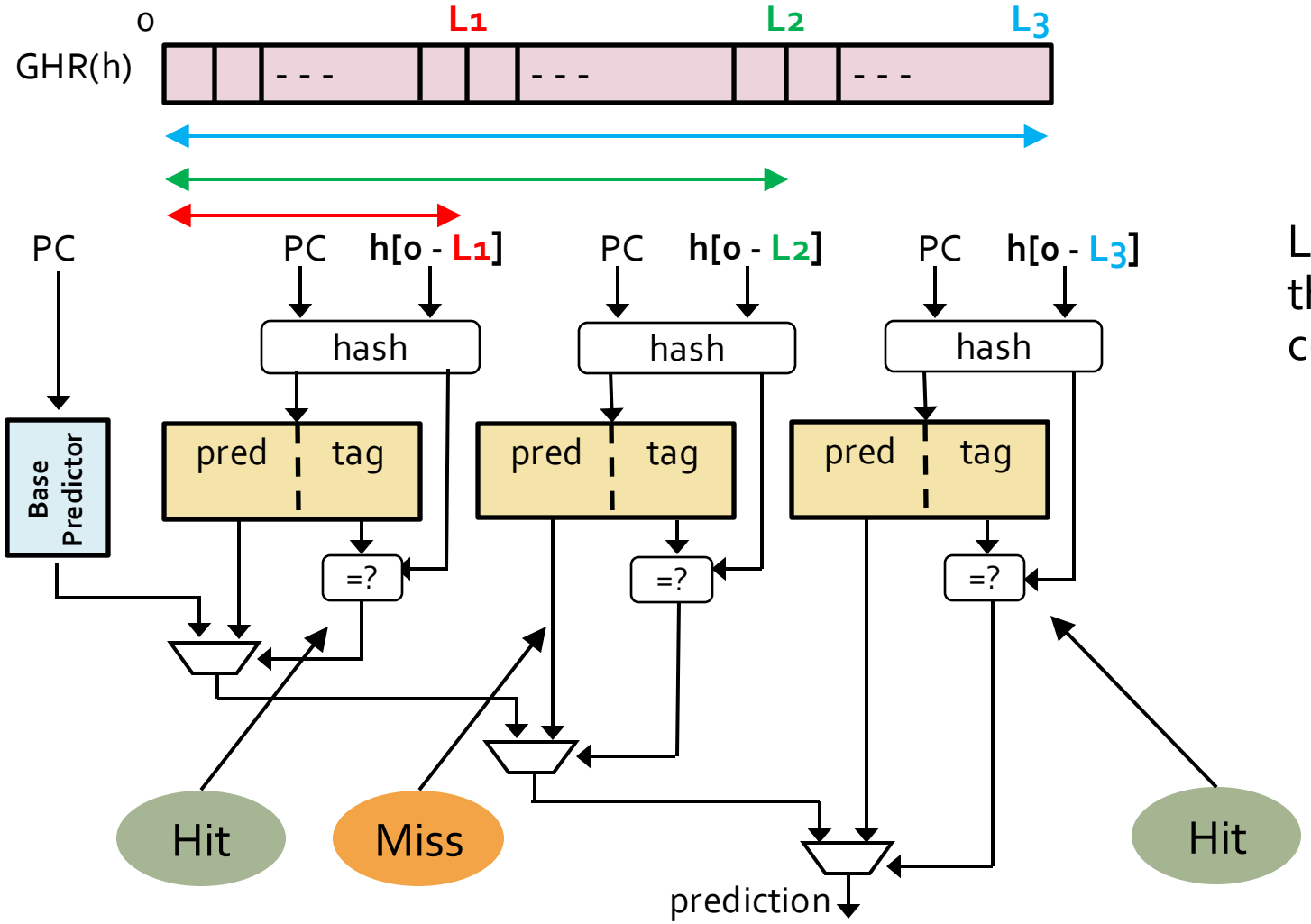
- Multiple tagged tables, use different global history lengths

- Set of history lengths forms a geometric series

{0, 2, 4, 8, 16, 32, 64, 128, 256, …, 2048}



Tagged Geometric History Length (TAGE)

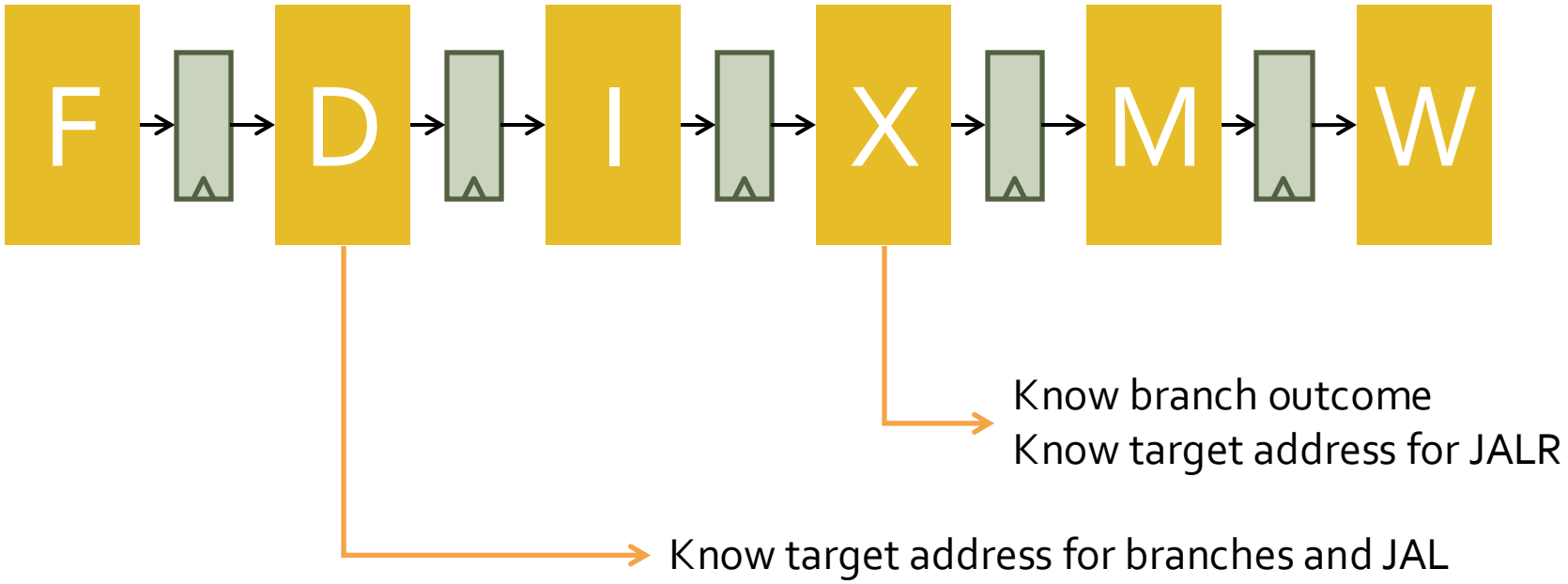


Longest matching table provides the prediction, subject to branch confidence

Agenda

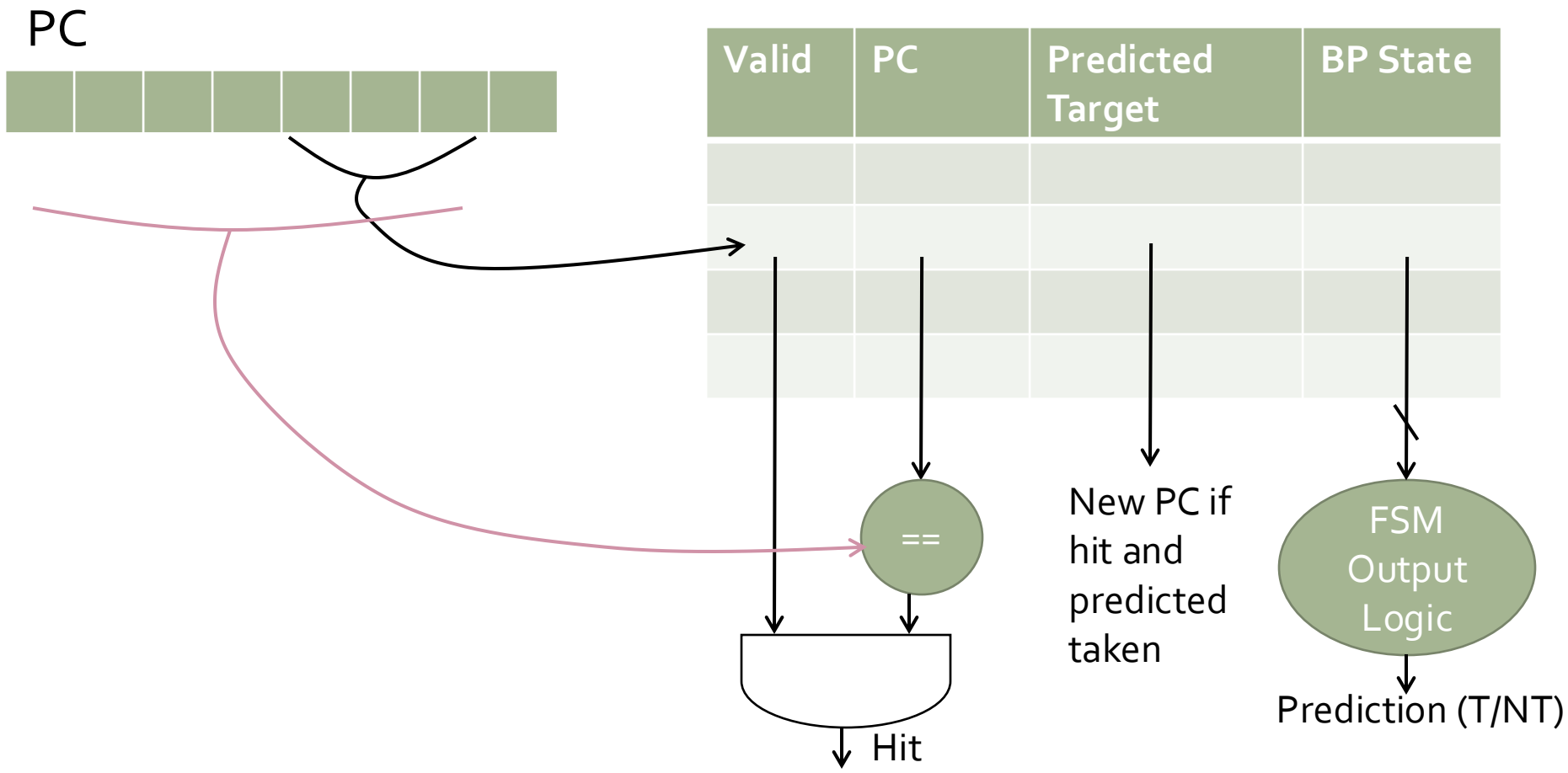
- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Predicting Target Address



Even with best possible prediction of branch outcome, still have to wait for branch target address to be determined

Branch Target Buffer (BTB)



Put BTB in Fetch Stage in parallel with PC+4 Speculation logic

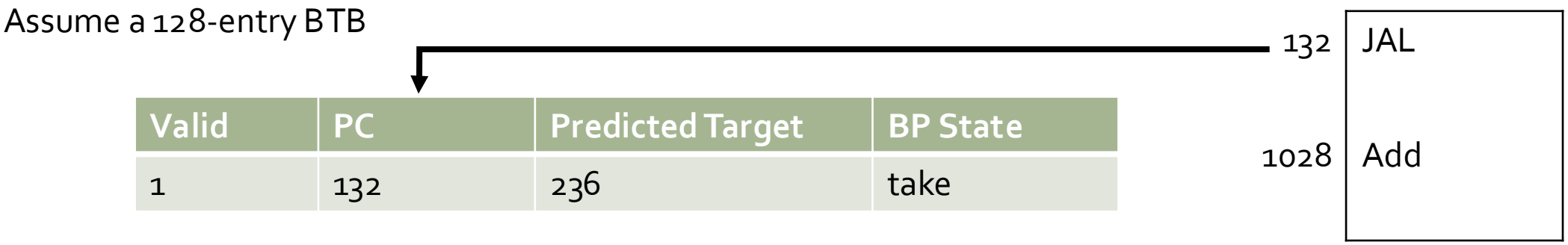
BTB is only for Control Instructions

- BTB contains useful information for branch and jump instructions only
  - Do not update it for other instructions
- For all other instructions the next PC is PC+4 !

How to achieve this effect without decoding the instruction?

Consulting BTB Before Decoding

Assume a 128-entry BTB



Valid	PC	Predicted Target	BP State
1	132	236	take

- The match for PC=1028 fails and 1028+4 is fetched
  → eliminates false predictions after ALU instructions
- BTB contains entries only for control transfer instructions
  → more room to store branch targets

Uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

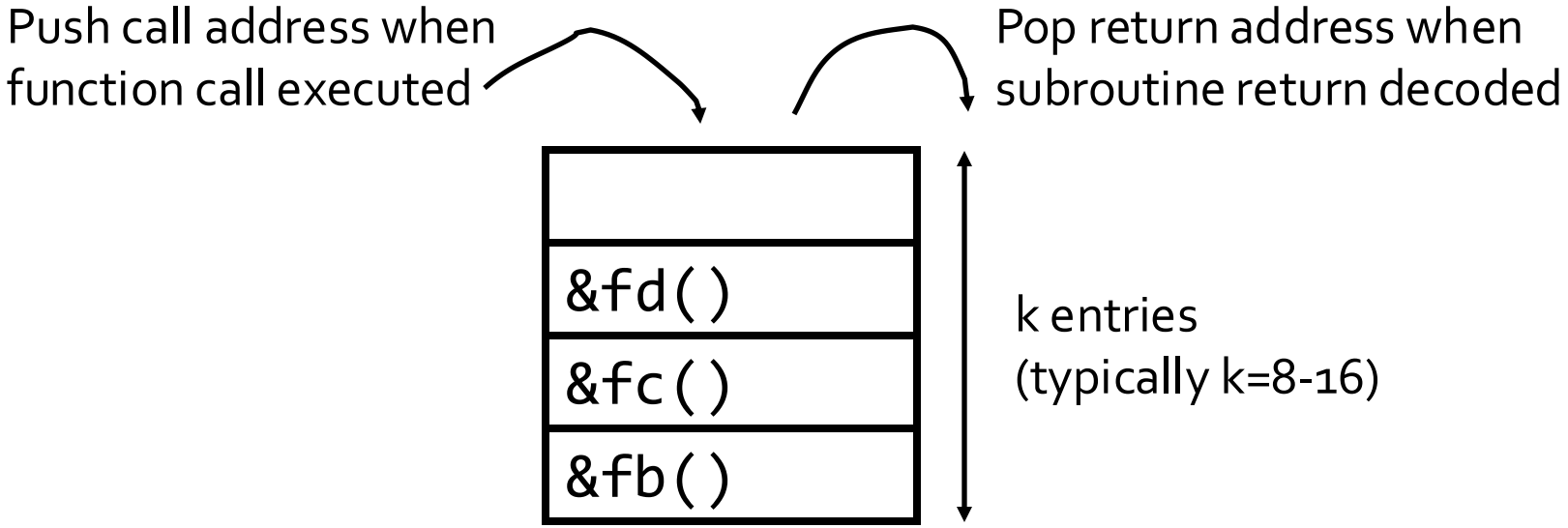
  $\Rightarrow$ **Often one function called from many distinct call sites!**

How well does BTB work for each of these cases?

Subroutine Return Stack

- Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



Recap

- Motivation for Long Pipelines
- Case Study: Intel Pentium 4
- Challenges for Long Pipelines
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
  - David Wentzlaff (Princeton)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- Cornell material derived from course ECE 4750
- Princeton material derived from course ECE 475