EECS 452 – Lecture 7
Branch Prediction

Instructor: Gokhan Memik

EECS Dept., Northwestern University

Roadmap

- The Need for Branch Prediction
- Dynamic Branch Prediction
- Control Speculative Execution

Instructions are like air

- If can't breathe nothing else matters
- If you have no instructions to execute no technique is going to help you
- 1 every 5 insts. is a control flow one
  - we'll use the term branch
  - Jumps, branches, calls, etc.
- Parallelism within a basic block is small
- Need to go beyond branches

Why Care about Branches?

- Example
- Roses are Red and Memory is slow, very very slow
  - 200 cycles and growing
  - One solution is to tolerate memory latencies
  - Tolerate?
    - Do something else
    - AKA find parallelism
    - Well, need instructions
  - How many branches in 200 instructions?
    - 40!

Branch Prediction

- Guess the direction of a branch
- Guess its target if necessary
- Fetch instructions from there
- Execute Speculatively
  - Without knowing whether we should
- Eventually, verify if prediction was correct
  - If correct, good for us
  - if not, well, discard and execute down the right path

For Example

```
while (l)
    if (l->data == 0)
        l->data++;
    l = l->next

loop:   beq     r1, r0, done
        ld      r2, 0(r1)
        bne     r2, r0, noinc
inc:    add     r2, r2, 1
        st      r2, 0(r1)
noinc:  ld      r1, 4(r1)
        bra     loop
done:
```

Terminology

- Lifetime of Branch
  - **Predict:** Guess where it goes
  - **Resolve:** Determine where it goes
- At Predict we start executing target instructions speculatively
- At Resolve we either leave those instructions untouched or we squash them
- Note: Resolve does not mean committed necessarily
  - We can “resolve” a speculatively executed branch
  - Later on this branch may be squashed due to a preceding branch that is resolved

For example...

```
loop:   beq    r1, r0, done
        ld     r2, 0(r1)
        bne    r2, r0, noinc     Predict taken (a)
noinc:  ld     r1, 4(r1)
        bra    loop              Pred. taken (b)
loop:   beq    r1, r0, done      Pred. NT (c)
        ld     r2, 0(r1)         (a)&(b) resolved OK
        bne    r2, r0, noinc     Predict taken (d)
noinc:  ld     r1, 4(r1)         (c) resolved OK
        bra    loop              Predict taken (e)
loop:   beq    r1, r0, done      Predict NT (f)
        ld     r1, 4(r1)         (d) mispredicted
inc:    add    r2, r2, 1
        st     r2, 0(r1)
noinc:  ld     r1, 4(r1)
        ...
```

Branch Prediction Steps

- Start with branch PC and answer:
- Q1? Branch taken or not?
- Q2? Where to?
- Q3? Target Instruction

- Why just PC?
- All must be done to be successful
- Let's consider these separately

Static Branch Prediction

- Static:
  - Decisions do not take into account dynamic behavior
  - Non-adaptive can be another term
- Always Taken
- Always Not-Taken
- Forward NT Backward T
- If X then T but if Y then NT but if Z then T
  - More elaborate schemes are possible
- Bottom line
  - Accuracy is high but not high enough
  - Say it's 60%
  - Probability of 100 instructions :
  - .6^20 = .000036

Branch Prediction Accuracy

- Probability of 100 insts (TODAY):
  - .6 = .000036
  - .7 = .00079
  - .8 = .011 or 1%
  - .9 = .12 or 12%
  - .95 = 36%
  - .98 = 66%
  - .99 = 82%
- Probability of 250 insts (SOON)
  - .9 = .9^250/5 = .9^50 = .0051
  - .95 = .08
  - .98 = .36
  - .99 = .6
- Assuming uniform distr.
- Not true but for the sake of illustration

Semi-Static Branch Prediction

- McFarling & Hennesy, ISCA 1986
- Use profile information
  - Branches tend to behave in a fixed way
  - Same way across different runs?
- Roughly 80% across different apps

Dynamic Branch Prediction

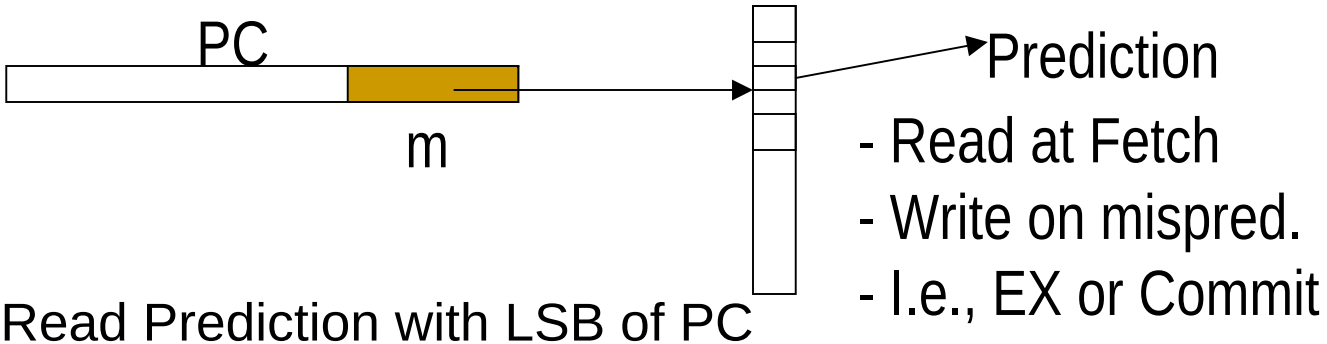
- Why?
  - Larger window → More opportunity for parallelism
- Basic Idea:
  - hardware guesses **whether** a branch will be taken, and if so **where** it will go
- Why it works? Program Behavior
  - **Past Branch Behavior STRONG indicator of future branch behavior**
  - **Branches tend to exhibit regular behavior**

Regularity in Branch Behavior

- Given Branch at PC X
- Observe it's behavior:
  - Q1? Taken or not?
  - Q2? Where to?
- In typical programs:
  - A1? Same as last time
  - A2? Same as last time

Last-outcome Branch prediction

- J. E. Smith, ISCA 1981
- Start with PC and answer whether taken or not
  - 1 bit information: T or NT (e.g., 1 or 0)
- Example Implementation: 2^m x 1 mem.



  - Read Prediction with LSB of PC
  - Change bit on misspredict ion
  - May use PC from wrong PC
    - aliasing: destructive vs. constructive
- Can we do better? Yes in a sec

Aliasing

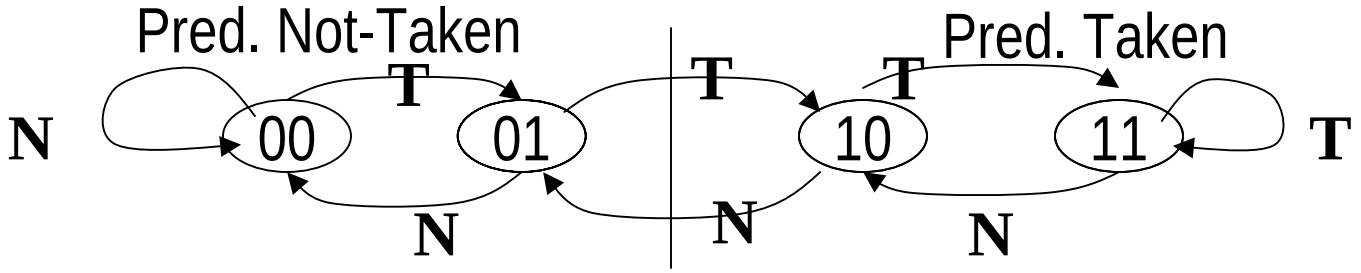
- Predictor Space is finite
- Aliasing:
  - Multiple branches mapping to the same entry
- Constructive
  - The branches behave similarly
    - May benefit accuracy
- Destructive
  - They don't
    - Will hurt accuracy
- Can play with the hashing function to minimize
  - Black magic
  - Simple hash (PC << 16) ^ PC works OK

Learning Time

- Number of times we have to observe a branch before we can predict it's behavior
- Last-outcome has very fast learning time
- We just need to see the branch at least once
- Even better:
  - initialize predictor to taken
  - Most branches are taken so for those learning time will be zero!

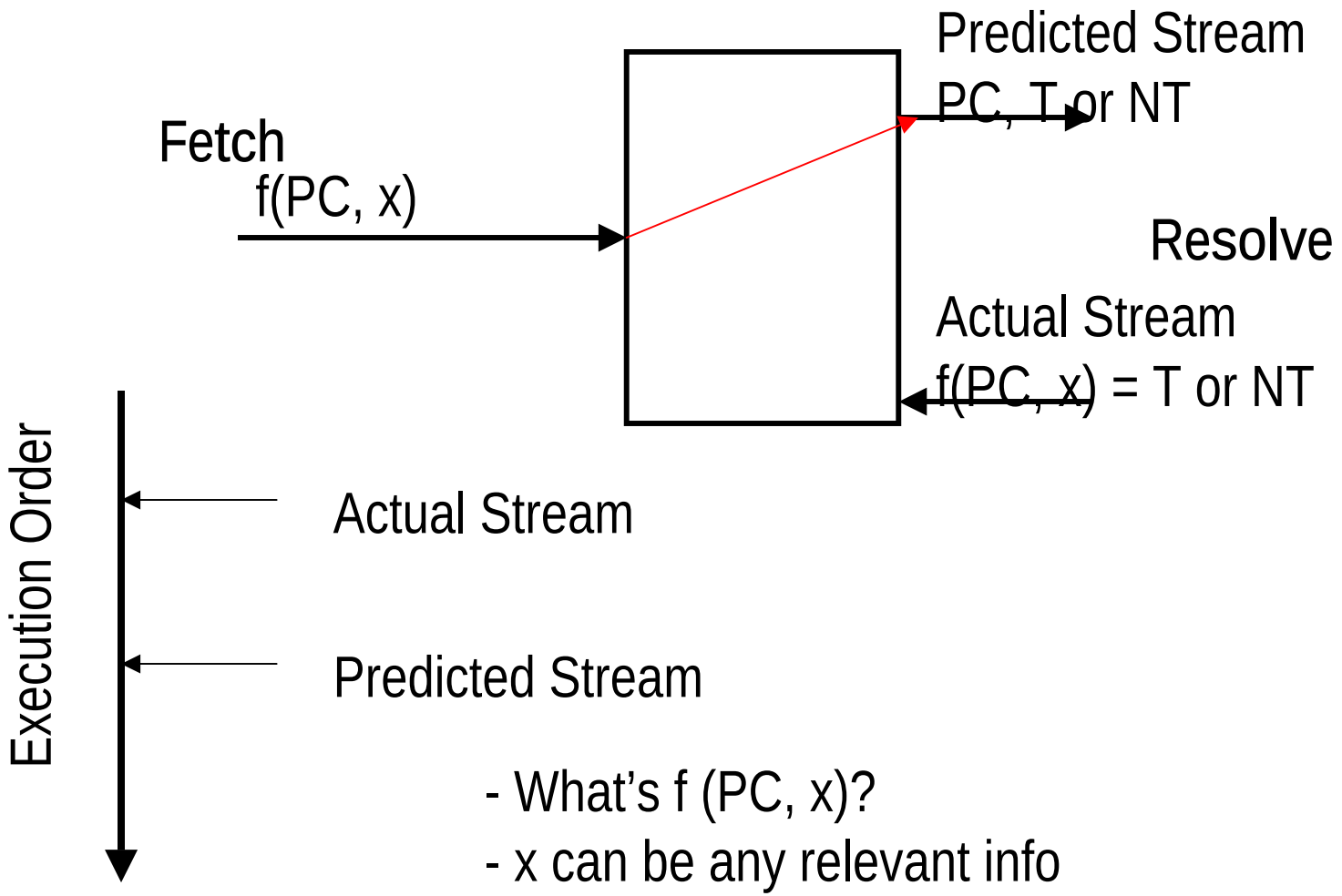
Saturating-Counter Predictors

- Consider strongly biased branch with infrequent outcome
- TTTTTTTT**N**TTTTTTTTT**N**TTTT
- Last-outcome will mispredict twice per infrequent outcome encounter:
- TTTTTTTT**N**TTTTTTTTT**N**TTTT
- Idea: Remember most frequent case
- Saturating-Counter: Hysteresis



- often called bi-modal predictor
- **Captures Temporal Bias**

A Generic Branch Predictor



- What's f (PC, x)?
- x can be any relevant info thus far x was empty

Correlating Predictors

- From program perspective:
  - Different Branches may be correlated
  - if (aa == 2) aa = 0;
  - if (bb == 2) bb = 0;
  - if (aa != bb) then …
- Can be viewed as a pattern detector
  - Instead of keeping aggregate history information
    - I.e., most frequent outcome
  - Keep exact history information
    - Pattern of n most recent outcomes
- Example:
  - BHR: n most recent branch outcomes
  - Use PC and BHR (xor?) to access prediction table

Pattern-based Prediction

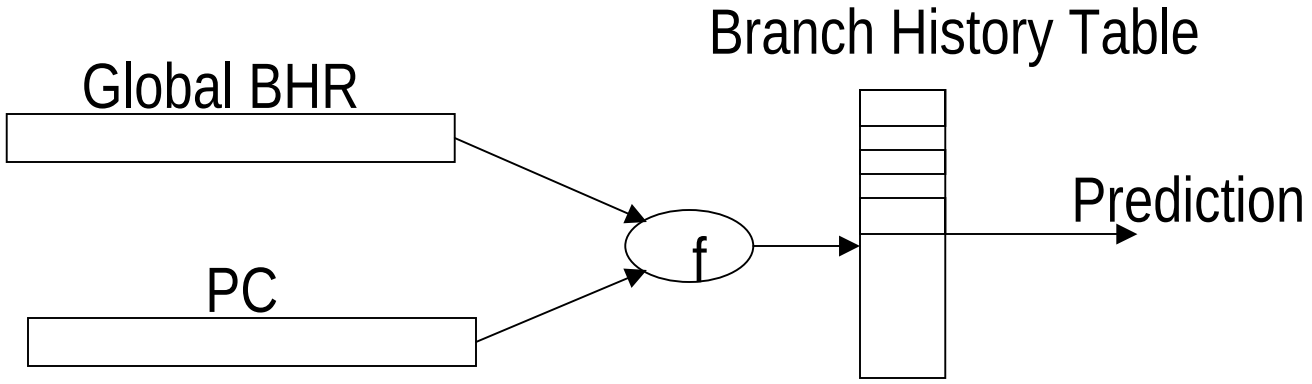
- Nested loops:

  for i = 0 to N

  for j = 0 to 3

  ...
- Branch Outcome Stream for j-for branch
  - 11101110111011101110
- Patterns:
  - 111 -> 0
  - 110 -> 1
  - 101 -> 1
  - 011 -> 1
- 100% accuracy
- Learning time 4 instances
- Table Index (PC, 3-bit history)

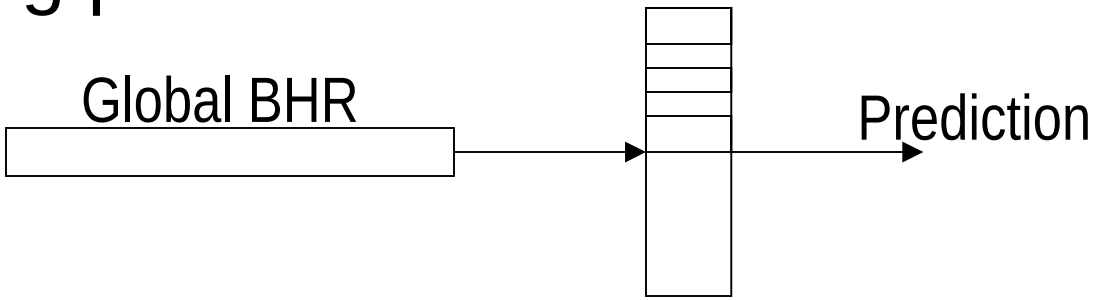
Gshare Predictor (McFarling, DEC)



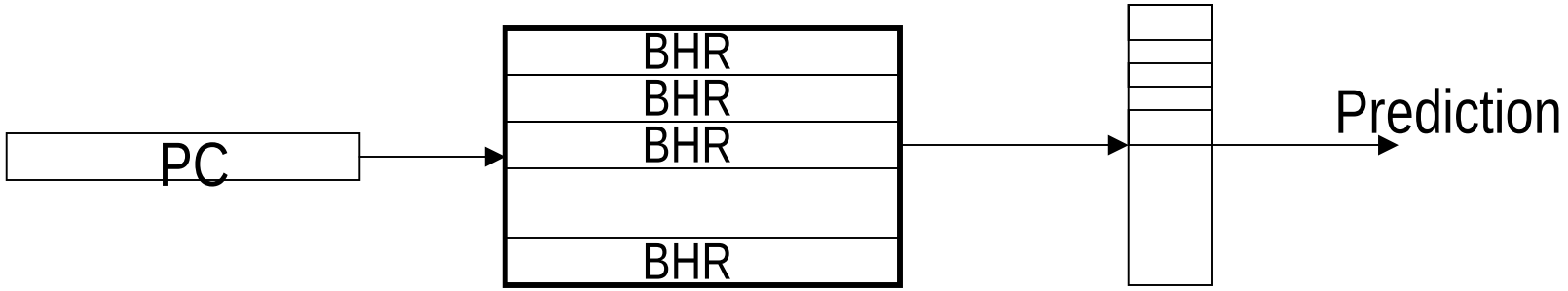
- PC and BHR can be
  - concatenated
  - completely overlapped
  - partially overlapped
  - xored, etc.
- How deep BHR should be?
  - Really depends on program
  - But, deeper increases learning time
  - May increase quality of information

Multi-Level Predictors (Yeh and Patt)

- GAg predictor

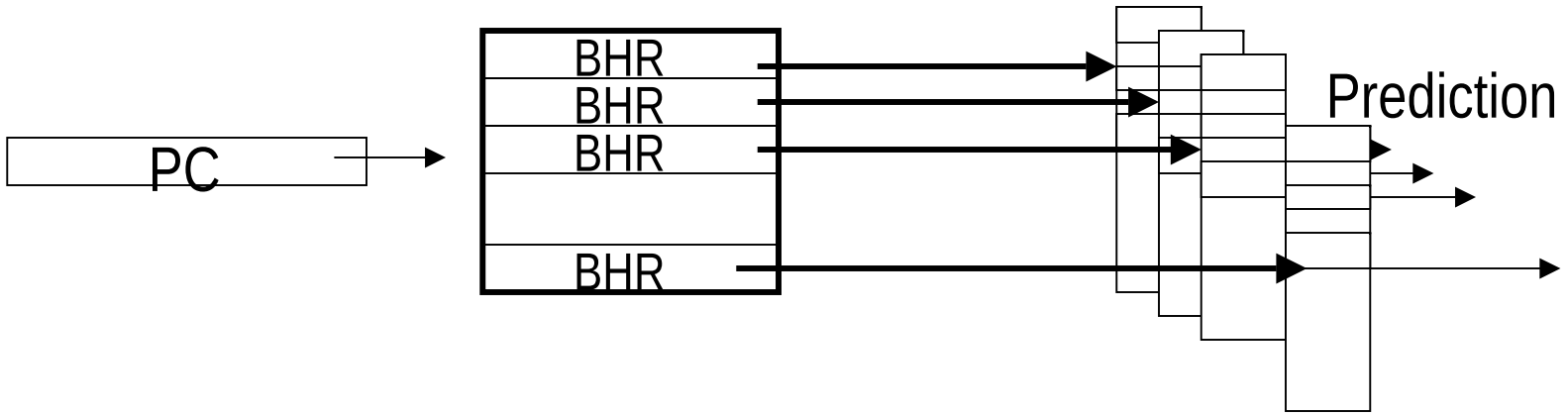


- PAg predictor



Multi-level Predictors, cont.

- PAp predictor
  - PC selects BHR
    - Separate prediction table per BHT



Multi-Method Predictors

- Some predictors work better than others for different branches
- Idea: Use multiple predictors and one that selects among them.
- Example:
  - Bi-modal Predictor
  - Pattern based (e.g., Gshare) predictor
  - Bi-modal Selector
  - Initially Selector Points to Bi-modal
  - If misprediction both predictor and selector are updated
  - Why? Gshare takes more time to learn

Other Branch Predictors

- The agree-Predictor
  - Whether static predictor same as dynamic behavior
  - Learning time
- Dynamic History Length Fitting
  - vary history depth during run-time
- Prediction and Compression
  - High correlation between the two
  - T. Mudge paper (ASPLOS 96) of multi-level predictors
  - Intuitively:
    - if compressible then high-redundancy
    - Or, automaton exists that has same behavior

Updates

- Speculatively update the predictor or not?
- Speculative: on branch complete
- Non-Speculative: on branch resolve
- Trace based studies
  - Speculative is better
  - Faster Learning
  - Not much interference

Branch Target Buffer

- 2nd step, where this branch goes
- Recall, 1st step: taken vs. not taken
- *Associate Target PC with branch PC*
- Target PC available earlier: derived using branch's PC.
  - No pipeline bubbles
- Example Implementation?
  - Think of it as a cache:
  - Index & tags: Branch PC (instead of address)
  - Data: Target PC
  - Could be combined with Branch Prediction

Branch Target Buffer - Considerations

- Careful:
  - Many more bits per entry than branch prediction buffer
- Size & Associativity
- Store not-taken branches?
  - Pros and cons.
  - Uniform
  - BUT, cost in wasted space

Branch Target Cache

- Note difference in terminology
- Start with Branch PC and produce
  - Prediction
  - Target PC
  - Target Instruction
- Example Implementation?
  - Special Cache
  - Index & tags: branch PC
  - Data: target PC, target inst. & prediction
- Facilitates “Branch Folding”, i.e.,
  - Could send target instruction instead of branch
  - “Zero-Cycle” branches
- Considerations: more bits, size & assoc.

Jump Prediction

- When?
  - Call/Returns
  - Direct Jumps
  - Indirect Jumps (e.g., switch stmt.)
- Call/Returns?
  - Well established programming convention
  - Use a small hardware stack
  - Calls push a value on top
  - Returns use the top value
  - NOTE: this is a prediction mechanism
  - if it's wrong it only impacts performance NOT correctness
  - Speculation? Keep track of changes

Indirect Jump Prediction

- Not yet used in state-of-the-art processors.
- Why? (very) Infrequent
- BUT: becoming increasingly important
  - OO programming
- Possible solutions?
- Last-Outcome Prediction
- Pattern-Based Prediction
- Think of branch prediction as predicting 1-bit values
- Now, think how what we learned can be used to predict n-bit values

Dynamic Branch Prediction Summary

- Considerations
  - No Correctness issues:
    - Results always correct
    - Incorrectly executed instructions are squashed
  - Don't slow down Clock Cycle (much?)
  - High Prediction accuracy
  - Fast on correct predictions
  - Not too slow on misspredictions
- Bottom line:
  - Useful for single-issue pipelines
  - Critical for multiple-issue machines
  - More so for larger instruction windows
  - E.g., given 90% accuracy what is the probability of having a 256 inst. window full?

Superscalar Processors: The Big Picture

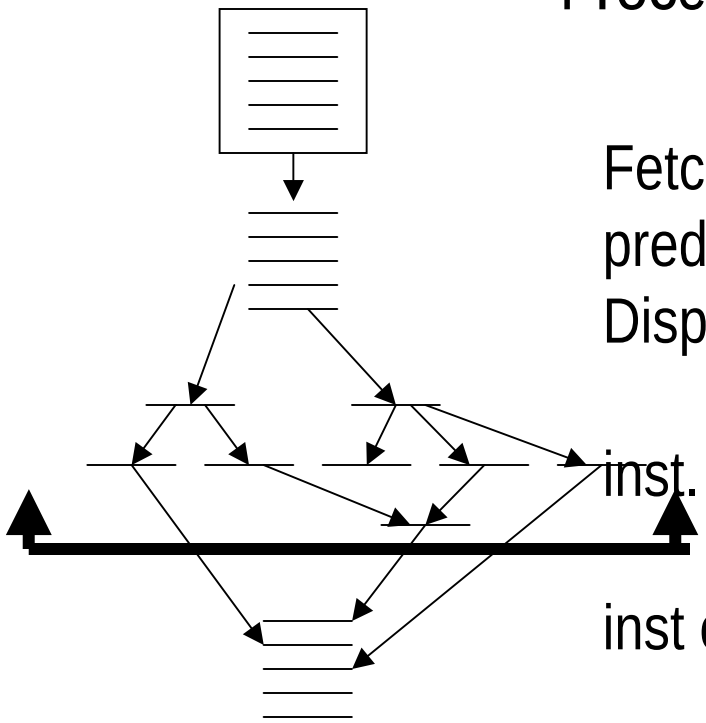
Program Form

Static program

dynamic inst. Stream (trace)

execution window

completed instructions



Processing Phase

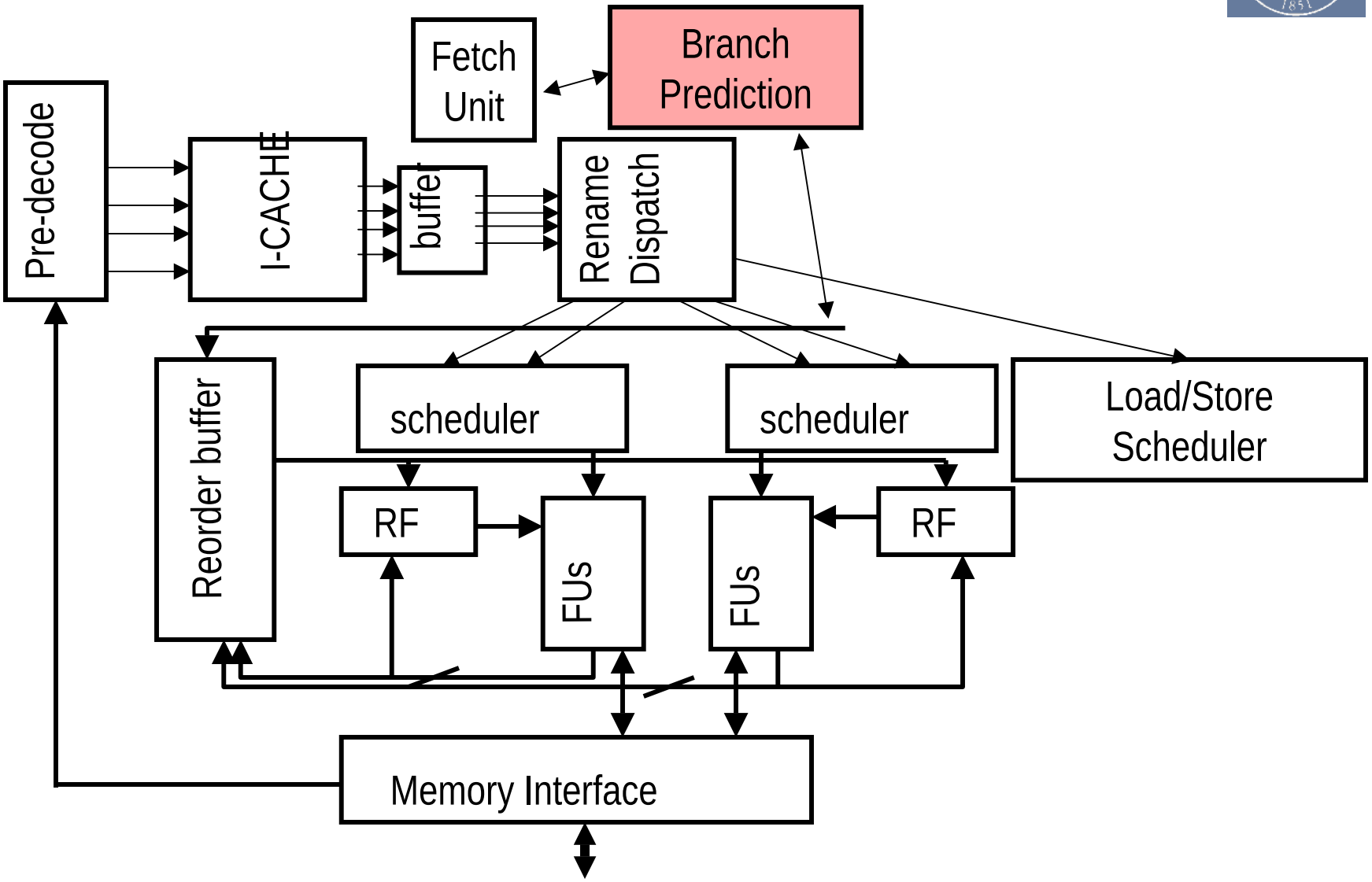
Fetch and CT prediction Dispatch/ dataflow

inst. Issue

inst execution

inst. Reorder & commit

A Generic Superscalar OOO Processor



Speculative Execution

- Execute Instructions without being sure that you should
  - Branch prediction:
    - instructions to execute w/ high prob.
  - Speculative Execution allows us to go ahead and execute those
  - We'll see other uses soon
    - Memory operations
- Notice that SE and BP are different techniques
  - BP uses SE
  - SE can be used for other purposes too

Execution of Control Speculative Code

- Idea: Allow execution but keep enough information to restore correct state on misprediction.
- Two approaches:
  - Maintain history with each instruction
    - e.g., copy target value before updating it
  - Two copies of data values:
    - Architectural and Speculative
- The second is the method of choice today. Why?
  - On misspeculation all incorrect instructions are squashed
  - I.e., discarded.
  - Execution resumes by fetching instructions from the correct control path

Speculative Execution, Example

```
I1:     LD      R6,     34(R2)
I2:     LD      R5,     38(R2)
I3:     ADD     R0,     R5,     R6
I4:     ADD     R2,     R2,     R1
I5:     BNEZ    R0,     -16
I6:     LD      R6,     34(R2)
I7:     LD      R5,     38(R2)
I8:     ADD     R0,     R5,     R6
```

Iteration i

Iteration i+1

	Register Rename Table							ROB		
	0	1	2	3	4	5	6	TR	OR	S?
	0	1	2	3	4	5	6			
I1	0	1	2	3	4	5	7	6	6	N
I2	0	1	2	3	4	8	7	5	5	N
I3	9	1	2	3	4	8	7	0	0	N
I4	9	1	10	3	4	8	7	2	2	N
I6	9	1	10	3	4	8	11	6	7	Y
I7	9	1	10	3	4	12	11	5	8	Y
I8	14	1	10	3	4	13	11	0	9	Y

TR=target reg (log), OR=old register (phys), S?=Control Speculative

Mis-speculation Handling

- Squash all after Branch:

	Register Rename Table							ROB		
	0	1	2	3	4	5	6	TR	OR	S?
	0	1	2	3	4	5	6			
I1	0	1	2	3	4	5	7	6	6	N
I2	0	1	2	3	4	8	7	5	5	N
I3	9	1	2	3	4	8	7	0	0	N
I4	9	1	10	3	4	8	7	2	2	N
I6	9	1	10	3	4	8	11	6	7	Y
I7	9	1	10	3	4	12	11	5	8	Y
I8	14	1	10	3	4	13	11	3	9	Y

- Requires backward walk of ROB, Why?
- Can we do better?
- If using RUU, then this is OK
  - Register Mapping in RUU

Mis-Speculation Handling, Contd.

- Save Whole RAT when Speculating a branch

	Register Rename Table						
	0	1	2	3	4	5	6
AP->	0	1	2	3	4	5	6
AP->	9	1	10	3	4	8	7
	9	1	10	3	4	8	7
AP->	14	1	10	3	4	13	11
	9	1	10	3	4	8	7

- Number of RATs limits # of speculative Branches
- Nice from a VLSI perspective

Change of Topic

- Multiple Instruction Fetch

The Need for Multiple-Instruction Fetch

- Can't execute more than we fetch
- Today: 4-way issue
  - Fetch at least 4 instructions per cycle
  - More like 8

- Three main issues. Per cycle:
  1. **Align instructions to feed scheduler**
  2. **Fetch Multiple Cache Lines?**
  3. **Predict Multiple Branches?**

Fetching Multiple Instructions & the I-Cache

- 4 instructions (4 bytes each) out of 32-byte line
  - Alignment restrictions also limits possible cases

				i-0	i-1	i-2	i-3
			i-0	i-1	i-2	i-3	
		i-0	i-1	i-2	i-3		
	i-0	i-1	i-2	i-3			
i-0	i-1	i-2	i-3				

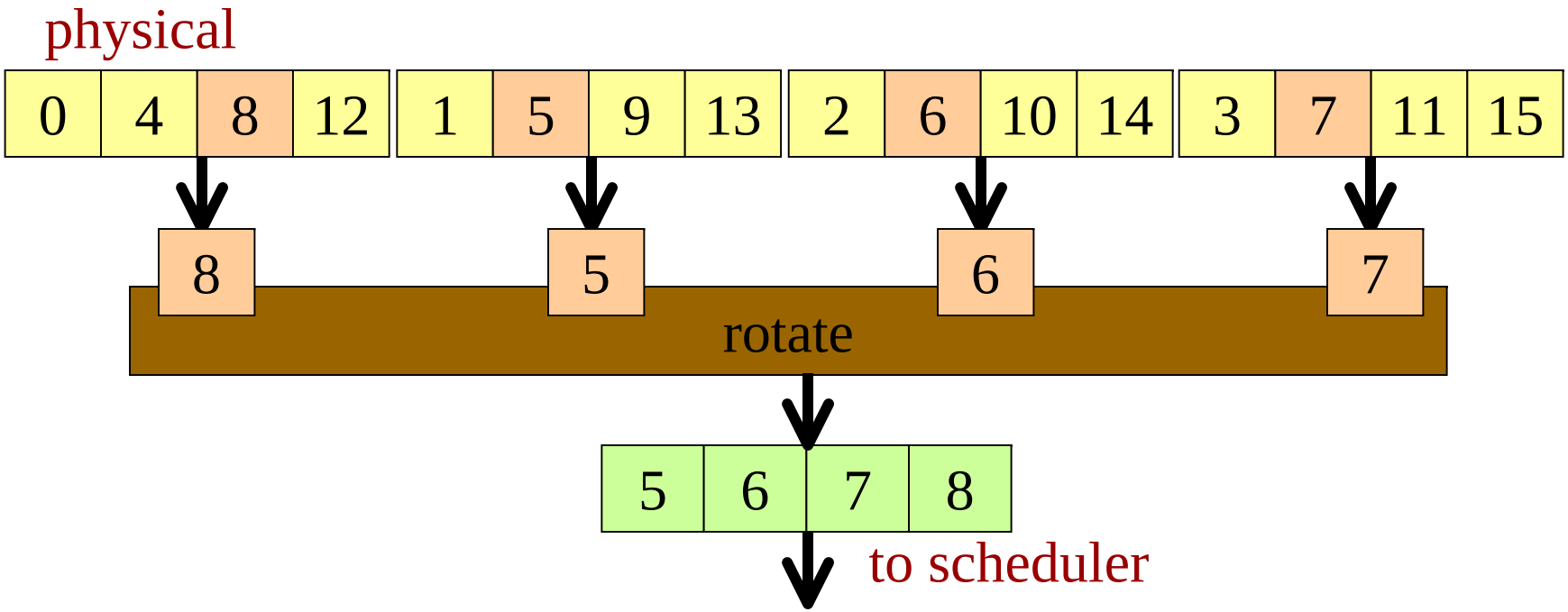
selection/alignment network

to scheduler

i-0	i-1	i-2	i-3

MIPS R1000 Solution

- 4 instructions (4 bytes each) out of 16-inst line (64 bytes)
- Take word with same (MOD 4) and place next to each other
  - Only one inst. per set per cycle (for up to 4 insts fetched)
- Select the 4 words
- Then rotate into place



Fetching 8 Instructions per Cycle

- The chunk of insts may reside in multiple cache lines:

continuous sequence

n					i-0	i-1	i-2	i-3
n+1	i-4	i-5	i-6	i-7				
want	i-0	i-1	i-2	i-3	i-4	i-5	i-6	i-7
m				i-0	i-1	i-2	i-3	i-4
m+1	i-5	i-6	i-7					
want	i-0	i-1	i-2	i-3	i-4	i-5	i-6	i-7

Solution #1: Do not support

- Recall, try easy thing first
  - Try this in SimpleScalar to see how much you lose
- Hardware detects that fetch set spills over
- Fetches only as many instructions as possible

Solution #2: Full Support

- Allow multiple cache line accesses per cycle
  - Can be expensive
  - Multiple access paths
  - Multi-ported cache
  - Banked
- Routing network complexity?

- Bottom line:
  - Can be made to work
  - Complex: slow and expensive
- Would prefer to do without

Full Support: How Many Cache Lines Per Cycle?

- If no control flow: 2 is enough
- But with control flow?
  - Worst case: 8 cache lines
    - Every inst a branch
    - Or every other instruction a branch that points to the last inst in a cache line (MOD 8 = 7)
- Since 1 in 5 insts is a branch it is to be expected that there are going to be about 2 branches in every 8 insts.

Multiple Branches per Cycle?

- We also need to predict multiple branches per cycle:
- Multi-port predictor
- History-based (pattern)
  - Precise
    - Wait for first prediction then do second
    - Do second with both possible outcomes for first and then select → expensive
  - Imprecise
    - Just use present history for both

Trace Cache

- Conventional cache:
  - *Stores instructions in static order*
- Trace cache:
  - *Stores instructions in the order they were **executed***

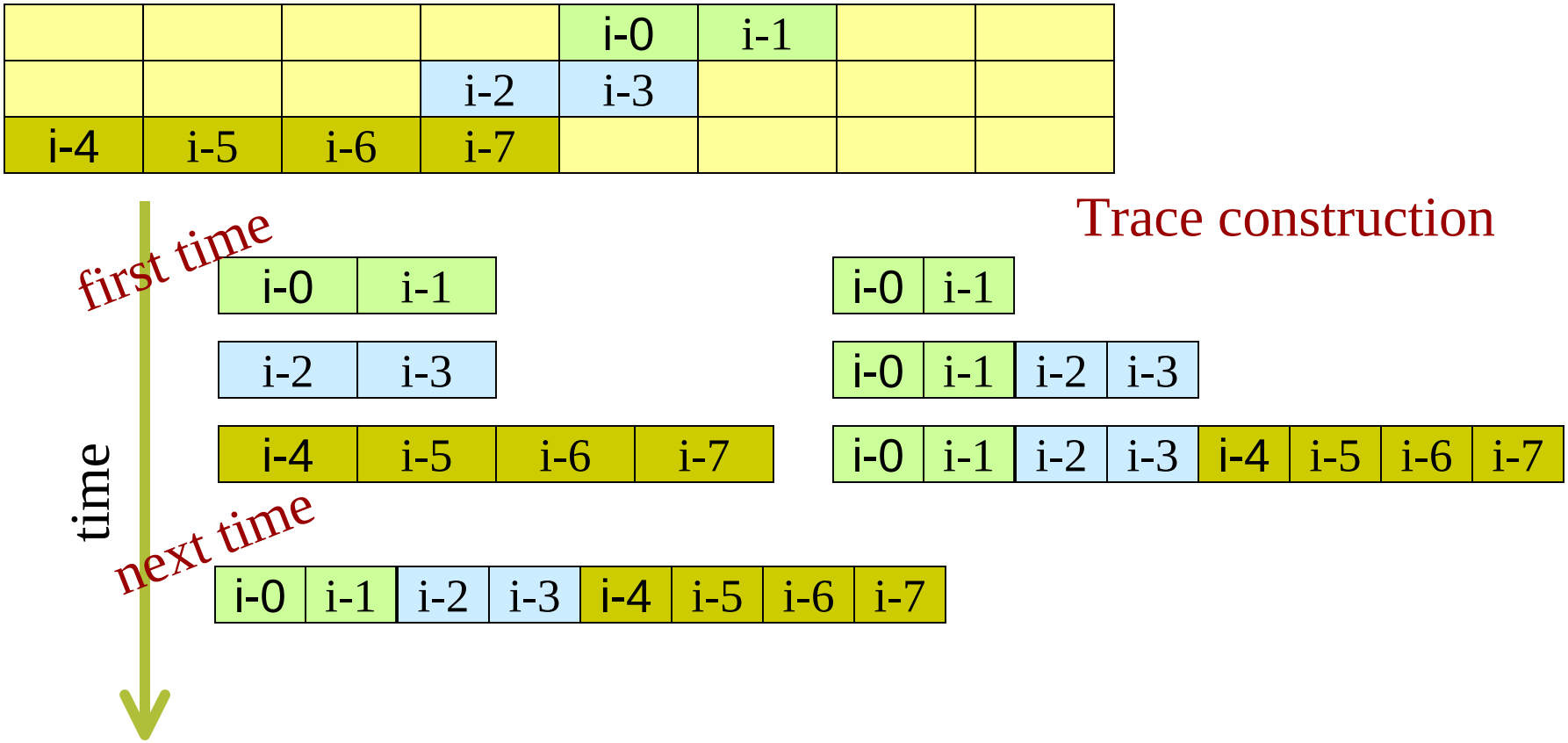
If we execute the same sequence again then the trace cache can provide all the instructions, nicely aligned in one access

- Trace:
  - ***Sequence of instructions as they were executed***

Trace Cache – Main Idea

- Optimize based on program behavior
  - Programs tend to execute the same trace many times and close in time
- Idea:
  - Pay the penalty once
    - Use conventional cache and branch predictor the first time
    - As instructions commit form trace
  - Next time use constructed trace
    - Implicitly predict multiple branches and fetch all instructions

Trace Cache Example



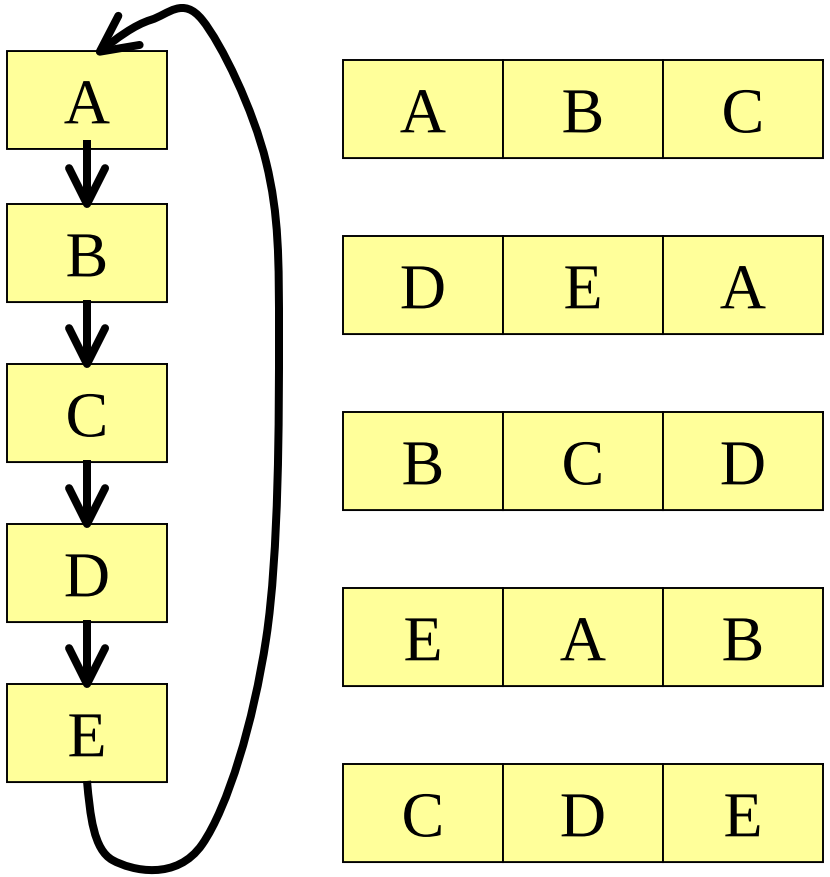
Trace cache also implicitly predicts multiple branches

Trace Cache Structures

- Trace Cache: Keeps the trace
  - lead PC
  - Bit vector for branch directions: **m-bits**
  - Branch count: < m (lg m bits)
  - Last inst a branch
    - Trace still correct if the last branch is miss-predicted
  - Fall through address
  - Target address
  - **n instructions**

Trace Cache Issues

- Same instructions may appear multiple times
- Careful trace selection can increase trace cache efficiency



Trace Selection

- When to Start and End a trace?
- Arbitrary points can lead to low storage efficiency and utilization
- Start at Branch targets
- Start at Function calls and returns

How to use the Trace Cache

- Use PC + multiple branch prediction bits as index
  - Predictions may come from separate predictor
- Alternatively, treat f(PC, multiple predictions) as the unit for the predictor

- Increase effectiveness by predicting two traces
  - Very likely
  - Somewhat likely

The P4 Trace Cache

- 12k uops
- X86 instructions are translated to sequences of uops
  - This process is expensive and complicated due to variable length encoding
- Cache past translations into the trace cache

Trace Cache Bibliography

[1] **A. Peleg, U. Weiser**, "*Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line",* US Patent number 5,381,533, Intel Corporation, 1994.

[2] **D. Friendly, S. Patel, and Y. Patt**, *"Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism",* in Proceedings of the 30th International Symposium on Microarchitecture, November 1997.

[3] **Q. Jacobson, E. Rotenberg, and J. Smith**, *"Path-Based Next Trace Prediction,"* in Proceedings of the 30th International Symposium on Microarchitecture, November 1997.

[4] **E. Rotenberg, S. Bennett, and J. Smith**, "*Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching",* in Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, 1996

Many other studies on improvements/alternatives