Processor FSM Design

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Abstract model of computer
- Single-cycle instruction execution
- Multi-cycle instruction execution
- Pipelined instruction execution

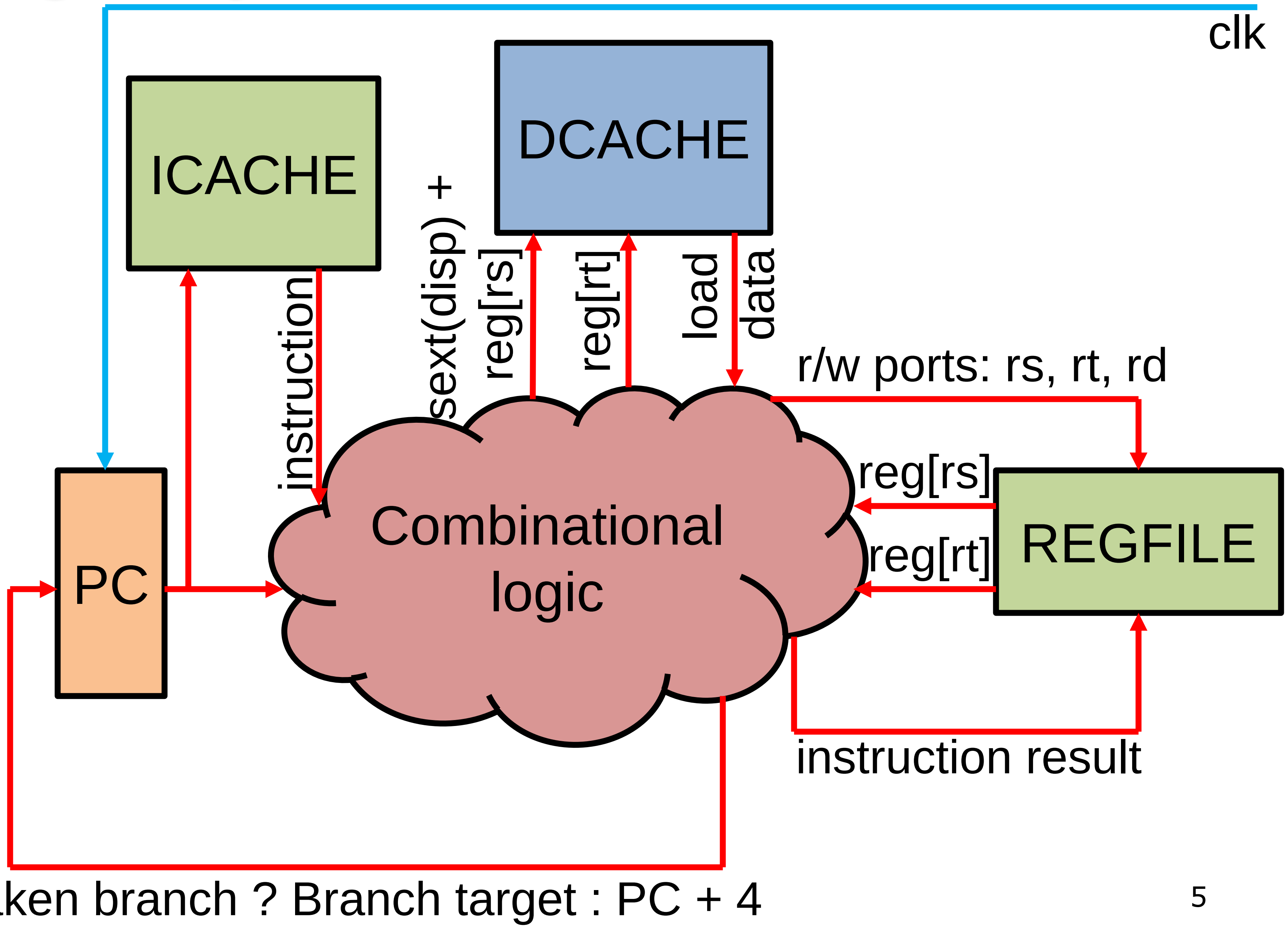
Abstract model of computer

- Each instruction undergoes five stages
  - Stage 0 (IF): fetch the instruction pointed to by program counter from memory
  - Stage 1 (ID/RF): decode the instruction to extract various fields and read source register operands
    - Source registers can be read in parallel with instruction decoding in MIPS due to fixed positions of rs and rt in all formats of the ISA
  - Stage 2 (EX): execute the instruction in ALU; compute address of load/store instructions; update program counter to PC+4 or branch target (if this instruction is control transfer ins)
  - Stage 3 (MEM): access memory if load/store instruction; use address computed in stage 2
  - Stage 4 (WB): write result back to destination register if the instruction produces a result

Single-cycle instruction execution

- Possible to implement all five stages as a big combinational logic
  - All the memory elements are also combinational meaning that the input address can be presented any time within a cycle and the output data is available within the same cycle
    - Register file (in stages 1 and 4), icache (in stage 0), and dcache (in stage 3) are the combinational memory elements
    - PC, icache[PC], reg[rs], reg[rt], dcache[disp(rs)] are inputs
      - dcache[disp(rs)] is a relevant input for loads
    - PC, reg[rd] or reg[rt], dcache[disp(rs)] are outputs
      - dcache[disp(rs)] is a relevant output for stores
  - The instruction opcode and function (for R format) fields decide the control of combinational logic e.g., which register operand(s) is/are valid sources, which ALU operation to invoke, etc.

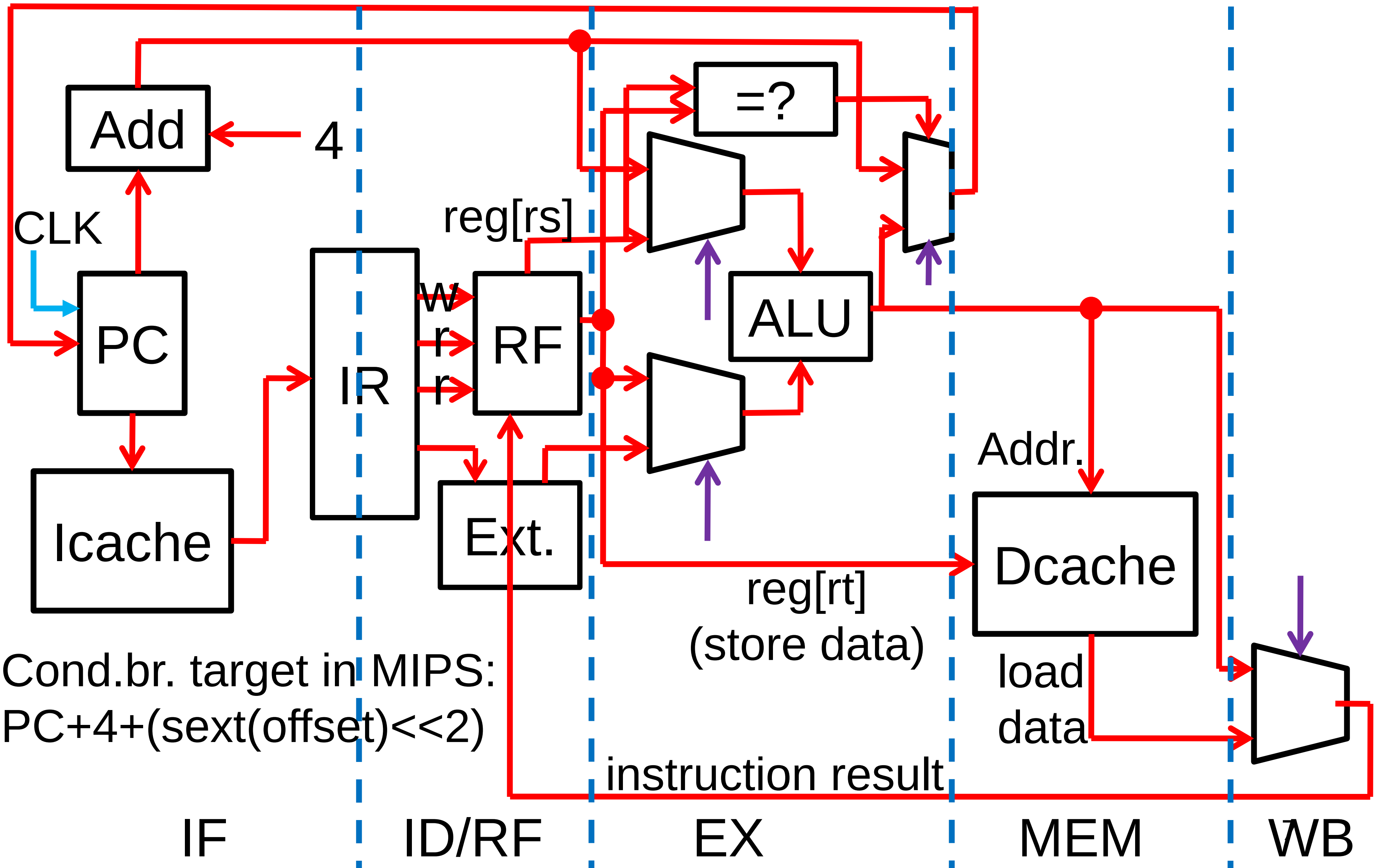
Single-cycle instruction execution



Single-cycle instruction execution

- Taken branch
  - A branch instruction whose condition dictates that the next instruction to be executed is the one at the branch target
    - beq \$1, \$2, label (the branch is taken if \$1 == \$2)
    - Unconditional direct and indirect jumps and procedure calls are always taken branches
- Not taken branch
  - A branch instruction whose condition dictates that the next instruction to be executed is the instruction right next to the branch
    - A la Frost's "Road Not Taken"
    - beq \$1, \$2, label (the branch is not taken if \$1 != \$2)
    - A conditional branch can be taken or not taken depending on how the condition evaluates at run time

Single-cycle instruction execution



Cond.br. target in MIPS:
PC+4+(sext(offset)<<2)

Single-cycle instruction execution

- Clock cycle time must be large enough to accommodate the lengthiest instruction
  - This is typically the load instruction whose critical path includes all five stages: IF: fetch instruction, ID/RF: decode while reading reg[rs] and sign extending displacement, EX: add reg[rs] with sext(disp) to generate address, MEM: read dcache to get load data, WB: write load data back to reg[rt]
  - Not a good idea since many other instructions take smaller amount of time
    - Control transfer instructions require only three stages
    - Stores require only four stages
    - All ALU instructions can bypass MEM stage

Multi-cycle instruction execution

- Each stage takes one cycle to execute
  - Processor is now a five state FSM and the memory elements are sequential
  - Need flip-flops at stage boundaries and each stage is a combinational logic
  - Clock cycle time = latency of the longest stage
  - Most instructions take five cycles
    - ALU instructions do nothing in MEM stage, but will have to go through it if FSM state transition is implemented using an incrementer
    - Some instructions may take longer e.g., multiply/divide, load/store (in case of dcache miss)
    - Some instructions may take shorter time e.g., stores take four cycles, control transfer instructions take three cycles
  - Overall CPI depends on instruction mix

Multi-cycle instruction execution

- Example
  - IF: 2 ns, ID/RF: 1 ns, EX: 1 ns, DMEM: 3 ns, WB: 1 ns
  - Branch frequency: 20%
  - Store frequency: 10%
  - Multiply/divide frequency: 5%, latency: 30 ns
  - Total instruction count: 100
- Multi-cycle
  - Cycle time: 3 ns, frequency: 333 MHz
  - CPI: 0.2*3+0.1*4+0.05*10+0.65*5 = 4.75
  - Execution time: 100*4.75*3 ns = 1425 ns
- Single-cycle
  - Cycle time: 8 ns, frequency: 125 MHz
  - CPI: 1*0.95+ceil(30/8)*0.05 = 1.15
  - Execution time: 100*1.15*8 ns = 920 ns

Multi-cycle instruction execution

- Same example with a balanced pipeline
  - IF: 2 ns, ID/RF: 2 ns, EX: 2 ns, DMEM: 2 ns, WB: 2 ns
  - Branch frequency: 20%
  - Store frequency: 10%
  - Multiply/divide frequency: 5%, latency: 30 ns
  - Total instruction count: 100
- Multi-cycle
  - Cycle time: 2 ns, frequency: 500 MHz
  - CPI: 0.2*3+0.1*4+0.05*15+0.65*5 = 5
  - Execution time: 100*5*2 ns = 1000 ns
- Single-cycle
  - Cycle time: 10 ns, frequency: 100 MHz
  - CPI: 1*0.95+ceil(30/10)*0.05 = 1.1
  - Execution time: 100*1.1*10 ns = 1100 ns (worse than multi-cycle)

Pipelined instruction execution

- Observations from multi-cycle design
  - In the second cycle, I know if it is a branch; if not, start fetching the next instruction?
  - When the ALU is doing an addition (say), the decoder is sitting idle; can we use it for some other instruction?
  - In summary, exactly one stage is active at any point in time: wastes hardware resources
- Form a pipeline
  - Process five instructions in parallel
  - Each instruction is in a different stage of processing (called pipe stage)

Pipelined instruction execution

- Individual instruction latency is five cycles, but ideally can finish one instruction every cycle after the pipeline is filled up
  - Ideal CPI of 1.0 at the clock frequency of multi-cycle design
  - Execution time is ideally one-fifth of the multi-cycle design
    - Instruction throughput improves five times (number of instructions completed in a given time)

I0	IF	ID/RF	EX	MEM	WB			
I1		IF	ID/RF	EX	MEM	WB		
I2			IF	ID/RF	EX	MEM	WB	
I3				IF	ID/RF	EX	MEM	WB
I4					IF	ID/RF	EX	MEM

time →

Pipelined instruction execution

- Gains
  - Extracting parallelism from a sequential instruction stream: known as instruction-level parallelism (ILP)
  - Can complete one instruction every cycle (ideally)
- Loss
  - Each pipe stage may get lengthened a little bit due to control overhead (skew time, setup time, propagation delay): limits the gain due to pipelining
  - Each instruction may take slightly longer for this reason
  - Bigger aggregate memory bandwidth: icache and dcache may miss in the same cycle
  - Pipeline hazards

Pipeline hazards: structural

- Structural hazards
  - Arises due to resource conflicts
  - Happens if the same resource is accessed in at least two stages of the pipe
  - Fewer resources than needed
  - Unpipelined functional units
  - Lack of memory or register file ports
- Why fewer resources?
  - Reduction in complexity (and power consumption)
  - Make the common case fast: pipelined divider may only waste silicon estate
- All types of hazards introduce stalls or pipeline bubbles

Pipeline hazard: control

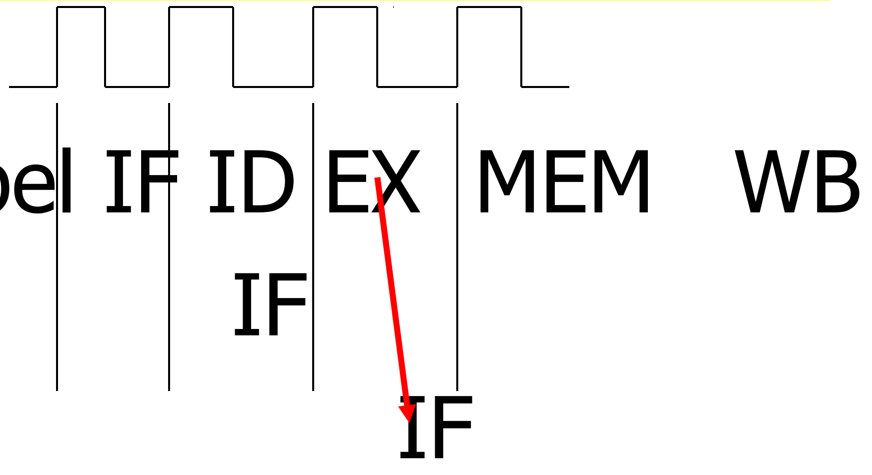
- Branches pose a problem

	Cycles $\longrightarrow$				
ins *i* (*beq* \$1, \$0, label)	IF	ID/RF	EX	MEM	WB
What to fetch?		IF			
What to fetch?			IF		
ins *i*+1 (target)				IF	

- Two pipeline bubbles: increases average CPI
- Can we reduce it to one bubble?

Branch delay slot

- MIPS R3000 has one bubble
  - Called branch delay slot
  - Exploit clock cycle phases
  - On the positive half compute branch condition
  - On the negative half fetch the target



beq $1, $0,label IF ID EX MEM WB

Bubble IF

Target IF

- The PC update hardware (selection between target and next PC) works on the lower edge

Branch delay slot

- Can we utilize the branch delay slot?
  - The delay slot is always executed (irrespective of the fate of the branch)
    - Reason why branch target is PC+4+(sext(offset)<<2)
  - Boost instructions common to fall through and target paths to the delay slot or from earlier than the branch

```
if (cond) {
   // fall through
}
else {
   // target
}
```

branch instruction

delay slot

Branch delay slot

- Can we utilize the branch delay slot?
  - Boost instructions common to fall through and target paths to the delay slot or from earlier than the branch
  - Not always possible to find
  - Must boost something that does not alter the outcome of the computation
  - If the BD slot is filled with useful instruction then we don't lose anything in CPI; otherwise we pay a branch penalty of one cycle

Branch prediction

- To increase the clock frequency, designers may choose a deeper pipeline
  - Subdivide the longest stage further
- If the number of pipe stages increases between IF and EX, branch penalty also increases
  - Pipe1: IF ID/RF EX … Br. penalty?
  - Pipe2: IF ID RF EX … Br. penalty?
  - Pipe3: IF1 IF2 ID RF1 RF2 EX … Br. penalty?
  - Assume no branch delay slot
    - Meaning that IF or EX cannot be accommodated in half cycle

Branch prediction

- To increase the clock frequency, designers may choose a deeper pipeline
  - Subdivide the longest stage further
- If the number of pipe stages increases between IF and EX, branch penalty also increases
- Today all processors rely on branch predictors that observe the behavior of individual branches and learn to predict their future behavior
  - Makes it possible to infer the next instruction's PC even before the branch executes
  - Pipeline must be flushed on a wrong prediction

Branch target buffer

- Branch target buffer
  - We can put a branch target cache in the fetcher
  - Also called branch target buffer (BTB)
  - Use the lower bits of the instruction PC to index the BTB
  - Use the remaining bits to match the tag
  - In case of a hit the BTB tells you the target of the branch when it executed last time
  - You can hope that this is correct and start fetching from that predicted target provided by the BTB
  - Later you get the real target, compare with the predicted target, and throw away the fetched instruction in case of misprediction; keep going if predicted correctly

Branch target buffer

- BTB is looked up with the PC of every instruction in parallel with fetching the instruction
  - On a hit, it provides two pieces of information: this instruction is a control transfer instruction and the target of this control transfer instruction seen last time
    - This target will be used to fetch in the next cycle
  - On a miss, the fetcher has no option but to fetch from the fall through path (PC+4) in the next cycle
  - A control transfer instruction is inserted in the BTB after the EX stage when its target is known
    - A lookup at this point may hit in the BTB; if the branch is not taken, the BTB entry is invalidated; otherwise the entry is updated with the taken target
    - If a lookup at this point misses in the BTB, a new entry is allocated provided the branch is taken

Branch target buffer

- Assume for a program
  - 90% of all control transfer instructions hit in the BTB
  - 90% of outcomes provided by BTB hits are correct
  - 20% of control transfer instructions that miss in the BTB result in taken branches
  - Fraction of bubbles saved = BTB prediction accuracy = 0.9*0.9+0.1*0.8 = 0.89
  - 11% branches suffer from mispredictions and will require some "recovery" mechanism for correct execution

Branch prediction

- BTB will work great for
  - Loop branches (how many misprediction?)
  - Subroutine calls
  - Unconditional branches
- What about *jalr* ?
- Conditional branch prediction
  - Rather dynamic in nature
  - The last target is not very helpful in general (if-then-else)
  - Need a direction predictor (predicts taken or not taken)
  - Once that prediction is available we can compute the target

Direction predictors

- How about static prediction?
  - Always not-taken (NT) or always taken (T): penalty?
  - Forward not-taken and backward taken (rationale?)
- Deeper pipelines
  - Define branch penalty
  - Big problem: deeper pipelines increase branch penalty
  - Must have better branch predictors for deeper pipeline

Control dependence

- Roughly every fifth instruction is a branch
  - Need to be on the right *control flow path*
    - This is the source of input to the pipeline
  - Static techniques are not enough
    - Need highly accurate dynamic predictors
  - Speculate past branches: Alpha 21264 allows 20 outstanding branches, MIPS R10000 allows only 4
    - Need to speculate past predicted branches in deeper pipelines (not a big issue in five-stage pipe)
  - Prediction accuracy?
    - Probability of a correct prediction is p
    - Probability of staying on correct path after n predictions p^n
    - What is minimum p if n is 4 (MIPS R10k)? If n is 20?

Direction predictors

- Let us encode each taken branch as 1 and not taken branch as 0
  - The behavior of a conditional branch can be represented as a binary string
  - Loop branch: 11111…110
  - Alternating if-else: 1010101010 or 0101010101
  - If-else branches can exhibit a wide variety of patterns

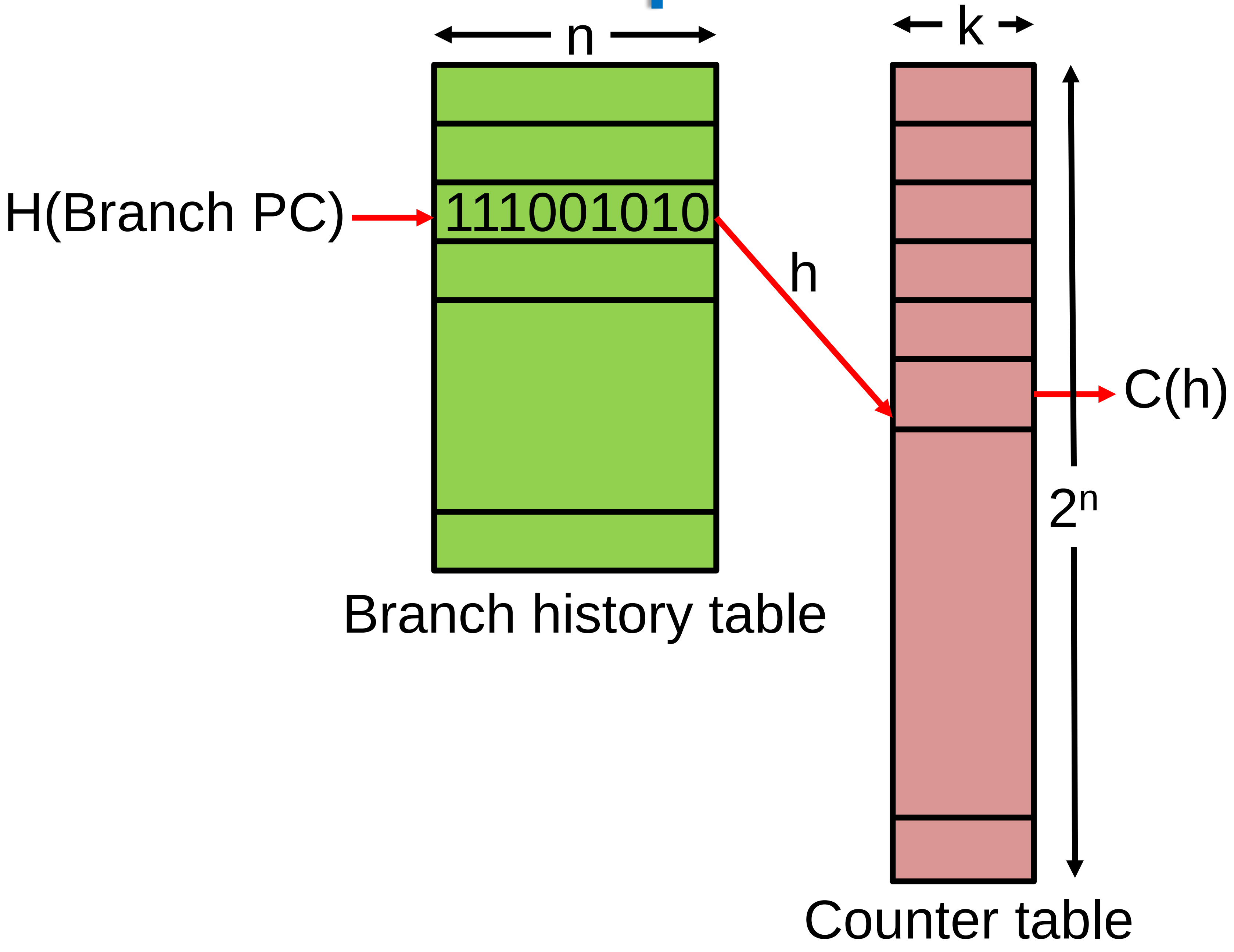
Direction predictors

- The problem of direction prediction is essentially design of an estimator that, given an n-bit history, tells us the next most likely outcome for a particular static branch instruction
  - All branch predictors compute the probability of seeing a zero or one, given the recent pattern history h of some limited length n
  - Suppose the number of time 0 appears after h is C0 and similarly define C1; the prediction is 1 if C1 >= C0 and 0 otherwise
    - Instead of having two counters, C1-C0 is maintained

Direction predictors

- Let the difference counter be C(h) for a certain history h: C(h) = C1(h) – C0(h)
  - Let C(h) be of k bits length (this is independent of the history length)
  - Can count from 0 to 2^k -1
  - On seeing 0 after h, decrement C(h); on seeing 1 after h, increment C(h)
  - Saturates at boundaries (a saturating counter)
    - Does not decrement below 0 or increment above 2^k -1
  - By examining C(h) at any point in time, we can say which outcome had higher likelihood in the last 2^k occurrences of history h
    - Need to shift the origin to mid-point 2^{k-1}
    - If C(h) is below mid-point, the prediction is zero; otherwise 1

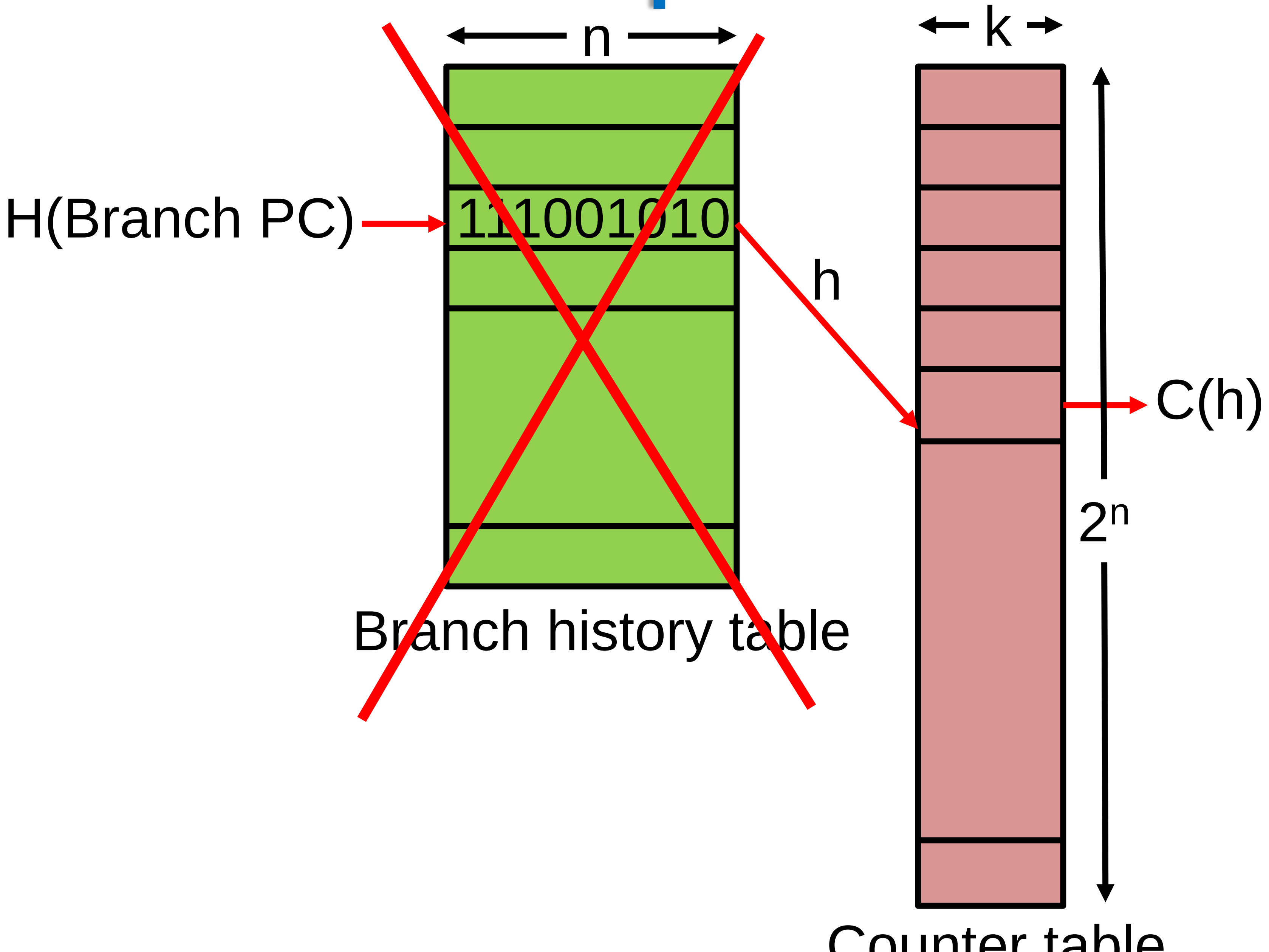
Direction predictors



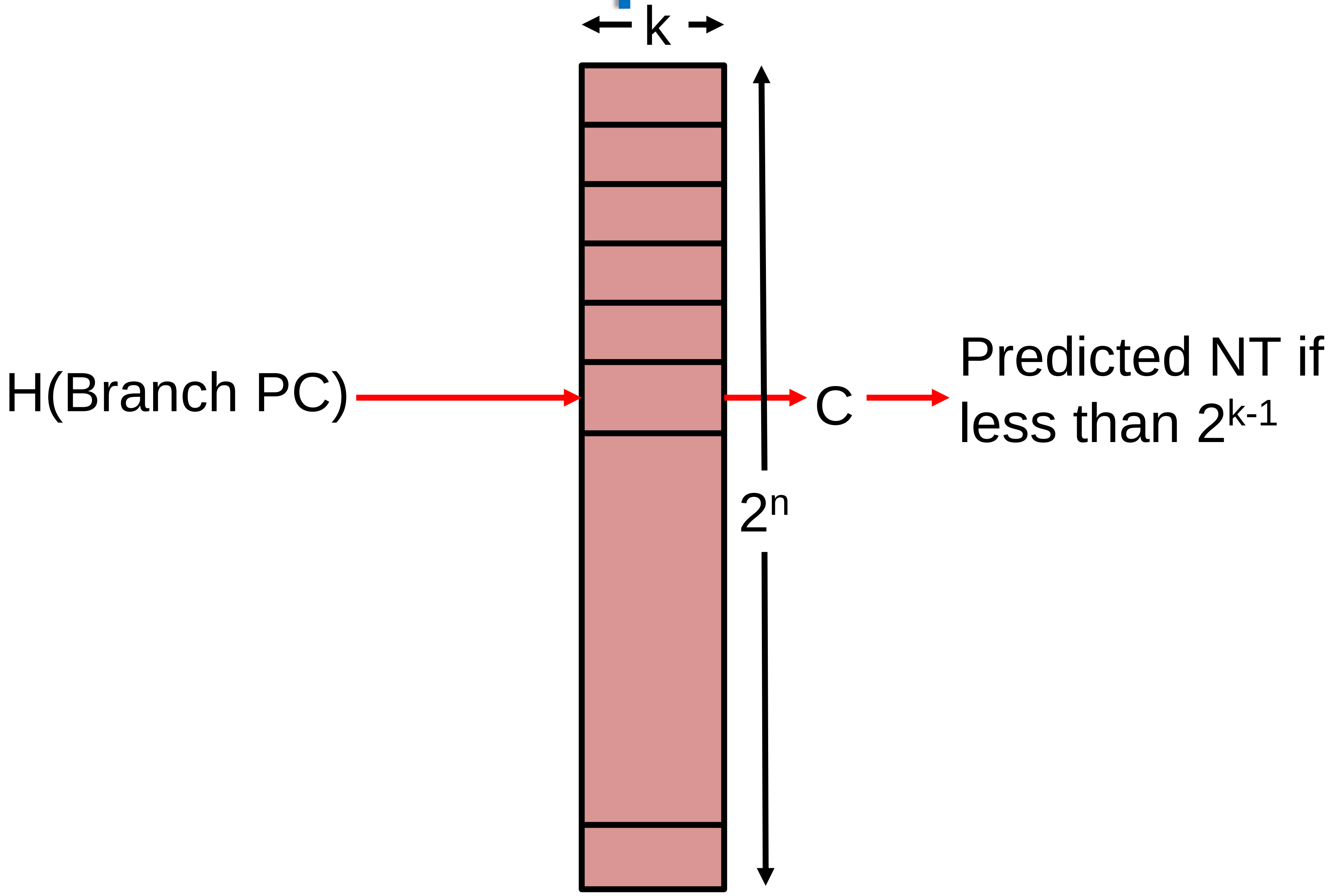
Direction predictors

- How many different histories? [Size of BHT]
  - No history? [no BHT] Two possible designs:
    - Just a global counter that counts occurrences of 0’s and 1’s and offers prediction based on that
      - Not very useful
    - One counter per branch (known as bimodal predictor)
      - Somewhat useful: helps identify largely bimodal distributions
  - One global history? [Size of BHT = 1]
    - Captures cross-correlation between branches
    - Since history is a sliding pattern, h will keep on changing
      - One counter for each history pattern? Use a hashmap
  - One local history per branch? Size of BHT?
    - Hash history patterns to counters
    - Loses global correlation

Bimodal predictors



Bimodal predictors



Pattern history table (PHT)

Bimodal predictors

- Simplest of the lot (used in MIPS R10000)
  - Maintains a pattern history table (PHT)
    - MIPS called it a branch history table (BHT)
  - Each entry is a saturating counter
  - Basic idea is to go with the most frequent pattern seen in the past
  - How do you index it?
  - Aliasing?
  - How wide is each counter?
  - Performance of loops (simplest kind of control flow)
  - Alternating branches?
  - Correlating branches?

Challenges

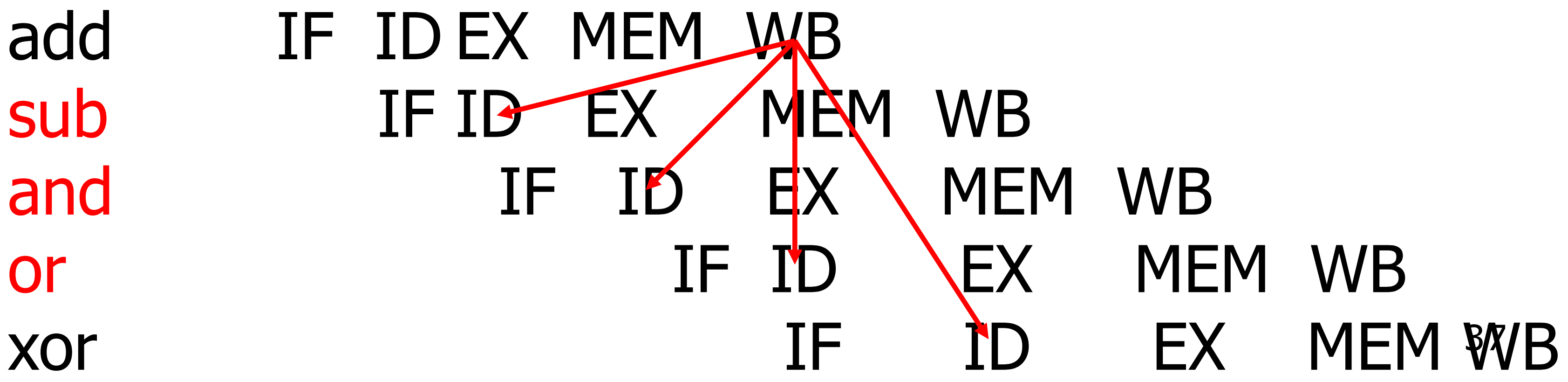
- Aliasing in pattern history table
  - Multiple branches mapping to the same counter
- Height of pattern history table is exponential in history length
  - Need long history for better prediction
  - Neural predictors help
- What are the best ways to combine multiple predictors?
  - Each predictor can be an expert for a certain type of branch

Pipeline hazard: data

- Pipelining disturbs the sequential thought-process
  - Data dependencies among instructions start to show up

```
add $1, $2, $3
sub $4, $1, $5
and $6, $1, $7
or  $8, $1, $9
xor $10, $1, $11
```

  - Result of add is needed by all instructions (RAW hazard: read after write hazard)



```
add     IF  ID EX  MEM  WB
sub          IF ID    EX     MEM  WB
and                IF   ID      EX     MEM  WB
or                         IF  ID      EX     MEM  WB
xor                              IF      ID      EX     MEM WB
```

Pipeline hazard: data

- How to avoid increasing CPI?
  - Stalling is clearly not acceptable
  - Phased register file solves three-cycle apart RAW
  - Can we forward the correct value just in time?

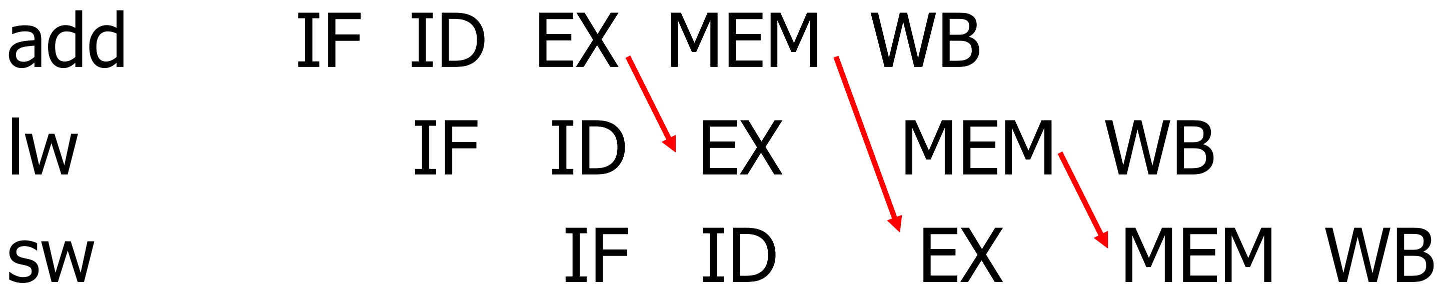
$$\begin{tikzcd}
\text{add} & \text{IF} & \text{ID} & \text{EX} \arrow[dr] & \text{MEM} \arrow[ddr] & \text{WB} & & \\
\text{sub} & & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} & \\
\text{and} & & & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB}
\end{tikzcd}$$

  - Read wrong value in ID/RF, but bypassed value overrides it (need a multiplexor for each ALU input to choose between the RF value and the bypassed value)

Data hazards

- MEM/WB to MEM bypass

```
add r1, r2, r3
lw    r4, 0(r1)
sw   r4, 20(r1)
```

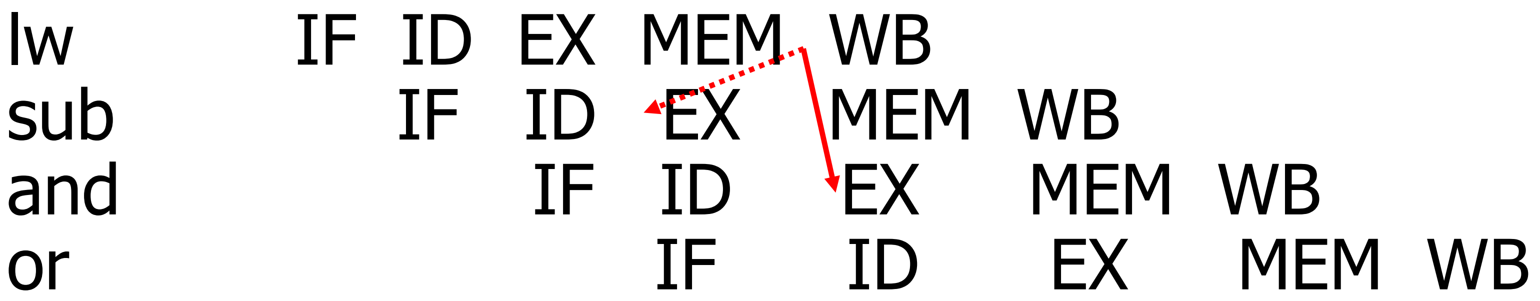


```
add     IF  ID  EX  MEM  WB
lw          IF  ID   EX     MEM  WB
sw              IF   ID     EX       MEM  WB
```

Pipeline hazard: data

- Can we always avoid stalling?

```
lw   $1, 0($2)
sub  $4, $1, $5
and  $6, $1, $7
or   $8, $1, $9
```



```
lw        IF  ID  EX  MEM  WB
sub           IF  ID   EX   MEM  WB
and               IF   ID        EX   MEM  WB
or                     IF        ID        EX   MEM  WB
```

  - Need some time travel (backwards)! Not yet feasible!!
  - One option: hardware *pipeline interlock* to stall the *sub* by a cycle
  - Early generations of MIPS (Microprocessor without Interlocked Pipe Stages) had the compiler to fill the *load delay slot* with something independent or a NOP

Hiding data hazard stalls

- In general, cache misses for load instructions can introduce large stalls in the pipeline
- Some of the established ways to hide some of these stalls
  - Execute instructions that do not depend on the load instruction that has missed in the cache
    - Requires ways to reorder the instruction sequence inside the processor
  - Prefetch data into cache before it is needed
    - Need to predict the sequence of data addresses and prefetch from memory hierarchy based on the prediction
  - Predict return value of the load instruction even before it executes

Pipelined instruction execution

- All three types of hazards can cause pipeline stalls and introduce bubbles in the pipeline depending on the pipeline organization
- Overall speedup of pipelining over multi-cycle non-pipelined implementation = number of pipe stages/(1 + average stall cycles per instruction)