CS 152 Computer Architecture and Engineering

Lecture 5 - Pipelining II
(Branches, Exceptions)

Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

http://inst.eecs.berkeley.edu/~cs152

Last time in Lecture 4

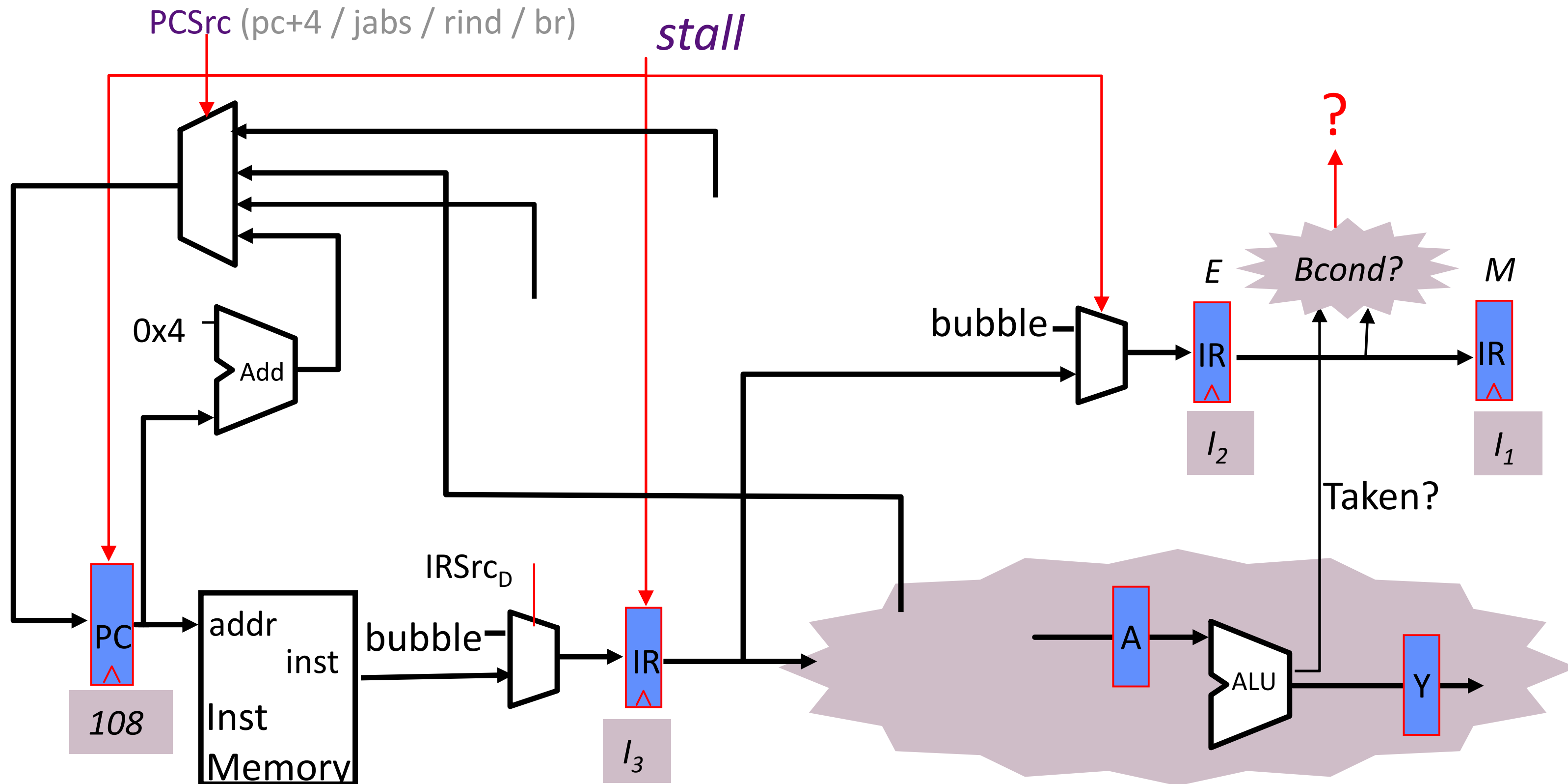
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Increases because of pipeline bubbles (Cycles/Instruction)

Reduces because fewer logic gates on critical paths between flip-flops (Time/Cycle)

- Pipelining increases clock frequency, while growing CPI more slowly, hence giving greater performance
- Pipelining of instructions is complicated by HAZARDS:
  - Structural hazards (two instructions want same hardware resource)
  - Data hazards (earlier instruction produces value needed by later instruction)
  - Control hazards (instruction changes control flow, e.g., branches or exceptions)
- Techniques to handle hazards:
  1) Interlock (hold newer instruction until older instructions drain out of pipeline and write back results)
  2) Bypass (transfer value from older instruction to newer instruction as soon as available somewhere in machine)
  3) Speculate (guess effect of earlier instruction)

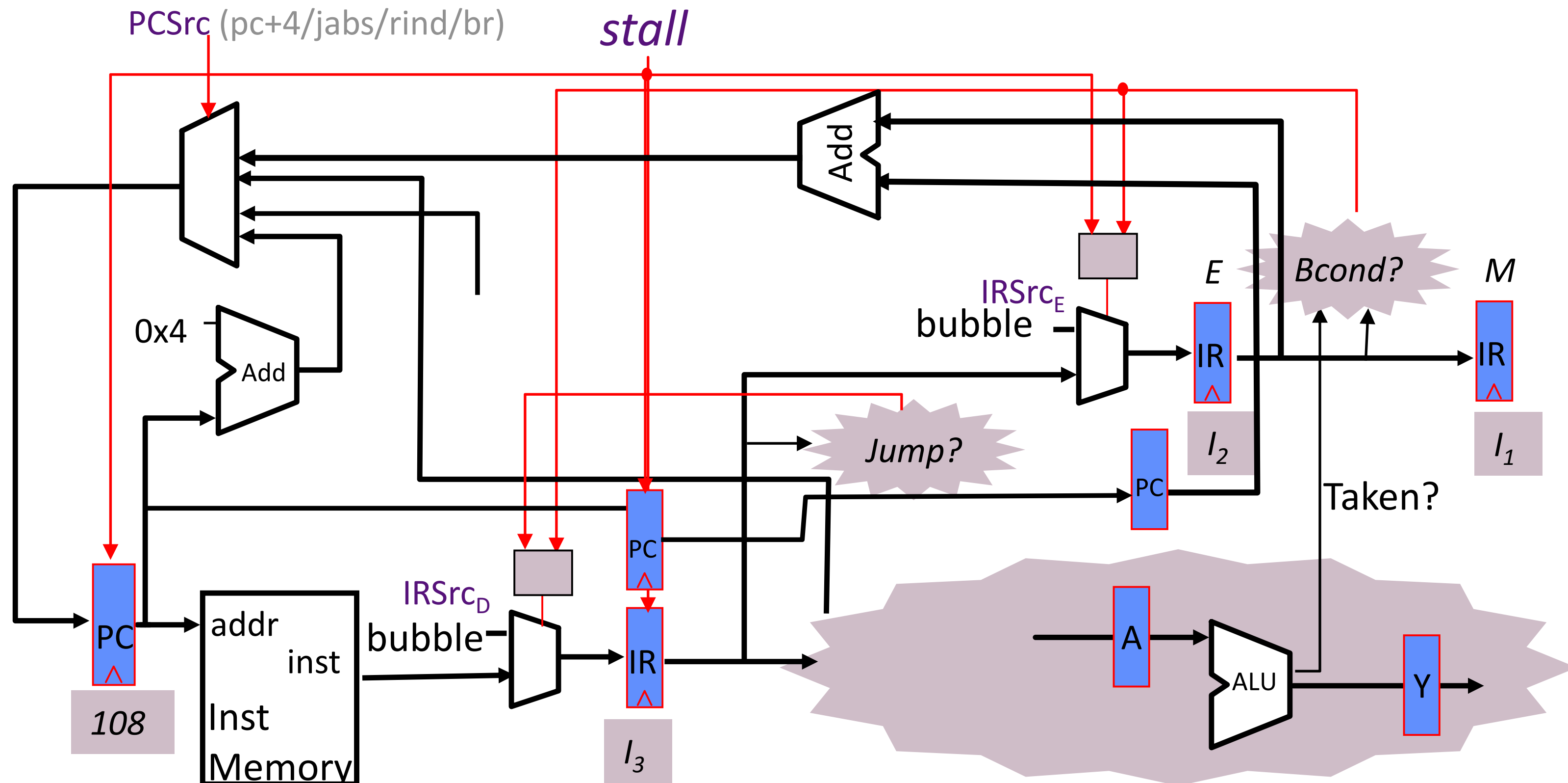
Pipelining Conditional Branches



If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid ⇒ *stall signal is not valid*

I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	300	ADD

Pipelining Conditional Branches

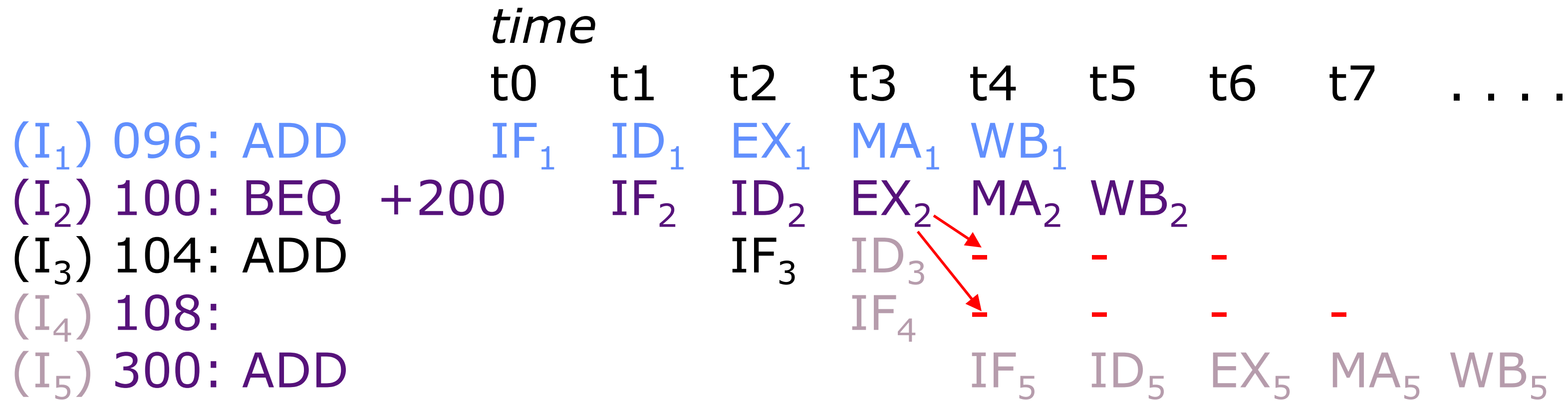


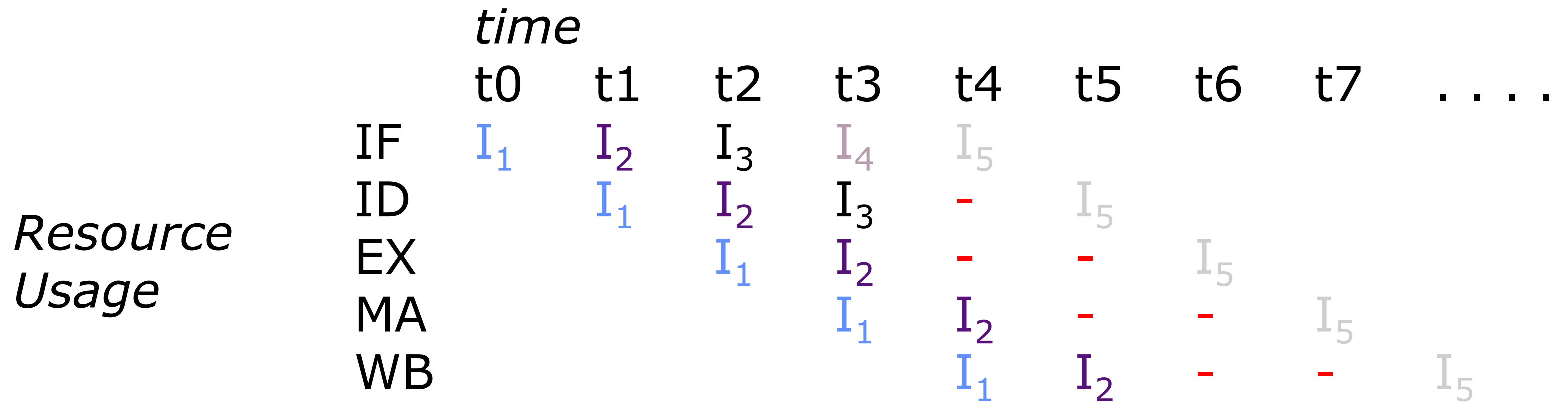
If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid ⇒ *stall signal is not valid*

$I_{1:}$	096	ADD
$I_{2:}$	100	BEQ x1,x2 +200
$I_{3:}$	104	ADD
$I_{4:}$	300	ADD

Branch Pipeline Diagrams
(resolved in execute stage)

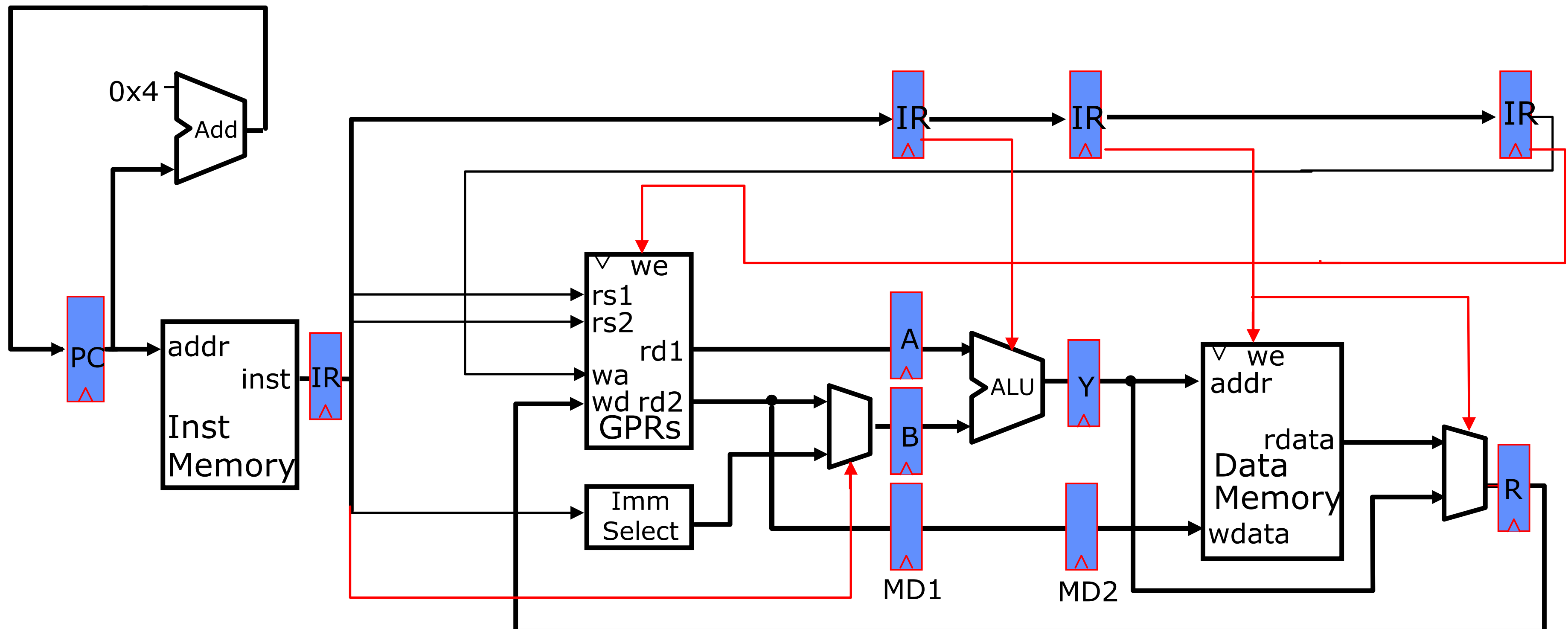




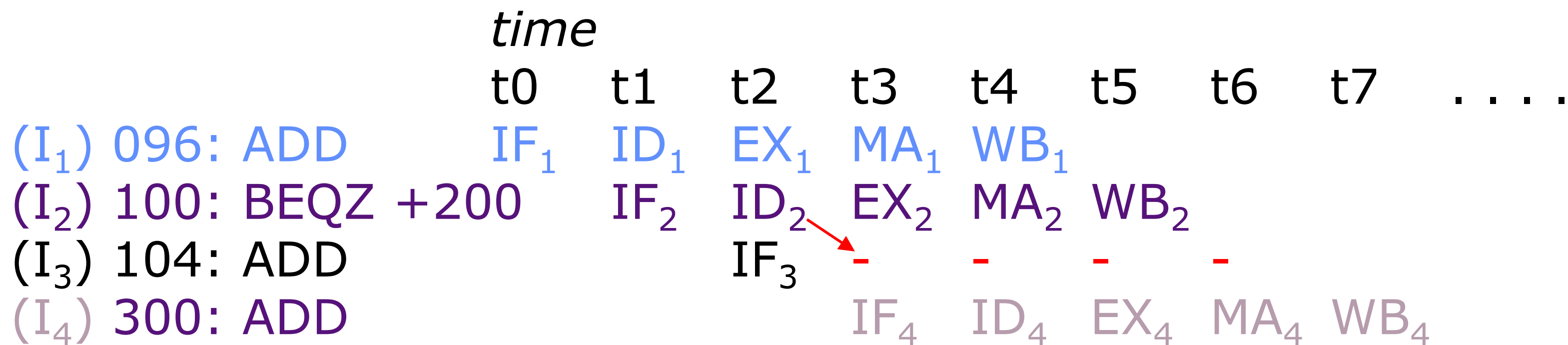
- ⇒ *pipeline bubble*

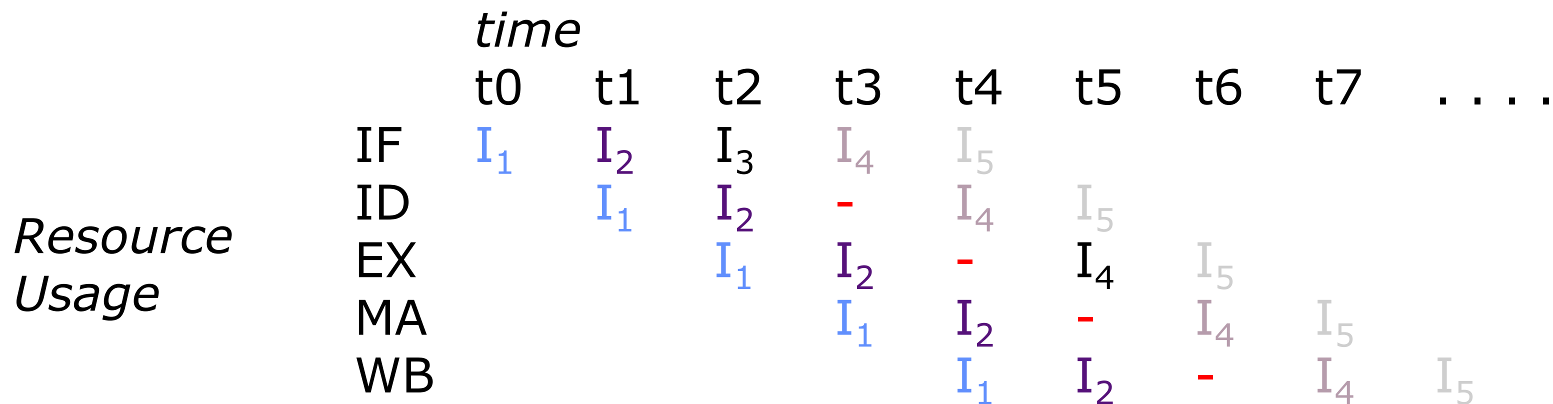
What If…

- We used a simple branch that compares only one register (rs1) against zero
- Can we do any better?



Use simpler branches (e.g., only compare one reg against zero) with compare in decode stage





- ⇒ *pipeline bubble*

What If… (2)

- Instead of flushing the pipeline in response to a control hazard, is there a way we can make the instruction following the branch always useful?

I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	300	ADD

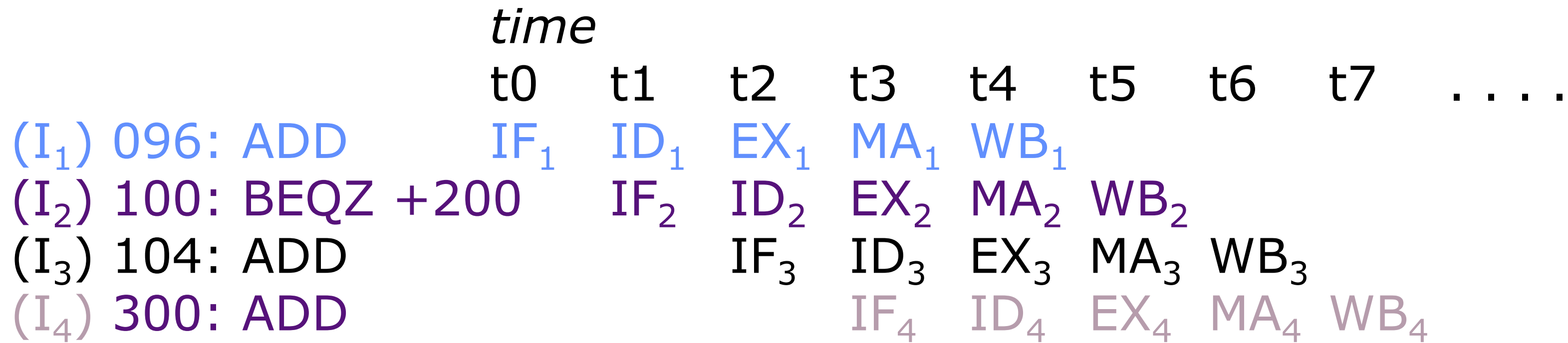
Branch Delay Slots (expose control hazard to software)

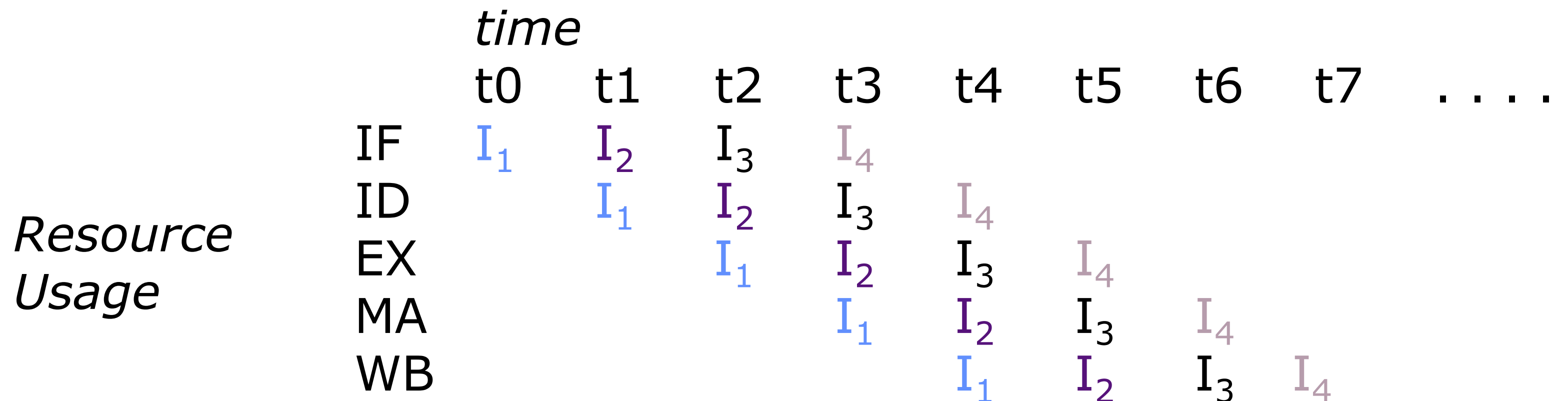
- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

I_1	096	ADD
I_2	100	BEQZ r1, +200
I_3	104	ADD
I_4	300	ADD

Delay slot instruction executed regardless of branch outcome (arrow pointing to I_3 104 ADD)

Branch Pipeline Diagrams
(branch delay slot)





Post-1990 RISC ISAs don't have delay slots

- Encodes microarchitectural detail into ISA
  - C.f. IBM 650 drum layout
- What are the problems with delay slots?
- Performance issues
  - E.g., I-cache miss or page fault on delay slot instruction causes machine to wait, even if delay slot is a NOP
- Complicates more advanced microarchitectures
  - 30-stage pipeline with four-instruction-per-cycle issue
- Complicates the compiler's job
- Better branch prediction reduced need for delay slots

Why an Instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II (pipeline interlocks added in hardware)
    - MIPS:"**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages"
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler. NOPs increase instructions/program!

RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

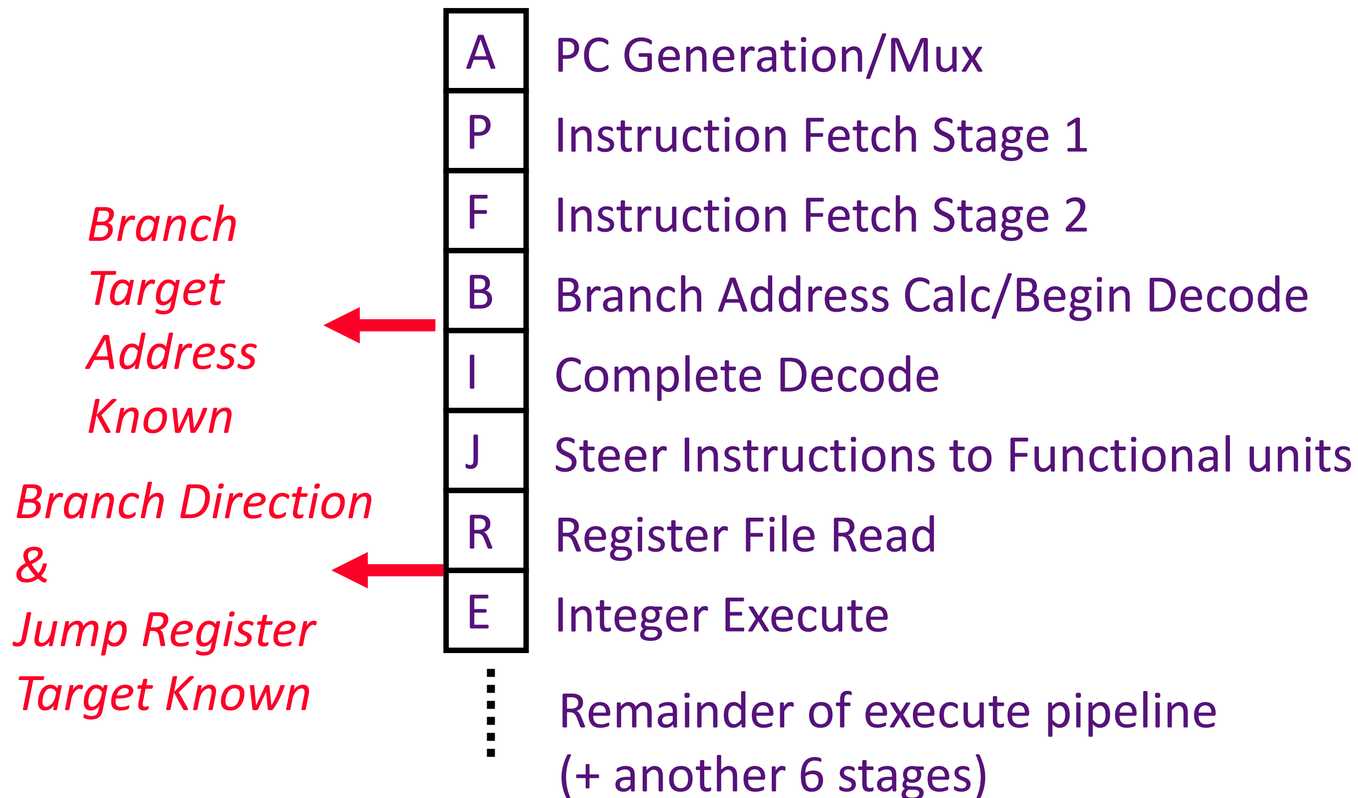
1) Is the preceding instruction a taken branch?

2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Reg. Fetch
B<cond.>	After Execute	After Inst. Decode

Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)



Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches - loop unrolling
    - Increases the run length
  - Reduce resolution time - instruction scheduling
    - Compute the branch condition as early as possible (of limited value because branches often in critical path through code)

- Hardware solutions
  - Find something else to do - delay slots
    - Replaces pipeline bubbles with useful work (requires software cooperation)
  - Speculate - branch prediction
    - Speculative execution of instructions beyond the branch

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:

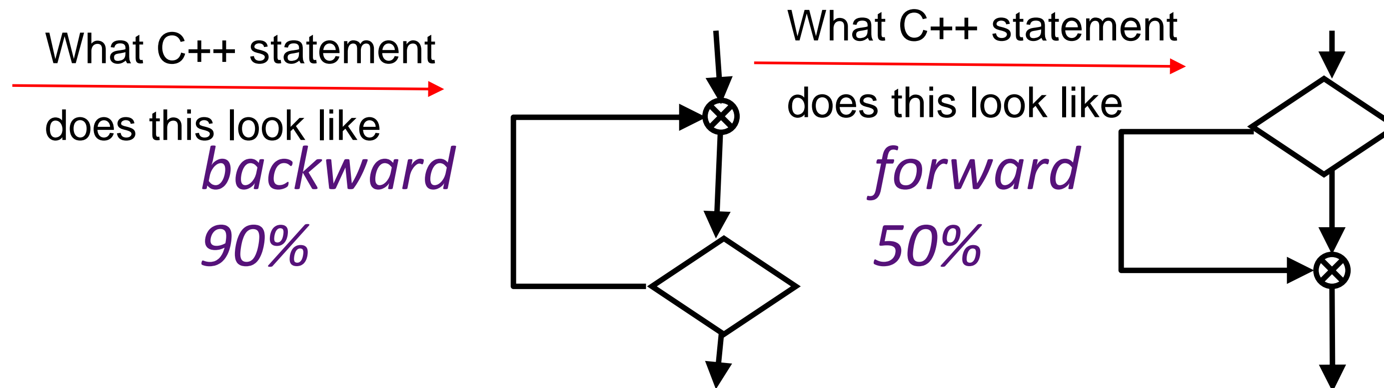
- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to that following branch

Static Branch Prediction

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



ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

Dynamic Branch Prediction learning based on past behavior

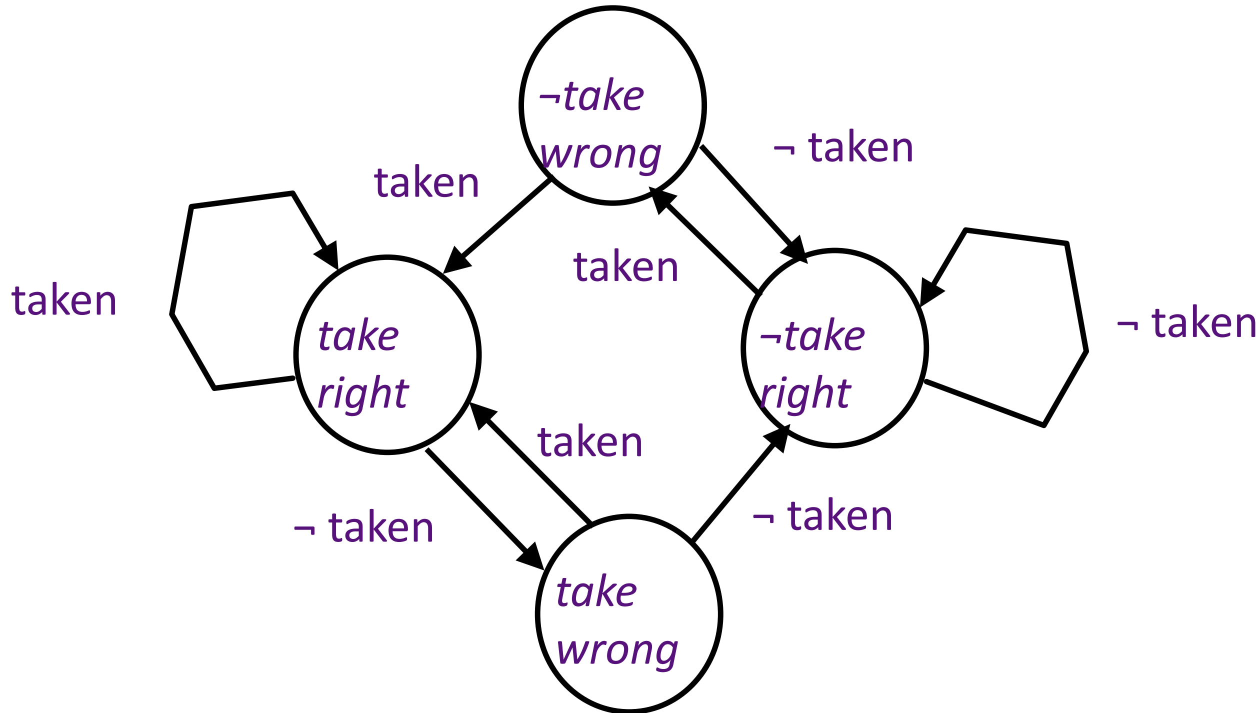
- Temporal correlation (time)
  - If I tell you that a certain branch was taken last time, does this help?
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation (space)
  - Several branches may resolve in a highly correlated manner
  - For instance, a preferred path of execution

What If… (3)

- Every branch remembers what it decided last time, and does the same thing
  - First, how do we implement this?
  - Is this effective?

Branch Prediction Bits

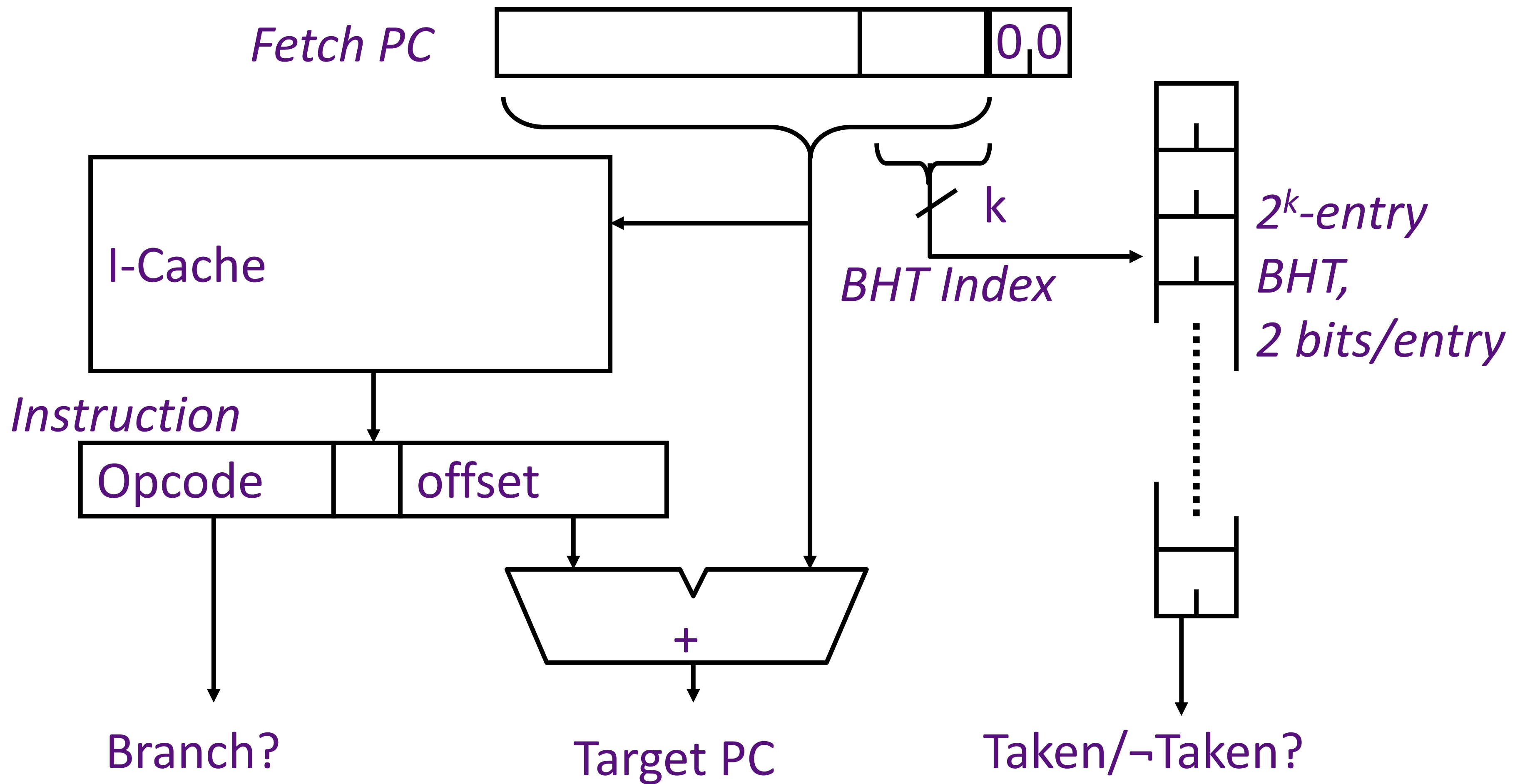
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



BP state:

(*predict* take/¬take) x (*last prediction* right/wrong)

Branch History Table



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

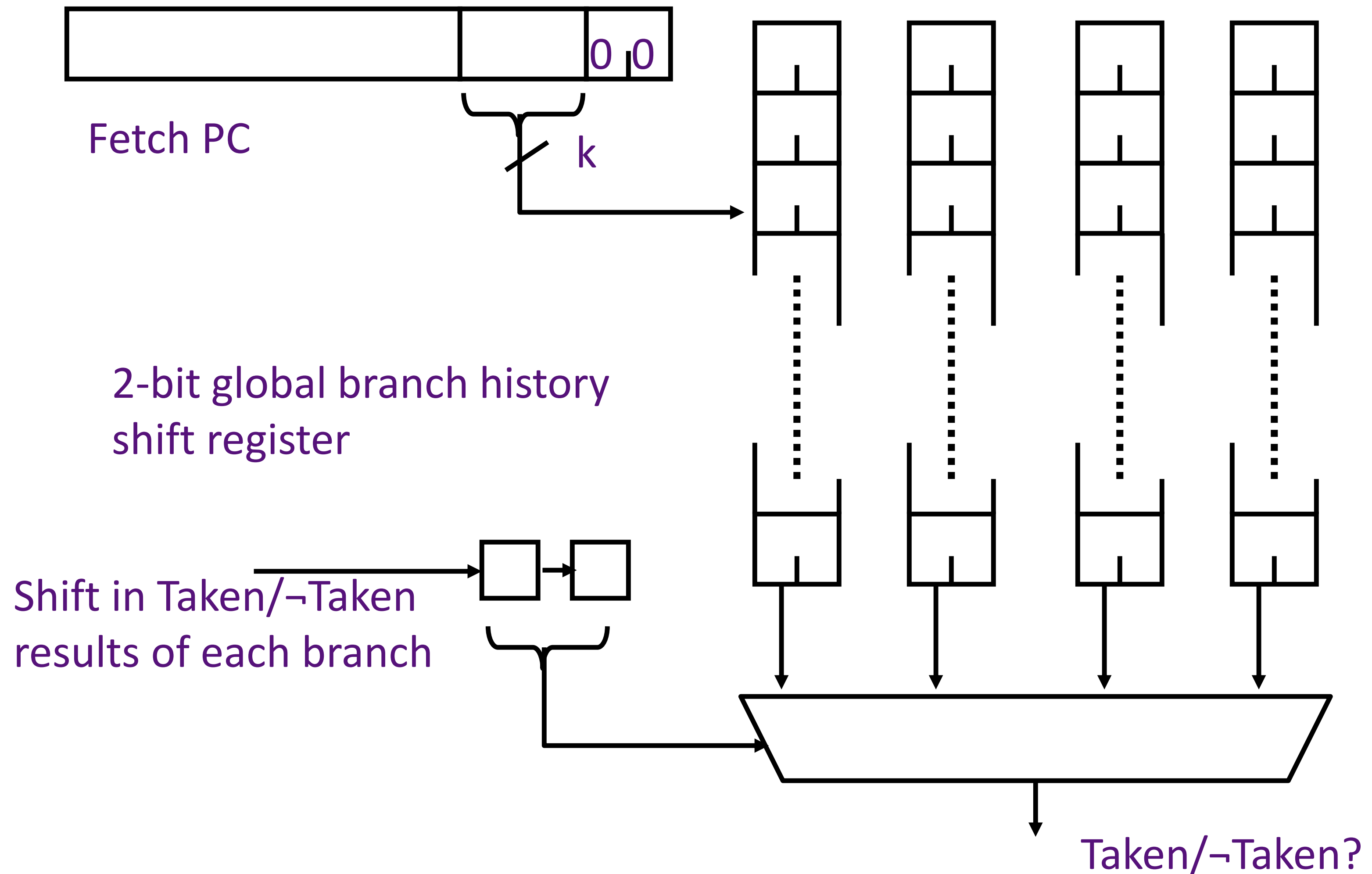
History register, H, records the direction of the last N branches executed by the processor

What If… (4)

- We wanted to use both spatial and temporal correlation

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Speculating Both Directions

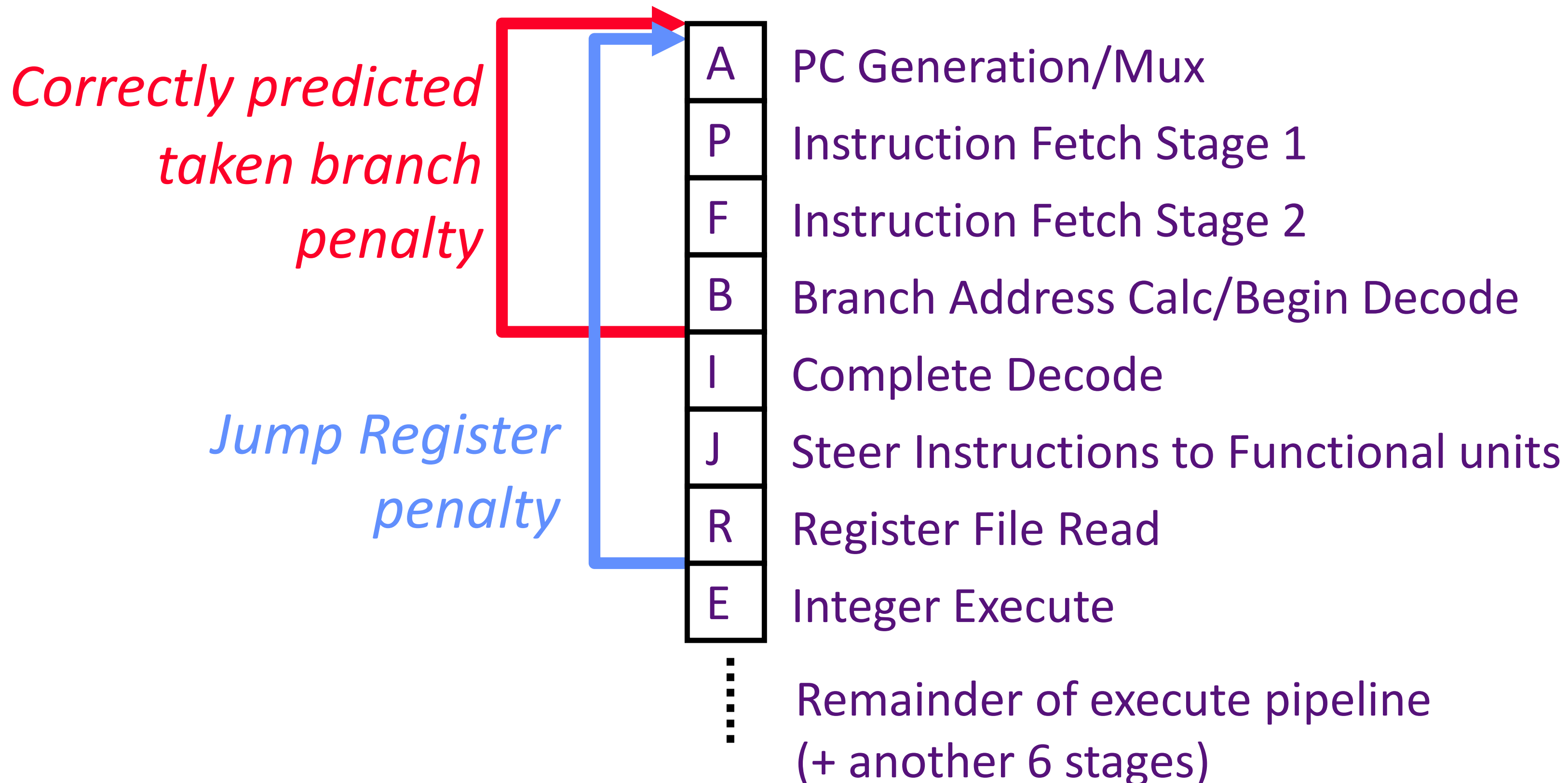
- An alternative to branch prediction is to execute both directions of a branch speculatively
  - resource requirement is proportional to the number of concurrent speculative executions
  - only half the resources engage in useful work when both directions of a branch are executed speculatively
  - branch prediction takes less resources than speculative execution of both paths
- With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction!
  - What would you choose with 80% accuracy?

Are We Missing Something?

- Knowing whether a branch is taken or not is great, but what else do we need to know about it?

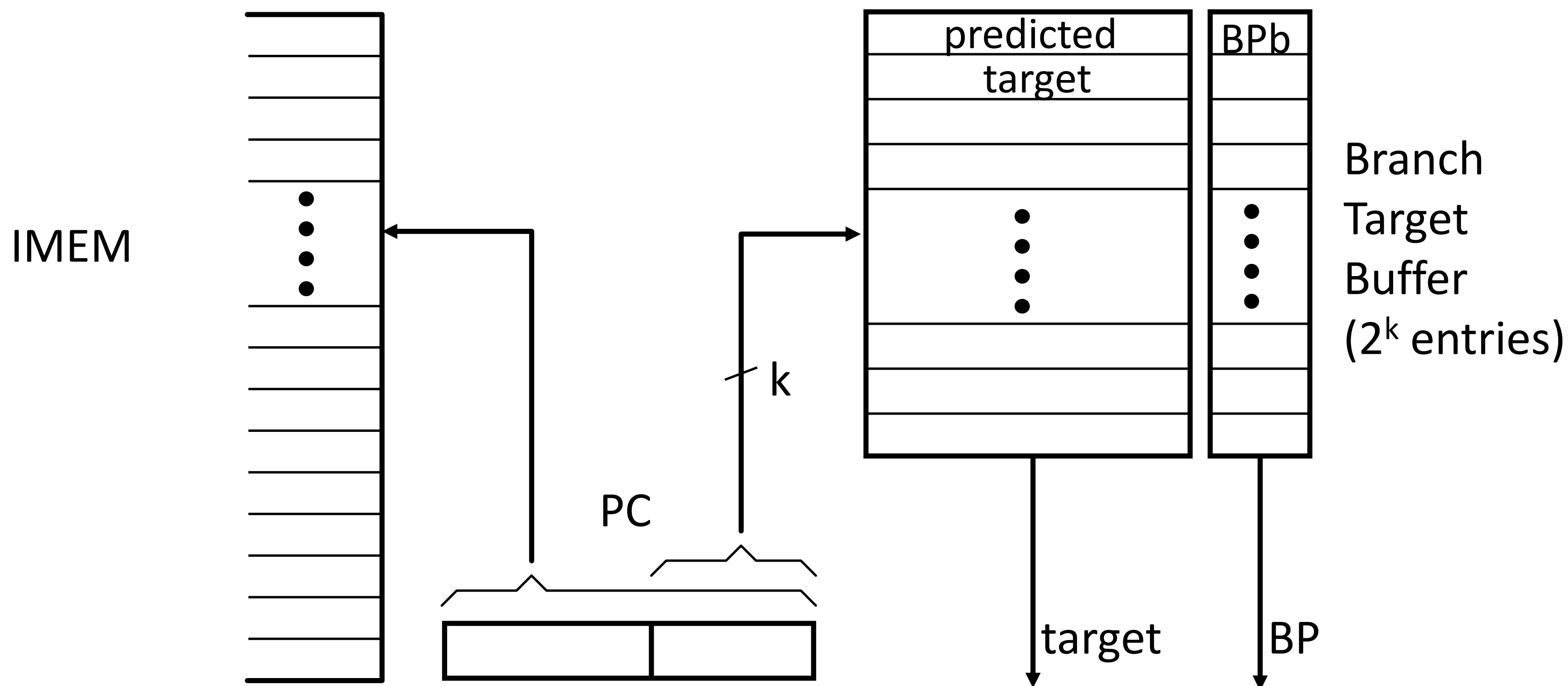
Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



UltraSPARC-III fetch pipeline

Branch Target Buffer

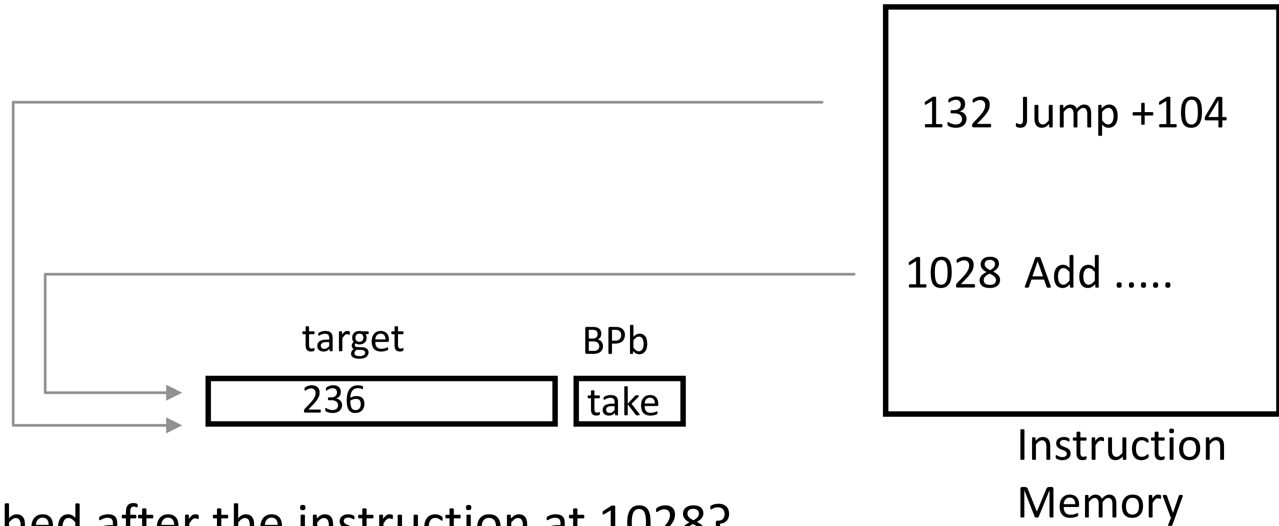


BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
Later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Address Collisions (Mis-Prediction)

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

=> *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?

BTB is only for Control Instructions

- Is even branch prediction fast enough to avoid bubbles?
- When do we index the BTB?
  - i.e., what state is the branch in, in order to avoid bubbles?

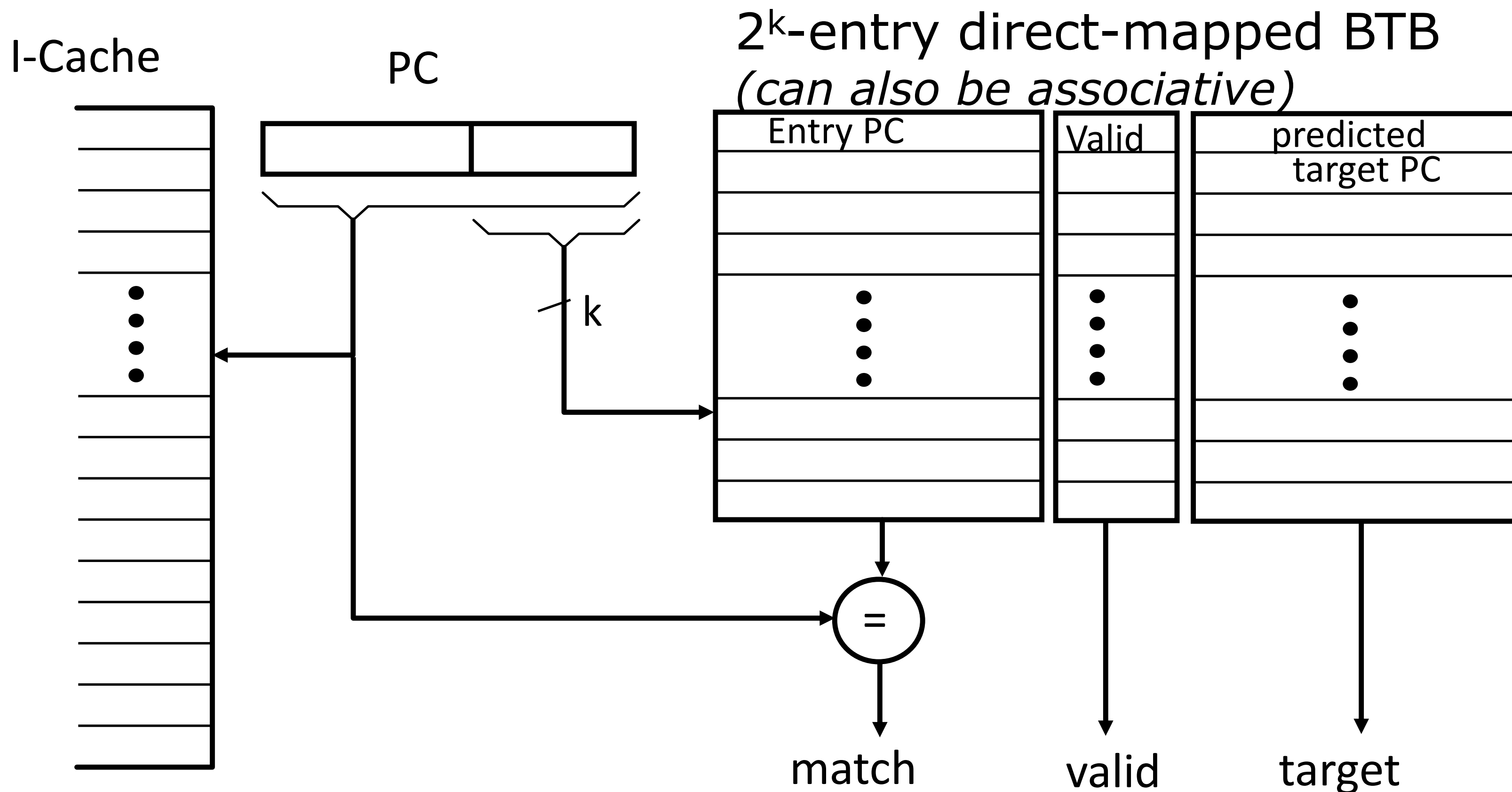
- BTB contains useful information for branch and jump instructions only

  => 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?*

Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

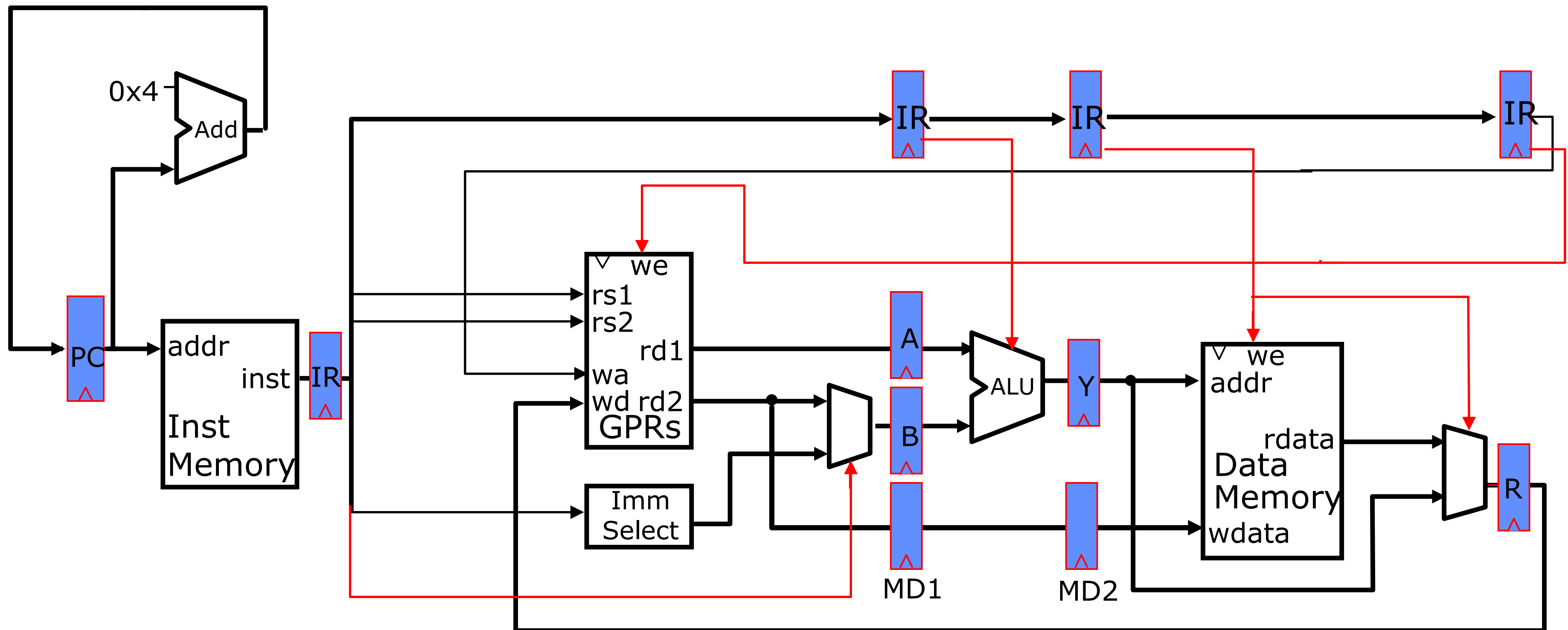
Are We Extending Cycle Time?

- By making instruction fetch more complicated?

- Thankfully no because the BTB can be accessed *in parallel* with the BTB

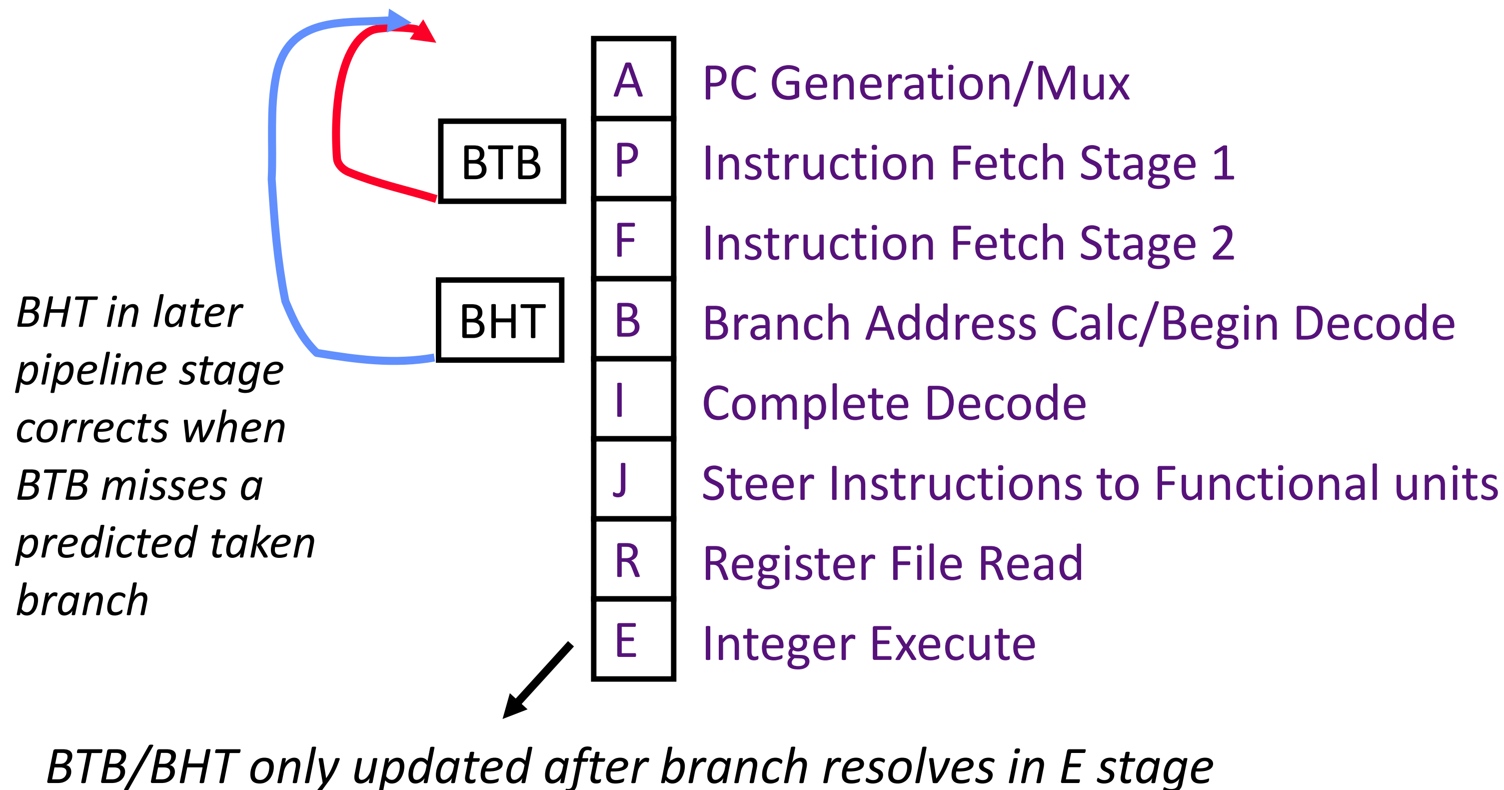
Are We Missing Something? (2)

- When do we update the BTB or BHT?



Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



BTB/BHT only updated after branch resolves in E stage

Uses of Jump Register (JR)

- 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.

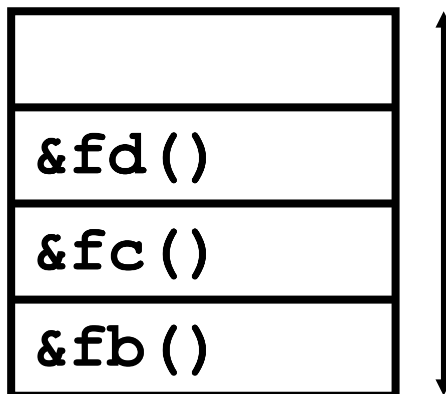
```
fa() { fb(); }

fb() { fc(); }

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

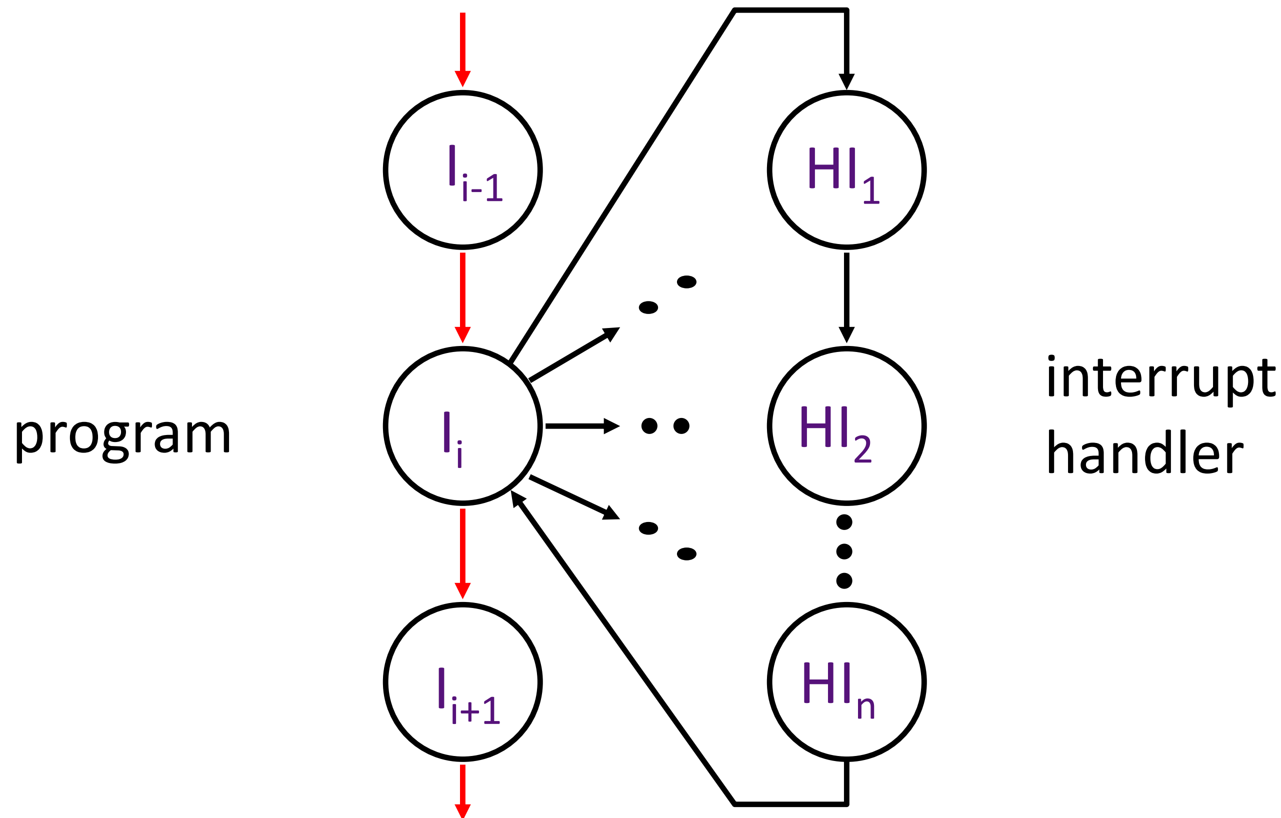
Push call address when function call executed

Pop return address when subroutine return decoded



k entries (typically k=8-16)

Interrupts:
altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

Causes of Interrupts

Interrupt: an *event* that requests the attention of the processor

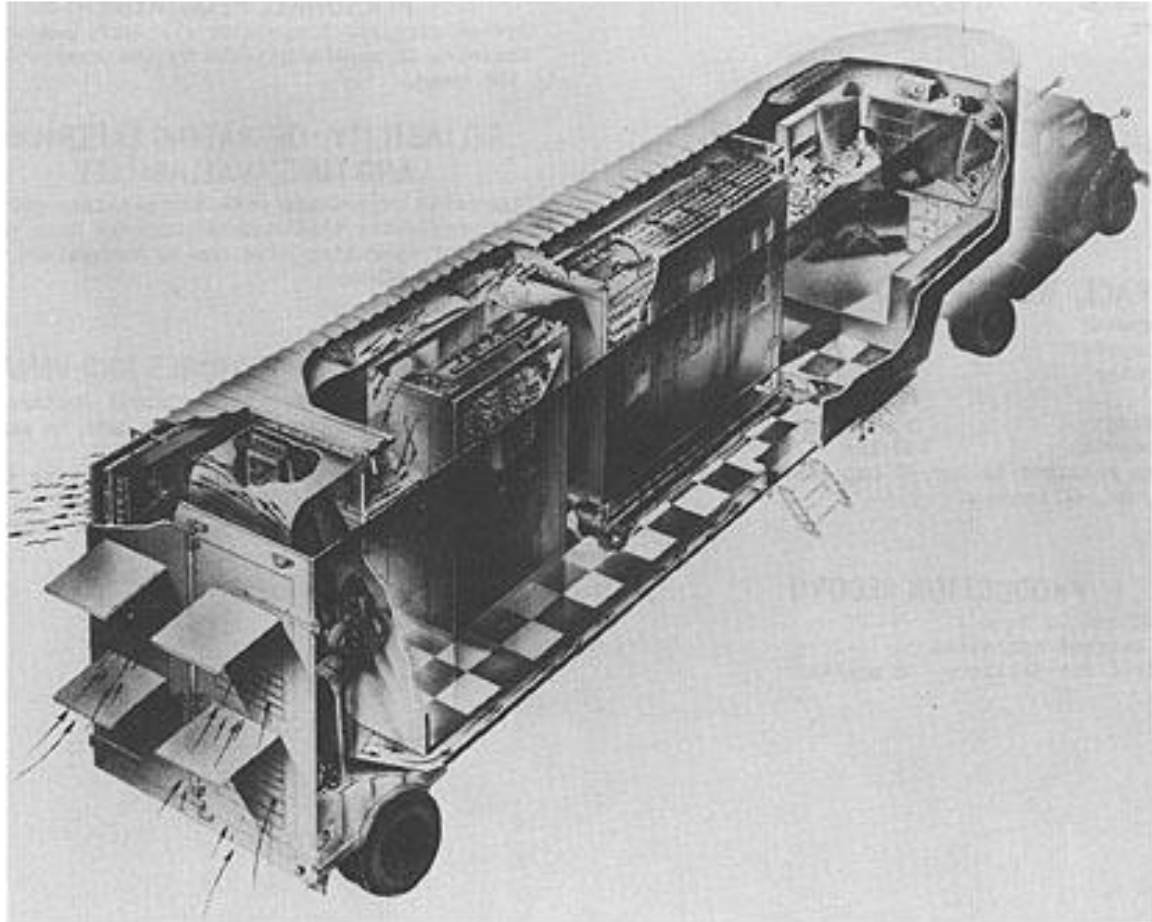
- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a. traps or exceptions)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults,
    TLB misses, protection violations
  - system calls, e.g., jumps into kernel

History of Exception Handling

- First system with exceptions was Univac-I, 1951
  – Arithmetic overflow would either
    • 1. trigger the execution a two-instruction fix-up routine at address 0, or
    • 2. at the programmer's option, cause the computer to stop
  – Later Univac 1103, 1955, modified to add external interrupts
    • Used to gather real-time wind tunnel data
- First system with I/O interrupts was DYSEAC, 1954
  – Had two program counters, and I/O signal caused switch between two PCs
  – Also, first system with DMA (direct memory access by I/O device)

[Courtesy Mark Smotherman]

DYSEAC, first mobile computer!



- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

[Courtesy Mark Smotherman]

Asynchronous Interrupts:
invoking the interrupt handler

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} (*precise interrupt)*
  - It saves the PC of instruction I_i in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

Precise and Imprecise

- Precise interrupts preserve the model that instructions execute in program-generated order

- Imprecise interrupts (and exceptions) do not

I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	300	ADD

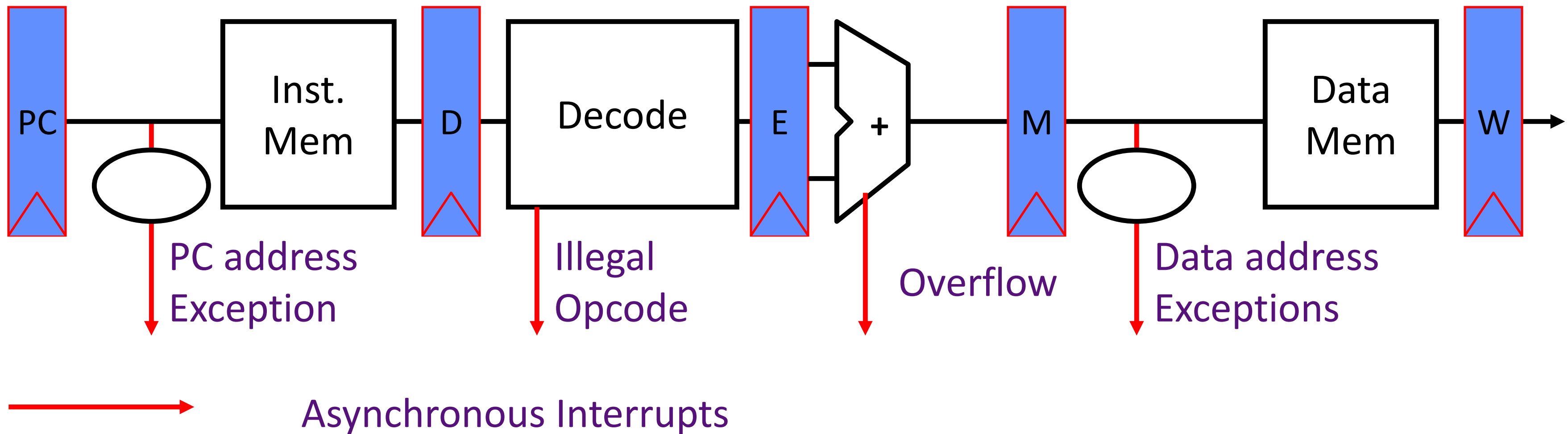
Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Interrupts

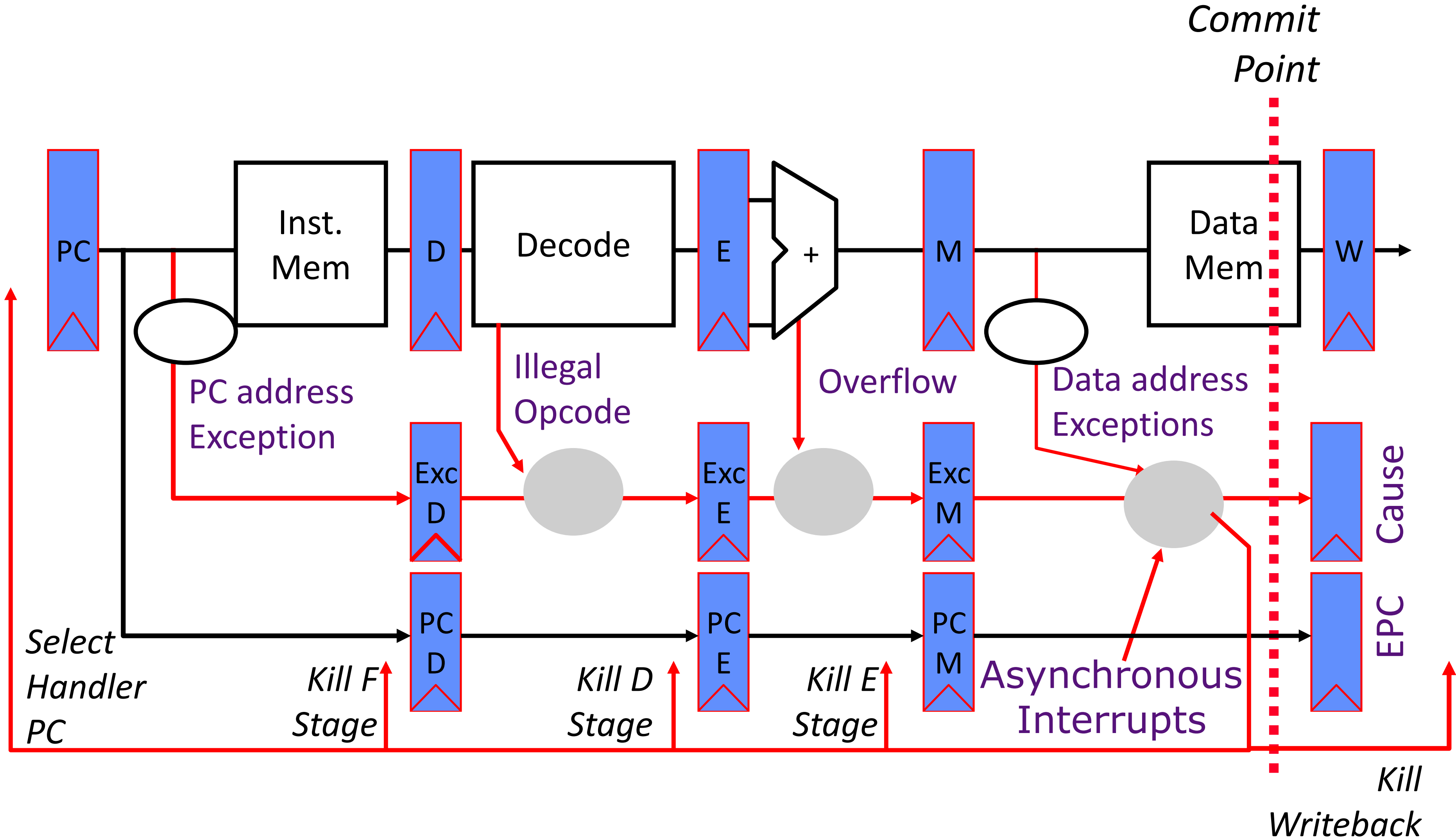
- A synchronous interrupt (exception) is caused by a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call trap, the instruction is considered to have been completed
  - a special jump instruction involving a change to privileged kernel mode

Exception Handling 5-Stage Pipeline



- When do we stop the pipeline for precise interrupts or exceptions?
- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception Handling 5-Stage Pipeline



Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point for instructions that will be killed (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

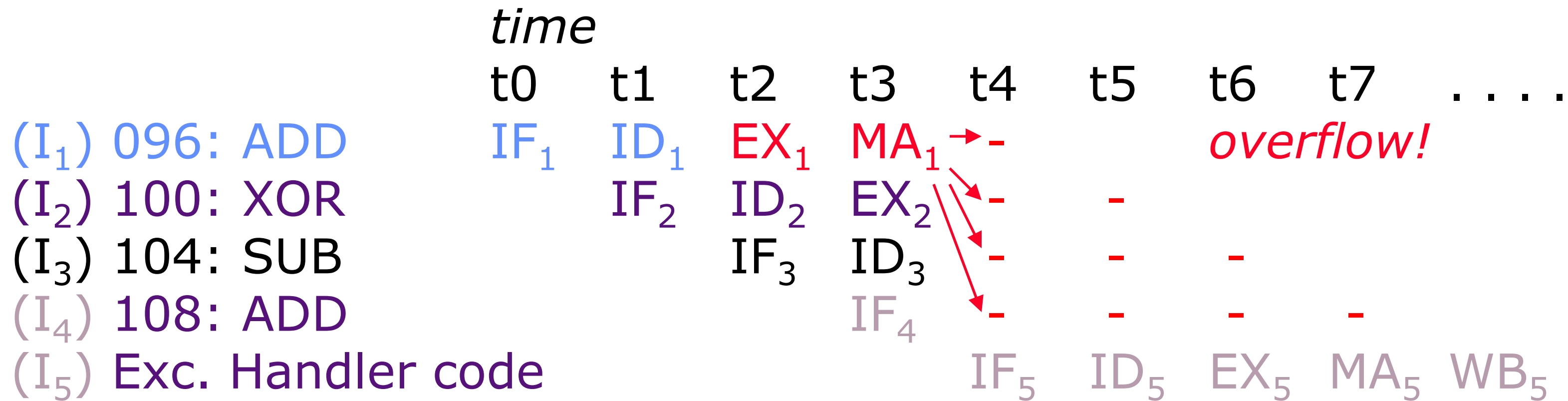
Speculating on Exceptions

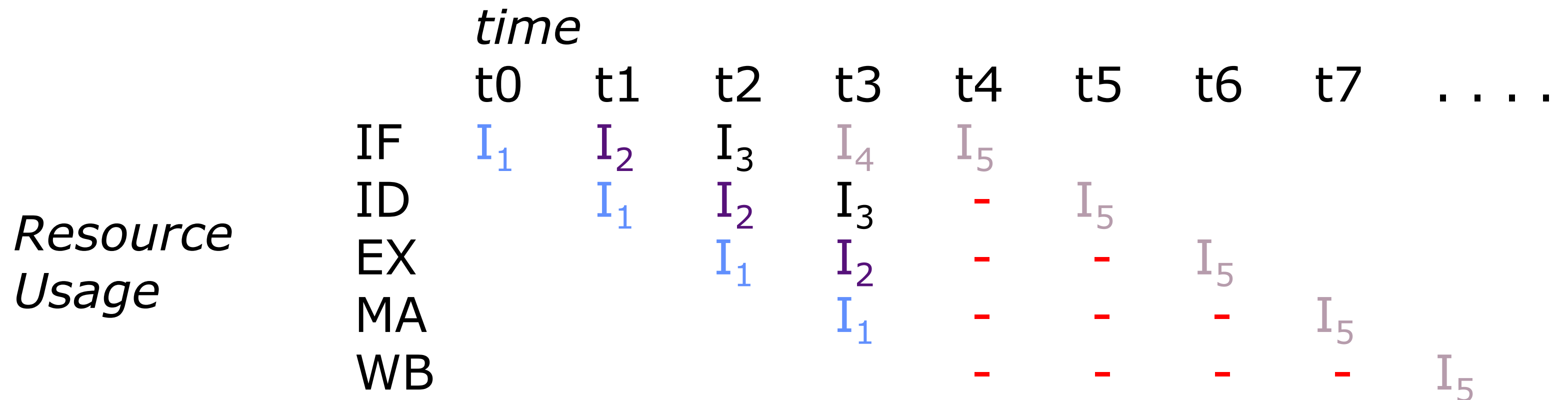
- Prediction mechanism
  - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline

What About Data Forwarding?

- Bypassing allows use of uncommitted instruction results by following instructions

Exception Pipeline Diagram





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)

- MIT material derived from course 6.823
- UCB material derived from course CS252