



Lecture 7: Branch Predictor

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University



Acknowledgements and Disclaimer

- Slides were developed in the reference with
 - CS 61C at UC Berkeley
 - <https://inst.eecs.berkeley.edu/~cs61c/sp23/>
 - CIS510 at Upenn
 - <https://www.cis.upenn.edu/~cis5710/spring2019/>
 - IF3 at the University of Edinburgh
 - https://www.inf.ed.ac.uk/teaching/courses/car/Notes/2017-18/lecture05-handling_hazards.pdf



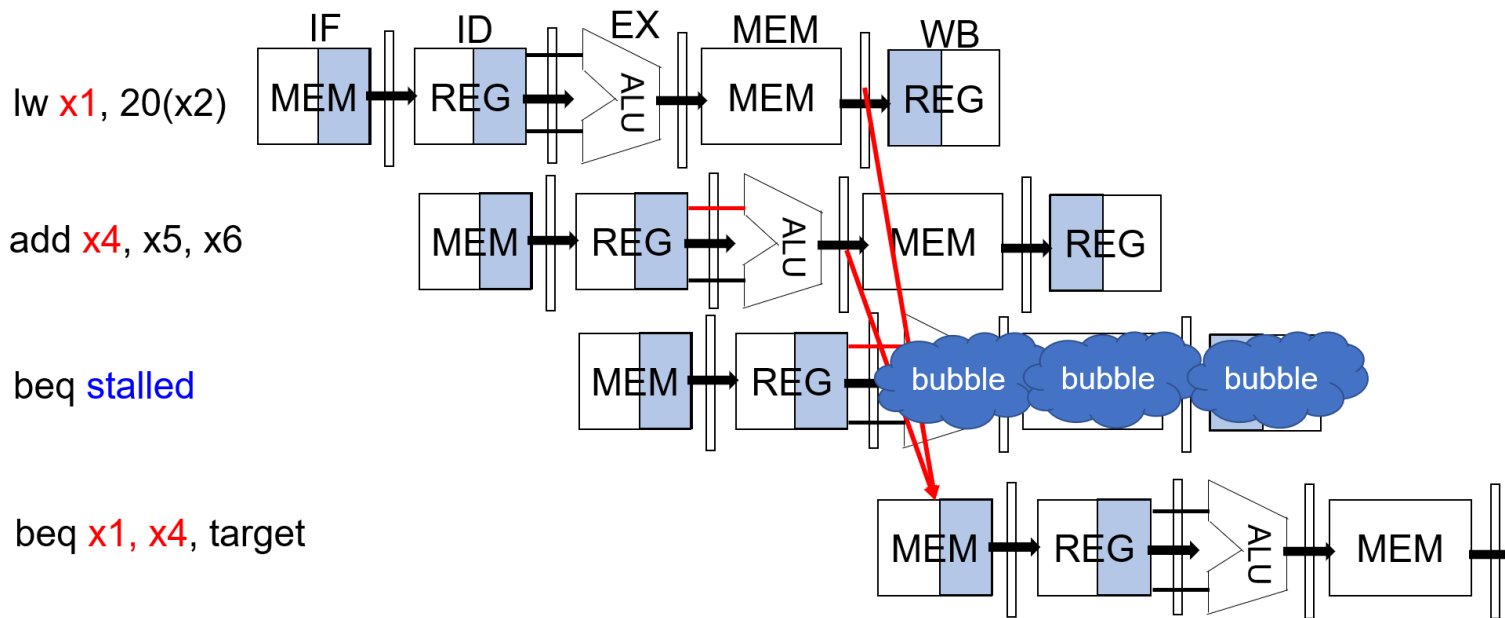
Outline

- Branch Predictor
- Branch Target Buffer
- Bimodal Branch Predictor
- Gshare History-Based Predictor
- Tournament Predictor
- Predication



Data Hazards for Branches

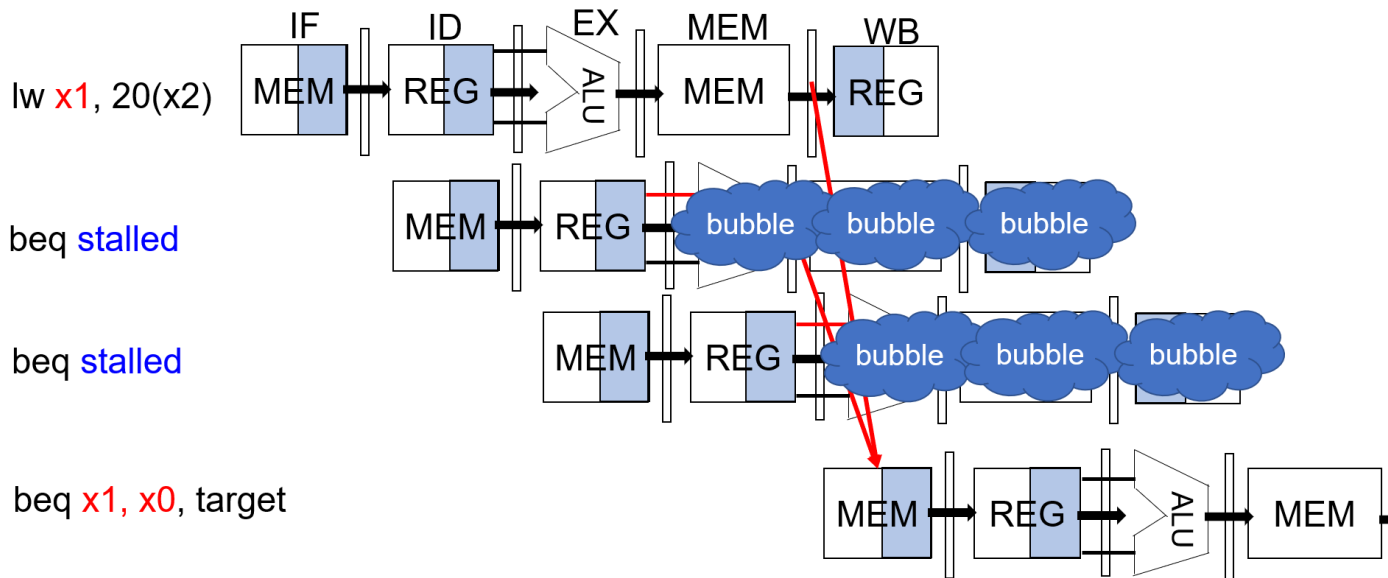
- If a comparison register is a destination of 2nd or 3rd preceding load instruction -> **Need 1 stall cycle**





Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction -> **Need 2 stall cycles**

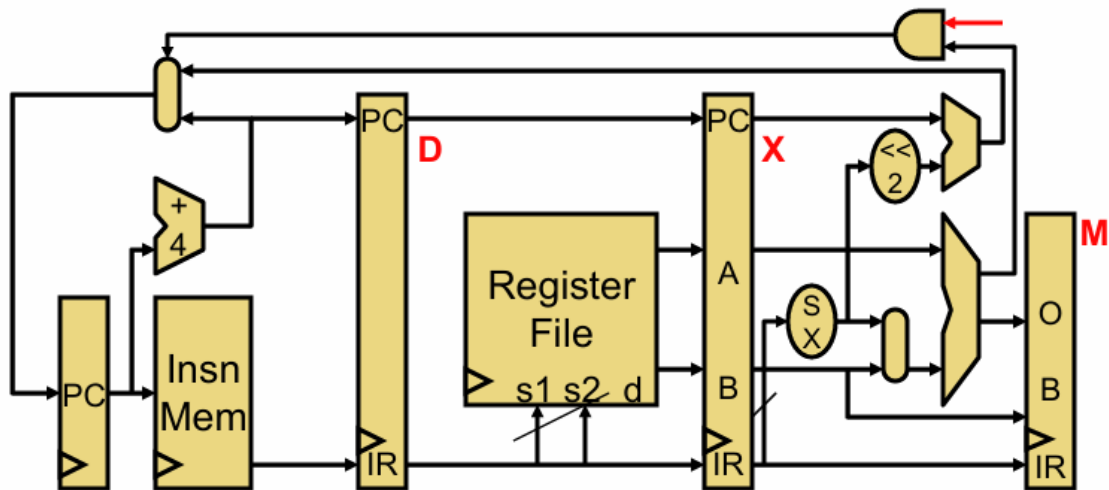




Branch Prediction

- **Branch speculation**

- Could just stall to wait for branch outcome (two-cycle penalty)
- Fetch past branch instructions before branch outcome is known
 - Default: assume **“not-taken”** (at fetch stage, can't tell it's a branch)





Branch Prediction

- **Speculative execution**

- Execute before all parameters known with certainty
- If speculation is **“correct”**
 - + Avoid stall, improve performance
- If speculation is **“incorrect”** (mis-speculation)
 - Must abort/flush/squash incorrect instructions
 - Must undo incorrect changes (recover pre-speculation state)



Branch Prediction

- **Control speculation mechanics**
 - Guess branch target, start fetching at guessed position
 - Doing nothing is implicitly guessing target is PC+4
 - Can actively guess other targets: **dynamic branch prediction**
 - Execute branch to verify guess
 - Correct speculation? Keep going
 - Mis-speculation? Flush mis-speculated instructions
 - Hopefully haven't modified permanent state (Regfile, DMem)
- + Happens naturally in in-order 5-stage pipeline



Branch Prediction

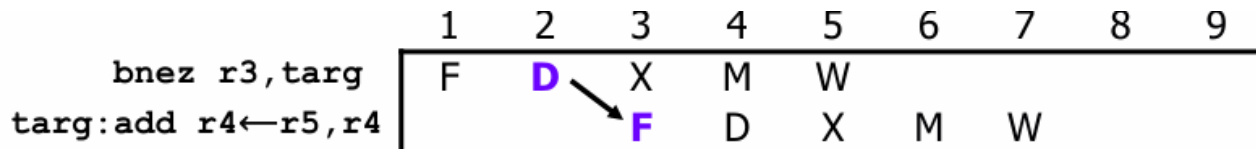
- **When to perform branch prediction?**

- Option #1: During decode

- Look at instruction opcode to determine branch instructions
- Can calculate next PC from instruction (for PC-relative branches)
- One cycle “mis-fetch” penalty even if branch predictor is correct

- Option #2: During fetch?

- How do we do that?
 - Branch predictor





Branch Recovery

- **Branch recovery**

- What to do when branch is actually taken
 - **Flush them**, i.e., replace them with **nops**
 - + They haven't written permanent state yet (regfile, DMem)
 - - Two cycle penalty for taken branches

Correct:

	1	2	3	4	5	6	7	8	9
<code>addi r3←r1,1</code>	F	D	X	M	W				
<code>bnez r3,targ</code>		F	D	X	M	W			
<code>st r6→[r7+4]</code>			F	D	X	M	W		
<code>mul r10←r8,r9</code>				F	D	X	M	W	

speculative



Branch Recovery

• Mis-speculation recovery

○ What to do on wrong guess

- Branch resolves in X (EXEC.) stage
- + Younger insts (in F, D) haven't changed permanent state
- **Flush** instructions currently in D and X (i.e, replace with **nops**)

	1	2	3	4	5	6	7	8	9
Recovery: <code>addi r3←r1,1</code>	F	D	X	M	W				
<code>bnez r3,targ</code>		F	D	X	M	W			
<code>st r6→[r7+4]</code>			F	D	--	--	--		
<code>mul r10←r8,r9</code>				F	--	--	--	--	
<code>targ:add r4←r4,r5</code>					F	D	X	M	W



Takeaway Questions

- Assume that
 - **Branch: 20%**, load: 20%, store: 10%, other: 50%
 - Say, 75% of branches are taken
 - What is the CPI?



Takeaway Questions

- Assume that
 - **Branch: 20%**, load: 20%, store: 10%, other: 50%
 - Say, 75% of branches are taken
 - What is the CPI?
 - **$CPI = 1 + 20\% * 75\% * 2 = 1.3$**
 - **Branches cause 30% slowdown**
 - Worse with deeper pipelines, why?
 - Can we do better than assuming branch is not taken?



Dynamic Branch Prediction

- Motivation
 - In deeper and superscalar pipelines, branch penalty is more significant
 - The branch predictor primarily operates during the Instruction Fetch (IF) stage of the processor pipeline
- Monitor branch behavior and learn
 - Key assumption: past behavior indicative of future behavior
- Predict the present branch using learning history
 - Identify individual branches by their PC or dynamic branch history



Dynamic Branch Prediction

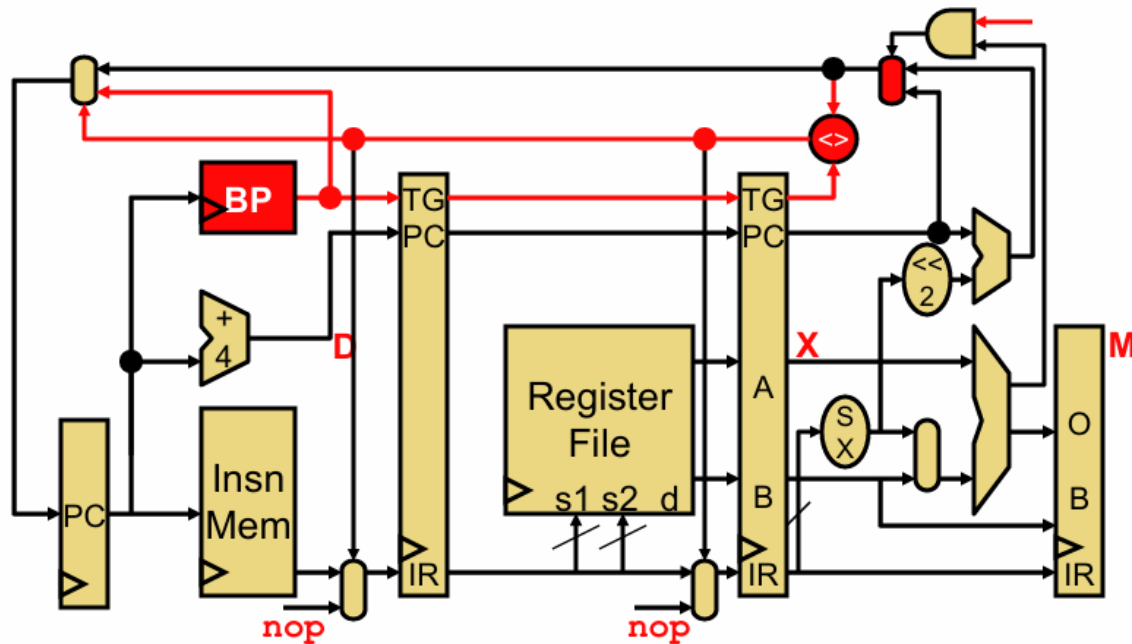
- Fetch (IF) stage
 - Branch predictor identifies if an instruction is a branch
 - Predict its target address to immediately fill the pipeline
- Decode (ID) stage
 - Branch testing and target address calculation
- Execute (EX) stage
 - The actual branch decision and target address are calculated
 - Allowing the processor to verify if the prediction made in the IF stage was correct



Dynamic Branch Prediction

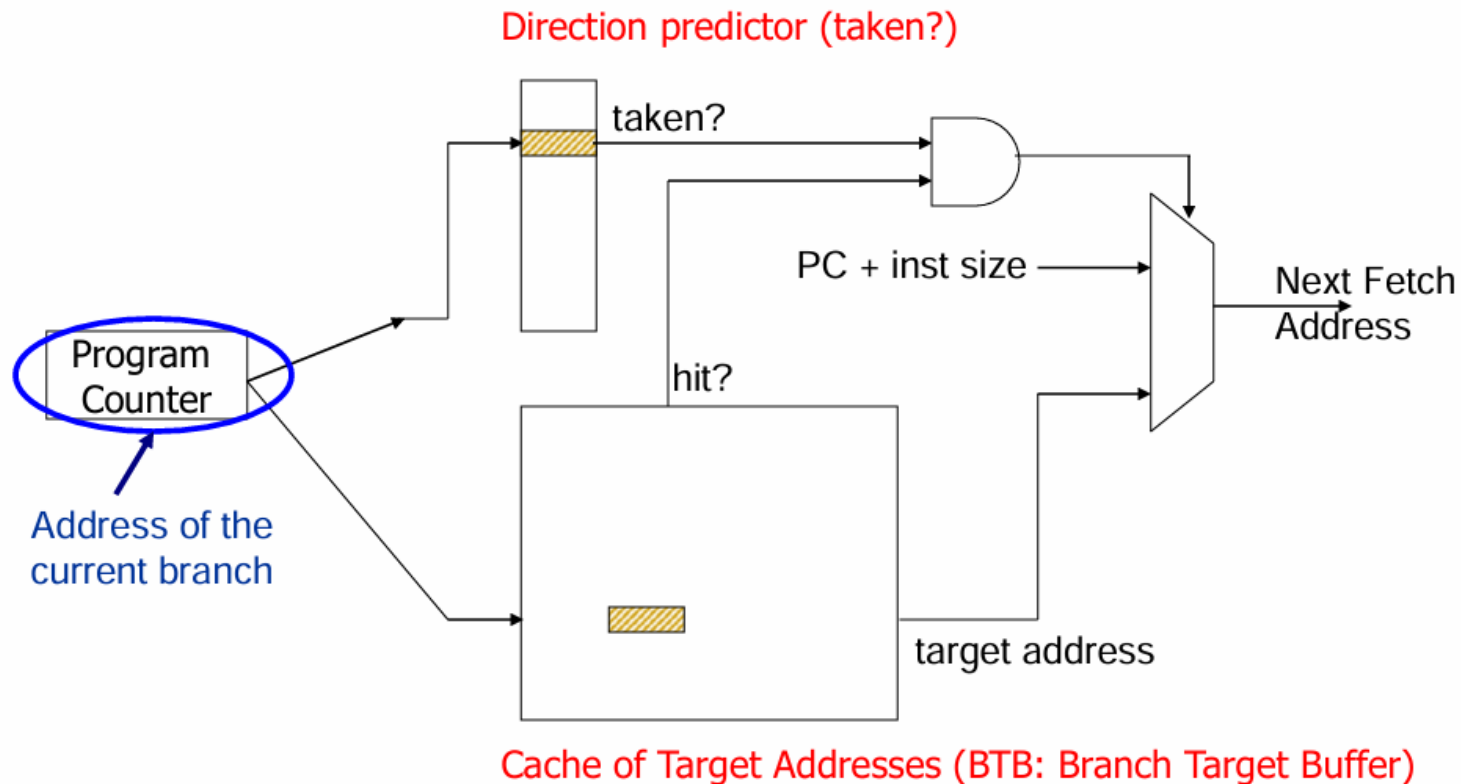
- **Dynamic branch prediction**

- Hardware guesses outcome
- Start fetching from guessed address
- Flush on **mis-prediction**





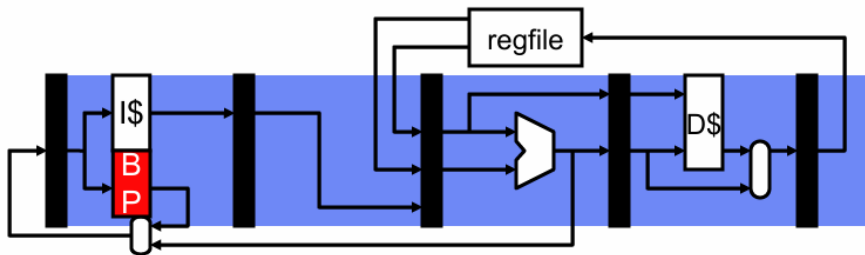
Dynamic Branch Prediction





Dynamic Branch Prediction

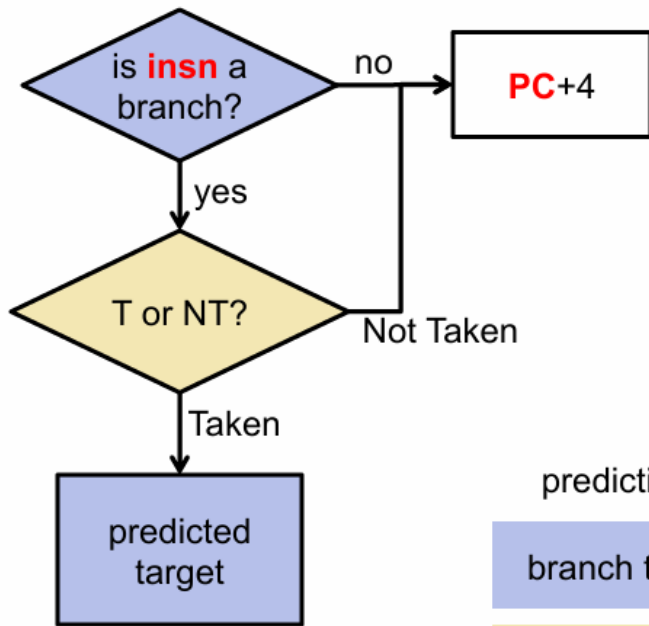
- **Dynamic Branch Prediction components**
 - Step #1: is it a branch?
 - Easy after decode ...
 - Step #2: is the branch taken or not taken?
 - Direction predictor (applies to conditional branch only)
 - Predict taken/not-taken
 - Step #3: if the branch is taken, where does it go?





Dynamic Branch Prediction

- **Branch Prediction steps**



- Which **insn**'s behavior are we trying to predict?
- Where does **PC** come from?

prediction source:

branch target buffer

direction predictor



Branch Target Buffer

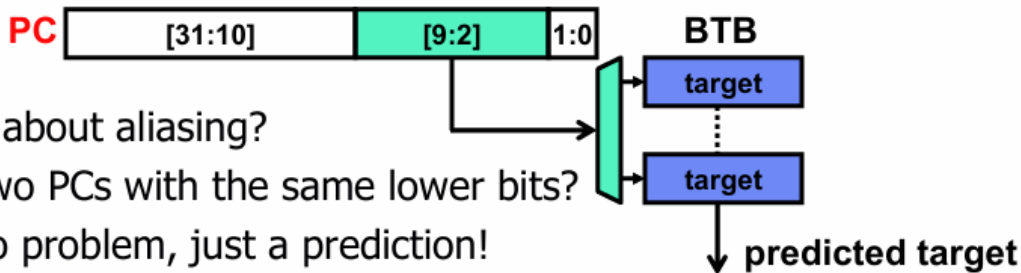
- Branch predictor shortcoming
 - Tell whether the branch will be taken or not
 - Say nothing about the target of the branch
 - To resolve a branch early we need to know both the outcome and the target
- Branch Target Buffer (BTB)
 - Store the likely target of the branch in a table (cache) indexed by the branch PC
 - BTB is accessed in the IF stage
 - Branch predictor is accessed later in the ID stage



Branch Target Buffer

- **Branch Target Buffer (BTB)**

- Learn from past, predict the future, record the past in a hardware
- Guess the future PC based on past behavior
- E.g. “Last time the branch X was taken, it went to address Y”
 - So, in the future, if address X is fetched, fetch address Y next
 - PC indexes table of bits target addresses
 - Branch will go to same place it went last time



- What about aliasing?
 - Two PCs with the same lower bits?
 - No problem, just a prediction!

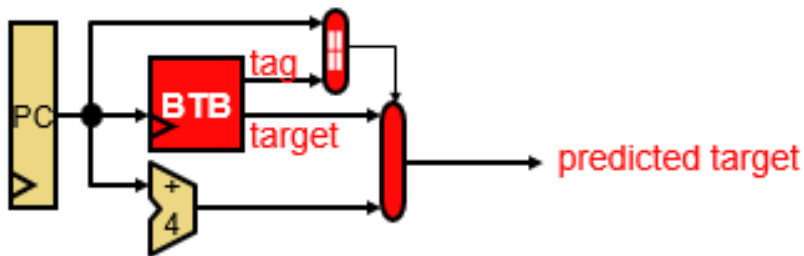


Branch Target Buffer

Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm

- **Branch Target Buffer (BTB)**

- At Fetch, how does insn know it's a branch and read BTB?
 - All insns access BTB in parallel with Imm Fetch
- BTB is used to predict which insn are branches
 - Implement by “tagging” each entry with its corresponding PC
 - Update BTB on every taken branch insn, record target PC
 - BTB[PC].tag = PC, BTB[PC].target = target of branch



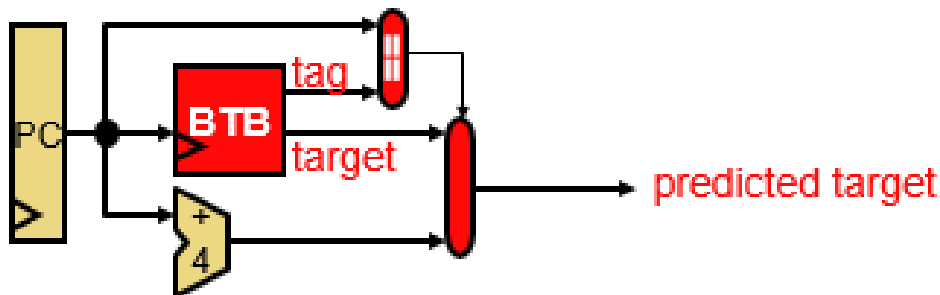


Branch Target Buffer

- **Branch Target Buffer (BTB)**

- All instructions access at Fetch in parallel with Imm
 - Check for tag match, signifies insn at that PC is a branch
 - Predict PC = (BTB[PC].tag == PC) ? BTB[PC].target: PC + 4

Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm





Branch Target Buffer

- **Why does a BTB work?**

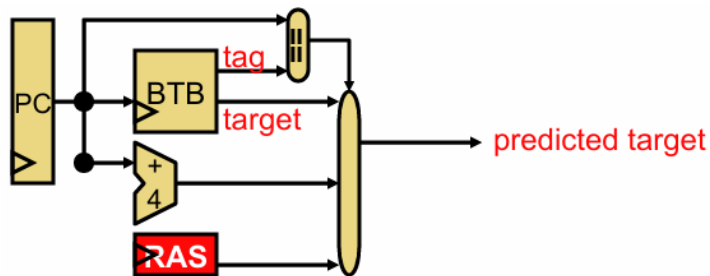
- Because most control instructions use **direct targets**
 - Target encoded in insn itself -> same “taken” target every time
- What about indirect targets?
 - Target held in a register -> can be different each time
 - Two indirect call idioms
 - Dynamic linked functions (DLLs): target always the same
 - Dynamic dispatched (virtual) functions: hard but uncommon
 - Two indirect unconditional jump idioms
 - Switches: hard but uncommon
 - Function returns: hard and common but ...



Branch Target Buffer

- **Return Address Stack (RAS)**

- Call instruction? $RAS[TopOfStack++] = PC+4$
- Return instruction? Predicted-target = $RAS[--TopOfStack]$
- How can we tell if an insn is a call/return before decoding it?
 - Access RAS on every insn BTB-style doesn't work
 - Another predictor (or put them in BTB marked as "return")
 - Or, **pre-decode bits** in insn mem, written when first executed

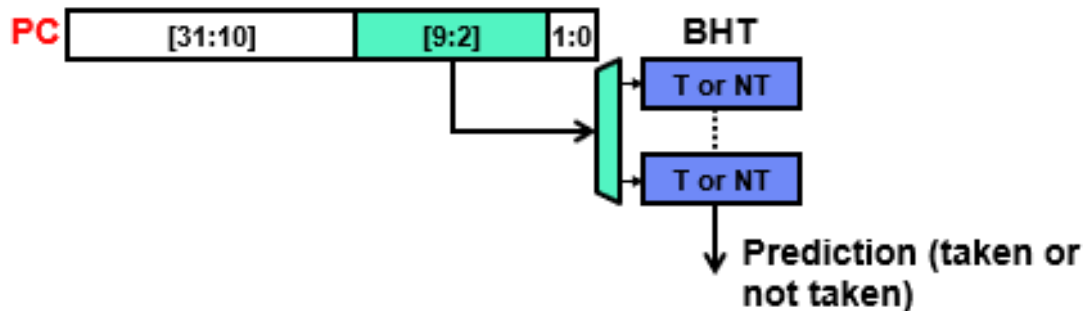




1-bit Branch Predictor

- **1-bit predictor: simplest predictor**

- PC indexes branch history table of bits (0 = N, 1 = T), no tags
- Essentially: branch will go same way it went last time





1-bit Branch Predictor

- 1 bit indicating Taken (1) or Not Taken (0)
- Branch prediction buffers
 - Match branch PC during IF or ID stage

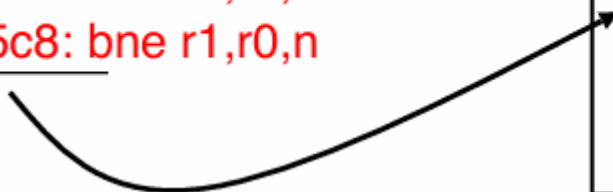
...

0x135c4: add r1,r2,r3

0x135c8: bne r1,r0,n

...

Branch PC	Outcome
0x135c8	0
0x147e0	1
...	...





1-bit Branch Predictor

- **1-bit direction predictor**

- Problem: **inner loop branch** below

```
for (i=0;i<100;i++)  
  for (j=0;j<3;j++)  
    // whatever
```

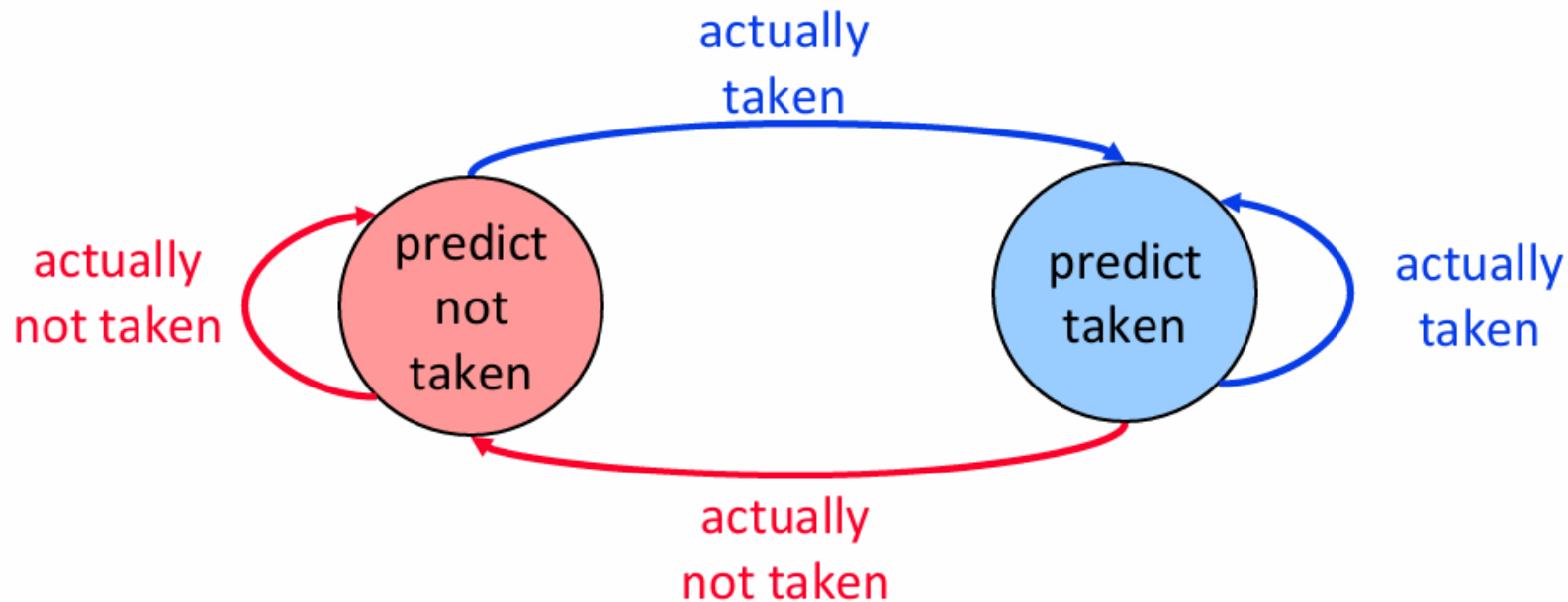
- The outcome is based on previous correct prediction
- Two “built-in” mis-predictions per inner loop iteration
- Branch predictor “changes its mind too quickly”

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	T	T	T	Correct
3	T	T	T	Correct
4	T	T	N	Wrong
5	N	N	T	Wrong
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	N	N	T	Wrong
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

6 incorrect prediction Result?



1-bit Branch Predictor FSM





1-bit Branch Predictor

- 1 bit indicating Taken (1) or Not Taken (0)
- Incur at least 2 mis-predictions per loop
 - **Problem: “unstable” behavior**

```
int  gtz = 0;
int  i   = 0;

while (i < 100) {
    x = a[i];
    if (x == 0)
        continue;
    gtz ++;
}
```



1-bit Branch Predictor

- **Problems**

- A last-time predictor changes its prediction from T->NT or NT->T too quickly
- Even though the branch may be mostly taken or mostly not taken

- **Solutions**

- Add hysteresis to the predictor so that prediction does not change one a single different outcome
- Use two bits to track the history of predictions for a branch instead of a single bit
- Can have 2 states for T or NT instead of 1 state for each



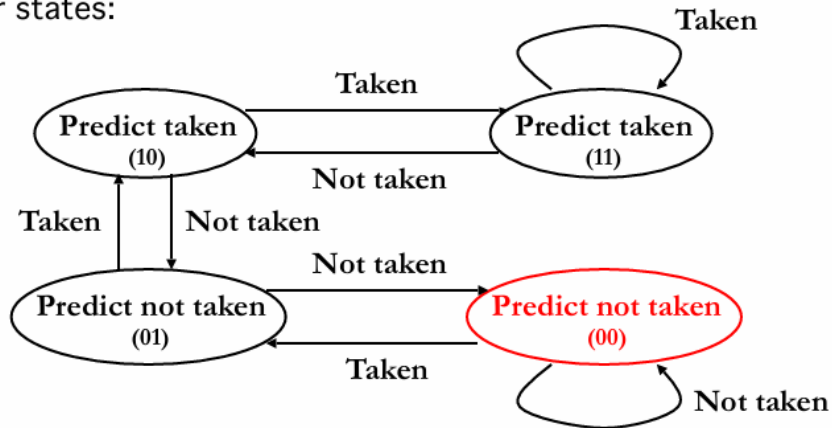
Bimodal Branch Predictor

- **Learn from past, predict the future**
 - Record the past in a hardware structure
- **Direction predictor (DIRP)**
 - Map conditional-branch PC to taken/not-taken (T/N) decision
 - Individual conditional branches often biased or weakly biased
 - 90%+ one way or the other considered “biased”
 - Why? Look back edges, checking for uncommon conditions



2-bit (Bimodal) Branch Predictor

- Predictor states:



- Learns biased branches
- N-bit predictor
 - Increment on Taken outcome and decrement on Not Taken outcome
 - If counter $> (2^{n-1}) / 2$ then taken, otherwise do not take
 - Takes longer to learn, but sticks longer to the prediction



2-bit (Bimodal) Branch Predictor

- **Two-bit saturating counters (2bc)**

- Replace each single-bit prediction
 - (0, 1, 2, 3) = (N, n, t, T)
- Adds “hysteresis”
 - Force predictor to mis-predict twice before “changing its mind”
 - One mis-predict each loop execution (rather than two)
 - + Fixes this pathology (which is not contrived)

5 incorrect prediction

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	n	N	T	Wrong
3	t	T	T	Correct
4	T	T	N	Wrong
5	t	T	T	Correct
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	t	T	T	Correct
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong



2-bit (Bimodal) Branch Predictor

Nested loop:

```
Loop1: ...  
    ...  
Loop2: ...  
    bne r1,r0,loop2  
    ...  
    bne r2,r0,loop1
```



- 1st outer loop execution

- 00 -> predict not taken; actually taken -> update to 01 (misprediction)
- 01 -> predict not taken; actually taken -> update to 10 (misprediction)
- 10 -> predict taken; actually taken -> update to 11
- 11 -> predict taken; actually taken
- ...
- 11 -> predict taken; actually not taken -> update to 10 (misprediction)



2-bit (Bimodal) Branch Predictor

Nested loop:

```
Loop1: ...  
    ...  
Loop2: ...  
    bne r1,r0,loop2  
    ...  
    bne r2,r0,loop1
```



- 2nd inner loop execution
 - 10 -> predict taken; actually taken -> update to 11
 - 11 -> predict taken; actually taken
 - ...
 - 11 -> predict taken; actually not taken -> update to 10 (misprediction)
- 2-bit branch predictors work very well for loop-intensive applications, why?



2-bit (Bimodal) Branch Predictor

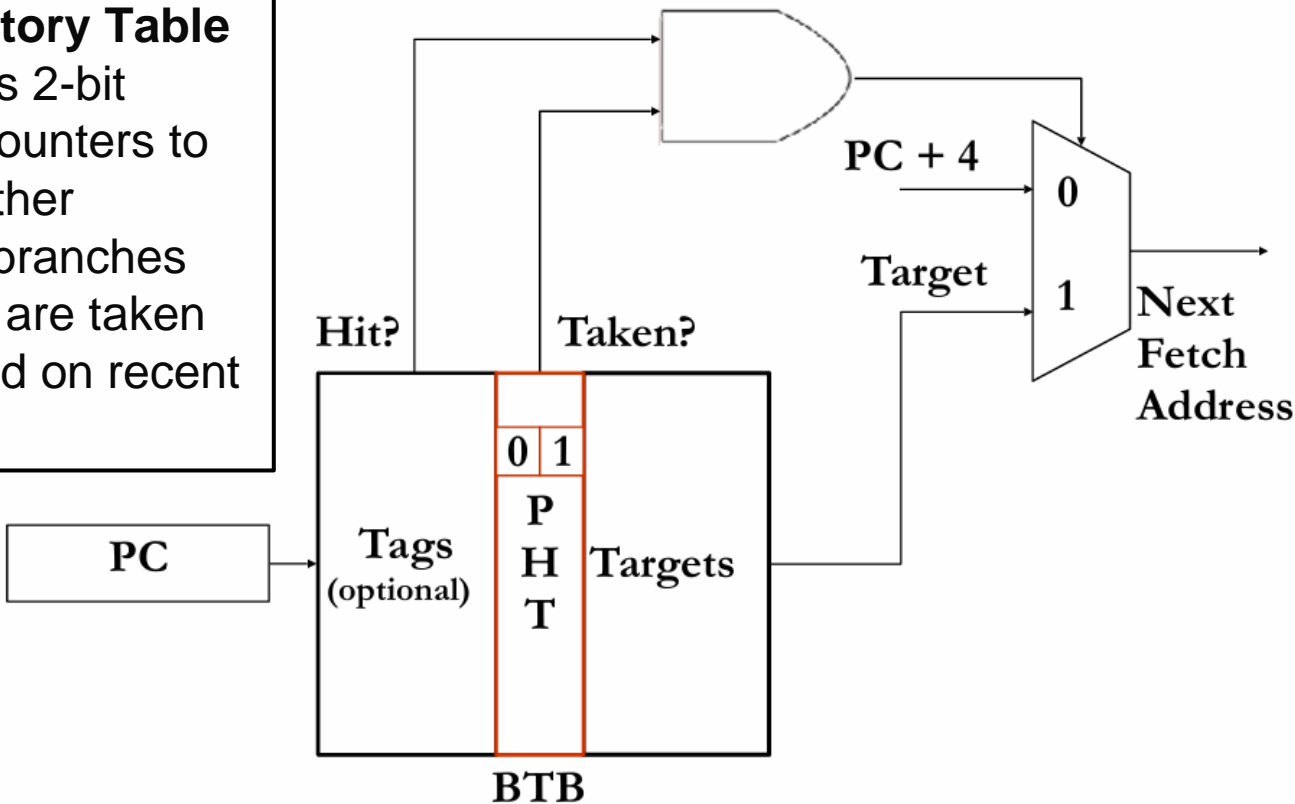
- **2-bit branch predictor**

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome
- Accuracy for a loop with N iterations = $(N-1)/N$
- TNTNTNTNTNTNTNTNTN -> 50% accuracy (assuming counter initialized to weakly taken)
- + Better prediction accuracy
- - More hardware cost (but counter can be part of a BTB entry)



2-bit (Bimodal) Branch Predictor

Pattern History Table (PHT) stores 2-bit saturating counters to predict whether conditional branches (like if-else) are taken or not, based on recent history





Correlating Predictors

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: branches are correlated!
 - Local:
 - A branch outcome maybe correlated with past outcomes of the same branch
 - Global:
 - A branch outcome maybe correlated with past outcomes of other branches



Correlating Predictors

- **Branches may be correlated**

- Consider:

```
for (i=0; i<1000000; i++) { // Highly biased
  if (i % 3 == 0) { // Locally correlated
    ...
  }
  if (random() % 2 == 0) { // Unpredictable
    ...
  }
  if (i % 3 == 0) { // Globally correlated
    ...
  }
}
```



Correlating Predictors (Local)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: outcomes of same branch correlated!
 - Branch outcomes:
 - 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0 ...
 - **Idea: exploit recent history of same branch in prediction**

```
while (i < 4) {  
    x = a[i];  
    if (x == 0)  
        continue;  
    gtz ++;  
}
```



Correlating Predictors (Global)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: Different branches maybe correlated!

```
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    ...}
```

If both branches are taken,
the last branch definitely not taken

```
char s1 = "Bob"
...
if (s1 != NULL)
    reverse_str(s1);
reverse_str(char *s) {
    if (s1 == NULL)
        return;
    ...}
```

s1 definitely not Null
in this calling context



Correlating Predictors (Global)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: Different branches maybe correlated!

```
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    ...}
```

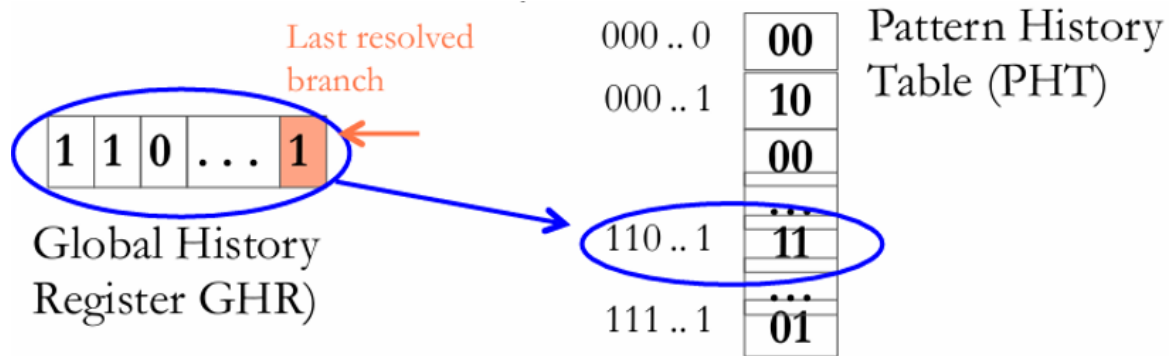
```
char s1 = "Bob"
...
if (s1 != NULL)
    reverse_str(s1);
    _____
reverse_str(char *s) {
    if (s1 == NULL)
        return;
    ...
```

Idea: exploit recent history of other branches in prediction



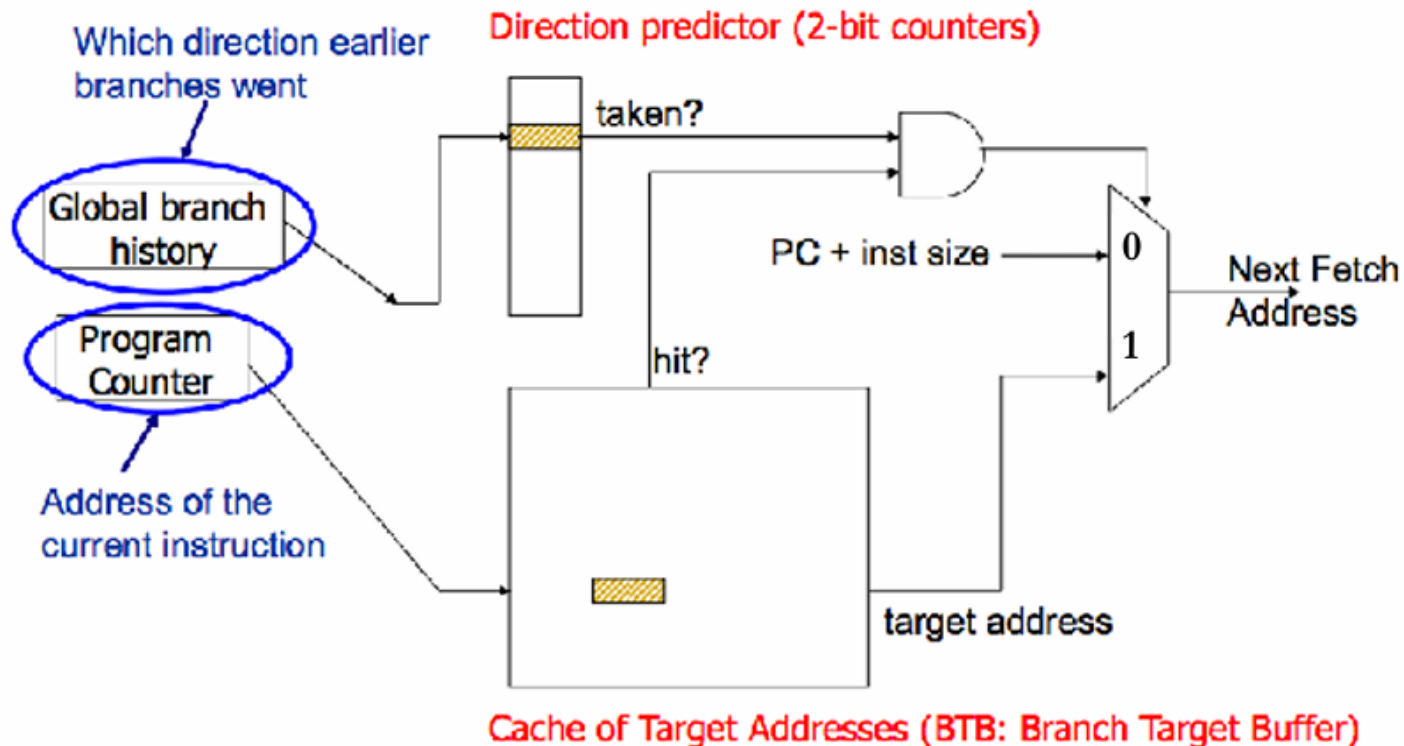
Global Two-level (Correlating) Predictor

- Prediction depends on the context of the branch
 - Context: history (T/NT) of the last N branches
 - First level of the predictor
 - Implemented as a shift register
- Prediction: 2-bit saturating counters
 - Indexed with the "global" history
 - Second level of the predictor





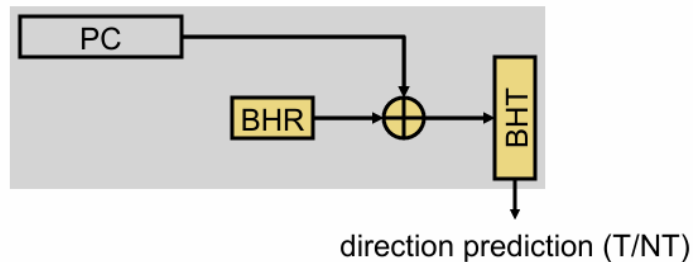
Global Two-level (Correlating) Predictor





Gshare History-Based Predictor

- Observation
 - branch outcomes are correlated
- Maintains recent branch outcomes in
 - **Branch History Register (BHR)**
 - In addition to BHT of counters (typically 2-bit saturate counters)
- How do we incorporate history into our predictions?
 - Use PC xor BHR to index into BHT
 - Input A: PC bits (the lower bits of the branch address)
 - Input B: The BHR stores a shift-register of the last N branch outcome
 - Logic: PC bits are XORed with BHR bits to create the index





Gshare History-Based Predictor

- Gshare working example
 - Assume program has one branch
 - BHT: one 1-bit DRIP entry
 - **3BHR**: last 3 branch outcomes
 - Train counter, **and** update BHR after each branch
 - The Gshare spreads out branch patterns more evenly across the table

Time	State	BHR	Prediction	Outcome	Result?
1	N	NNN	N	T	wrong
2	N	NNT	N	T	wrong
3	N	NTT	N	T	wrong
4	N	TTT	N	N	correct
5	N	TTN	N	T	wrong
6	N	TNT	N	T	wrong
7	T	NTT	T	T	correct
8	N	TTT	N	N	correct
9	T	TTN	T	T	correct
10	T	TNT	T	T	correct
11	T	NTT	T	T	correct
12	N	TTT	N	N	correct



Bimodal and Global 2-level

- Bimodal (2-bit) branch predictor
 - + Good for biased branches
 - + No interference
 - - Cannot discern patterns
- Global 2-level Branch Predictor
 - + Leverages correlated branches
 - + Identifies patterns
 - - Cannot always take advantage of biased branches
 - - Interference



Interference in Global 2-level Predictor

Pattern History Table

Global History Register



1	1	0
0	1	1
1	1	2
0	1	3
1	1	4
1	0	5
1	1	6
0	1	7

- Branch A is always Not Taken when GHR is 101
- Branch B is a loop with a million iterations
- Branch A and Branch B can interfere in entry 5 of the PHT

Interference (aliasing) happens when:

Two unrelated branches (different PCs) happen to produce the same XOR hash

Biased branches pollute the PHT!!!!

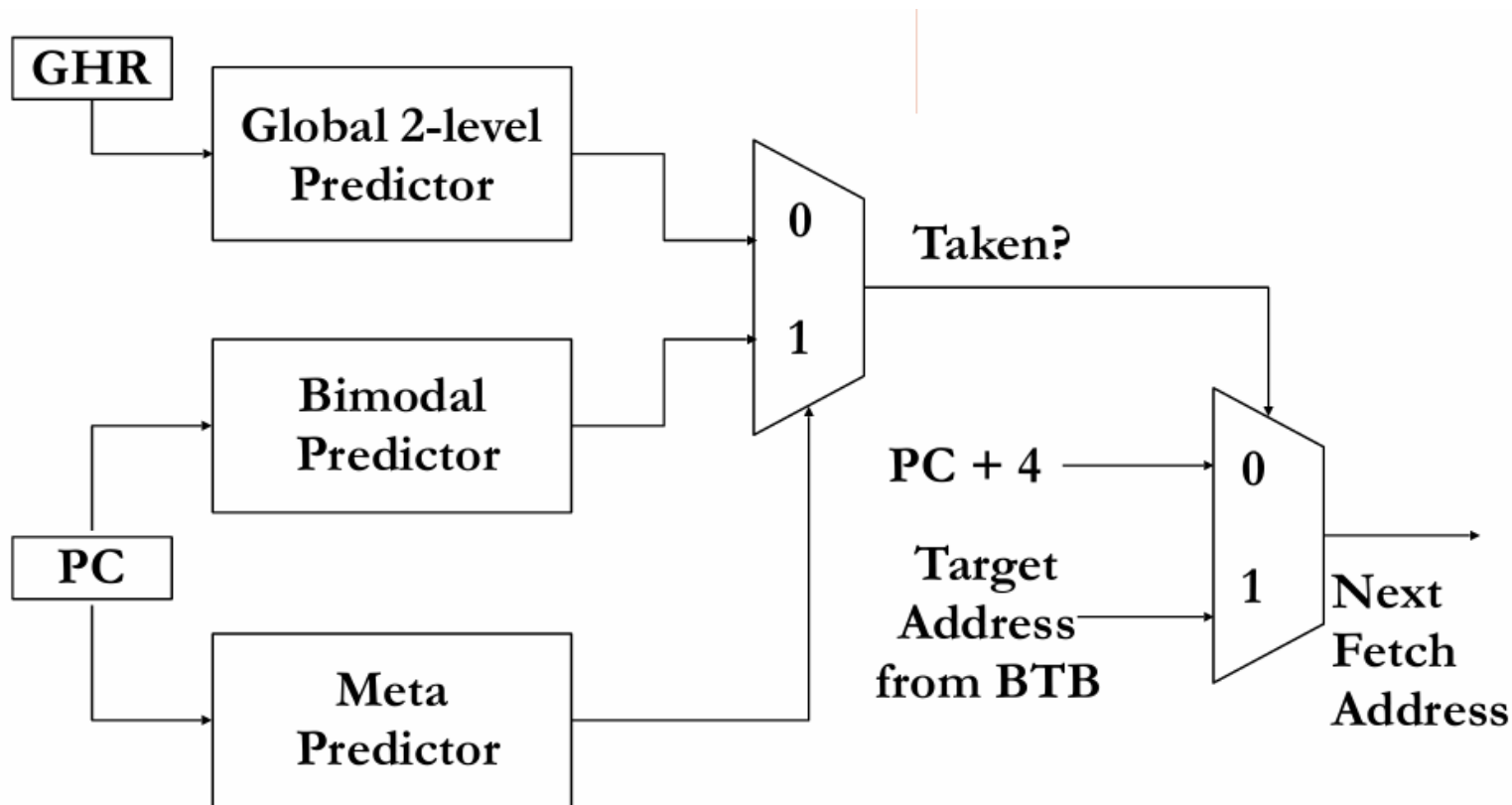


Tournament Predictor

- Observations
 - Most branches are biased (e.g. 99% Taken)
 - Filter the biased branches with a simple predictor (e.g. Bimodal)
- How does Tournament Predictor work
 - Run two different branch predictors in parallel
 - Use global 2-level predictor for branches that are hard to predict
 - Use a meta-predictor (selector) to decide which one to trust based on their past performance
 - The meta-predictor is a PHT of 2-bit saturating counters



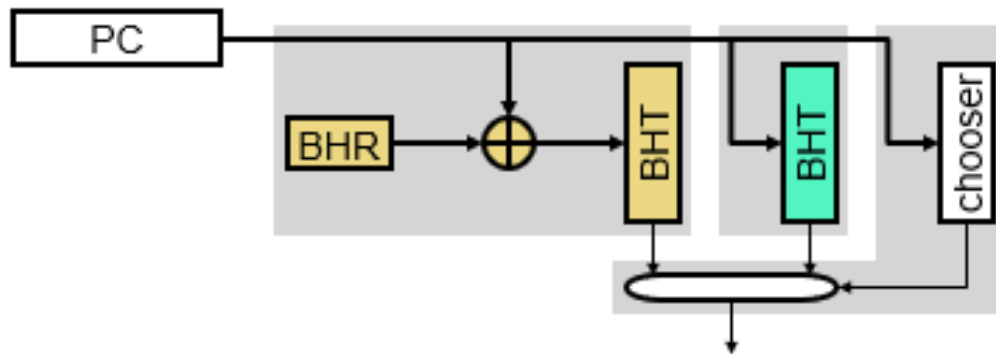
Tournament Predictor





Tournament Predictor

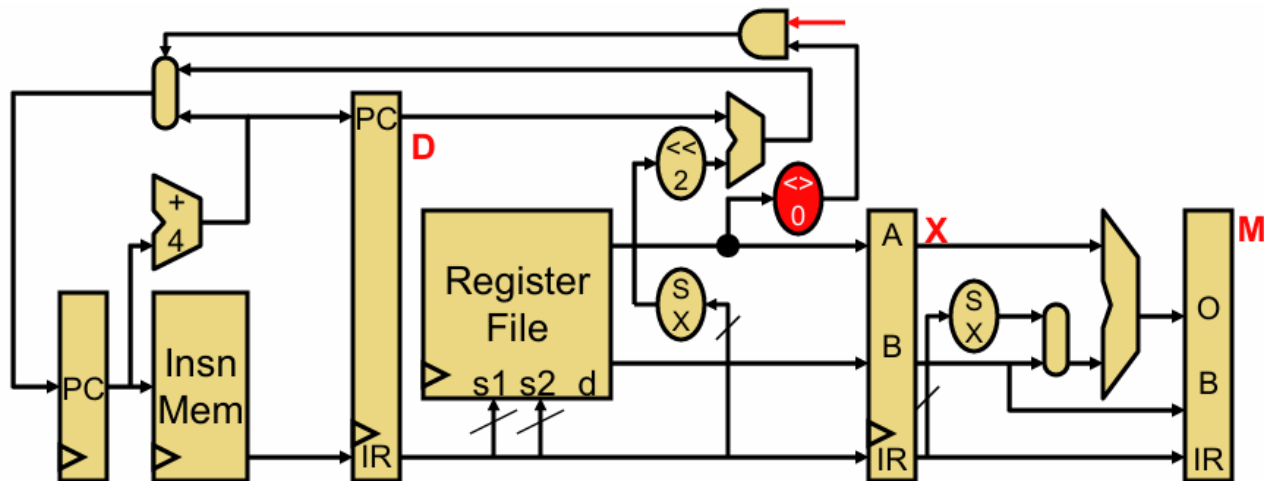
- Hybrid (tournament) predictor
 - + Correlated predictor can be made smaller, handles fewer branches
 - + 90 – 95% accuracy





Reduce Branch Penalty

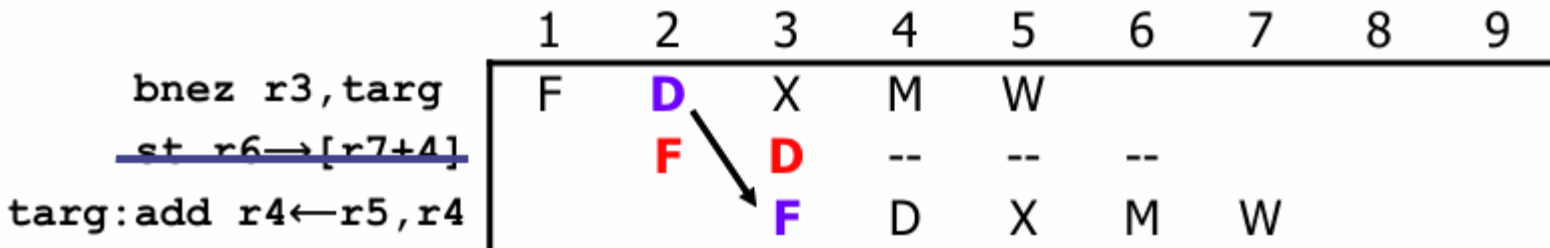
- **Fast branch: can decide at Decode, not Exec. Stage**
 - Test must be comparison to zero or equality
 - + New taken branch penalty is 1
 - - Additional instructions (slt) for more complex tests, must bypass to Decode





Reduce Branch Penalty

- **Fast branch: targets control-hazard penalty**
 - Basically, branch instructions that can resolve at D, not X
 - - Must bypass into decode stage





Reduce Branch Penalty

- **Fast branch performance**

- Assume: Branch: 20%, 75% of branches are taken
 - $CPI = 1 + 20\% * 75\% * 1 = 1.15$
 - **15% slowdown** (better than the 30% from before)
- Fast branches assume only simple comparisons
 - Not fine for ISAs with “branch if \$1 > \$2” operations
 - In such cases, say 25% of branches require an extra instruction
 - $CPI = 1 + (20\% * 75\% * 1) + 20\% * 25\% * 1$ (extra insn) = **1.2**
- Another options
 - Delayed branch or branch delay slot



Predication

- Complex predicates are converted into multiple branches
 - If $((a == b) \ \&\& \ (c < d) \ \&\& \ (a > 5000)) \ \{ \dots \}$
 - 3 conditional branches
- Problem:
 - This increases the number of control dependencies
- Idea
 - **Combine predicate operations to feed a single branch instruction**
 - Predicates stored and operated on using **condition register**
 - A **single branch** checks the value of the combined predicate
 - **+ Fewer branches in code -> fewer mispredictions/stalls**



Predication

- **Predicate**

- A Boolean condition (true/false) that determines if an instruction should be executed
 - If the predicate is false, the instruction is treated as a NOP
 - Ideal for the short if-else blocks
- A branch is a control-flow instruction that decides where the program jumps next based on a condition
 - Change the PC to a new address



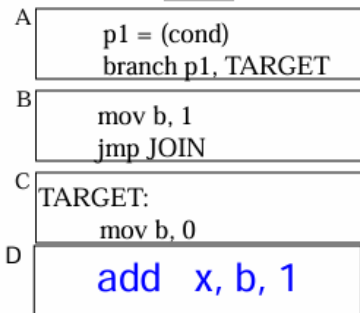
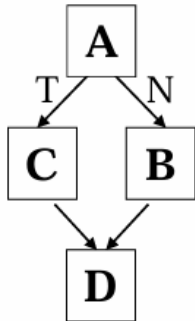
Predication

- Idea: compiler converts control dependence into data dependence ->
branch is eliminated

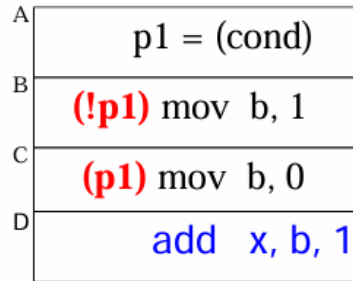
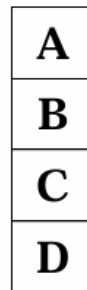
<https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module7.4.2-predicated-execution.pdf>

(normal branch code)

```
if (cond) {  
    b = 0;  
}  
else {  
    b = 1;  
}
```



(predicated code)





Predication

- Idea: compiler converts control dependence into data dependence -> branch is eliminated
 - Each instruction has a predicate bit set based on the predicate computation
 - Only instructions with TRUE predicates are committed (Other turned into NOPs)
 - - Possible unnecessary work
 - If the first predicate is false, no need to compute other predicates



Predication

- Instead of predicting which way we're going, why not go **both ways**?

```
// C code          ; original RV          ; imaginary predicated RV
if (a >= b) {      blt x1,x2,else      slt p1,x1,x2
  x += y;          add x3,x3,x4      add.!p1 x3,x3,x4
} else {           j after          sub.p1 x3,x3,x5
  x -= z;          else:
}                  sub x3,x3,x5
                  after:
```



Predication

- Advantages
 - + Eliminates mispredications for hard-to-predict branches
 - + No need for branch prediction for some branches
 - + Good if misprediction cost $>$ useless work due to prediction
 - + Enable code optimizations hindered by the control dependency
 - + Can move instructions more freely within predicted code



Predication

- Disadvantages
 - - Cause useless work for branches that are easy to predict
 - - **Reduce performance if misprediction cost < useless work**
 - - Additional hardware and ISA support
 - - Cannot eliminate all hard to predict branches
 - Loop branches?



Takeaway Questions

- Dynamic branch prediction
 - 20% of instruction branches
 - Simple predictor: branches predicted with 75% accuracy
 - What is the CPI when using such a simple predictor?
 - More advanced predictor: 95% accuracy
 - What is the CPI when using such an advanced predictor?



Takeaway Questions

- Dynamic branch prediction
 - 20% of instruction branches
 - Simple predictor: branches predicted with 75% accuracy
 - What is the CPI when using such a simple predictor?
 - $\text{CPI} = 1 + (20\% * 25\% * 2) = 1.1$
 - More advanced predictor: 95% accuracy
 - What is the CPI when using such an advanced predictor?
 - $\text{CPI} = 1 + (20\% * 5\% * 2) = 1.02$



Exceptions & Interrupts

- Handle unexpected events from within the processor
 - Control System Register (CSR) tackles these events in the processor
- **Exception**
 - Arises within the CPU
 - E.g. undefined opcode, system calls
- **Interrupt**
 - From an external I/O controller
 - Raised by UART, SPI, GPIO, keyboard, network card ...



Handling Exception

- When an exception occurs
 - Save the address of the unfortunate instruction in the
 - In RISC-V: supervisor exception cause register (SEPC)
 - 64/32-bit register
 - Capture the address of the interrupted instructions
 - Transfer control to the operating system at some specified address
 - The operating system can terminate the program or may continue its execution



Handling Exception

- How to communicate the reason for exception
 - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
 - Holds a field that indicates the reason for the exception (64-bit)
 - Records the highest priority exception in a clock cycle if more than one exception occurs
 - Exception code field:
 - 2 for undefined opcode
 - 12 for hardware malfunction
 - A single entry point for all exceptions
 - OS decodes the status register to find the cause
 - E.g. Jump to the handler with the address
 - 0000 0000 1C09 0000_{hex}



Handling Exception

- How to communicate the reason for exception
 - Vectored interrupt
 - Handler address determined by the cause
 - Define exception vector addresses exception types

Exception type	Exception vector address to be added to a Vector Table Base Register
Undefined instruction	00 0100 0000 _{two}
System Error (hardware malfunction)	01 1000 0000 _{two}

- Instruction either
 - Deal with the interrupt or
 - Jump to real handler



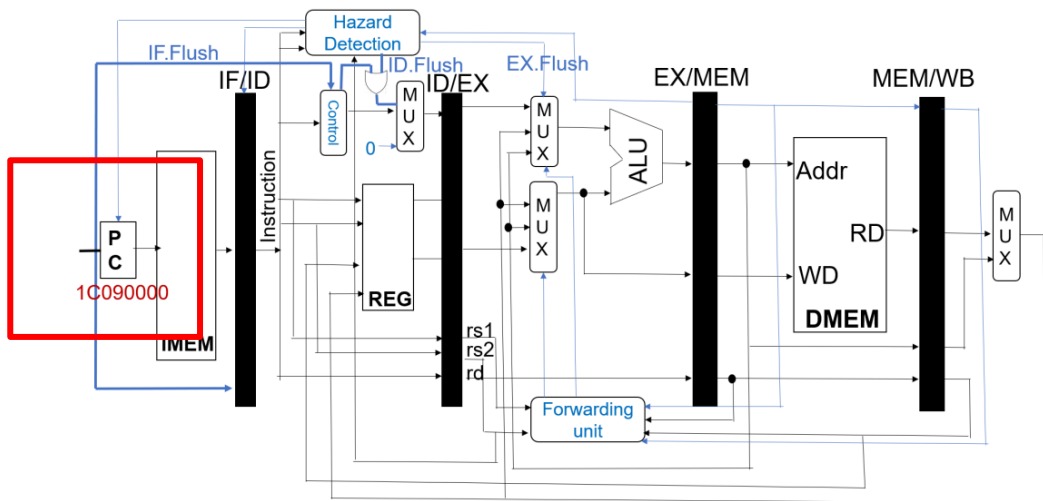
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
 - Take corrective action
 - Use sret instruction to return to program
- Otherwise
 - Terminate program
 - Report error using SEPC, SCAUSE registers, ...



Exceptions in a Pipeline

- When exception is detected during the EX stage
 - A hardware malfunction exception occurs in the `add` instruction
 - Assume the hardware malfunction is detected in the EX stage
 - `1C09 0000hex` is forced into the PC



Given this instruction sequence

`40hex sub x11, x2, x4`

`44hex and x12, x2, x5`

`48hex or x13, x2, x5`

`4Chex add x1, x2, x1`

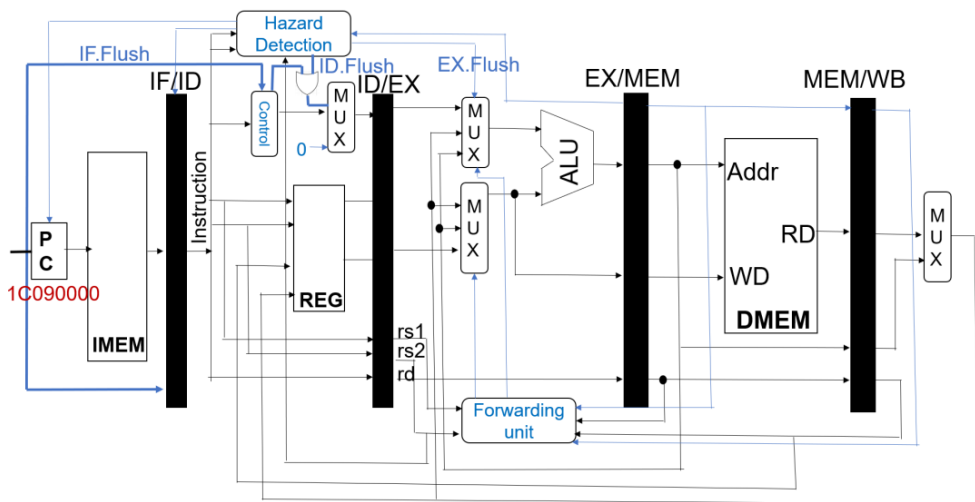
`50hex sub x15, x6, x7`

`54hex lw x16, 100(x7)`



Exceptions in a Pipeline

- When exception is detected during the EX stage
 - **Step 1:** Use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage



Given this instruction sequence

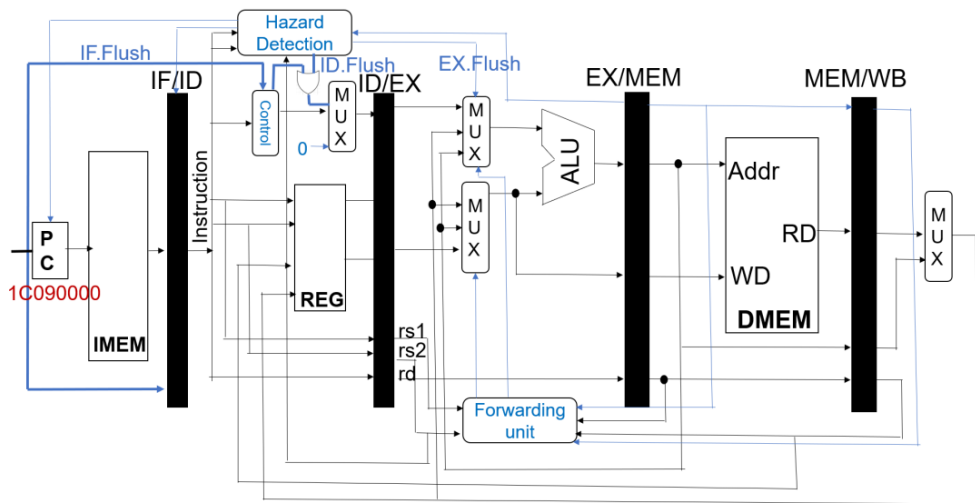
```
40hex sub x11, x2, x4
44hex and x12, x2, x5
48hex or x13, x2, x5
4Chex add x1, x2, x1
50hex sub x15, x6, x7
54hex lw x16, 100(x7)
```

Flush



Exceptions in a Pipeline

- When exception is detected during the EX stage
 - Fetch the first instruction of the exception handler (sw x26, 1000(x0)) from the instruction location $1C09\ 0000_{hex}$
 - **Step 2:** Complete the previous instructions



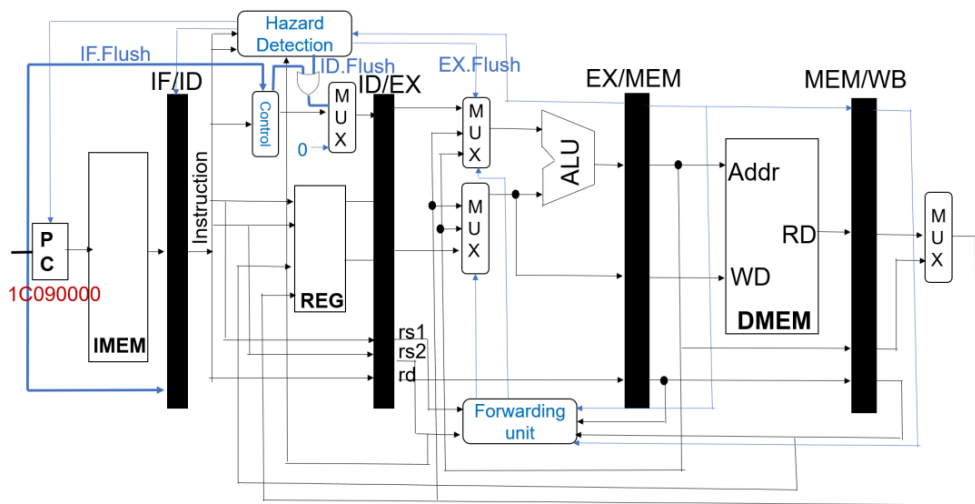
Given this instruction sequence

```
40hex sub x11, x2, x4
44hex and x12, x2, x5
48hex or x13, x2, x5
4Chex add x1, x2, x1
50hex sub x15, x6, x7
54hex lw x16, 100(x7)
```



Exceptions in a Pipeline

- When exception is detected during the EX stage
 - **Step 3:** Flush the “add” instruction and the subsequence instructions
 - Flush add, sub, lw instructions



Given this instruction sequence

40_{hex} sub x11, x2, x4

44_{hex} and x12, x2, x5

48_{hex} or x13, x2, x5

4C_{hex} add x1, x2, x1

50_{hex} sub x15, x6, x7

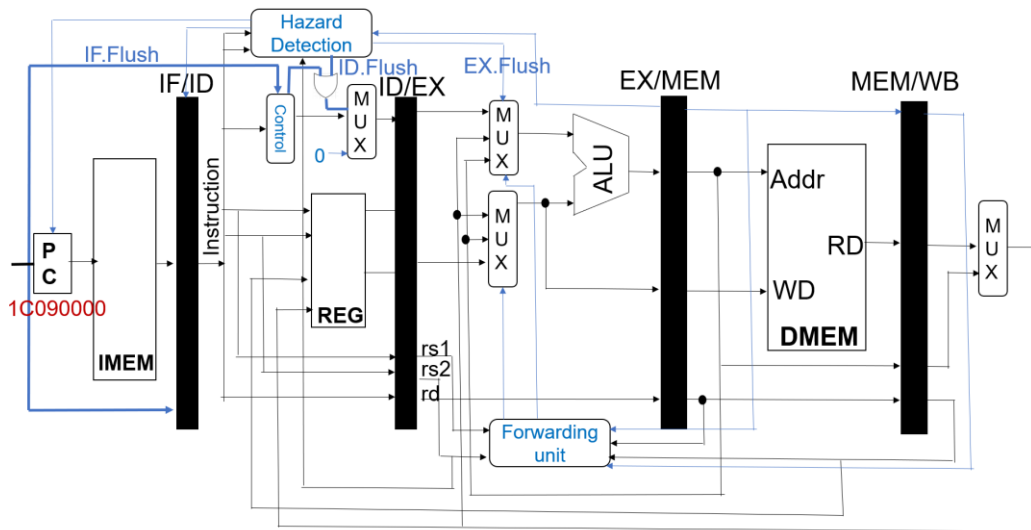
54_{hex} lw x16, 100(x7)

Flush



Exceptions in a Pipeline

- When exception is detected during the EX stage
 - **Step 4:** Save the address of the offending instruction in the SEPC
 - Save ($4C_{\text{hex}}$) address of add instruction in the SEPC register



Given this instruction sequence

```
40hex sub x11, x2, x4
44hex and x12, x2, x5
48hex or x13, x2, x5
4Chex add x1, x2, x1
50hex sub x15, x6, x7
54hex lw x16, 100(x7)
```



Multiple Exceptions

- Pipelining overlaps multiple instructions
 - Could have multiple exceptions at once
- Simple approach
 - Deal with exception from earliest instruction
 - Flush subsequence instructions
 - “Precise” exception
- In complex pipelines
 - Multiple instructions issued per cycle
 - Out-of-order completion
 - Maintaining precise exceptions is difficult!



Imprecise Exceptions

- What is imprecise exceptions?
 - When a processor cannot identify the exact instruction that caused an error
- When the imprecise exception occurs?
 - Instructions completes out of order during out-of-order execution
 - The PC might point to a later instruction than the one caused the fault, preventing easy resumption



Conclusion

- Pipeline challenge is hazards
 - Forwarding helps with many data hazards
 - Delayed branch helps with control hazard in 5 stage pipeline
 - Load delay slot / interlock necessary
- More aggressive performance
 - Superscalar
 - Out-of-order execution