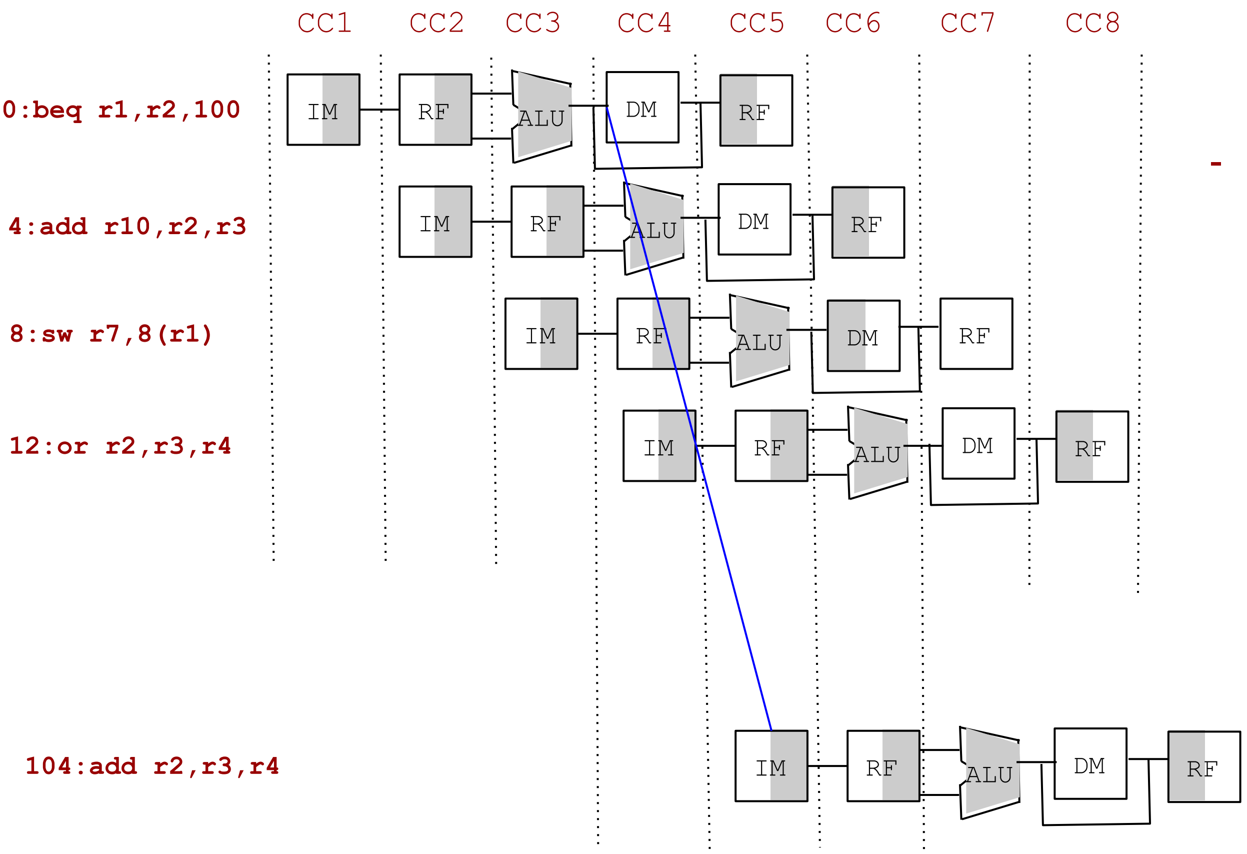
Computer Architecture

Control Hazard and Branch Prediction

Debadatta Mishra, CSE, IITK

Control hazard (racap)



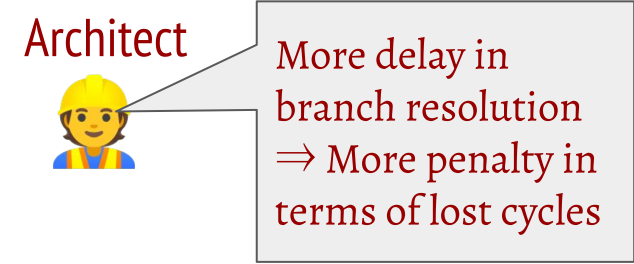
- Assuming `r1=r2`, next instruction will be fetched from memory address 104 in CC5
- What is the impact on pipeline?
  - Three partially executed instructions need to be discarded
  - Frequent reset of the pipeline results in loss of performance

Control hazard (racap)

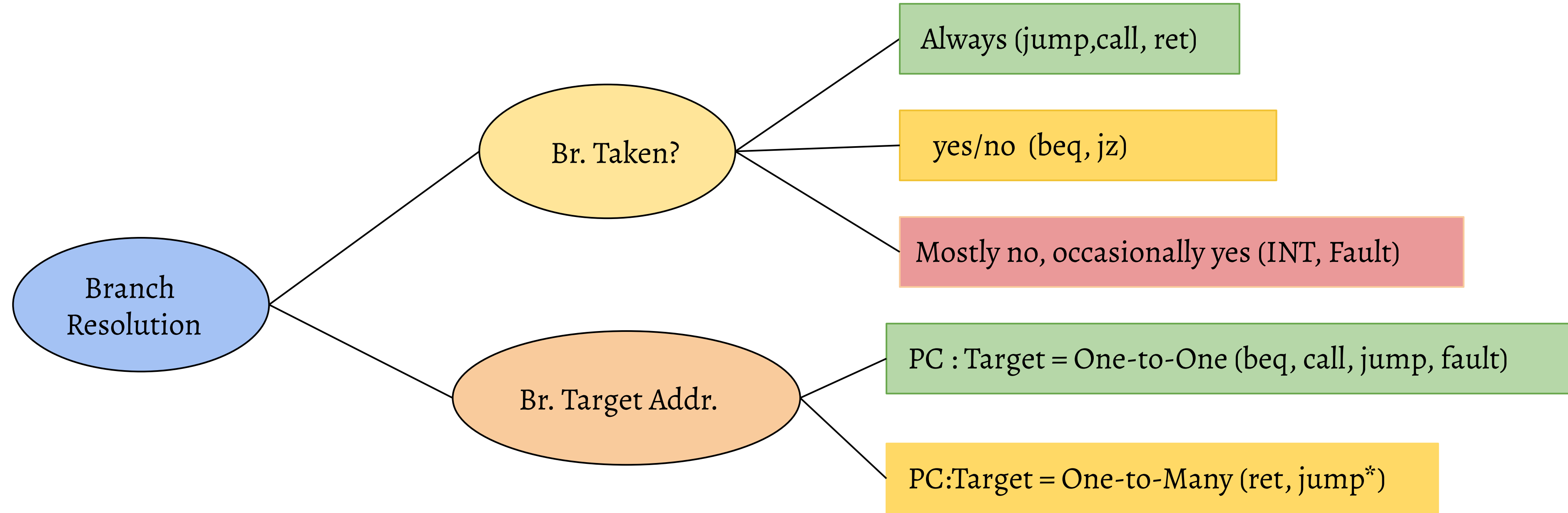
CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8

```
0:beq r1,r2,100      IM RF ALU DM RF
4:add r10,r2,r3      IM RF ALU DM RF
8:sw r7,8(r1)        IM RF ALU DM RF
12:or r2,r3,r4       IM RF ALU DM RF
104:add r2,r3,r4     IM RF ALU DM RF
```

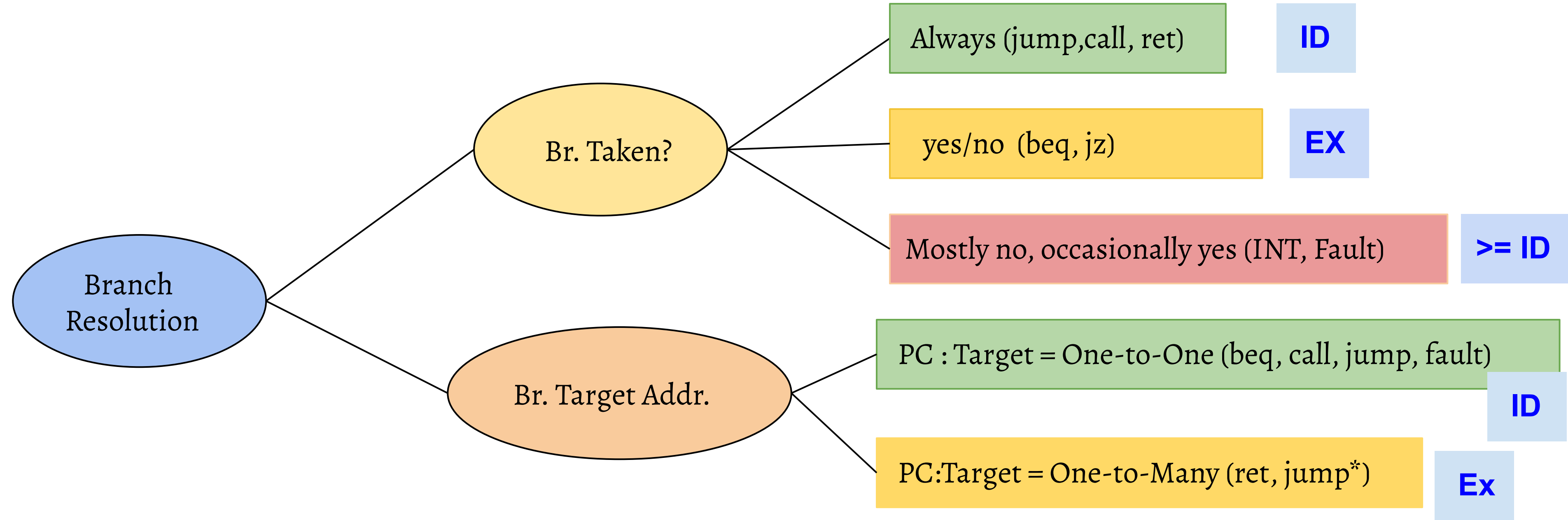
- Assuming `r1=r2`, next instruction will be fetched from memory address 104 in CC5
- What is the impact on pipeline?
  - Three partially executed instructions need to be discarded
  - Frequent reset of the pipeline results in loss of performance



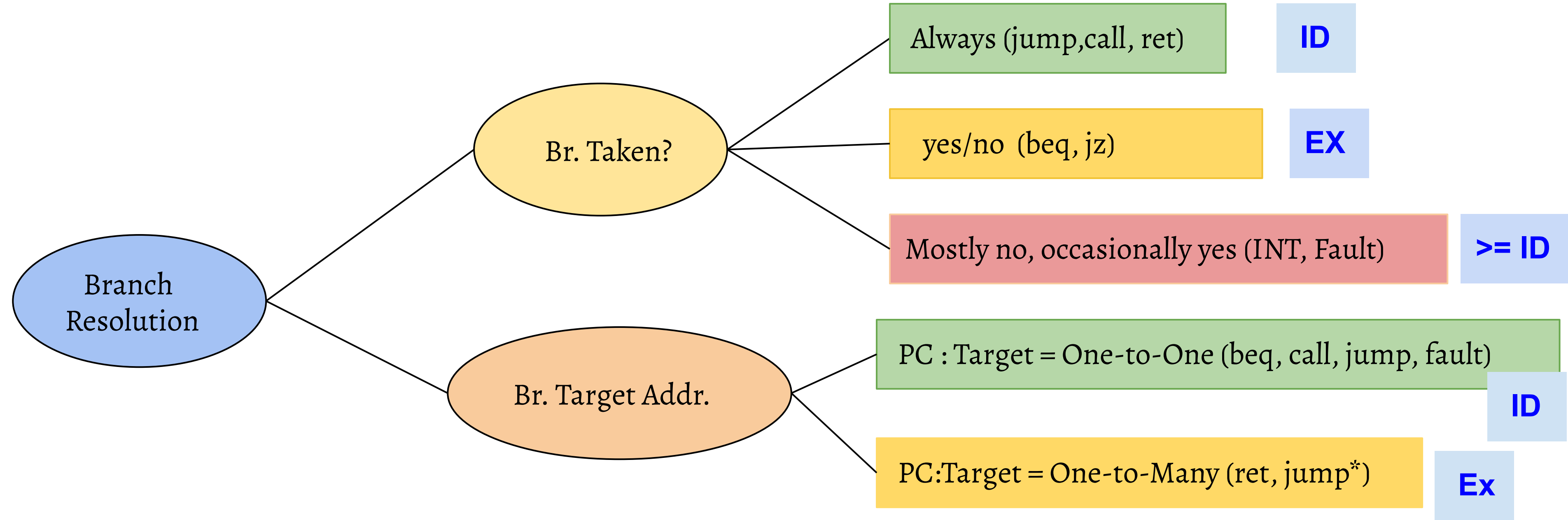
Taxonomy of branches



Taxonomy of branches

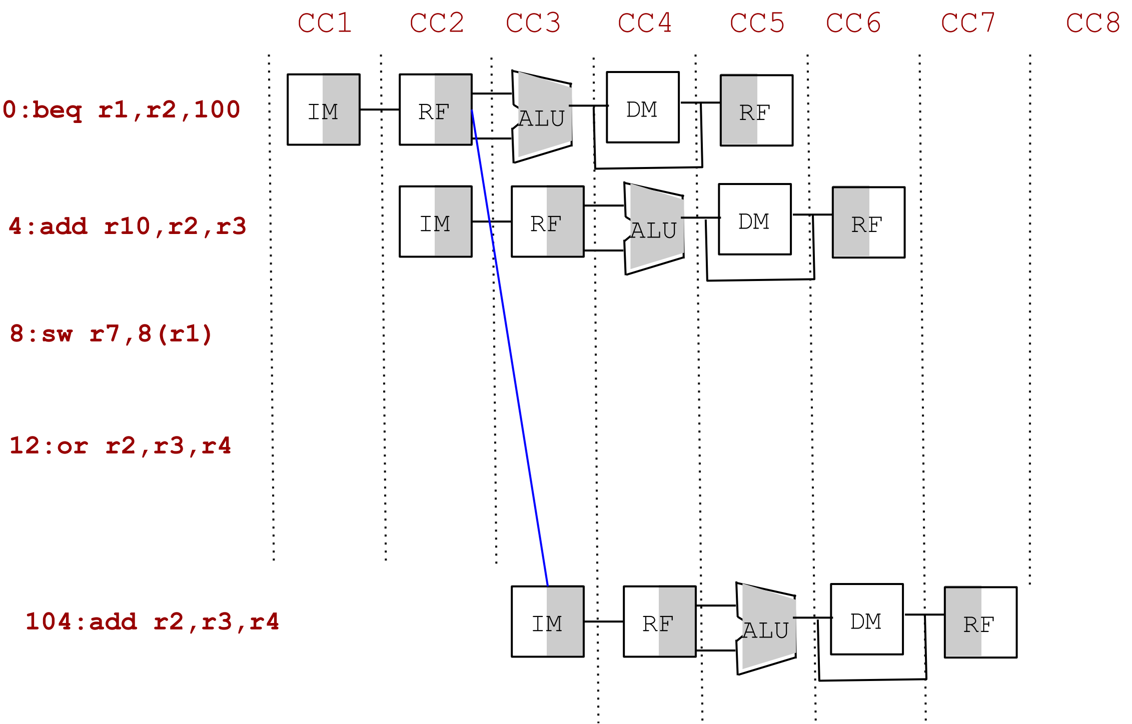


Taxonomy of branches



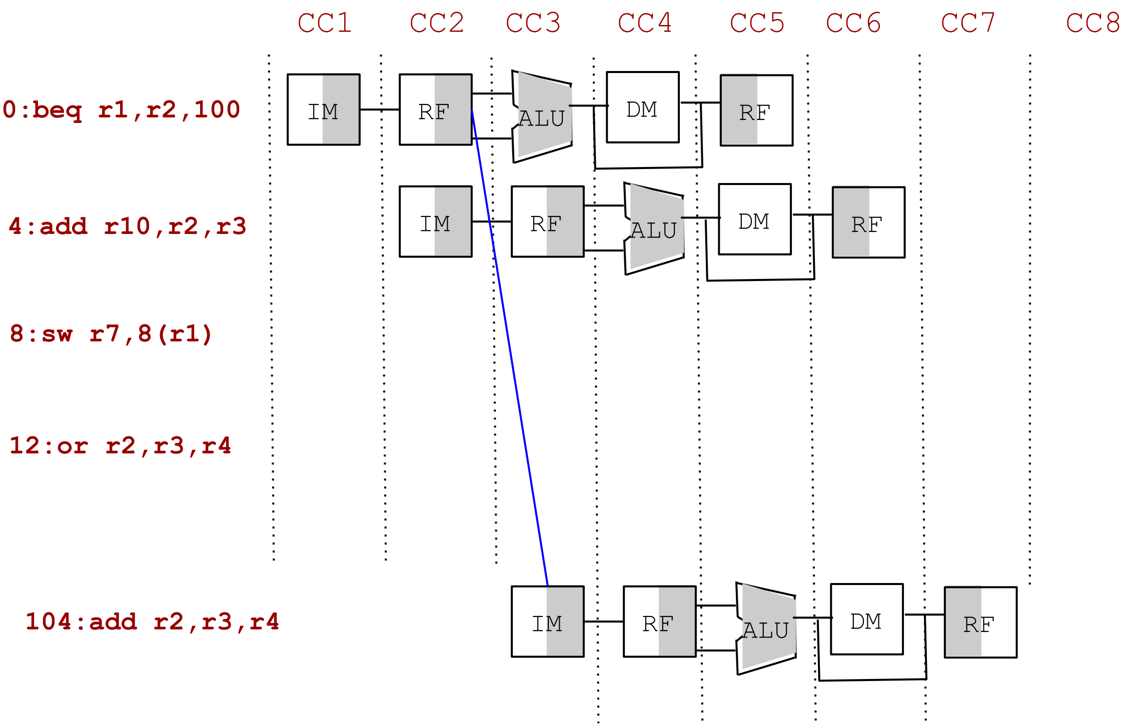
- Pipeline performance loss = MAX{*stage deciding branch taken or not, stage providing target addr*}
- For conditional jumps, "evaluation of condition" is the first target of optimization

Solution approach: Resolve early



- Assume additional circuits to evaluate the condition, resolve target address and update PC in the ID stage
- Cycles lost = ?
- Side-effects?

Solution approach: Resolve early



- Assume additional circuits to evaluate the condition, resolve target address and update PC in the ID stage
- Cycles lost = 1 (when taken)
- Side-effects? Can introduce new data hazards, e.g., branch instruction now needs data earlier

Solution approach: Resolve early + Delay Slot

- Idea: Place one instruction after the branch instruction. Preferably the instruction should perform some useful work. Otherwise, place an instruction which if executed wrongly, does not impact the correctness of the program.

Solution approach: Resolve early + Delay Slot

- Idea: Place one instruction after the branch instruction. Preferably the instruction should perform some useful work. Otherwise, place an instruction which if executed wrongly, does not impact the correctness of the program.

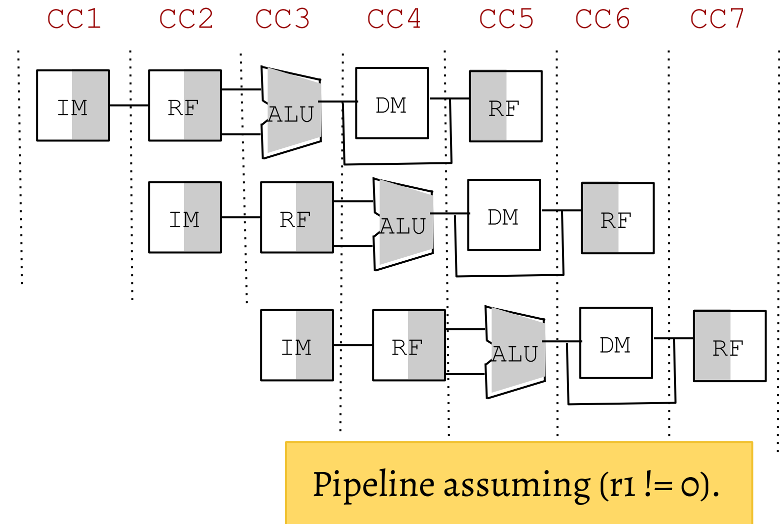
```
add r3,r1,r2
if(r1 == 0){
  [Delay slot]
  sub r7, r1, r2
}

L1: add r7,r0,r8
```

```
bneq r1,r0,100

add r3,r1,r2


L1: add r7,r0,r8
```



Solution approach: Resolve early + Delay Slot

- Idea: Place one instruction after the branch instruction. Preferably the instruction should perform some useful work. Otherwise, place an instruction which if executed wrongly, does not impact the correctness of the program.

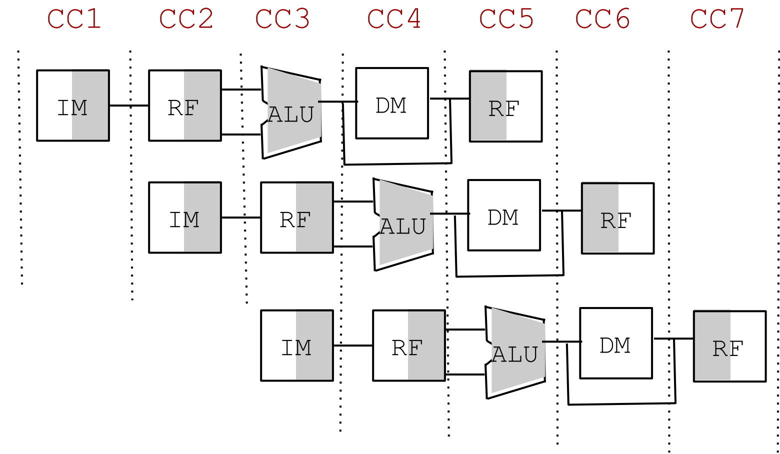
```
sub r1,r1,r2
if(r1 == 0){
  [Delay slot]
  sub r7, r1, r2
}

L1: add r7,r0,r8
```

```
bneq r1,r0,100

add r7,r0,r8

sub r7,r1,r2
```



Pipeline assuming (r1 = 0).

Any other possible solution?

Solution approach: Resolve early + Delay Slot

- Idea: Place one instruction after the branch instruction. Preferably the instruction should perform some useful work. Otherwise, place an instruction which if executed wrongly, does not impact the correctness of the program.

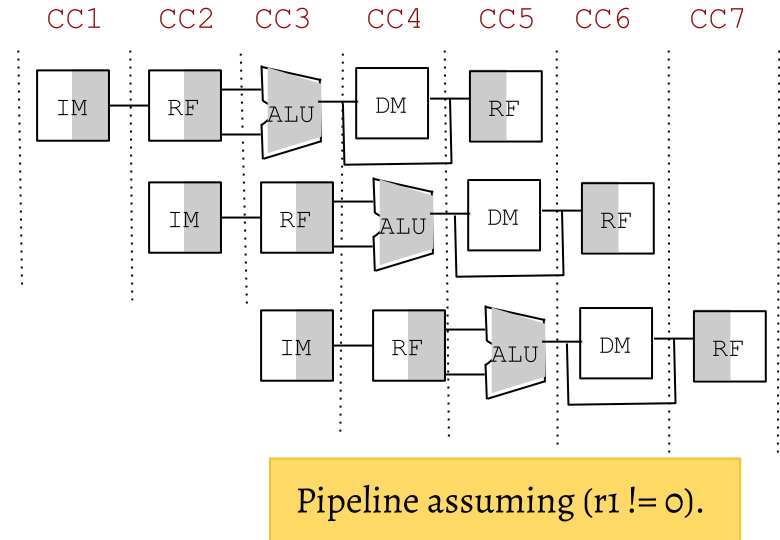
```
sub r1,r1,r2
if(r1 == 0){
  [Delay slot]
  sub r7, r1, r2
}

L1: add r7,r0,r8
```

```
bneq r1,r0,100

sub r7, r1, r2

L1: add r7,r0,r8
```



Solution approach: Resolve early + Delay Slot

- Idea: Place one instruction after the branch instruction. Preferably the instruction should perform some useful work. Otherwise, place an instruction which if executed wrongly, does not impact the correctness of the program.

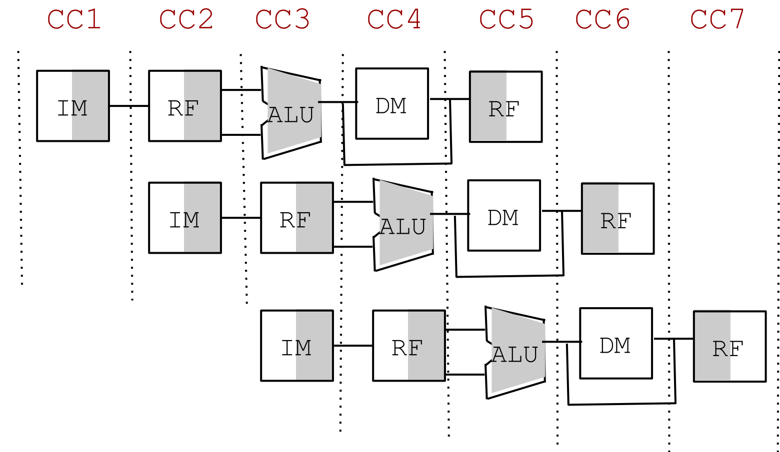
```
add r3,r1,r2
if(r1 == 0){
  [Delay slot]
  sub r7, r1, r2
}

L1: add r7,r0,r8
```

```
bneq r1,r0,100

add r3,r1,r2

L1: add r7,r0,r8
```



- The compiler/assembler can choose an instruction such that
  - An independent instruction before the branch
  - An instruction executing wrongly does not impact the correctness but is some useless operation

Programming/compiler techniques

- Avoid/reduce branches
    - Use indexing into const. values

```
switch(digit){
case '1':
            strcat(op, "one");
            break;
case '2':
            strcat(op, "two");
            Break;
…
}
```

Programming/compiler techniques

- Avoid/reduce branches
  - Use indexing into const. values

```
switch(digit){
case ‘1’:
            strcat(op, “one”);
            break;
case ‘2’:
            strcat(op, “two”);
            Break;
…
}
```

```
char *ptr[10]={“zero”, “one”, … “nine”};

strcat(op, ptr[digit]);
```

Programming/compiler techniques

- Avoid/reduce branches
  - Use indexing into const. values
  - Replace {branch+arithmetic} by {arithmetic+logical)

```
if(val < max){
   newval += max - val;
}
```

Programming/compiler techniques

- Avoid/reduce branches
  - Use indexing into const. values
  - Replace {branch+arithmetic} by {arithmetic+logical)

```
if(val < max){
   newval += max - val;
}
```

```
newval += (-(val < max)) & (max-val)
```

Programming/compiler techniques

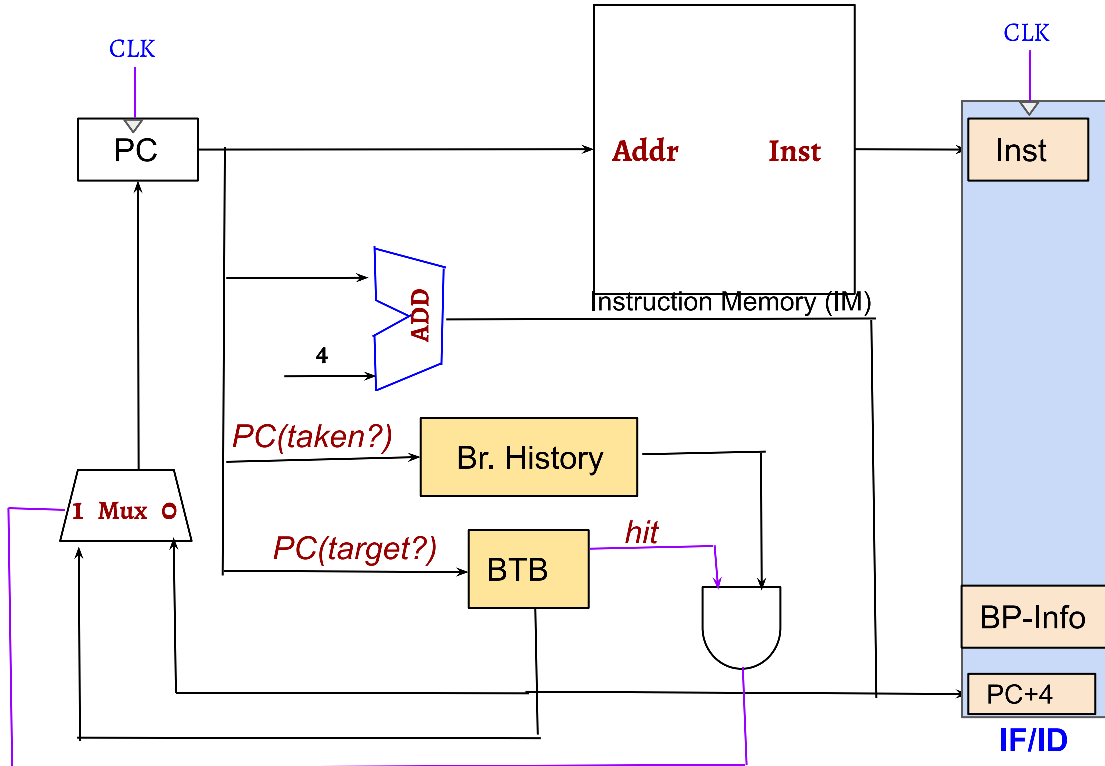
- Avoid/reduce branches
  - Use indexing into const. values
  - Replace {branch+arithmetic} by {arithmetic+logical)
- Static branch prediction
  - Profile the branch behavior with different types of input and generate branch hints through the ISA
  - Program analysis based
  - Programmer can also help by using hints (e.g., gcc pragmas)

```
if(condition){
   //Some code
}
```

```
if(likely(condition)){
    // Some code
}
```

Dynamic branch prediction

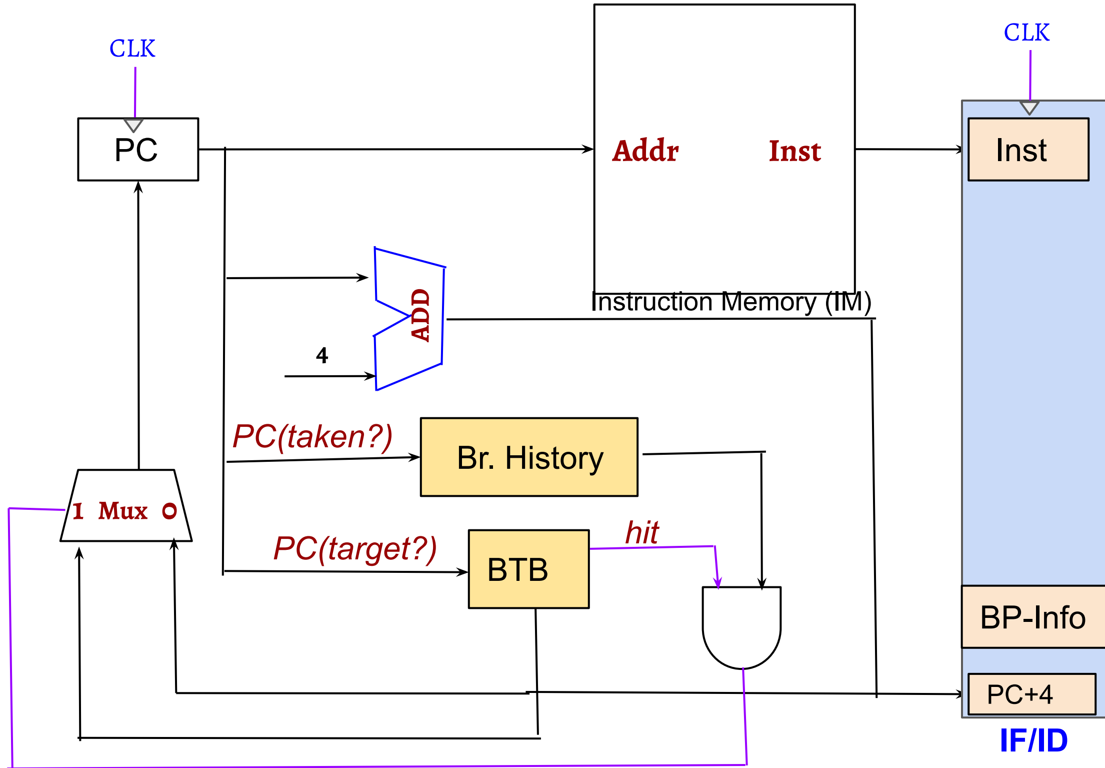
Idea: **Predict** the branch outcome and branch target address during the fetch cycle of the branch instruction. Typically, prediction is based on some history (local and/or global).



- Branch History: Predict the branch result based on its history
- Branch Target Buffer(BTB): Lookup the target address
- What information (BP-Info) is forwarded and why?

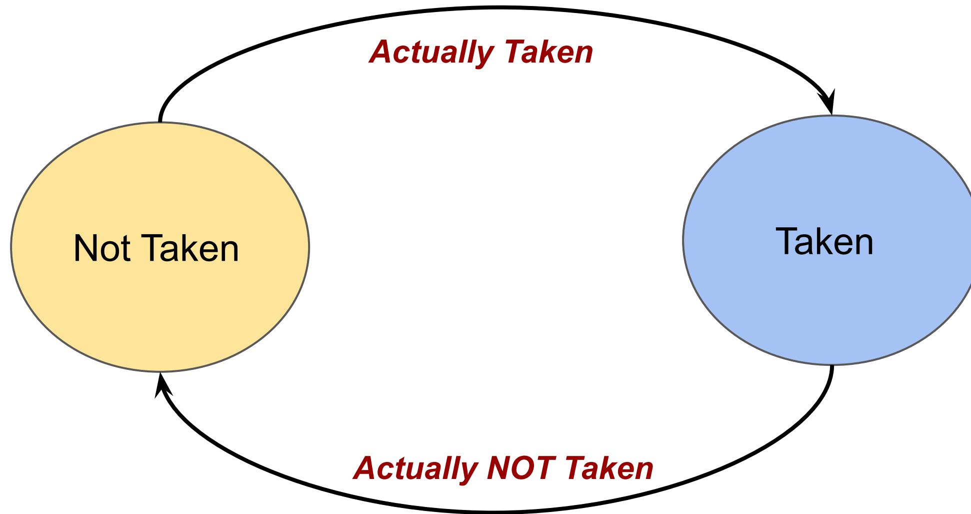
Dynamic branch prediction

Idea: **Predict** the branch outcome and branch target address during the fetch cycle of the branch instruction. Typically, prediction is based on some history (local and/or global).



- Branch History: Predict the branch result based on its history
- Branch Target Buffer(BTB): Lookup the target address
- What information (BP-Info) is forwarded and why? Information regarding the entry in the BTB and Br. history to update it based on the actual values (if required).

Prediction using single-bit history (last time)

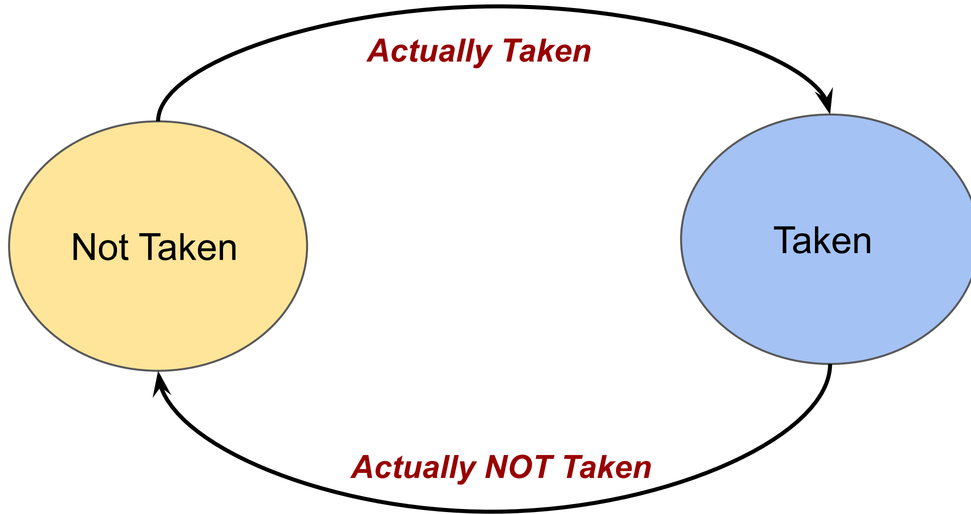


Branch Target Buffer (BTB)

PC′	Target Addr.	BHT
0x1230	0x1250	0x1

- The history state changes on wrong prediction
- Depending on the scheme, the lookup key can be lower bits of PC combined with other (e.g., execution history)

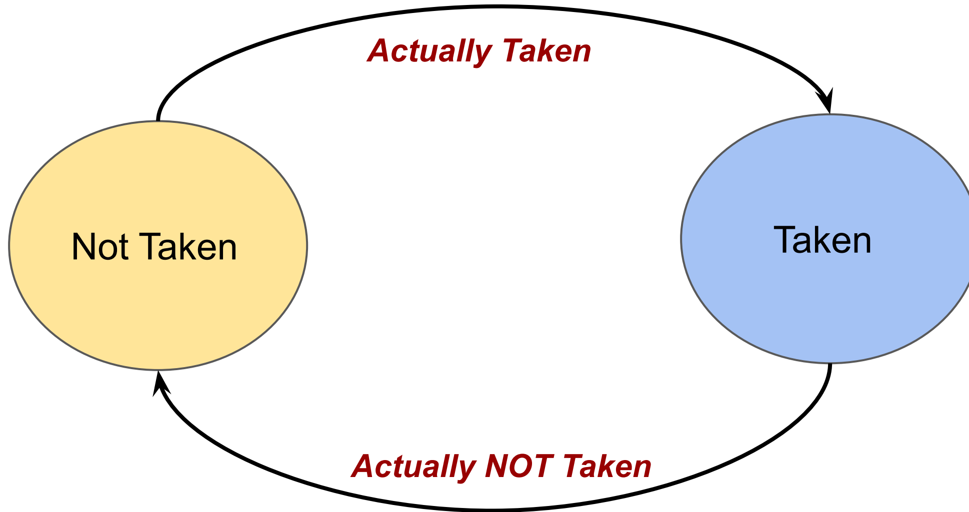
Prediction using single-bit history (last time)




```
for(i=0; i < 100; ++i){ // B1
  for(j=0; j<10; j++){ // B2
      if(j % 2) // B3
          do_something();
  }
}
```

- What will be the branch prediction accuracy for B1, B2 and B3?

Prediction using single-bit history (last time)

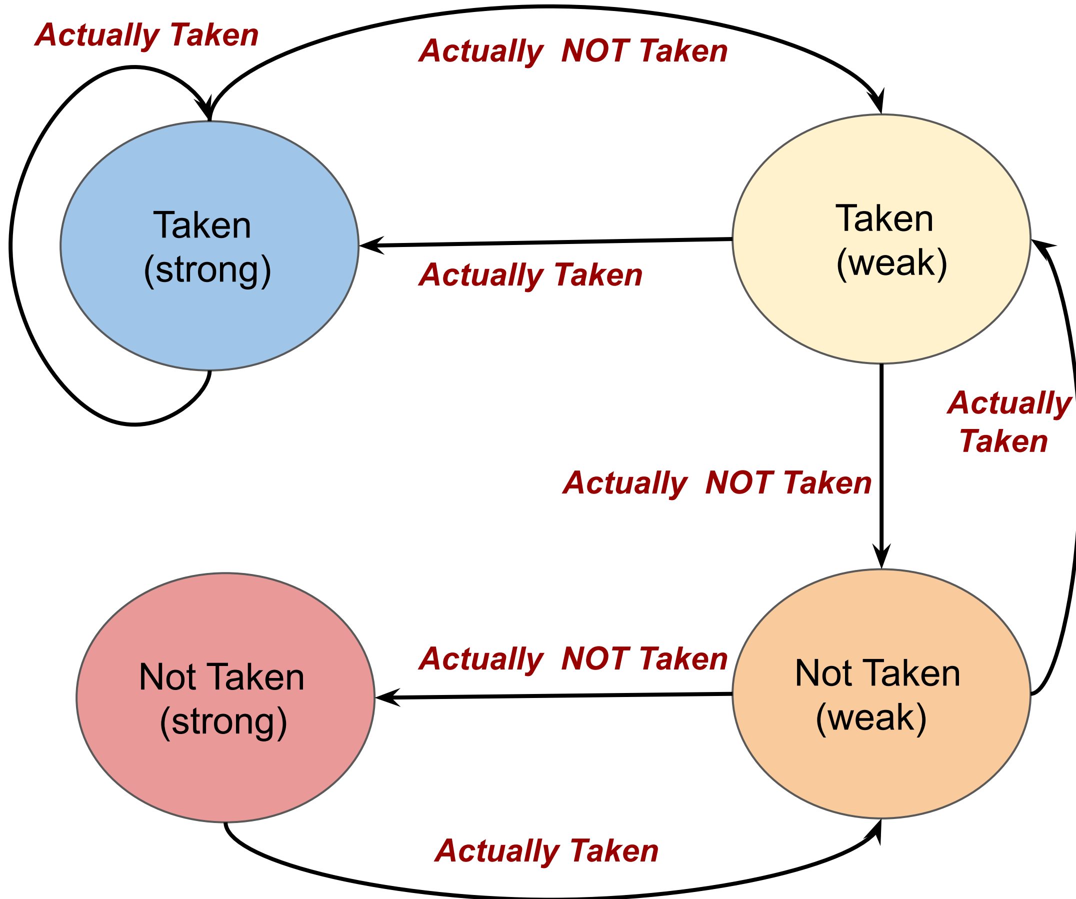



```
for(i=0; i < 100; ++i){ // B1
  for(j=0; j<10; j++){ // B2
      if(j % 2) // B3
          do_something();
  }
}
```

- What will be the branch prediction accuracy for B1, B2 and B3?

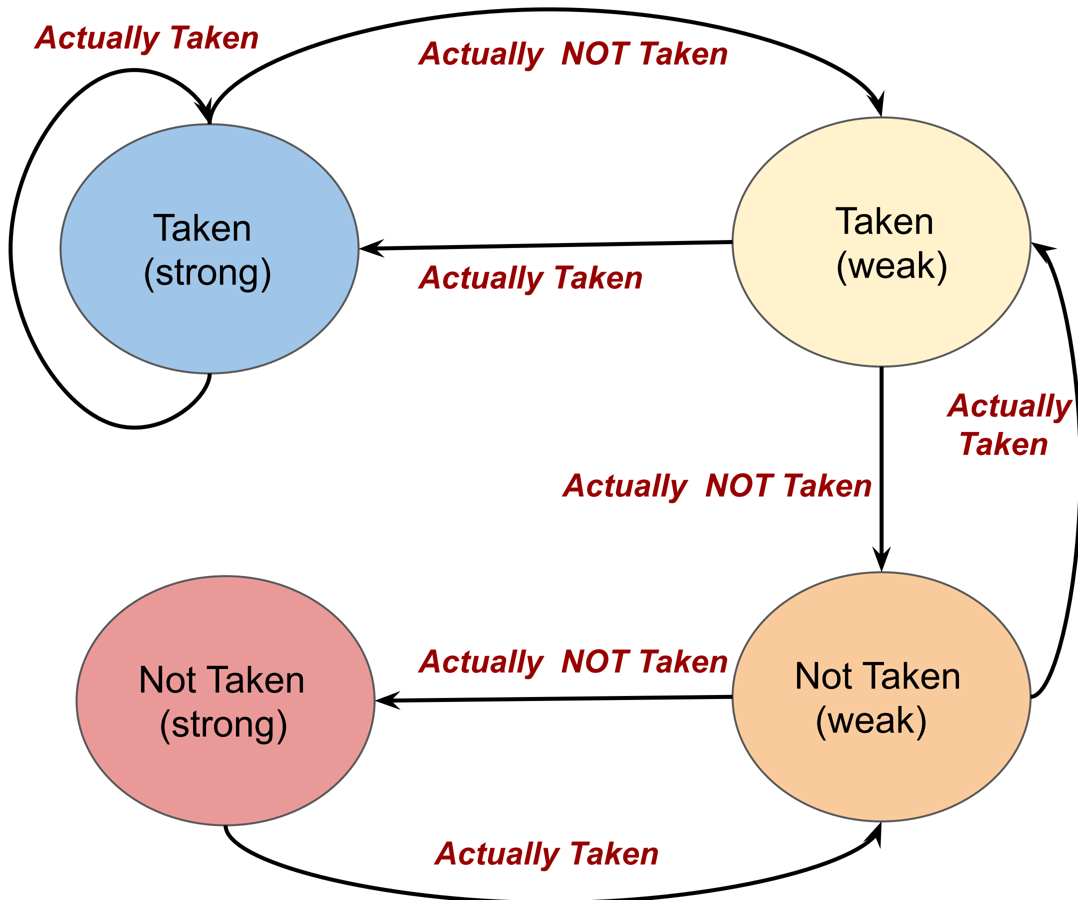
B1 = 98%, B2=80%, B3=0%

Prediction using two-bit history (second chance)



- Allow two mispredictions before changing the prediction

Prediction using two-bit history (second chance)




```
for(i=0; i < 100; ++i){ // B1
  for(j=0; j<10; j++){ // B2
      if(j % 2) // B3
          do_something();
  }
}
```

- What will be the branch prediction accuracy for B1, B2 and B3? ~100% for B1 and B2. For B3, it will be 50% assuming starting state to be "weakly not taken"

Global history

- One issue with 2-bit predictor is its complete reliance on history of the branch only. It does not consider the code path followed before reaching the branch.

```
if(a > b) // B1
    a = a << 2;
else
    a = a << 3 - 1;
if(a & 0x1) // B2
     b = a + b;
```

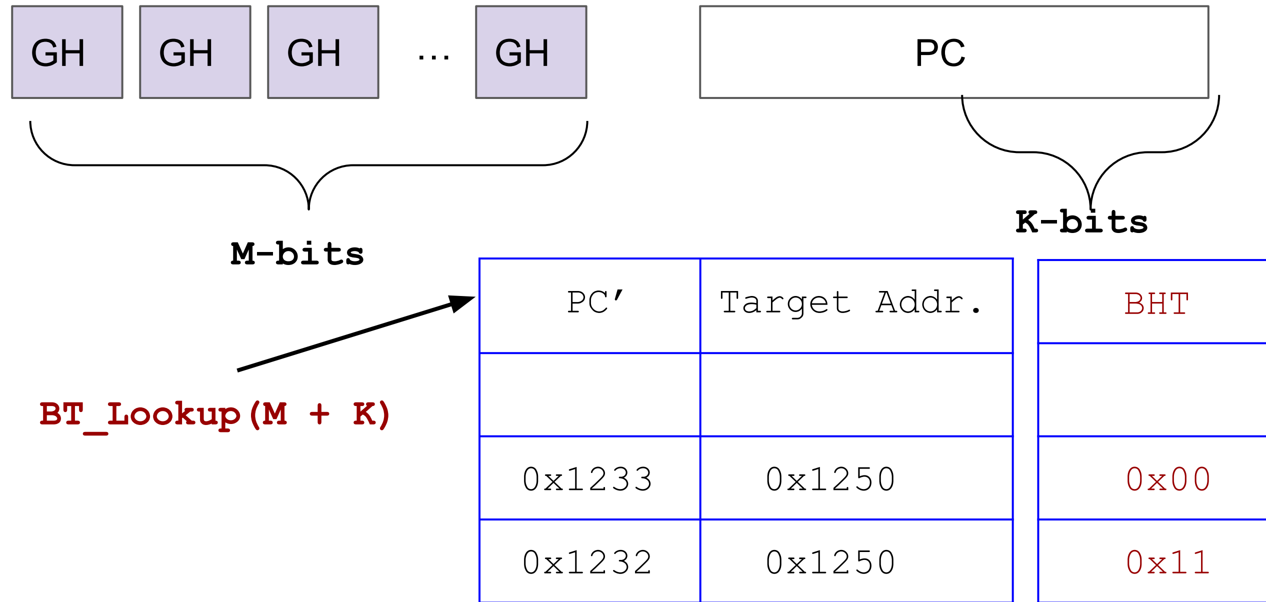
Global history

- One issue with 2-bit predictor is its complete reliance on history of the branch only. It does not consider the code path followed before reaching the branch.

```
if(a > b) // B1
    a = a << 2;
else
    a = a << 3 - 1;
if(a & 0x1) // B2
     b = a + b;
```

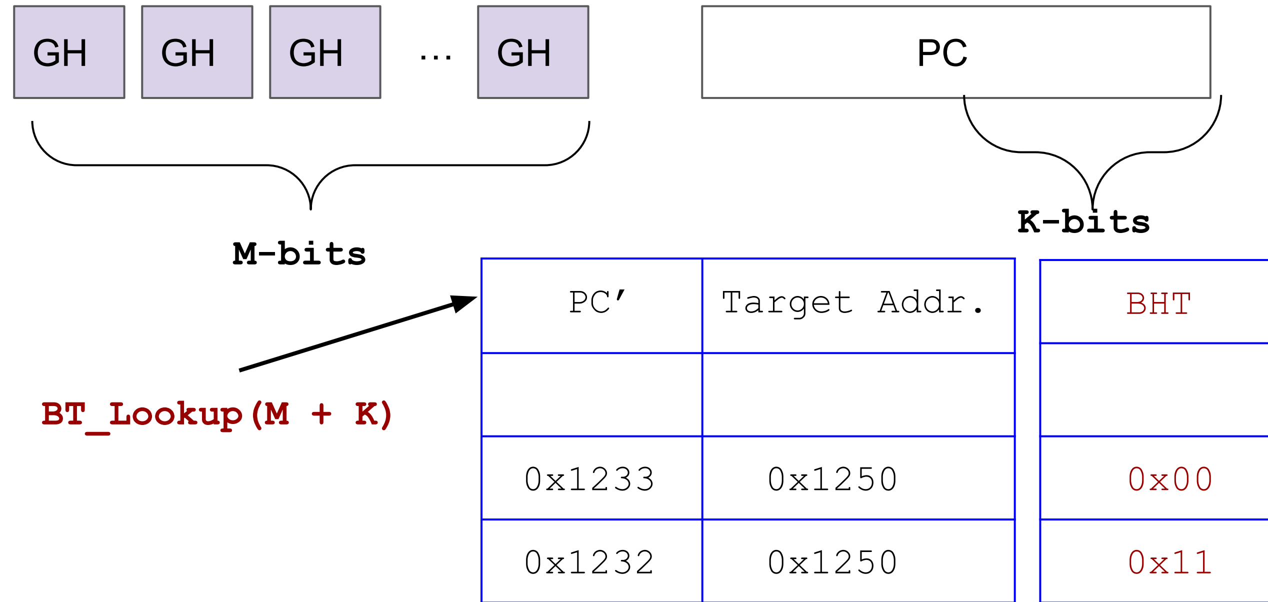
- In the above example, **B2** is taken only if **B1** is not taken. Branch predictor should use this knowledge somehow to improve prediction accuracy.

Two-level predictor



- For a single branch, there can be 2^M entries
- Each entry maintains local history using N bits (N=2 for a two bit predictor)
- Other advantages?

Two-level predictor



- For a single branch, there can be 2^M entries
- Each entry maintains local history using N bits (N=2 for a two bit predictor)
- Other advantages? Global history → Execution Flow → Target of indirect jumps

Tournament predictor

- 2-Level predictor uses extra bits to maintain state
- Example: To maintain 16K entries for 2-bit branch predictor with 3-bits of global history consumes 8x more bits compared to a simple 2-bit branch predictor

Tournament predictor

- 2-Level predictor uses extra bits to maintain state
- Example: To maintain 16K entries for 2-bit branch predictor with 3-bits of global history consumes 8x more bits compared to a simple 2-bit branch predictor

- Basis: Local history is sufficient for many branches. Global history requires extra space and therefore should be used only for branches influenced by global history.
- Additional circuits may be used to select local, global or combination of local and global for for branch prediction
- Choice of the predictor can be based on the historical accuracy of the predictors

Other type of branches

- Indirect branches
  - Commonly used in object oriented programming languages
  - PIE code generation also uses indirect branches
  - Typically unconditional jumps $\Rightarrow$ Branch target to be predicted
- Execution flow can be used to capture the history of branches (global history can be useful in this case to certain extent)

Other type of branches

- Indirect branches
  - Commonly used in object oriented programming languages
  - PIE code generation also uses indirect branches
  - Typically unconditional jumps ⇒ Branch target to be predicted
- Execution flow can be used to capture the history of branches (global history can be useful in this case to certain extent)

- Return from function
  - Unconditional jumps
  - Return address can be known (at the time of call)
- Return address is pushed onto a stack (on call) and used as the branch target address

Intel Branch Predictors

- **Local Patterns**
- **Global Patterns**
- **Dynamic loop count**

- Best prediction is used, based on prediction accuracy
- Separate prediction units for stack and indirect branches