# Branch Prediction and Predication

CS 1541
Wonsun Ahn

University of Pittsburgh

# Branch Prediction

# Solution 3: Branch Prediction

- Wrong path instructions create bubbles and decrease utilization

- What if …
  - We were able to **predict** the branch outcome?
  - Without computing branch outcome at EX stage?

- What if we could make the prediction at **IF** stage?
  - We can start fetching on the correct path at very next cycle!
  - If our prediction turns out incorrect, **flush** at that point.

- HDU can **flush pipeline** of wrong path instructions, just like before
  - Misprediction becomes a performance, **not a correctness issue**
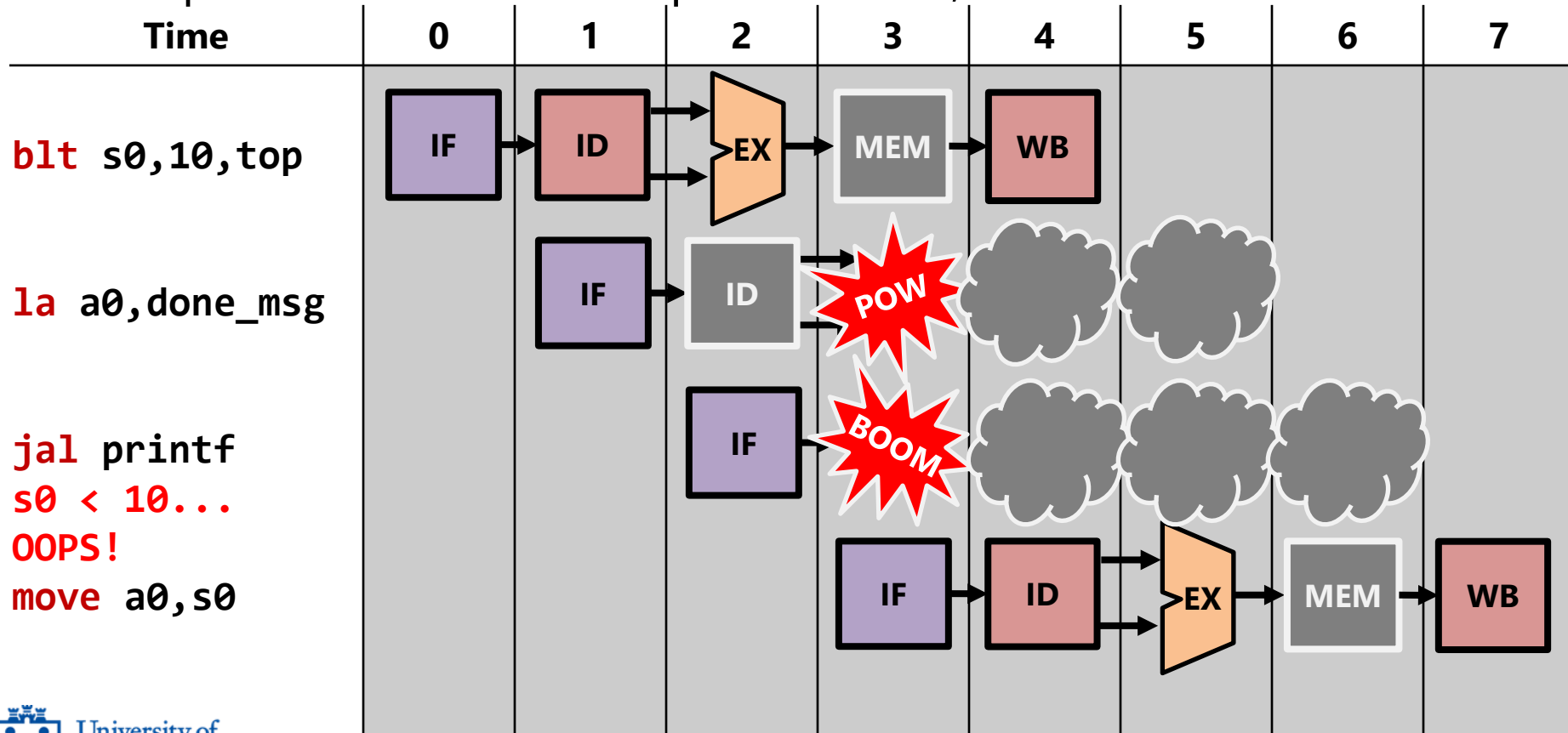
- We have been doing a form of branch prediction all along!
  - We assumed that all branches will be ***not taken***

- Two simple policies:
  - Predict ***not taken***: continue fetching PC + 4, flush if taken
    *Pro*: Can start fetching the next instruction immediately
    *Con*: ~67% of branches are taken (due to loops) → many flushes
  - Predict ***taken***: fetch branch target, flush if not taken
    *Pro*: ~67% of branches are taken (due to loops) → less flushes
    *Con*: ID stage must decode branch target before fetch → bubble

- Both are non-ideal: there are better ways to predict!

# Types of Branch Prediction

- ***Static Branch Prediction***
  - Predicting branch behavior based on code analysis
  - **Compiler** gives hints about branch direction through ISA
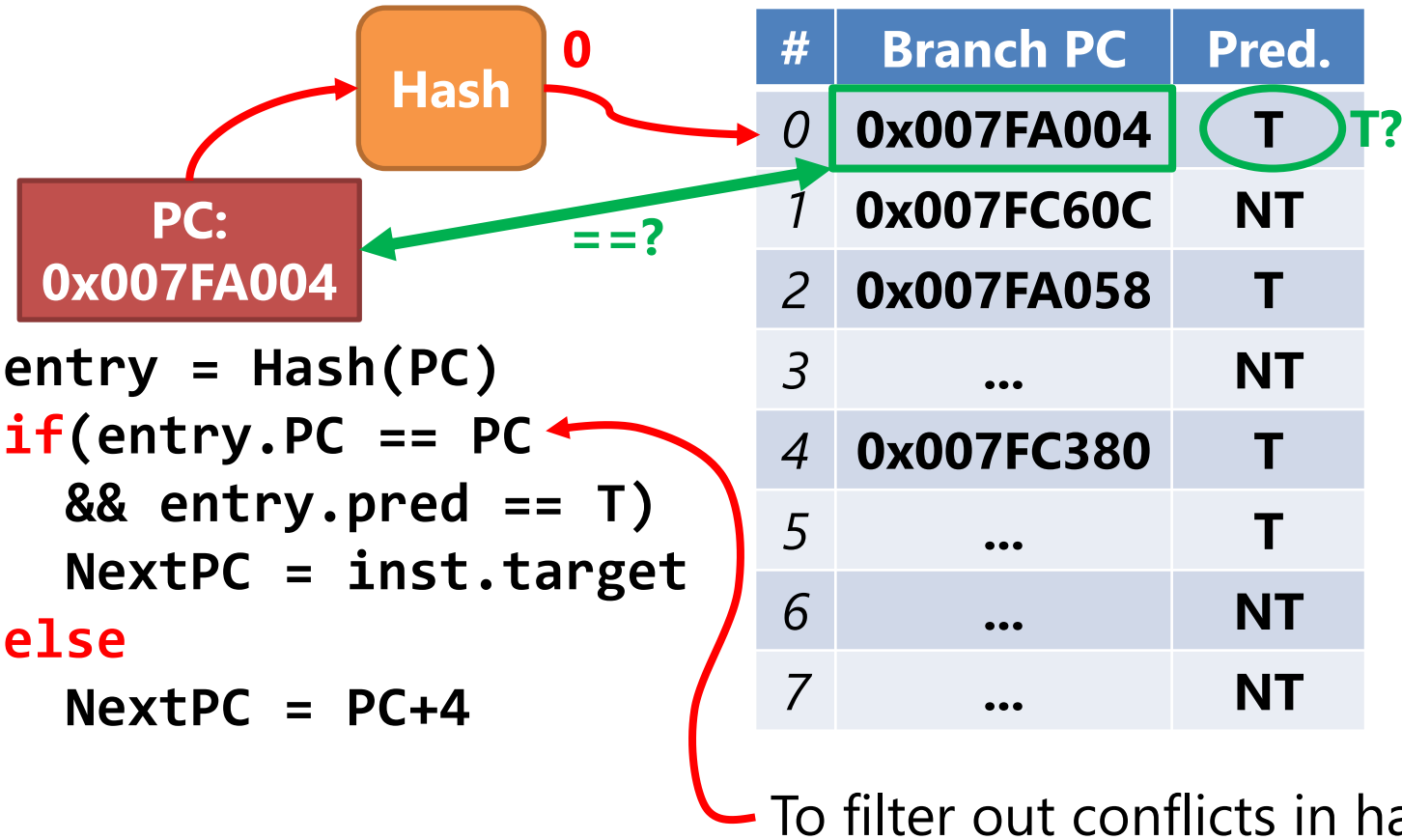  - Not used nowadays due to inaccuracy of compiler predictions

- ***Dynamic Branch Prediction***
  - Predicting branch behavior based on (dynamic) branch history
  - Typically using **hardware** that tracks history information
  - *Premise*: **history repeats itself**
    - Branches not taken in the past → likely not taken in the future (e.g. branches to error handling code)
    - Branches taken in the past → likely taken in the future (e.g. branch back to the next iteration of the loop)

University of Pittsburgh

- BHT stores Taken (**T**) or Not Taken (**NT**) history info for each branch
  o If branch was taken most recently, **T** is recorded
  o If branch was not taken most recently, **NT** is recorded

- BHT is indexed using PC (Program Counter)
  o Each branch has a unique PC, so a unique entry per branch

- BHT, being hardware, is limited in capacity
  o Cannot have a huge table with all PCs possible in a program
  o Besides, not every PC address contains a branch
  o Best to use **hash table** to map branch PCs to (limited) entries

Hash

**0**

PC: 0x007FA004

**==?**

| # | Branch PC | Pred. |
|---|-----------|-------|
| 0 | **0x007FA004** | T  T? |
| 1 | **0x007FC60C** | NT |
| 2 | **0x007FA058** | T |
| 3 | ... | NT |
| 4 | **0x007FC380** | T |
| 5 | ... | T |
| 6 | ... | NT |
| 7 | ... | NT |

```
entry = Hash(PC)
if(entry.PC == PC
   && entry.pred == T)
   NextPC = inst.target
else
   NextPC = PC+4
```
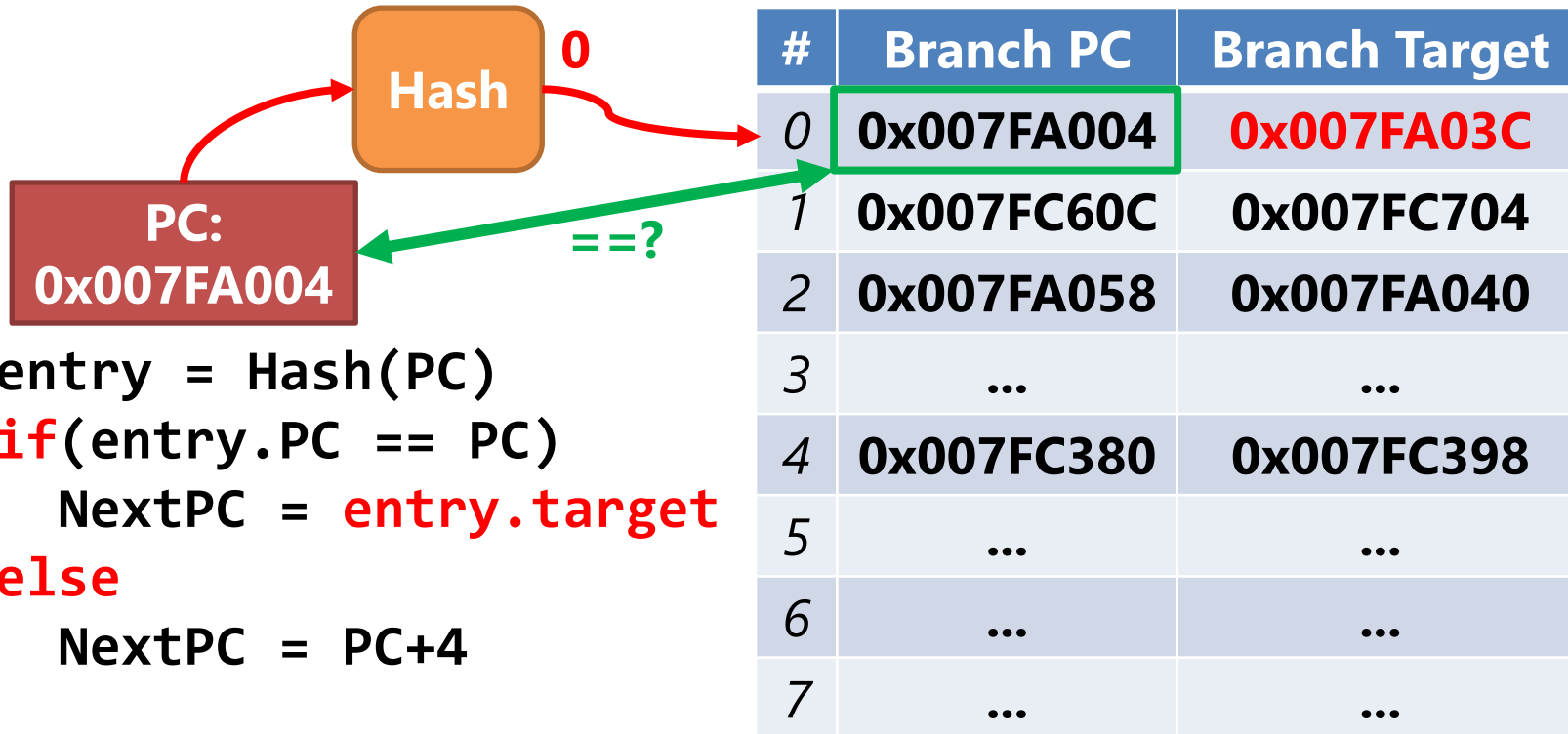
To filter out conflicts in hash function

University of Pittsburgh

- Ideally, we would like know what next to fetch at the **IF** stage
  - So that correct instruction is immediately fetched in next cycle

- BHT can give us branch direction **IF** stage
  - All the information needed is the PC (which is available at IF)

- But also need the **branch target** to know what to fetch
  - Must wait until the **ID** stage for branch target to be decoded
  - If **NT** in BHT: no need to wait (branch target is irrelevant)
    But if **T** in BHT: need to wait until ID stage

- That introduces a **bubble** for **taken branches**
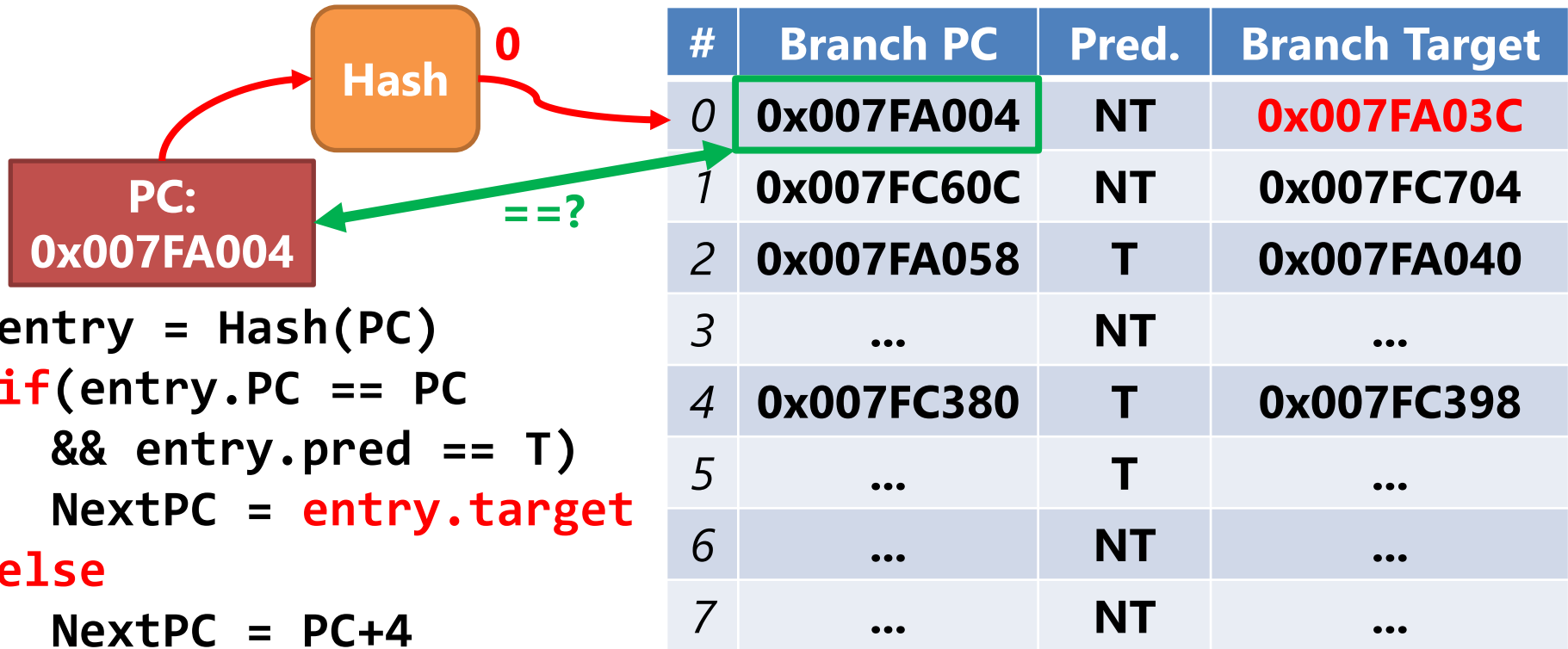
- BTB stores **branch target** for each branch

- BTB is also indexed using PC of branch using a hash table

- BTB allows branch target to be known at the IF stage
  - **No need to wait until ID stage** for branch target to be decoded

```
entry = Hash(PC)
if(entry.PC == PC)
    NextPC = entry.target
else
    NextPC = PC+4
```

| # | Branch PC | Branch Target |
|---|-----------|---------------|
| 0 | **0x007FA004** | **0x007FA03C** |
| 1 | **0x007FC60C** | **0x007FC704** |
| 2 | **0x007FA058** | **0x007FA040** |
| 3 | ... | ... |
| 4 | **0x007FC380** | **0x007FC398** |
| 5 | ... | ... |
| 6 | ... | ... |
| 7 | ... | ... |

University of Pittsburgh

Hash **0**

PC:
0x007FA004

==?

```
entry = Hash(PC)
if(entry.PC == PC
   && entry.pred == T)
   NextPC = entry.target
else
   NextPC = PC+4
```

| # | Branch PC | Pred. | Branch Target |
|---|-----------|-------|---------------|
| 0 | 0x007FA004 | NT | 0x007FA03C |
| 1 | 0x007FC60C | NT | 0x007FC704 |
| 2 | 0x007FA058 | T | 0x007FA040 |
| 3 | … | NT | … |
| 4 | 0x007FC380 | T | 0x007FC398 |
| 5 | … | T | … |
| 6 | … | NT | … |
| 7 | … | NT | … |

University of
**Pittsburgh**

Assuming that branch condition and target are resolved in ID stage
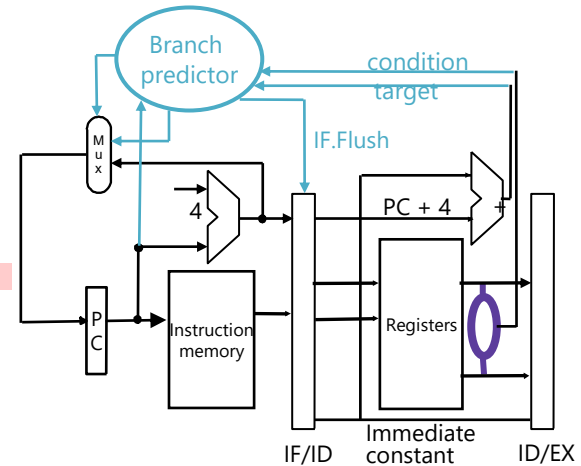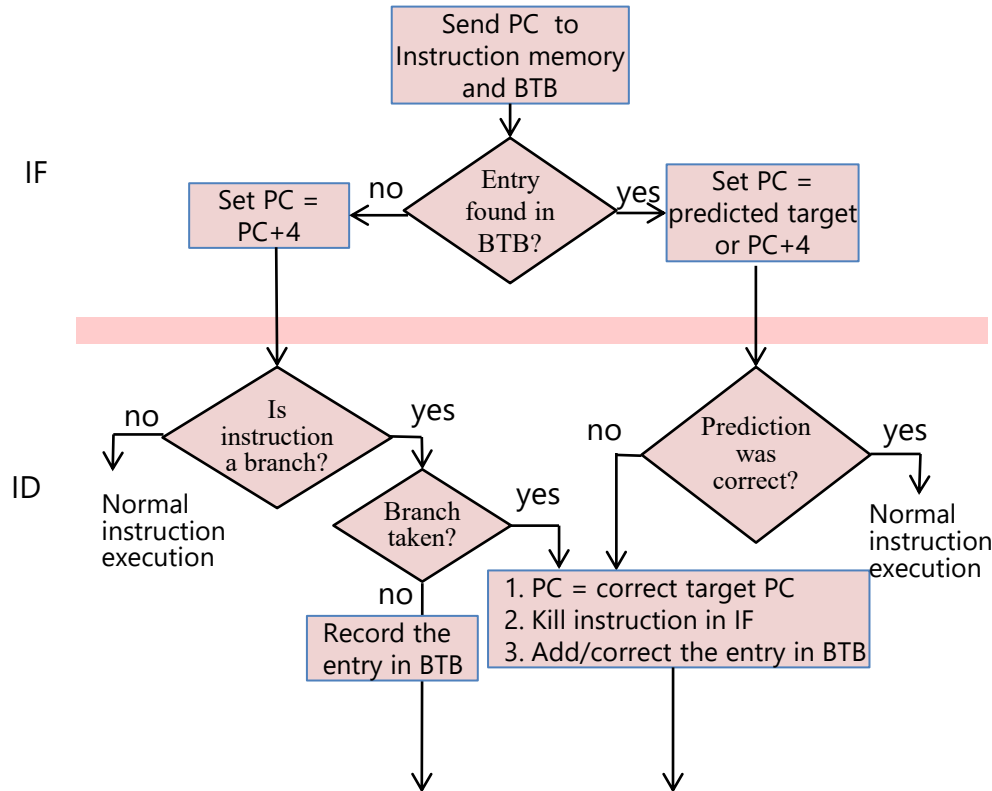
- Is 1-bit (T / NT) enough history to make a good decision?

- Take a look at this example:

```
for (j=0; j<100; j++) {
  for (i=0; i< 5; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
  }
}
```

| Predicted | - | T | T | T | **T** | **NT** | T | T | T | **T** | **NT** | T | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Actual | T | T | T | T | NT | T | T | T | T | NT | T | T | T |

this branch is predicted wrong twice every inner loop invocation (every 5 branches)

- It would have been better to stay with T than flip back and forth!

- Idea behind the 2-bit predictor: make predictions more stable
  - So that predictions don't flip immediately

- State transition diagram of 2-bit predictor:



change in prediction
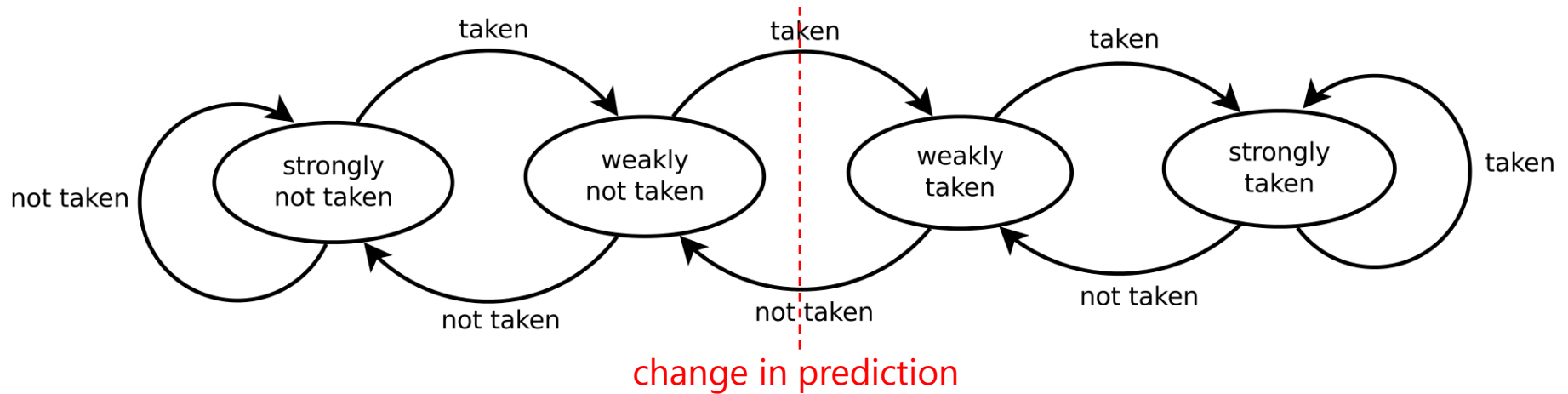
- Can be implemented using a 2-bit saturating counter
  - Strongly not taken: 00
  - Weakly not taken: 01
  - Weakly taken: 10
  - Strongly taken: 11

# 2-bit BHT Predictor

- How well does the 2-bit predictor do with our previous example?

- Our previous example:

```
for (j=0; j<100; j++) {
    for (i=0; i< 5; i++) {
        A[i] = B[i] * C[i];
        D[i] = E[i] / F[i];
    }
}
```

| | Weakly Taken | Strongly Taken | Strongly Taken | Strongly Taken | Weakly Taken | Strongly Taken | Strongly Taken | Strongly Taken | Strongly Taken | Weakly Taken | Strongly Taken | Strongly Taken |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Predicted** | - | T | T | T | **T** | T | T | T | T | **T** | T | T |
| **Actual** | T | T | T | T | NT | T | T | T | T | NT | T | T |

this branch is predicted wrong **only once** every inner loop invocation (every 5 branches)

- Does it help beyond 2 bits? (e.g. 3-bit predictor, or 4-bit predictor)
  - Empirically, no. 2 bits already cover loop which is most common.
  - 2 bits + large BHT gets you **~93% accuracy**

- We need other tricks to improve accuracy!

University of Pittsburgh

- Here is an example where 1-bit BHT predictor fails miserably

```
for (j=0; j<100; j++) {
  if (j % 2) {
  }
}
```

You get the prediction **wrong every single time**!

| Predicted | - | **NT** | **T** | **NT** | **T** | **NT** | **T** | **NT** | **T** | **NT** | **T** | **NT** | **T** |
|-----------|---|--------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|-------|
| Actual | NT | T | NT | T | NT | T | NT | T | NT | T | NT | T | NT |

- And a 2-bit predictor doesn't do very well either

```
for (j=0; j<100; j++) {
  if (j % 2) {
  }
}
```

You get the prediction **wrong every other time**!

| Predicted | - | **NT** | NT | **NT** | NT | **NT** | NT | **NT** | NT | **NT** | NT | **NT** | NT |
|-----------|---|--------|----|--------|----|--------|----|--------|----|--------|----|--------|----|
| Actual | NT | T | NT | T | NT | T | NT | T | NT | T | NT | T | NT |

  o Would a 3-bit predictor do any better?
- Idea: Base prediction on a **pattern** found in history of branches!
  o Rather than relying on a single prediction for a branch
  o If History: T → predict NT, if History: NT → predict T

University of **Pittsburgh**

- **Correlating Predictor**: Uses patterns in past branches for prediction
  - o Often branch behavior more complex than just taken or not taken
  - o Often correlates to a pattern of past branches

- Pattern may exist in two ways:
  - o Pattern in **local** branch history (history of only current branch)
  - o Pattern in **global** branch history (history of all branches)

- Maintaining longer history allows detection of longer patterns
  - o Local branch history for each branch maintained at all times
  - o One global branch history maintained at all times

University of Pittsburgh

- With a local branch history of 1, can predict perfectly!

```
for (j=0; j<100; j++) {
  if (j % 2) {
  }
}
```

Predict with branch PC + 1 local branch history

| Predicted | - | - | - | T | NT | T | NT | T | NT | T | NT | T | NT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Actual | NT | T | NT | T | NT | T | NT | T | NT | T | NT | T | NT |

- Local branch history changes as such:
  - **NT** → **T** → **NT** → **T** → **NT** → **T** → **NT** → **T** → **NT** → **T** → **...**
- Prediction based on branch PC and local branch history:
  - PC: if (j % 2) + History: **NT**          → Prediction: **T**
  - PC: if (j % 2) + History: **T**          → Prediction: **NT**

- You need a local branch history of 2 for this one.

```
for (j=0; j<100; j++) {
  if (j % 3) {
  }
}
```

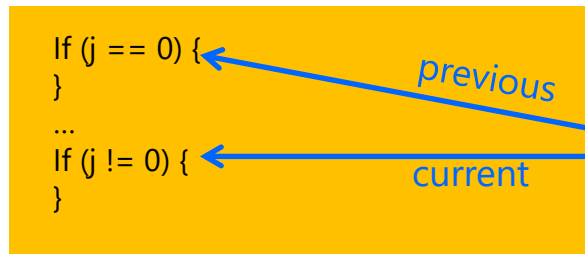Predict with branch PC + 2 local branch history

| Predicted | - | - | - | - | - | T | NT | T | T | NT | T | T | NT |
|-----------|----|----|----|-----|----|----|-----|----|----|-----|----|----|-----|
| Actual | NT | T | T | NT | T | T | NT | T | T | NT | T | T | NT |

- Local branch history changes as such:
  - **NT, T → T, T → T, NT → NT, T → T, T → T, NT → NT, T → ...**
- Prediction based on branch PC and local branch history:
  - PC: if (j % 3) + History: **NT, T**       → Prediction: **T**
  - PC: if (j % 3) + History: **T, T**         → Prediction: **NT**
  - PC: if (j % 3) + History: **T, NT**       → Prediction: **T**
  - PC: if (j % 3) + History: **NT, NT**      → No prediction

- Knowing the result of other branches in your history also helps

```
If (j == 0) {
}
...
If (j != 0) {
}
```
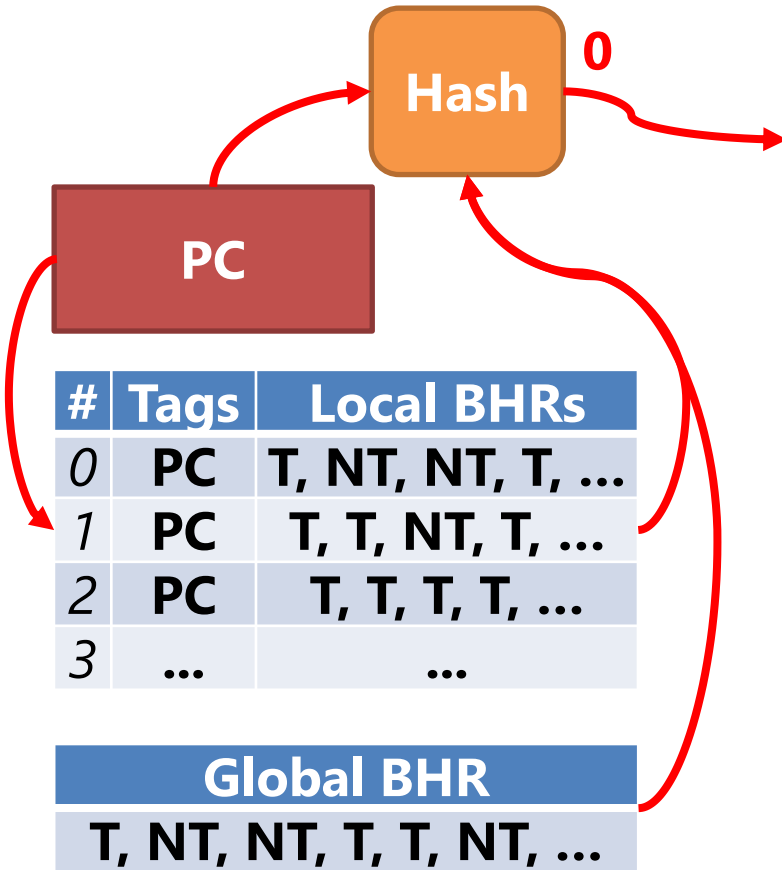
*previous*

*current*

Knowing result of a **previous different branch** in your history helps in predicting **current** branch!

- This is called **global branch history** (involves all branches).

- Can be helpful when local branch history can't capture pattern.

University of Pittsburgh

- **Correlates** prediction with branch **history** as well as branch PC
  - o Local branch history + Global branch history
  - o An entry with matching history gives more precise prediction!

- Now, instead of indexing into BHT by branch PC only
  - o Use hash(PC, Local branch history, Global branch history)

- History is stored in register called Branch History Shift Register (BHR)
  - o T/NT bit is shifted on to BHR whenever branch is encountered
  1. One Global BHR (there is just one global history)
  2. Multiple Local BHRs (local histories for each branch PC)

# Correlating Predictors



| # | Tags | Local BHRs |
|---|------|-----------|
| 0 | PC | T, NT, NT, T, ... |
| 1 | PC | T, T, NT, T, ... |
| 2 | PC | T, T, T, T, ... |
| 3 | ... | ... |

**Global BHR**

**T, NT, NT, T, T, NT, ...**

| # | Tags | Pred. | Branch Target |
|---|------|-------|---------------|
| 0 | PC+History | 01 | 0x007FA03C |
| 1 | PC+History | 00 | 0x007FC704 |
| 2 | PC+History | 11 | 0x007FA040 |
| 3 | ... | 01 | ... |
| 4 | PC+History | 10 | 0x007FC398 |
| 5 | ... | 00 | ... |
| 6 | ... | 10 | ... |
| 7 | ... | 11 | ... |

- Can reach up to **97% accuracy!**

- **jal funcAddr**: function call
  o Stores PC+4 to **$ra** and jumps to funcAddr

- **jr $ra**: function return
  o Jumps to function return address stored in **$ra**

- **jr $ra** makes life difficult for the BTB
  o Unlike other branches, branch target is not an immediate value! (Jumping to a variable target is called an ***indirect branch***)
  o **$ra** can change dynamically depending on call site
  o BTB which relies on jump target being constant

- Target of **jr** is predicted using the ***Return Stack Buffer***
  o Not the Branch Target Buffer (BTB)

- Since functions return to where they were called every time, it makes sense to cache the return addresses in a stack

```
4AB33C jal someFunc
4AB340 beq v0, $0, blah
       ...
       someFunc:
            ...
          jr $ra
```

When we encounter the jal, push the return address.

When we encounter the jr $ra, pop the return address. Easy!

| 40CC00 |
| 46280C |
| 4AB108 |
| 4AB340 |
| 000000 |
| 000000 |
| 000000 |
| 000000 |

- On misprediction or stack overflow, empty stack
  - Not a problem since this is for prediction anyway

- Now, $CPI = CPI_{nch} + \alpha * \pi * K$
  - $CPI_{nch}$ : CPI with no control hazard
  - $\alpha$ : fraction of branch instructions in the instruction mix
  - $\pi$ : **probability a branch is mispredicted**
  - K : penalty per pipeline flush

- With deep pipelines, mispredictions can have outsized impact

> **Example:** If 20% of instructions are branches and the misprediction rate is 5%, and pipeline flush penalty 20 cycles, then:
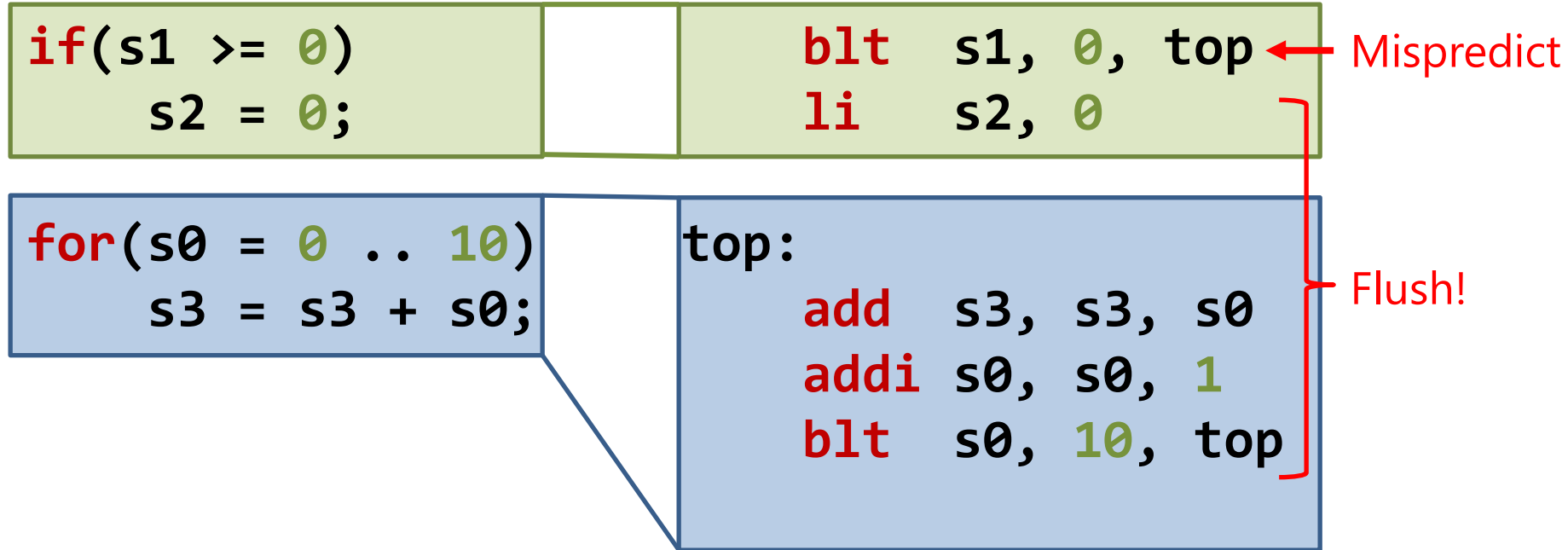>    $CPI = CPI_{nch} + 0.2 * 0.05 * 20 = CPI_{nch} + 0.2$ cycles per instruction

- Problem is a small percentage of hard to predict branches
  - How do we deal with these?

# Predication

# Branch Mispredictions have Outsized Impact

- Assume a deep pipeline and `if(s1 >= 0)` is hard to predict

```
if(s1 >= 0)            blt   s1, 0, top      ← Mispredict
    s2 = 0;            li    s2, 0
```

```
for(s0 = 0 .. 10)    top:
    s3 = s3 + s0;         add   s3, s3, s0
                          addi  s0, s0, 1       Flush!
                          blt   s0, 10, top
```

- On a misprediction, every following instruction is flushed
  - Not only the control dependent instructions (`li s2, 0`)
  - But also multiple iterations of the "bystander" loop that were fetched

University of Pittsburgh

- **Predicate**: a Boolean value used for conditional execution
  - o Instructions that use predicates are said to be **predicated**
  - o A predicated instruction will modify state only if predicate is true
  - o ISA is modified to add predicated versions for all instructions

- Example of code generation using predication:
```
pge  p1, s1, 0      # Store boolean s1 >= 0 to predicate p1
li.p s2, 0, p1      # Assign 0 to s2 if p1 is true
sw.p s3, 0(s4), p1 # Store s3 to address 0(s4) if p1 is true
```

- Now there is no branch.  It is just straight-line code!
  - o Control dependencies have been converted to data dependencies

- Now there are no branches!

```
if(s1 >= 0)                    pge   p1, 0, s1
    s2 = 0;                    li.p  s2, 0, p1
```

```
for(s0 = 0 .. 10)      top:
    s3 = s3 + s0;              add   s3, s3, s0
                               addi  s0, s0, 1
                               blt   s0, 10, top
```
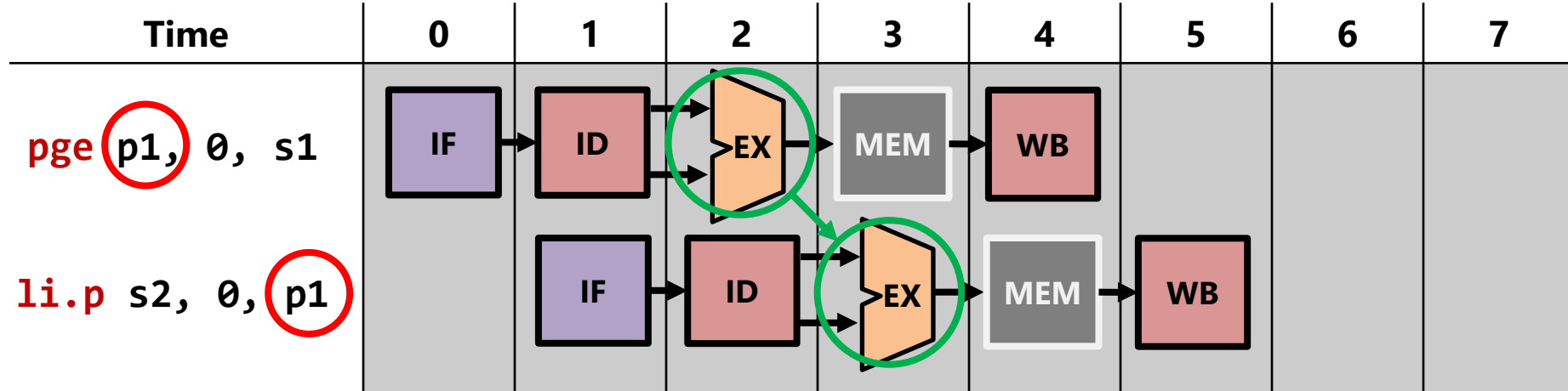
- Drawback: even if branch not taken, `li.p` fetched (acts like a bubble)
  - But often worth it for hard to predict branches!
  - For easy to predict branches, often not worth it.

University of Pittsburgh

# What does predication mean for the pipeline?
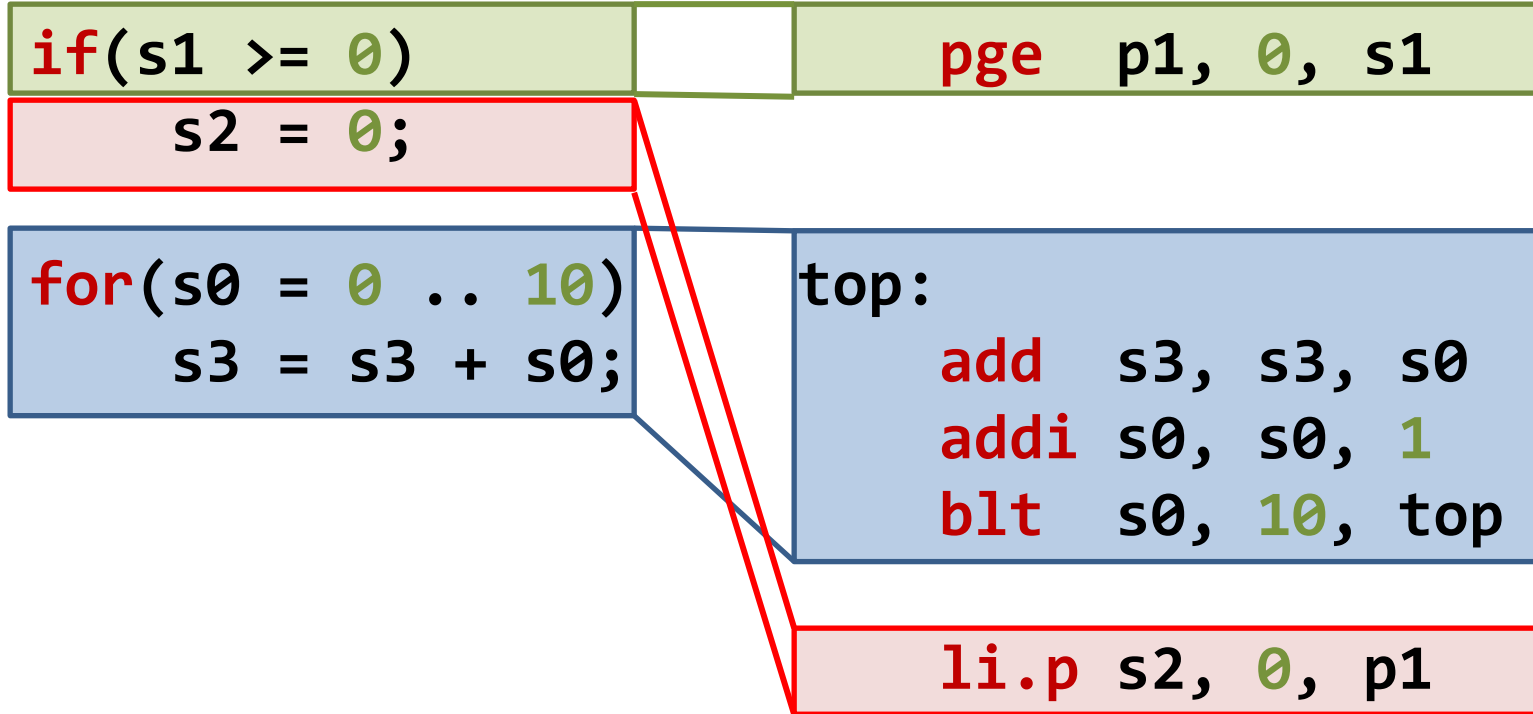
- Again, predicates are registers just like any other register
- Predicate dependencies work just like other data dependencies

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| pge p1, 0, s1 | IF | ID | EX | MEM | WB | | | |
| li.p s2, 0, p1 | | IF | ID | EX | MEM | WB | | |

- With data forwarding, no stalls required!
  - Predicate forwarded to `li.p` EX stage
  - Later predicate enables/disables regwrite control in `li.p` WB stage

University of Pittsburgh

- Compiler can schedule instruction more freely!

| | | |
|---|---|---|
| `if(s1 >= 0)` | | `pge   p1, 0, s1` |

```
      s2 = 0;
```

| | |
|---|---|
| `for(s0 = 0 .. 10)` | `top:` |
| `   s3 = s3 + s0;` | `    add   s3, s3, s0` |
| | `    addi  s0, s0, 1` |
| | `    blt   s0, 10, top` |

```
          li.p s2, 0, p1
```

- Low-power compiler-scheduled processors often support predicates

# Predication in the Real World

- Predication is only beneficial for hard to predict branches

- So how does the compiler figure out the hard to predict branches?
  - Through code analysis
  - Through software profiling (model a branch predictor)

- Supported in various ISAs
  - ARM allows most instructions to be predicated
  - Intel x86 has conditional move instructions (cmov)
  - SIMD architectures use predication in the form of a logical mask
    - Only data items that are not masked are updated
    - Intel AVX vector instructions
    - GPU instructions (e.g. CUDA)