Review

- Last Class:
  - Register renaming to resolve false dependences.
- Today's class:
  - Data hazard continued
  - Branch prediction
- Announcement and reminder
  - Hw-1 will be distributed today on canvas.
  - Hw-1 dues Feb 9th 11:59pm. Upload your answers to canvas.
  - I will hold **virtual** office hours during February. Same office hour zoom link on canvas.

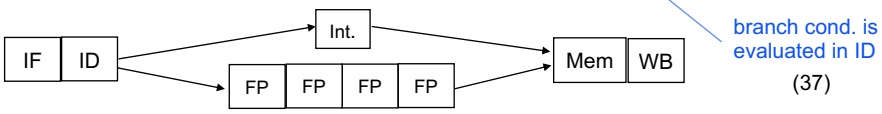
(36)

The effect of data dependencies (hazards)

- Consider the following loop, which adds a constant to the elements of an array.

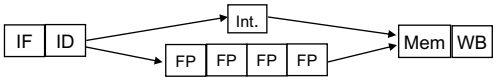
```
Loop:  fld     F0, 0(R1)     ; Load an element
       fadd.d  F4, F0, F2    ; add a scalar, in F2, to the element
       fsd     F4, 0(R1)     ; store result
       addi    R1, R1, #-8   ; update loop index, R1
       bne     R1, R2, Loop  ; branch if have not visited all elements
```

- Assume a MIPS pipeline with the following latencies (among others)
  - Latency = 3 cycles if an **FP ALU op** follows and depends on an **FP ALU op**.
  - Latency = 2 cycles if an **FP store** follows and depends on an **FP ALU op**.
  - Latency = 1 cycle if an **FP ALU op** follows and depends on an **FP load**.
  - Latency = 1 cycle if a **branch** follows and depends on an **Integer ALU op.**



(37)

Pipeline stalls due to data hazards




```
Loop:   fld      F0, 0(R1)      ; Load an element
        stall
        fadd.d   F4, F0, F2     ; add a scalar to the array element
        stall
        stall
        fsd      F4, 0(R1)      ; store result
        addi     R1, R1, #-8    ; update loop index, R1
        stall
        bne      R1, R2, Loop   ; branch if have not visited all elements
```

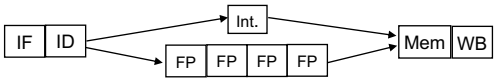
9 clock cycles per iteration

- Latency = 3 cycles if an **FP ALU op** follows and depends on an **FP ALU op**.
- Latency = 2 cycles if an **FP store** follows and depends on an **FP ALU op**.
- Latency = 1 cycle if an **FP ALU op** follows and depends on an **FP load**.
- Latency = 1 cycle if a **branch** follows and depends on an **Integer ALU op**.

(38)

Basic compiler techniques for exposing ILP (§3.2)

Reorder the statements :




```
Loop:   fld      F0, 0(R1)
        addi     R1, R1, #-8
        fadd.d   F4, F0, F2
        stall
        stall
        fsd      F4, 8(R1)
        bne      R1, R2, Loop
```

Note the change from 0 to 8

7 clock cycles per iteration

- Latency = 3 cycles if an **FP ALU op** follows and depends on an **FP ALU op**.
- Latency = 2 cycles if an **FP store** follows and depends on an **FP ALU op**.
- Latency = 1 cycle if an **FP ALU op** follows and depends on an **FP load**.
- Latency = 1 cycle if a **branch** follows and depends on an **Integer ALU op**.

(39)

Loop Unrolling (assume no pipelining)

```
Loop:  fld     F0, 0(R1)
       fadd.d  F4, F0, F2
       fsd     F4, 0(R1)
       addi    R1, R1, #-8
       fld     F0, 0(R1)
       fadd.d  F4, F0, F2
       fsd     F4, 0(R1)
       addi    R1, R1, #-8
       bne     R1, R2, Loop
```

Save 0.5 instruction per iteration

```
Loop:  fld     F0, 0(R1)
       fld     F6, -8(R1)
       fadd.d  F4, F0, F2
       fadd.d  F8, F6, F2
       addi    R1, R1, #-16
       fsd     F4, 16(R1)
       fsd     F8, 8(R1)
       bne     R1, R2, Loop
```

Save 1 instruction per iteration

- Need to worry about boundary cases (strip mining??)
- Can reorder the statements if we use additional registers.
- What limits the number of times we unroll a loop?
- Note that loop iterations were independent

  for i = 1, 100

  x(i) = x(i) + c

Executing the unrolled loop on a pipeline

- Latency = 3 cycles if an **FP ALU op** follows and depends on an **FP ALU op**.
- Latency = 2 cycles if an **FP store** follows and depends on an **FP ALU op**.
- Latency = 1 cycle if an **FP ALU op** follows and depends on an **FP load**.
- Latency = 1 cycle if a **branch** follows and depends on an **Integer ALU op**.

```
Loop:  fld     F0, 0(R1)
       fld     F6, -8(R1)
       fadd.d  F4, F0, F2
       fadd.d  F8, F6, F2
       addi    R1, R1, #-16
       fsd     F4, 16(R1)
       fsd     F8, 8(R1)
       bne     R1, R2, Loop
```

Potential problem if one cycle is required between integer op. and store (see comment bellow).

4 (or 4.5) clock cycles per iteration

- What can you do if there is no forwarding path from the int/Mem buffer to the ID/int buffer?
  - Stall for one cycle
  - Replace 16(R1) in the first fsd statement by 0(R1). (safe?)
  - Delayed branch (2 delayed slots)?
- What do you do if the latency = 3 cycles when “fsd” follows “fadd.d”?

Control dependences

- Determine the order of instructions with respect to branches.

if P1 then S1 ;
if P2 then S2 ;

S1 is control dependent on P1 and
S2 is control dependent on P2 (and P1 ??).

- An instruction that is control dependent on *P* cannot be moved to a place where it is no longer control dependent on *P* , and visa-versa

Example 1:

```
    add   R1,R2,R3
    beq   R4, R0, L
    sub   R1,R1,R6
L:  …
    or    R7,R1,R8
```

"or" instruction depends on the execution flow

Example 2:

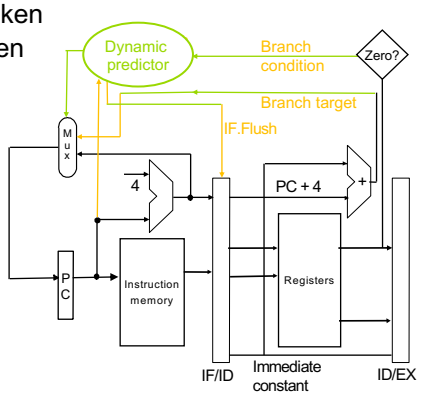
```
    add   R1,R2,R3
    beq   R12, R0, skip
    sub   R4,R5,R6
    add   R5,R4,R9
skip:
    or    R7,R8,R9
```

Possible to move "sub" before the branch (if R4 is not used after skip)

(42)

Branch prediction (§3.3)

- Static branch prediction (built into the architecture)
  - ➢The default is to assume that branches are not taken
  - ➢May have a design which predicts that branches are taken
- It is reasonable to assume that
  - ➢forward branches are often not taken
  - ➢backward branches are often taken
- May come up with more static predictors based on branch directions.
- Profiling is the standard technique for predicting the probability of branching.
- Dynamic predictors rely on the history to predict the future branch direction



(43)

Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – *always* predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
   - If taken, flush instructions after the branch (earlier in the pipeline, later in the code)
     - in IF, ID, and EX stages if branch logic in MEM – three stalls
     - In IF and ID stages if branch logic in EX – two stalls
     - in IF stage if branch logic in ID – one stall
   - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
   - restart the pipeline at the branch destination

(44)

Static Branching Structures

- Always predict NT works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead every time through the loop

```
Loop: beq $1,$2,Out
      1nd loop instr
            .
            .
            .
      last loop instr
      j  Loop
Out:  fall out instr
```

- Always predict NT doesn't work well for "bottom of the loop" branching structures
  - Guess wrong every time through the loop except the last time (when we fall out of the loop)

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

(45)

0-Bit Predictors, Loop for 10 Iterations

iteration	predict	actual	predict	actual
1	NT	T	T	T
2	NT	T	T	T
3	NT	T	T	T
4	NT	T	T	T
5	NT	T	T	T
6	NT	T	T	T
7	NT	T	T	T
8	NT	T	T	T
9	NT	T	T	T
10	NT	NT	T	NT
	10% accuracy		90% accuracy	

```
  pre-loop instr
Loop:
  1st loop instr
  2nd loop instr
        .
        .
        .
  last loop instr
  bne $1,$2,Loop
  post-loop instr
```

(46)

Other Branch Prediction Possibilities

2. Predict taken (T) – predict branches will always be taken
   - BUT predict taken *always* incurs one stall cycle (*if* branch destination hardware has been moved to the ID stage)
   - Is there a way to "cache" the address of the branch target instruction, or *better yet* the actual branch target instruction itself ?? Yes!

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution

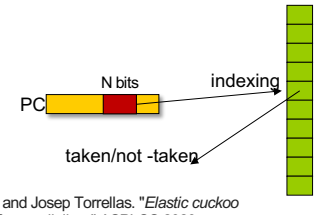
3. Dynamic branch prediction – predict branches at run-time using *run-time* information

Dynamic Branch Prediction

- Different than attaching a prediction with each branch
- Performance depends on the accuracy of prediction and the cost of miss-prediction.

- **Branch prediction buffer (Branch history table - BHT)**:
  - 1-bit table (cache) indexed by some bits of the address of the branch instructions (can be accessed in decode stage) → hashing
  - Record whether or not the branch was taken last time
  - May have collision (how to resolve? Cuckoo hashing, overheads?).
  - Will cause two miss-predictions in a loop (at start and end of loop).

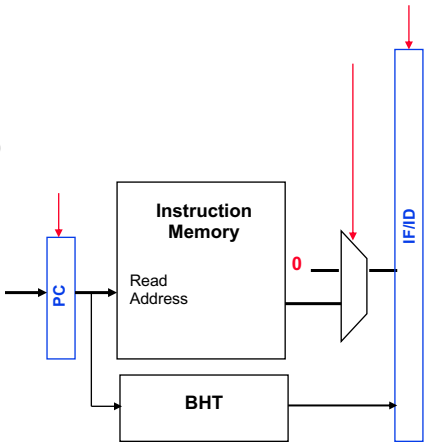
```
L1:  .......
L2:  .......
     .......
     beqz   R2, L2
     .......
     beqz   R1, L1
```



Skarlatos, Dimitrios, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. "*Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism.*" ASPLOS 2020
Bingyao Li, Jieming Yin, Youtao Zhang, Xulong Tang "*Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design.*" MICRO 2021

Dynamic Branch Prediction Buffer

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the low order bits of the PC, contains bit(s) (passed to the ID stage through the IF/ID pipeline register) that tell whether the branch was taken or not the last time it was executed
- 4,096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
- 4,096 about as good as infinite table, but 4,096 is a lot of hardware

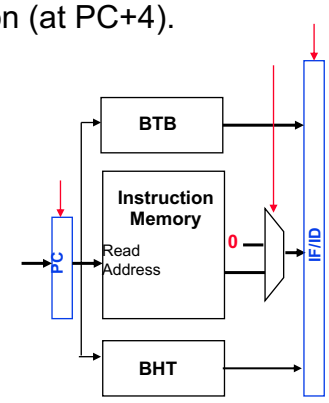


Branch History Table

- The BHTs prediction bits may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but that doesn't affect correctness, just performance
  - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
  - Problems? (side-channel security!)
  - Evtyushkin, Dmitry, Ryan Riley, Nael CSE and ECE Abu-Ghazaleh, and Dmitry Ponomarev. "Branchscope: A new side-channel attack on directional branch predictor." ACM SIGPLAN Notices 53, no. 2 (2018): 693-707.
  - Abu-Ghazaleh, Nael, Dmitry Ponomarev, and Dmitry Evtyushkin. "How the spectre and meltdown hacks really worked." IEEE Spectrum 56, no. 3 (2019): 42-49.
- If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)

But Also Must Know the Branch Target

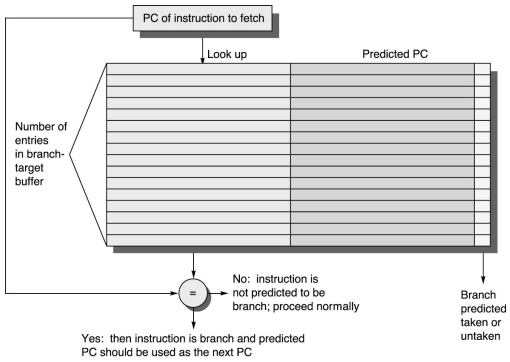
- A branch target buffer (BTB) in the IF stage can cache the branch target address, but remember we also need to fetch the next sequential instruction (at PC+4).
  - Would need a two read port IM
- Or, the BTB can cache the actual branch target instruction while the Instruction Memory is fetching the next sequential instruction
  - ID stage can then select between PC+4 and branch target instruction



- If we predict correctly, stalls can be avoided no matter which direction the branch goes

Branch target buffers(from §3.9)

- Store the address of the branch's target, in addition to the prediction.

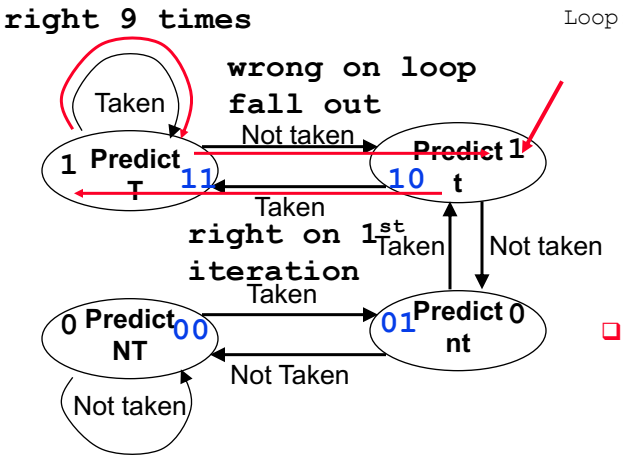


- Can determine the target address while fetching the branch instruction
- Offset or absolute PC?
- Can we use partial bits in the PC?
- how do you even know that the instruction is a branch?
  - BAC (branch address calculator) in ID stage to update the BTB (Intel)

(52)

2-bit Dynamic Branch Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- The BHT also stores the initial state

2-bits branch predictors

- change your prediction only if you miss-predict twice
- helps if a branch changes directions occasionally (ex. Nested loops)
- In general, *n*-bit predictors are called *Local Predictors*.
  - Use a saturated counter (++ on correct prediction, -- on wrong prediction)
  - n-bit prediction is not much better than 2-bit prediction (n > 2).
  - A BHT with 4K entries is as good as an infinite size BHT
  - Miss-predict when gets the wrong branch in BHT or a wrong prediction.

Correlating Branch predictors (global predictors)

- Hypothesis: recent branches are correlated (behavior of recently executed branches affects prediction of current branch).

- Example 1:

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

```
        addi   R3,R1, -2
        bnez   R3, L1 …      <= B1
        add    R1, R0, R0
L1:     addi   R3, R2, -2
        bnez   R3, L2        <= B2
        add    R2, R0, R0
L2:     sub    R3, R1, R2
        beqz   R3, L3        <= B3
```

If B1 is not taken and B2 is not taken, then B3 will be taken
If B1 and B2 are taken, then B3 will probably not be taken,

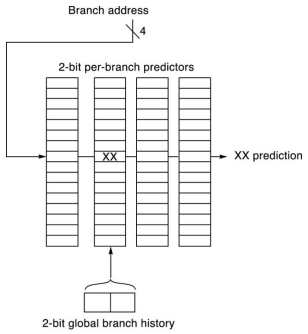
- Example 2:

```
If (d == 0) d = 1 ;
if (d == 1) .....
```

(56)

Correlating Branch predictors

- Keep history of the *m* most recently executed branches in an *m*-bit shift register.
- Record the prediction for each branch instruction, and each of the 2^m combinations.
- In general, (*m,n*) predictor means record last *m* branches to select between 2^m history tables each with *n*-bit predictor.
- Simple access scheme (double indexing).
- A (*0,n*) predictor is a local *n*-bit predictor.
- Size of table is $N \; n \; 2^m$, where *N* is the number of table entries.
- There is a tradeoff between *N* (determines collision), *n* (accuracy of local predicion) and *m* (determines history).

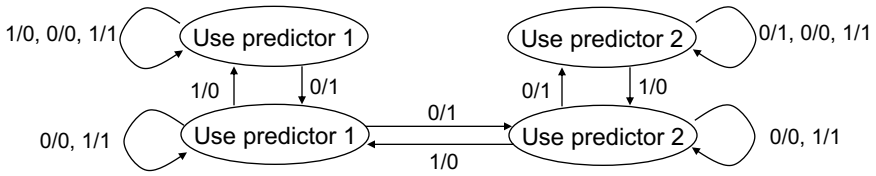


A (2,2) predictor

(57)

Tournament predictor (hybrid local-global predictors)

- Combines a global predictor and a local predictor with a strategy for selecting the appropriate predictor (multilevel predictors).

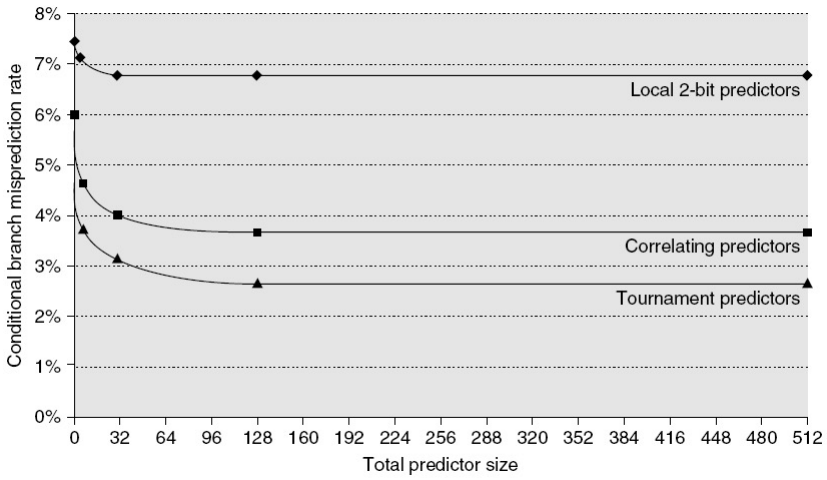


p_1/p_2 == predictor 1 is correct/predictor 2 is correct

- The Alpha 21264 selects between
  - a (12,2) global predictor with 4K entries
  - a local predictor which selects a prediction based on the outcome of the last 10 executions of any given branch.
- Weights for the closer branches?

(58)

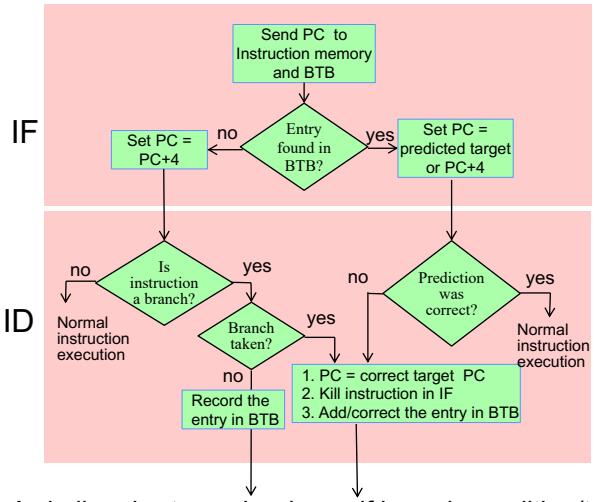
Performance of Branch predictors

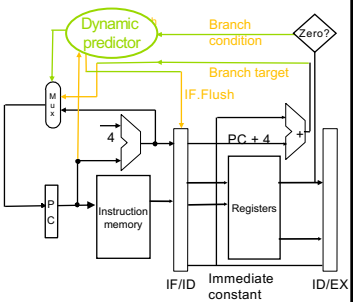


(59)

Branch prediction & pipelining

Assuming that branch condition and target are resolved in ID stage





A similar chart may be drawn if branch condition/target are resolved in EX

(60)

Evaluation example:

- Assume
  - access branch target buffer in IF stage
  - branch condition determined in ID
  - branch address determined in EX stage

- What is the branch penalty if:
  - penalty for correct prediction = 0 cycle
  - penalty for wrong prediction = 1 (or 2) cycles for non-taken (or taken) branch (assuming that target is not stored in BTB if “predict not taken”)
  - penalty if cannot predict and the branch is taken = 2 cycles
  - branch taken frequency = 60%
  - BTB hit rate = 80% (assume not taken in case of inability to predict)
  - BTB prediction accuracy = 90%
  - What is the presentation of the correct instruction is fetched?

- May store the target instruction and not only the address - useful when access of table needs more than one cycle.(61)