CS x290/ ECE x100

Pipelining

Spring 2020

1

New (tiny) ISA for example

- Fixed 32-bit encoding per instruction
  - Next PC = PC+4 is always the next instruction
- 32, 32-bit registers, plus PC and IR
- Arithmetic (e.g., ADD, AND, OR, SUB, etc):

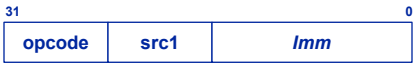
  - Format: ADD R*dest*, R*src1*, R*src2*
  - Regs[dest] = Regs[src1] + Regs[src2]
- Load word:

  - Format: LW R*dest*, *Imm*(R*src1*)
  - Regs[dest] = Memory[Regs[src1] + signextend(Imm)]

3

New (tiny) ISA (cont.)

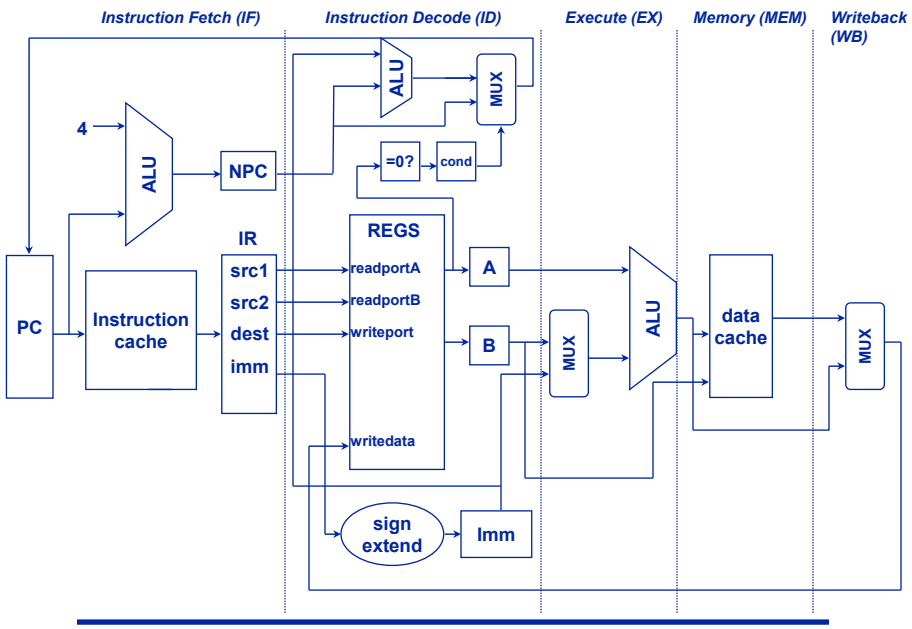
- Store word: same encoding as LW
  - Format: SW R*dest*, *Imm*(R*src1*)
  - *NOTE* Rdest *is a source, etc.!*
  - Mem[Regs[src1] + signextend(Imm)] = Regs[dest]
- Branches



  - Format: B<cond> R*src1*, *Imm*
  - *Example:* BEQZ R3, Label
  - if (Regs[src1] == 0) then
    - PC = NPC + signextend(Imm)

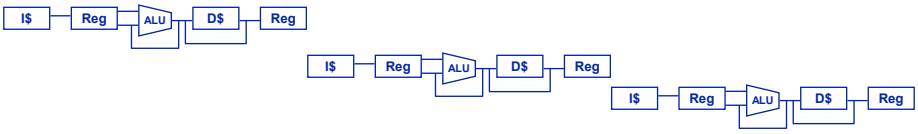
    "Imm" here is sometimes called "Pcoffset"

Now let's build the hardware for this…



Analysis

- All instructions (except branch & store):
  - IF, ID, EX, MEM*, WB
  - Each piece takes 1 cycle
- (*note we don't need MEM for all ops, but we will include it for simplicity and balance)
- CPI = 5
- Graphically (time flowing left to right):

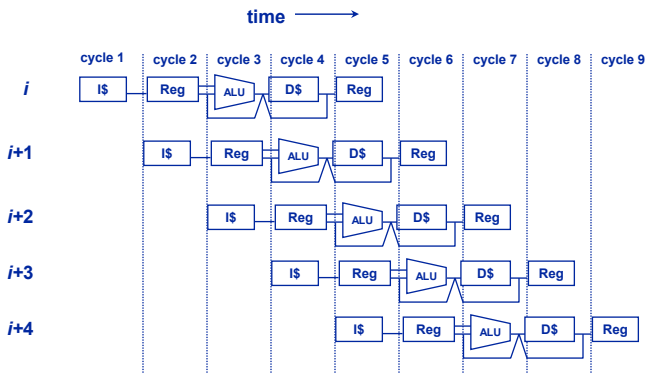


PIPE-6

6

Pipelining speeds this up!

- Isolate each of IF, ID, EX, MEM, WB with latches
- When instruction *i* is in WB, *i*+1 is in MEM, etc.
- Graphically:



PIPE-7

7

Pipeline diagrams

Time:	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
SUB		IF	ID	EX	MEM	WB			
LW			IF	ID	EX	MEM	WB		
ADD				IF	ID	EX	MEM	WB	
SW					IF	ID	EX	MEM	WB

This is a standard way to show pipeline behavior

Pipeline speedup (no stalls)

- For a pipeline of *n* stages for a long program:

$$\text{speedup} = \frac{\text{exec. time unpipelined}}{\text{exec time pipelined}} = \frac{IC * CPI_{unpipe} * CT}{IC * CPI_{pipe} * CT}$$
$$= \frac{CPI_{\text{unpipe}}}{CPI_{\text{unpipe}}/n + T_{\text{latch}}}$$
$$= n \quad (\text{ideal case, } T_{\text{latch}}\text{=0, ignore pipeline fill time})$$

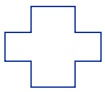
(Detail for computer engineers: The latch (level-triggered) could instead be a flip-flop (edge triggered))

Pipeline hazards

- A hazard reduces the performance of the pipeline
  - A problem caused by both hardware design and the program's characteristics
- Three kinds:
  - Data hazards - Dependencies between instructions prevent their overlapped execution
  - Structural hazards – Not enough hardware resources for all combinations of instructions
  - Control hazards - A branch instruction may change the PC, but not until stage 4. What do we fetch before that?

Program characteristics, Hazards and Stalls

Program characteristics
(e.g., dependencies, location of brances, "my car has bald tires")



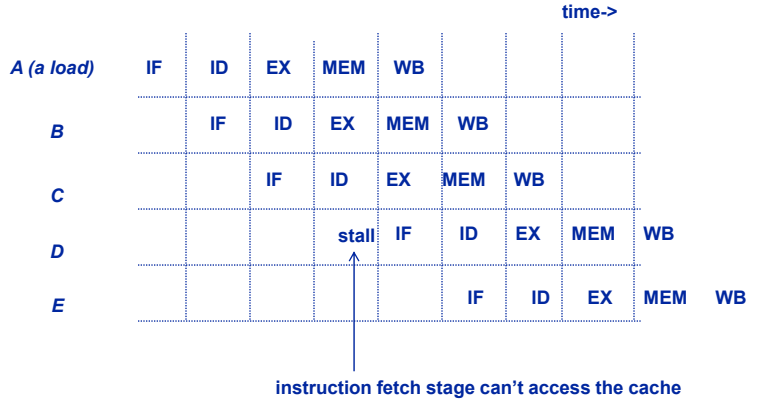
Hardware Hazards
(e.g., control hazards, structural hazards, data hazards, "the roads are icy")

Stalls
(e.g., "bridge freezes before road: my car crashes")

- Not all program characteristics lead to hardware hazards!
- Not all hardware hazards lead to stalls!

Structural hazards

- Uncommon for our pipeline example
- Consider a pipeline with a unified data+instruction cache:



Modeling stalls

$$\text{speedup} = \frac{\text{CPI}_{\text{unpipe}}}{\text{CPI}_{\text{pipe}}}$$

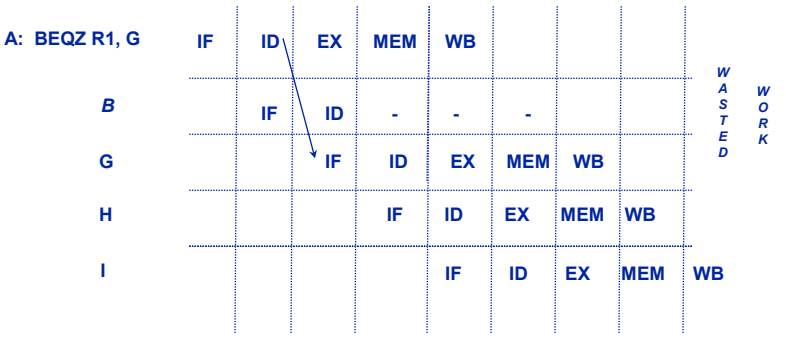
$$\text{CPI}_{\text{pipe}} = \text{CPI}_{\text{nostall}} + (\text{ave. stall cycles per instruction})$$

$$= 1 + (\text{ave. stall cycles per instruction})$$

$$= 1 + \text{stalls}$$ where 'stalls' is an abbrev for ave stall cycles per inst.

$$\text{speedup} = \frac{\text{CPI}_{\text{unpipe}}}{\text{CPI}_{\text{pipe}}} = \frac{n}{1 + \text{stalls}}$$

Control Hazards



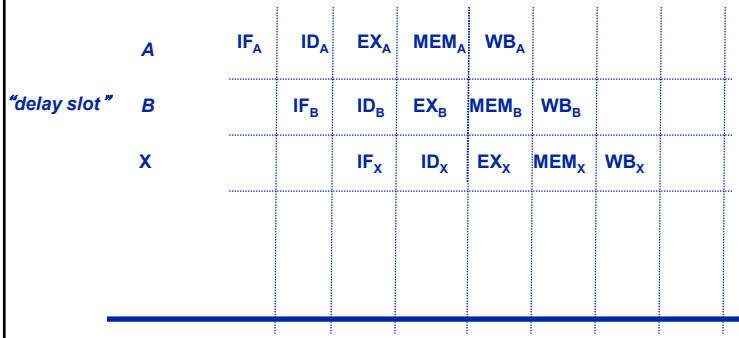
- NPC = PC+4 not always the brightest choice

PIPE-16

16

Reducing branch penalty via the compiler: Delay slots

- Change the meaning of a branch so that next instruction after branch holds something useful



PIPE-17

17

Delay slots

- Add *n* slots to cover *n* holes
- ISA is changed to mean "*n* instructions after any branch are always executed"
- Problem:
  - ISA feature that encodes pipeline structure: Difficult to maintain across generations
  - Typically can fill:
    - 1 slot 75-80% of time
    - 2 slots about 30% of time
    - >2 slots almost never (*fahgettaboutit!*)

GOOD SCENARIO
(branch doesn't depend on ADD)

```
ADD  R4, R2, R3          BEQZ  R1, X
BEQZ  R1, X      ⇨       ADD  R4, R2, R3
NOP
```

BAD SCENARIO
(branch depends on ADD)

```
ADD  R1, R2, R3          ADD  R1, R2, R3
BEQZ  R1, X      ⇨       BEQZ  R1, X
NOP                      NOP
```

PIPE-18

18

"Squash" slots

- If you can't fill slot(s) from before the branch, use instructions from either:
  - Target of branch (if frequently taken)
  - Fall-through of branch (if frequently not-taken)
- Example: fill from target (branch is frequently taken)

```
BEQZ  R1, X                BEQZ  R1, Y      (change target to Y)
NOP                        SUB  R4, R2, R3  (copy of X)
                    ⇨
X:  SUB  R4, R2, R3        X:  SUB  R4, R2, R3
Y:  ...                    Y:  ...
```

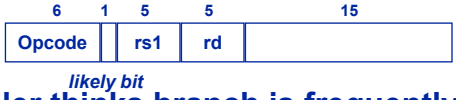
- Disadvantages
  - Only works if slot instruction is safe to execute (ie, no exceptions occur) when branch goes the opposite (infrequent) direction

PIPE-19

19

Squash slots (cont.)

- Each branch instruction now has two opcodes: BEQZ-likely and BEQZ-unlikely



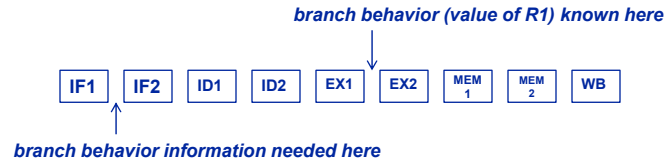
- If compiler thinks branch is frequently taken
  - Compiler sets *likely bit* = 1 (use "BEQZ-likely")
  - Compiler fills "squash slots" from target
  - Hardware knows to squash slot(s) if branch not-taken
- If compiler thinks branch is frequently not-taken
  - Compiler sets *likely bit* = 0 (use "BEQZ-unlikely)
  - Compiler fills "squash slots" from fall-through
  - Hardware knows to squash slot(s) if branch taken
- If likely bit set unintelligently, most squash slot instructions must be squashed!

Methods for setting likely bit

- Likely bit is essentially a *static branch prediction*
  - Compiler makes a prediction that is fixed for that branch
  - Likely bit = 1 means predict taken
  - Likely bit = 0 means predict not-taken
- Static branch prediction methods (compiler branch prediction)
  - Heuristics
    - Common one: Backward taken/Forward not taken
      - Could just use the sign bit of the branch offset=> eliminate likely bit from the instruction encoding!
  - Profiling
    - Run program once, see what the branches do, recompile and set likely bits

What about a deeper pipeline?

hypothetical pipeline of the future: consider BREQZ R1, G



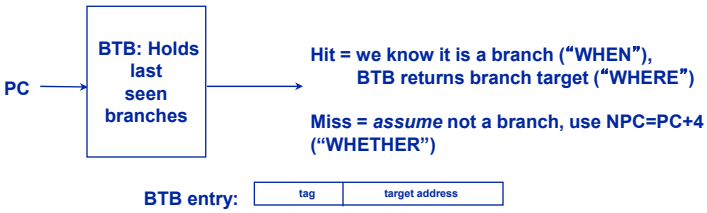
- Next generation processor wants deeper pipeline
  - Old "delay slot" code no longer binary compatible!
- Solution:
  - Do away with delay slots and squashing branches– these go into the dustbin of history!

Reducing Branch Stalls: The 3 W's

- Issues with Whether to branch:
  - (For conditional branches) Predict if it will branch or not, *before* execution
- Issues with Where to branch to:
  - Predicting where the branch will go (if taken)
- Issues with When the branch happens:
  - When to detect an undecoded instruction is a branch
- Optimal: try to determine all three in IF1 stage
  - Won't work perfectly (it's a *prediction*), but we can try our best

Branch Target Buffer

- What does Instruction Fetch stage know about an instruction?
  - Not much! Only the address of the instruction (PC)
  - So…Keep table of last known branch targets around
  - Table is written to by stage that knows the actual branch behavior (e.g., ID2, EX, -- depends on pipeline design)
  - Traditional name for this is a *Branch Target Buffer (BTB)*



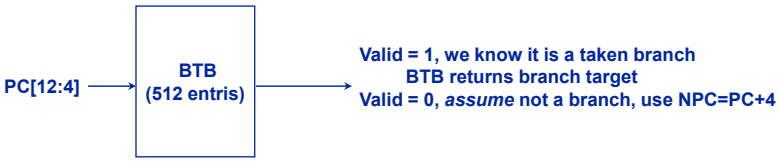
BTB entry: | tag | target address |

PIPE-24

24

Branch Target Buffer (cont.)

- Instead of the tag comparison, just hash the PC
- This works because a lookup of the wrong entry just causes a misprediction (ie, a performance hit but the code still works!)
- Example hash: use middle bits from PC, e.g., if table is 512 entries, use bits PC[12:4]



BTB entry: | Valid bit | target address |

PIPE-25

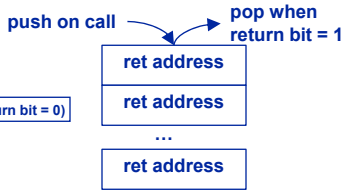
25

Predicting *Where* with returns

- Problem: A lot of branches are return instructions – BTB fails on these because target address is dependent on call location – it keeps changing
  - Holding the last target address is a poor predictor unless the call site is always the same!
- Solution: Keep a hardware "stack" of return addresses
  - Push return address when a "call" instruction is seen
  - Pop buffer on returns to get prediction
  - Empirically need 4 to 8 entries for integer code

Each entry in BTB now contains:

Tag bits *or* valid bit	return bit	Target address (only when return bit = 0)



PIPE-26

26

About the return address stack

Problem: stack is a fixed size, not ∞

Two program behavior scenarios:

1. Shallow call depth ≤ N
2. Very deep call depth >> N

A hardware stack of depth N will do a good job in scenario 1, but it will not in scenario 2.

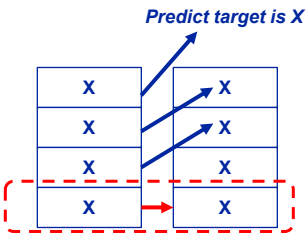
When does scenario 2 happen?

- It's most common in *recursive code*
- In recursive code, the stack grows very deep
- BUT, the stack contains the same value (e.g., X)

  stack would be full of many copies of X!

Solution: Always Copy the bottom of the hardware stack on Pops

```
Recursive code:
        call fact(n-1)
X:  next instruction
```

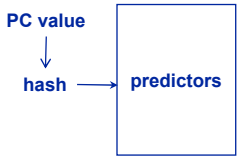


Issues with *Whether*

- Predicting conditional branches
  - And sometimes unconditional branches if needed before the instruction's opcode is decoded
  - Our BTB does the equivalent of "whatever happened last" or a 1-bit scheme

Hardware branch prediction

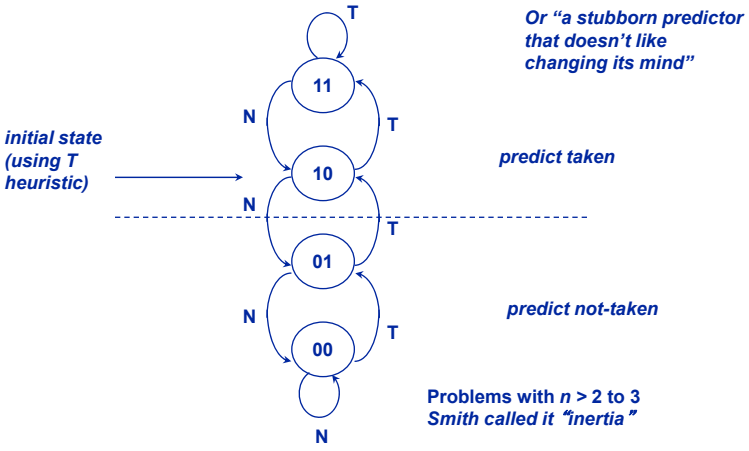
- OFTEN is a separate table, the branch prediction buffer:



- Only answers the "WHETHER branch is taken or not" question
- Typically it has no tags and conflicts are allowed

Smith *n*-bit "bimodal" counter predictor

- Replace prediction *bit* with *n*-bit counter:



PIPE-30

30

Smith counter cont.

- Why it works:
  - Conditional branches at the end of a loop are taken often but only not taken once
    - "T T T T T N T T T T T N…
    - Counter would be 01 11 11 11 11 10 11 11 11 11 10
    - The infrequent N does not chance the prediction
  - Conditional branches that guard an error condition are rare:
    - If (error) { handle error }
    - N N N N N N T N N N N N …
    - The rare T does not change the prediction
  - Smith called this property "inertia"
- Ravi Nair of IBM Research simulated all possible 2-bit state machines and found variants of the Smith counter were the best predictors
  - Variation was the initial state choice

PIPE-31

31

Example of Smith counter

	T	T	T	T	T	T	N	T	T	N	N	N	N	N	N	N	T	T
previous state	10	11	11	11	11	11	11	10	11	11	10	01	00	00	00	00	00	00	01
new state	11	11	11	11	11	11	10	11	11	10	01	00	00	00	00	00	00	01	10

5 mispredictions out of 19 branch executions

	N	T	N	T	N	T	N	T	N	T	N	T	N	T	N	T	N	T
previous state	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01
new state	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10

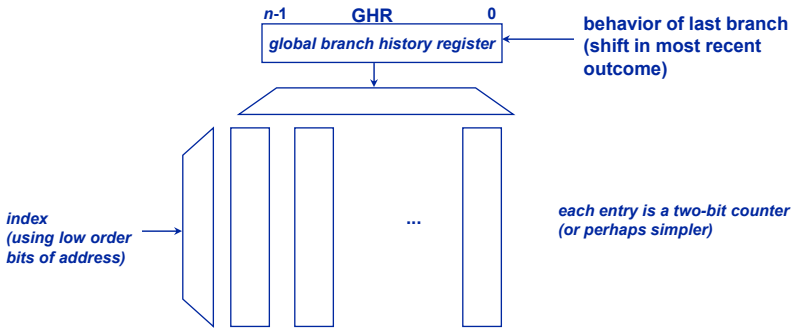
19 mispredictions out of 19: the infamous *"toggle branch"*

PIPE-32

32

Improving the Smith counter

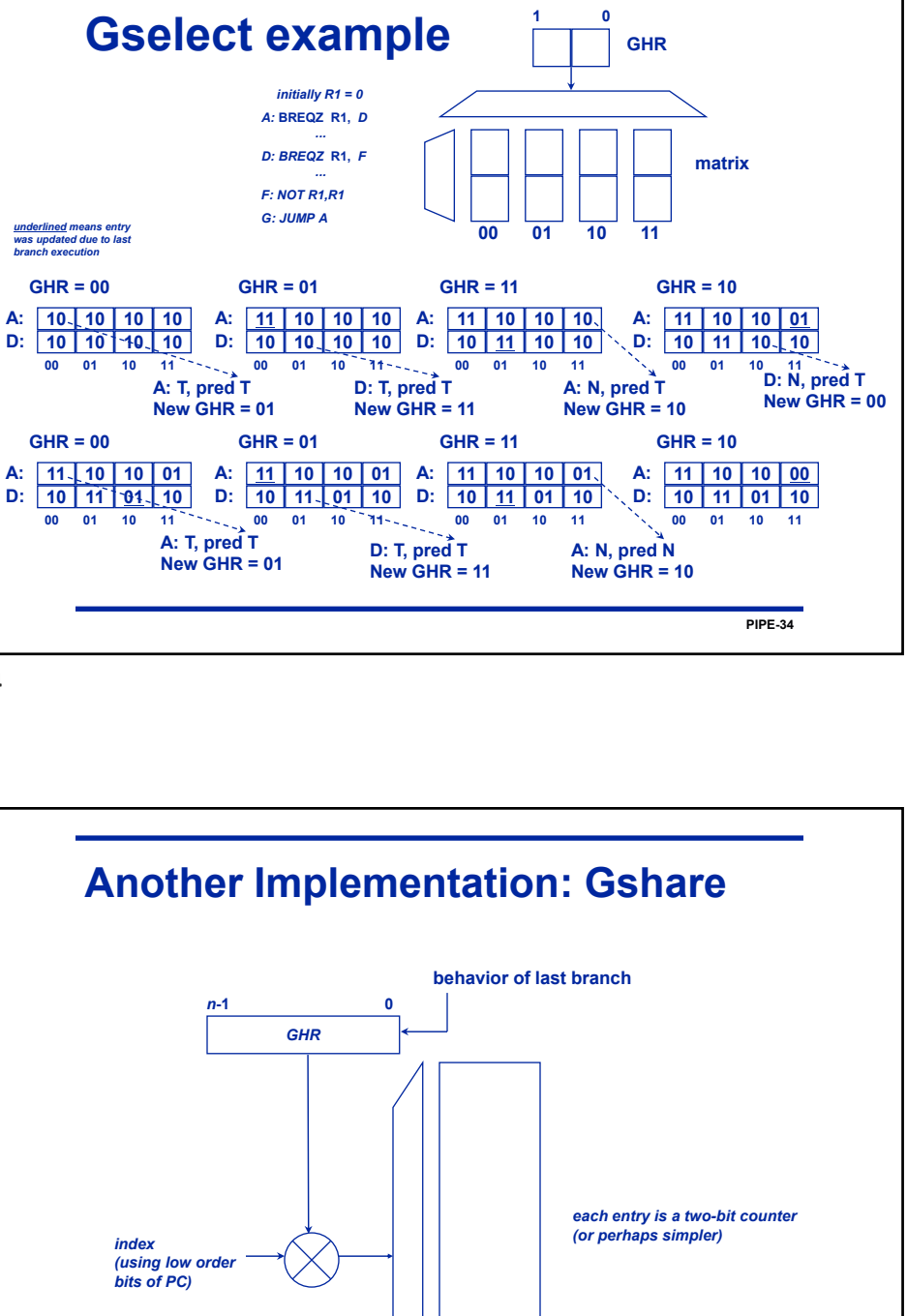
- Idea: Capture correlations between different branches
- Gselect:



PIPE-33

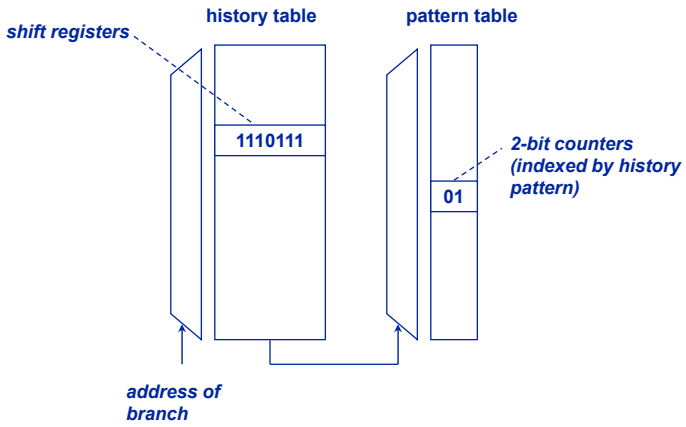
33

Gselect example



Another Implementation: Gshare

Yeh/Patt predictor



Idea: Associate predictions with branch histories, not branch addresses (use different indexing scheme)

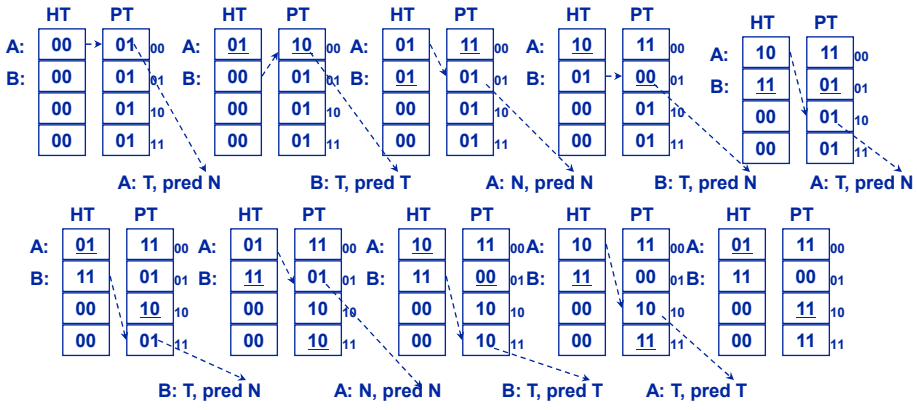
PIPE-36

36

Yeh/Patt Example: toggle branch

A: TNTNTNTNTNTNTNTNTN
B: TTTTTTTTTTTTTTTTTTTT

Note: We're using Not-Taken heuristic to initialize counters to 01 (weakly not-taken).



4 mispredictions in this case.
In general: provides 96-98% accuracy for integer code

PT entries 01, 10 are "trained" for A and 11 is "trained" for B

PIPE-37

37

Importance of branch prediction

CPI = 1 + stalls

If a branch is seen 20% of the time, and

The branch predictor is A% accurate, and

The penalty for a misprediction (pipeline depth dependent) is P, then

CPI = 1 + 0.2*(1-A)*P

Say A = 90% and P = 3 cycles, CPI = 1.06

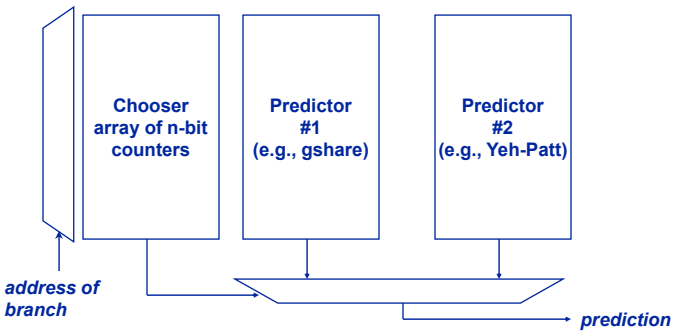
Now say P = 6, CPI = 1.12

Remember that CPU Time = IC * CPI * CT

Want low CPI (good branch prediction) and low CT (deep pipelines)

Branch prediction accuracy allows deeper pipelines, thus faster processors!

Hybrid predictors



- Both predictors supply a prediction-- pipeline uses only one
- Chooser ("confidence predictor") updated based on which predictor was correct
  - Increment chooser counter if #1 was correct, decrement if #2 was correct

Program Characteristics: Types of dependencies

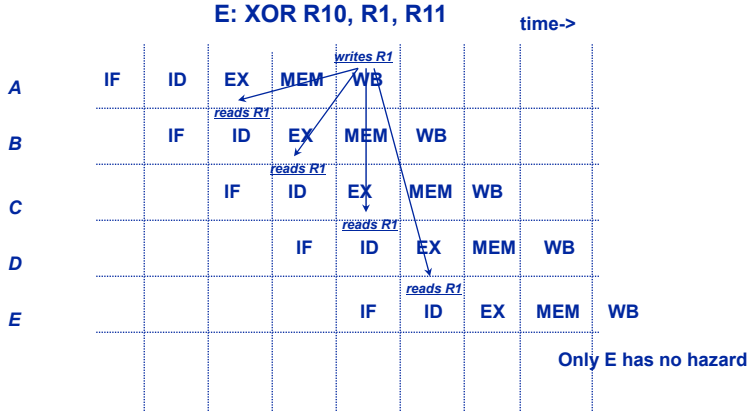
- True-dependence (pure-dependence, flow-dependence)
  - ADD R1,R2,R3
  - SUB R4,R5,R1
  - Stall if pipeline has RAW hazards
- Anti-Dependence
  - ADD R3,R2,R1
  - SUB R1,R4,R5
  - Stall if pipeline has WAR hazards
  - Due to reuse: Removed by using another register
- Output-Dependence
  - ADD R1,R2,R3
  - SUB R1,R4,R5
  - Stall if pipeline has WAW hazards
  - Due to reuse: Removed by using another register

Data Hazards

- Recall that hazards reduce the potential speedup of pipelining
  - Dependencies between instructions prevent their overlapped execution
- Kinds of data hazards
  - Read After Write (RAW)
    - Caused by pure dependence
  - Write after Read (WAR)
    - Caused by anti-dependence
  - Write After Write (WAW)
    - Caused by output-dependence
  - Read after Read (RAR)???
    - Really a structural hazard on read ports

Data hazards

A: ADD R1, R2, R3
B: SUB R4, R5, R1
C: AND R6, R1, R7
D: OR R8, R1, R9
E: XOR R10, R1, R11

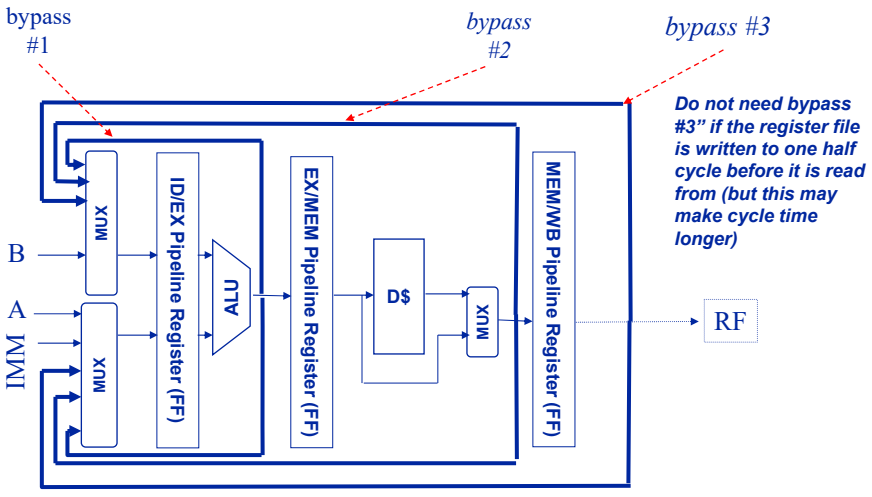


Only E has no hazard

PIPE-43

43

Data forwarding



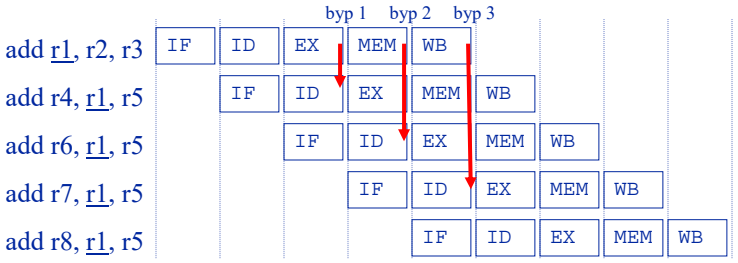
Do not need bypass #3" if the register file is written to one half cycle before it is read from (but this may make cycle time longer)

PIPE-44

44

Data forwarding (cont.)

- Need bypass3 ONLY if we *do not* follow the pipeline model where we *write registers during the first half of a clock cycle and read them in the second*

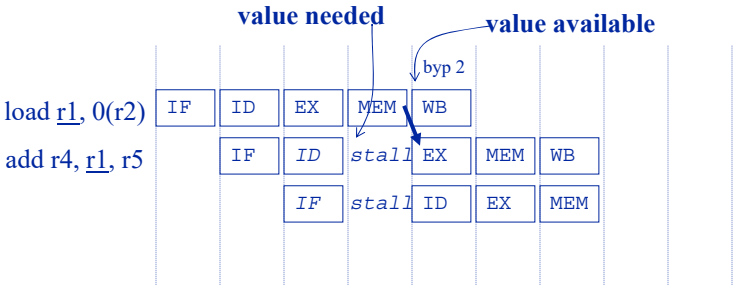


PIPE-45

45

Stalls due to data hazards

- In our pipeline:
  - Most RAW hazards => no stall
  - Loads cause 1-cycle stall (cache hit)
    - This RAW hazard is so common, it's sometimes given its own name, a "load-use hazard"
  - Can "feature" the hazard by telling programmer that load values are not available until AFTER the next instruction – called a "Load Delay Slot"



PIPE-46

46

Other data hazards

- WAR (write-after-read)
  - A ADD r1, r2, r3
  - B ADD r2, r4, r5
  - Hazard if B writes R2 before A reads R2
  - Modify example pipeline so that Icache misses do not stall instruction fetch, and you may get this hazard to occur:

PIPE-47

47

Other data hazards (cont.)

- WAW (write-after-write)
  - A ADD r1, r2, r3
  - B ADD r1, r4, r5
  - Hazard if B writes R1 before A writes R1
  - Result: later instructions see wrong value in the register
  - Occurs if instructions can write register file out-of-order
  - Using the same pipeline that does not stall fetch on Icache miss:

PIPE-48

48

Other data hazards (cont.)

- Handling WAR/WAW hazards
  - Stall the later instruction (stalls in WB stage)
  - Compiler: don't reuse registers (not always possible, because you run out of registers...)
  - Hardware: register renaming (*see next major topic – ILP*)

And finally...

- CPU time = IC x CPI x CT
- CPI= 1 + stalls
  = 1 + structural_stalls + br_stalls + data_stalls
- Best CPI is CPI = 1
- ... Right???