CS359: Computer Architecture [1]

Lecture Notes [2]

April 28th, Spring 2020

[1] Course Instructors: Yanyan Shen
TA: Xian Zhou, Weiyu Cheng

[2] Authors: Zehao Wang, Chi Zhang

Keyphrases: Branch Prediction, Dynamic Scheduling, Scoreboarding

1 *Branch Prediction*

A loop program written in C:

```
for (int i = 0; i < 100; i++) {
    // do something.
}
```

can be compiled into:

branch to the beginning

```
    addi r10, r0, 100
    addi r1, r0, r0
L1:
    #do something
    addi r1, r1, 1
    bne r1, r10, L1
```

branch to exit the loop

```
    addi r10, r0, 100
    addi r1, r0, r0
L1:
    #do something
    addi r1, r1, 1
    beq r1, r10, FINISH
    j L1
FINISH:
```

1.1 *Static Branch Prediction*

1. **Definition** An "uni-directional, always predict taken" method.
2. **Observation** If branch is taken every time, there are: 99/100 correct predictions (first program), and 1/100 correct predictions (second program).
3. For such **loop program**, this method can go into extermes.
4. Only used as a **fall-back technique** with dynamic branch prediction when there is no information for dynamic predictors to use.

Drawbacks

- Predict taken incurs one stall cycle in the pipeline(if branch is resolved in ID stage)
- Always branch to the same address. Any methods to store it?
- Branch penalty increases in deeper pipelines for CISC(e.g. x86_64, etc.)

1.2 *Simplest Dynamic Branch Prediction*

1. Use **latest outcome** as prediction result. This changes very often.
2. **Indexed by some bits** in the branch PC.
3. **Collision** may incur because some instructions share the same lower bits of address.

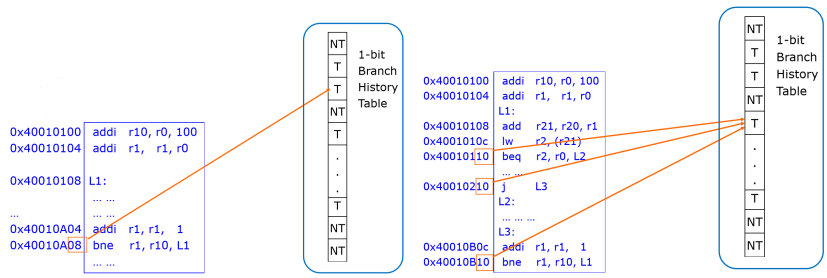


Figure 1: Collision in dynamic branch prediction

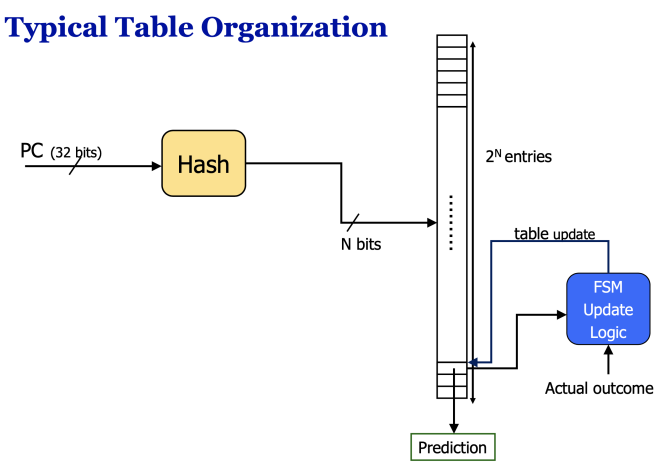


Figure 2: Typical Table Organization

1.3 *1-bit Finite State Machine of the Simplest Predictor*

1. 1-bit FSM is used to update branch logic for the simplest predictor
2. It is a 2-state machine
3. Its status changes fast.
4. **Observation** 1-bit FSM results in two wrong predictions, one at the end of the first loop, the other at the beginning of the next loop.
5. **Question** Can we further improve the performance by reducing wrong predictions?

6. **Answer** 2-bit FSM can solve the problem well.

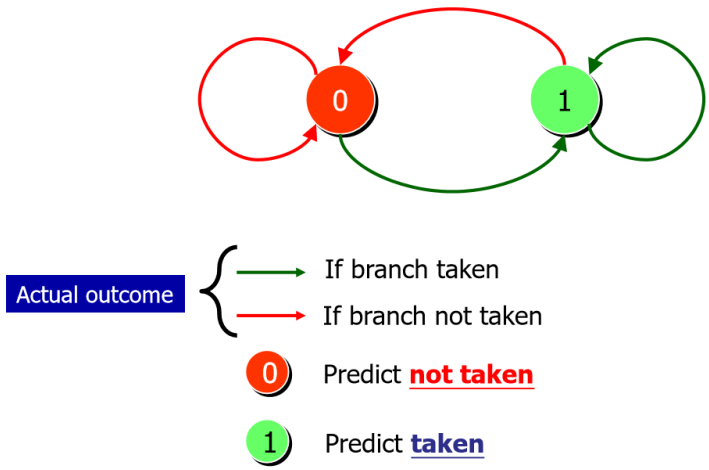


Figure 3: 1-bit Finite State Machine

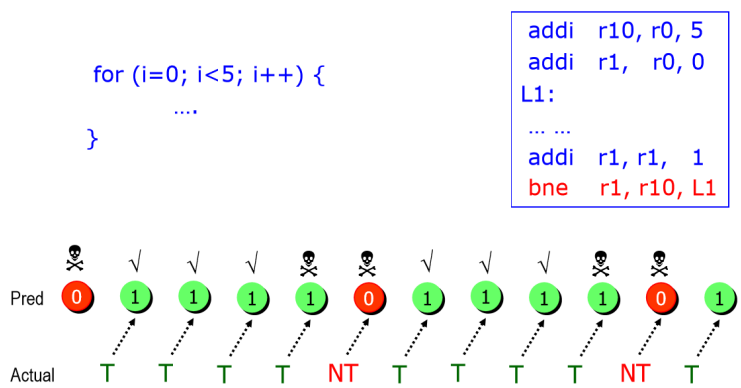


Figure 4: Example for the usage of 1-bit FSM

1.4 *2-bit Saturating Up Down Counter Predictor*

1. **Observation** It cause only one wrong prediction at the end of a loop and no wrong prediction at the beginning of a new loop.

2. Wrong prediction is destined to happen since computer is not human brain: it does not know what it is actually doing.

3. In this way, wrong prediction can be reduced to as few as possible, so the penalty is the least, causing highest performance.

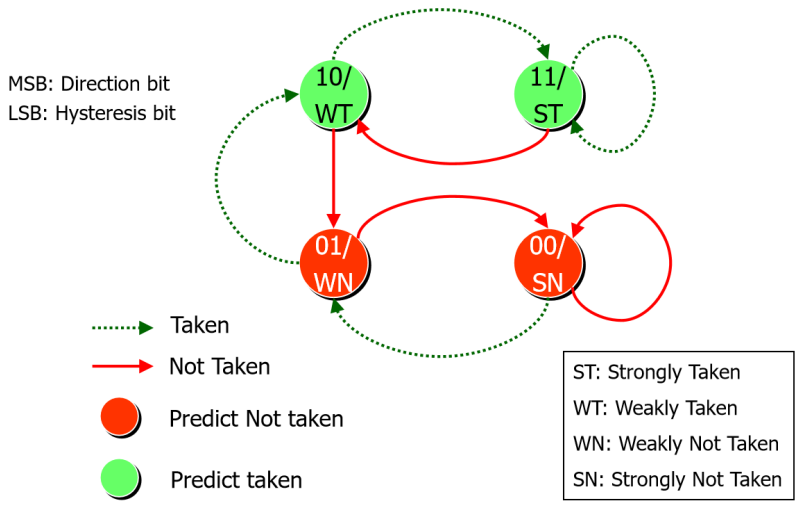


Figure 5: 2-bit Finite State Machine

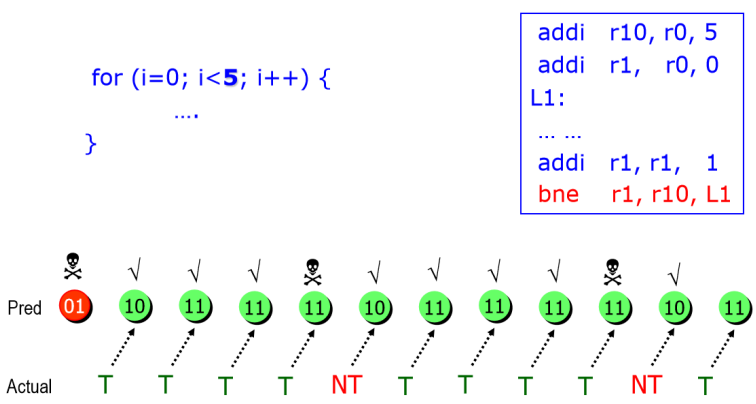


Figure 6: Example for the usage of 2-bit FSM

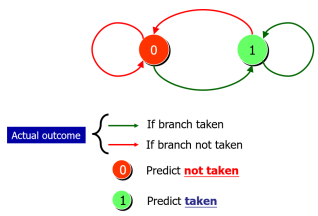
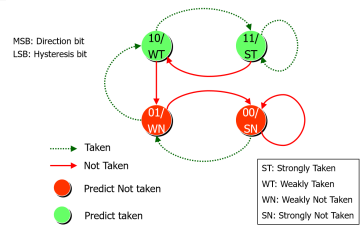
1.5 Summary

Static prediction and dynamic prediction with 1-bit FSM cause more penalty than dynamic prediction with 2-bit FSM. With the development of integrated circuit, the once expensive but more flexible dynamic prediction becomes cheaper and cheaper, making itself quite common in modern processors.

Two-level adaptive predictor can further improve accuracy by learning history pattern

	Static	Simplest Dynamic	2-bit Saturating
Description	Always take / not take	Take if taken last time	Take if counter >= 2

Check	take = true / false
	(or other stateless function)
		if (taken[PC])
		take = true;
		else take = false;
			if (counter[PC] >= 2)
			take = true;
			else take = false;

Update		taken[PC] = actually_taken?	if (actually_taken?) ++counter[PC]; else - -counter[PC]; counter limited to [0, 3]
FSM			
Accuracy	good	better	best

2 Dynamic Scheduling

2.1 Motivation

Observation: we can re-order instructions without affecting the result.

```
div_result = num1 / num2;
add_result = div_result + num3;
other_op = op1 - op2;
```

We can reduce stalled cycles by re-ordering `other_op` before `add_result`.

```
div_result = num1 / num2;
other_op = op1 - op2;
add_result = div_result + num3;
```

2.2 Dynamic Scheduling

Rearrange order of instructions to reduce stalls while maintaining data flow.

- Pros: compiler doesn't need to know micro-architecture, handle cases where dependencies are unknown at compile time
- Cons: substantial increase in hardware complexity, complicates exceptions

2.3 *Scoreboarding Approach*

We'll introduce scoreboarding in two parts: components and book-keeping.

Components

- **Instruction Status** saves what state each instruction is in.
- **Function Unit** execute instructions (load memory, do calculation).
- **Register Status** saves what FU will write to this register.

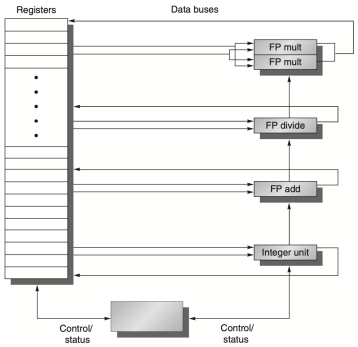


Figure 7: Components of Scoreboarding

Book-keeping Steps

- **Issue** Wait until function unit is available (structural hazard), and check if destination register is in use (WAW hazard).
  `LD F6, 45(R3)`

FU	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	(check)								

F0	F2	F4	F6	R3
			(check)	Add

FU	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	Load	F6			Add		No	

- **Read Operands** Wait until other FU completes by checking R_j, R_k of itself, set R_j, R_k to no, and put result in F_j, F_k (RAW hazard).

FU	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	Load	F6			Add		Yes	

FU	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	Load	F6	R3		Add		No	

F0	F2	F4	F6	R3
			Integer	

- **Execute complete** Dependent on FU. Each FU will take some cycles to complete its operation. Notify the scoreboard.
- **Write Result** wait until other instructions read this instruction's destination, 1. tell other FU that data is ready, 2. write data to register file, 3. mark itself as idle (WAR hazard)

Example: Multi 1 is waiting for Multi 2 before it can read F6

FU	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes								
Mult 1	Yes				F6	Mult2		Yes	

(2 cycles)

FU	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult 1	Yes				F6			No	

F0	F2	F4	F6	R3

Instruction status	Wait until	Bookkeeping

Issue	Not busy [FU] and not result [D]	`Busy[FU]←yes; Op[FU]←op; Fi[FU]←D;`
		`Fj[FU]←S1; Fk[FU]←S2;`
		`Qj←Result[S1]; Qk← Result[S2];`
		`Rj← not Qj; Rk← not Qk; Result[D]←FU;`

Read operands	Rj and Rk	`Rj← No; Rk← No; Qj←0; Qk←0`
Execution complete	Functional unit done	
Write result	$\forall f$((Fj[f] ¦ Fi[FU] or Rj[f] = No) & (Fk[f] ¦ Fi[FU] or Rk[f] = No))	$\forall f$(`if Qj[`f`]=FU then Rj[`f`]←Yes);` $\forall f$(`if Qk[`f`]=FU then Rk[`f`]←Yes);` `Result[Fi[FU]]← 0; Busy[FU]← No`

Figure 8: Scoreboarding Steps

2.4 *Quick Review*

	Single-Cycle Multi-Cycle (baseline)	5-level Pipeline	Scoreboarding	Tomasulo
Instruction Issue	In-order	In-order	In-order	In-order
Instruction Commit	In-order	In-order	Out-of-order	In-order (with reorder buffer / ROB)
Execution	In-order	In-order	Out-of-order	Out-of-order
RAW Data Hazard	No	Bypassing + Forwarding + Stall	Stall (Read Operands)	Stall (Execute Stage)
WAW Data Hazard	No	No (in-order commit)	Stall (Issue)	Register Renaming (Issue Stage)
WAR Data Hazard	No	No (always read before write)	Stall (Write Result)	Register Renaming (Issue Stage)
Control Hazard	No	Stall / Prediction + Flush / Delayed slot	Not mentioned	Speculation + Flush ROB
Structural Hazard	No	Stall (IF & MEM, muti-cycle EX)	Stall (Issue)	Stall (Issue Stage)
Exception	Easy	Not hard	Not mentioned	Complicated