Performance

cheat sheet!

IRON RULE: Execution Time = #Instructions*CPI*Clock Period = #Cycles*Clock Period

Clock Period

- Single Cycle: Latency of Slowest <u>Instruction</u>
- Multi Cycle: Latency of Slowest <u>Cycle</u> (generally slowest datapath component)

CPI

- Single Cycle: 1 (it's in the name)
- Multi Cycle: #Cycles/#Instructions

#Cycles

- Single Cycle: #Instructions
- Multi Cycle: #Instructions * sum over all opcodes(%opcodes*cycles for opcode)

#Cycles Per Opcode (MULTICYCLE ONLY)

- add/nor/sw/beq: 4 cycles
- lw: 5 cycles
- noop/halt: 2 cycles
- jalr: Don't worry about it

Components used by each Opcode

Instr.	Read PC, Access Instr. Mem.	Read Reg.	ALU	Data Mem. Access	Write PC	Write Reg.
add	✔	✔	✔			✔
nor	✔	✔	✔			✔
lw	✔	✔	✔	✔ (Read)		✔
sw	✔	✔	✔	✔ (Write)		
beq	✔	✔	✔x2		✔	
jalr	✔	✔			✔	✔
noop	✔					
halt	✔					

Datapath Benchmarking

We are using a benchmarking program that executes 10,000 instructions. 25% are `lw`, 15% `sw`, 30% `add` / `nor`, 20% `beq`, and 10% `noop` / `halt`.

Each use of the ALU costs **10ns**, reading memory costs **50ns**, writing memory costs **60ns,** register file reads and writes cost **5ns**, and everything else costs **0ns**.

Key for today: What is the runtime of this benchmark on different systems?

Optimal Performance Benchmarking

What is the optimal achievable serial runtime of this benchmark?

Main idea: only use the wire latency for each instruction.

`add / nor`	(50 + 5 + 10 + 5)ns = 70ns	`beq`	(50 + 5 + 10)ns = 65ns
`lw`	(50 + 5 + 10 + 50 + 5)ns = 120ns	`noop / halt`	50ns
`sw`	(50 + 5 + 10 + 60)ns = 125ns		

Total = 10,000 (0.25 ⨯ 120ns (lw) + 0.15 ⨯ 125ns (sw) +
0.3 ⨯ 70ns (add / nor) + 0.2 ⨯ 65ns (beq) +
0.1 ⨯ 50ns (noop / halt))
= 877,500ns

Pipeline Performance

#Cycles = #instr + #pipeline_stages - 1

(for the 5-stage pipeline) -> #Cycles = #instr + 4

Base CPI = #Cycles / #Instr ≈ 1

Then add 3 S's: Startup cycles, Stalls, Squashes

EX: For a program with 1000 instructions with <u>no hazards</u>, we would need **1004 cycles** to execute and the CPI would be **1004/1000 = 1.004**

We Can Guess If a Branch Is Taken Using Predictors

In order to predict a branch, we need to infer three things:

1. Whether the instruction is a branch *(We haven't Decoded yet!)*
2. If the branch instruction will be taken or not
3. Where the branch will jump to if it is taken

If we can't figure out some of this information, we cannot make a prediction, often revert to assuming it will not be taken

For example, if we don't know where a branch will jump to, we can never predict it as taken, since we won't know where to fetch from next

Static Branch Predictors Don't Change Prediction State

Always Not Taken: Predicts that a branch will never be taken (Project 3)

Always Taken: Predicts that a branch will always be taken

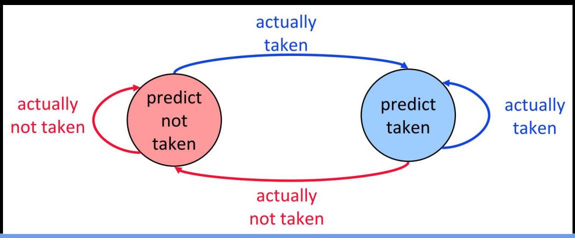
Backwards Taken, Forwards Not Taken: Predicts a branch will be taken or not based on if it branches forwards or backwards

These predictions can all be done at compile time

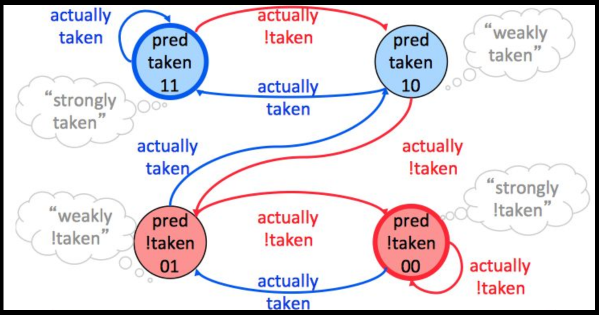
Dynamic Predictors Take Advantage of the Past

Last Time Predictor: Predict based on whatever the branch did last time

Problem: Could cause 0% accuracy for "T N T N T N T ..."



FSM For a 2-bit Branch Predictor


actually taken
pred taken 11
actually !taken
pred taken 10
"weakly taken"
"strongly taken"
actually taken
actually taken
actually !taken
"strongly !taken"
"weakly !taken"
pred !taken 01
actually !taken
pred !taken 00
actually !taken
actually taken