A Fault Tolerant Superscalar Processor

[Based on "Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor" by *V. Reddy* and *E. Rotenberg* (2008)]

PRESENTED BY
NAN ZHENG

[Part of slides borrowed from V. Reddy's slides in DSN2008]

Outline

- Introduction
  - FT in processors: why
  - Superscalar processors: what and why
  - Conventional processor FT, related drawbacks
    - Hardware & info & time redundancy
    - The need for a regimen-based FT

Outline (Cont.)

- Regimen-based FT (RFT) by *Reddy* and *Rotenberg* (2008)
  - FT regimen
    - Inherent Time Redundancy (ITR)
    - Register Name Authentication (RNA)
    - Timestamp-based Assertion Check (TAC)
    - Sequential PC Checks (SPC)
    - Register Consumer Counter (CC)
    - BFT Verify (BTBV)
  - Simulation Approach & Result
- Summary

Introduction

- Why Fault Tolerance (FT) in processors:
  - Critical charge decreases with processor die area (quadratically), i.e, making easier to flip a bit.
  - Cosmic rays in atmosphere being a source
- Superscalar processors: what and why
  - What?
    - Processors that exploit ILP by fetching & executing multiple instructions per cycle from a sequential instruction stream.
  - Why?
    - Almost all modern processors are superscalar

Introduction (Cont.)

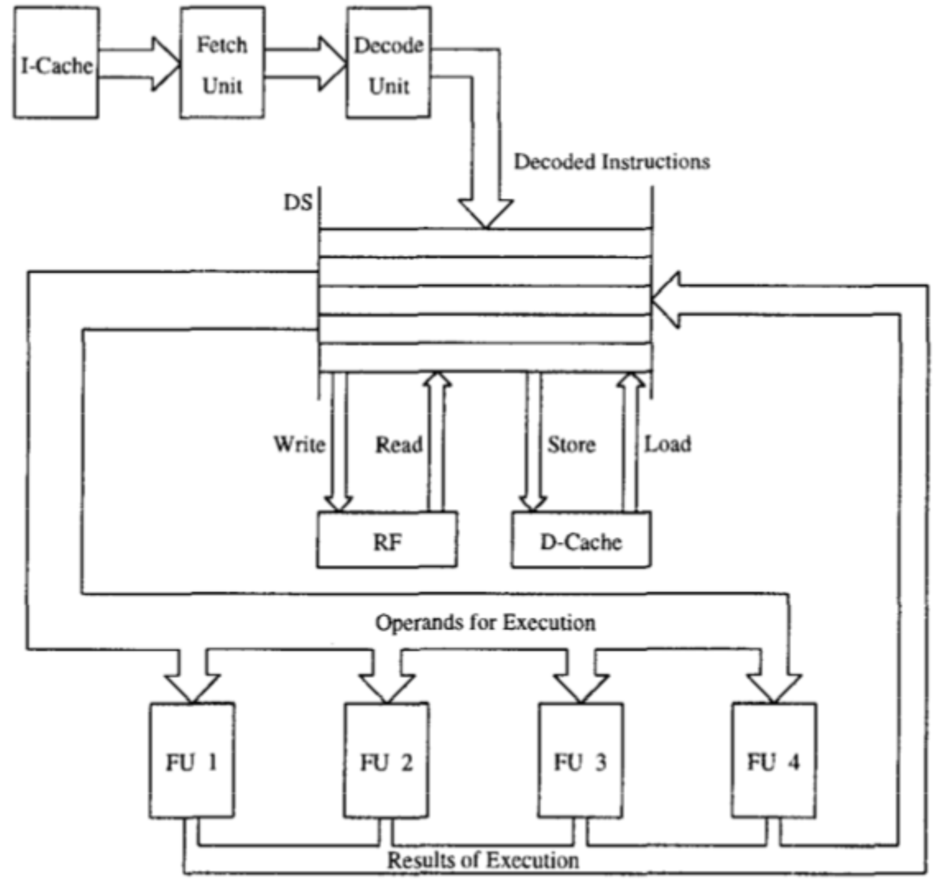


Figure 1: Block Diagram of a Dynamically Scheduled Superscalar Processor

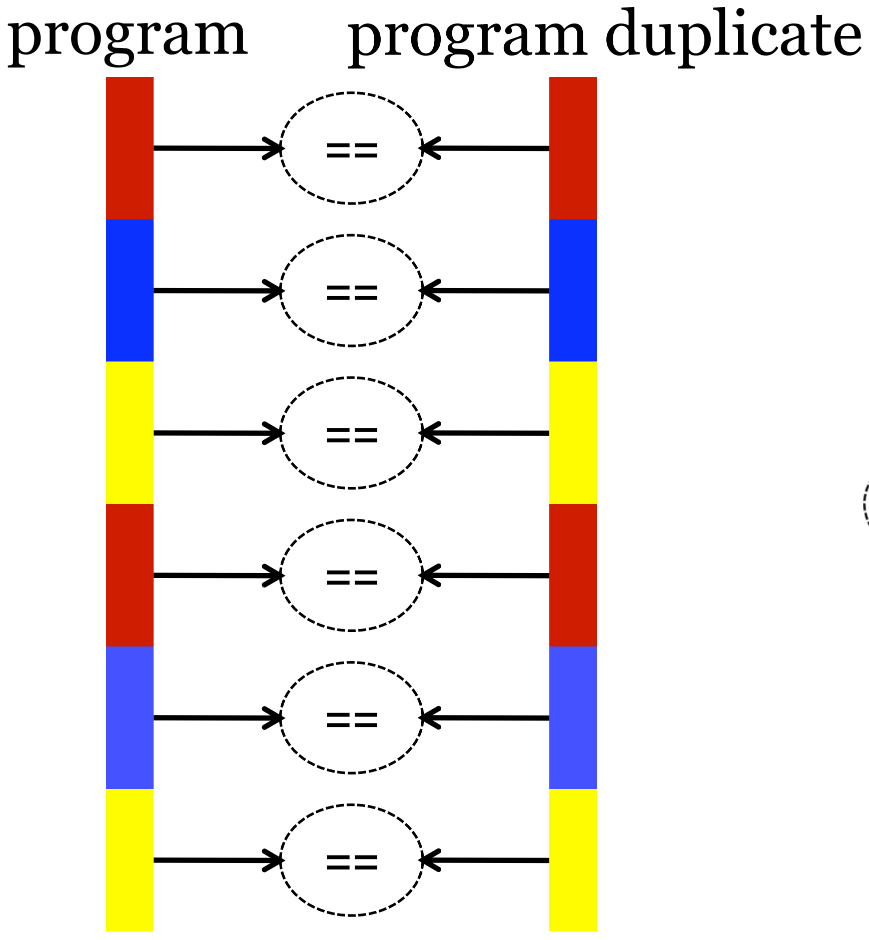
Introduction (Cont.)

- Conventional FT schemes in processors
  - Basic idea: some form of *redundancy*
  - Hardware redundancy
    - Additional FU especially for redundancy execution
    - Drawbacks: silicon area overhead, not for commercial processors
  - Information redundancy
    - Error-correcting code (ECC) in memory
    - Control flow based signals
    - Checksums for algorithm-based FT
  - Time redundancy
    - Instruction re-execution
    - Retrasmission of data…
  - Note:
    - Additional overheads in silicon area, pipeline stalls …
    - Only focused on FUs, errors can also occur in DU, DS and RF
    - Need a systematic suite of fault checks to achieve maximum coverage over all pipeline stages, and minimum overhead at the same time

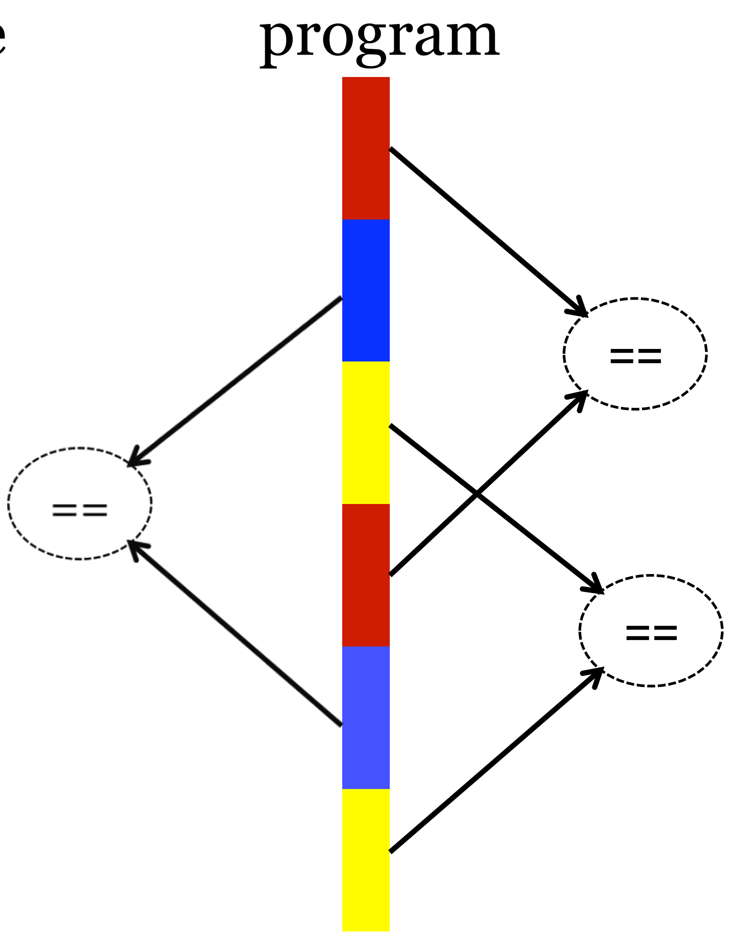
Regimen-based FT

- Overview on FT regimen:
  - Inherent Time Redundancy (ITR)
  - Register Name Authentication (RNA)
  - Timestamp-based Assertion Check (TAC)
  - Sequential PC Check (SPC)
  - Register Consumer Counter (CC)
  - Confident Branch Misprediction (ConfBr)
  - BTB Verify (BTBV)
- Individuals explained next…

Inherent Time Redundancy (ITR)



Conventional time redundancy



Inherent time redundancy

Inherent Time Redundancy (ITR)

- A decode signature is maintained per instruction
  - Signature is updated at last use of a decode signal
- At retirement, instruction signatures are combined into trace signatures
  - A trace ends at branch or 16 instructions
- Trace signatures are stored in a *ITR cache*
- Each new trace signature is checked with the copy in ITR cache
  - Cache miss does not directly cause fault coverage loss
  - Later hit to a previously missed signature detects faults in either the current or previous signature

RNA & TAC

- Register Name Authentication (RNA)
  - Detects faults in destination register mappings of instructions
  - Checks consistencies in rename unit

- Timestamp-based Assertion Check (TAC)
  - Detect faults in the issue unit
    - Checks if there's sequential order among data dependent instructions
  - Implementation:
    - Check: Instr's Timestamp >= Prod. Timestamps

Sequential PC Check (SPC)

- Detects faults affecting sequential control flow
- Asserts that a committing instr.'s PC matches the retirement PC
- Implementation
  - Maintain retirement program counter (PC)
  - For non-branch instr., increment retirement PC by instr. size
  - For branch instr., update retirement PC with calculated PC
  - Check: committing instr. PC match retirement PC

CC & ConfBr

- Register Consumer Counter (CC)
  - Detects faults in source register mappings after register renaming
  - Implementation:
    - One counter per physical register
    - Increment counter of source register at rename stage
    - Assert counter of source register > 0 at register read stage
    - Decrement counter of source register after register read
- Confident Branches Misprediction (ConfBr)
  - Detects faults affecting values that influence branch outcomes
  - Implementation
    - Identify highly-predictable branches using ‘confidence’ counters
    - Misprediction of a confident branch may be symptomatic of a fault

BTB Verify (BTBV)

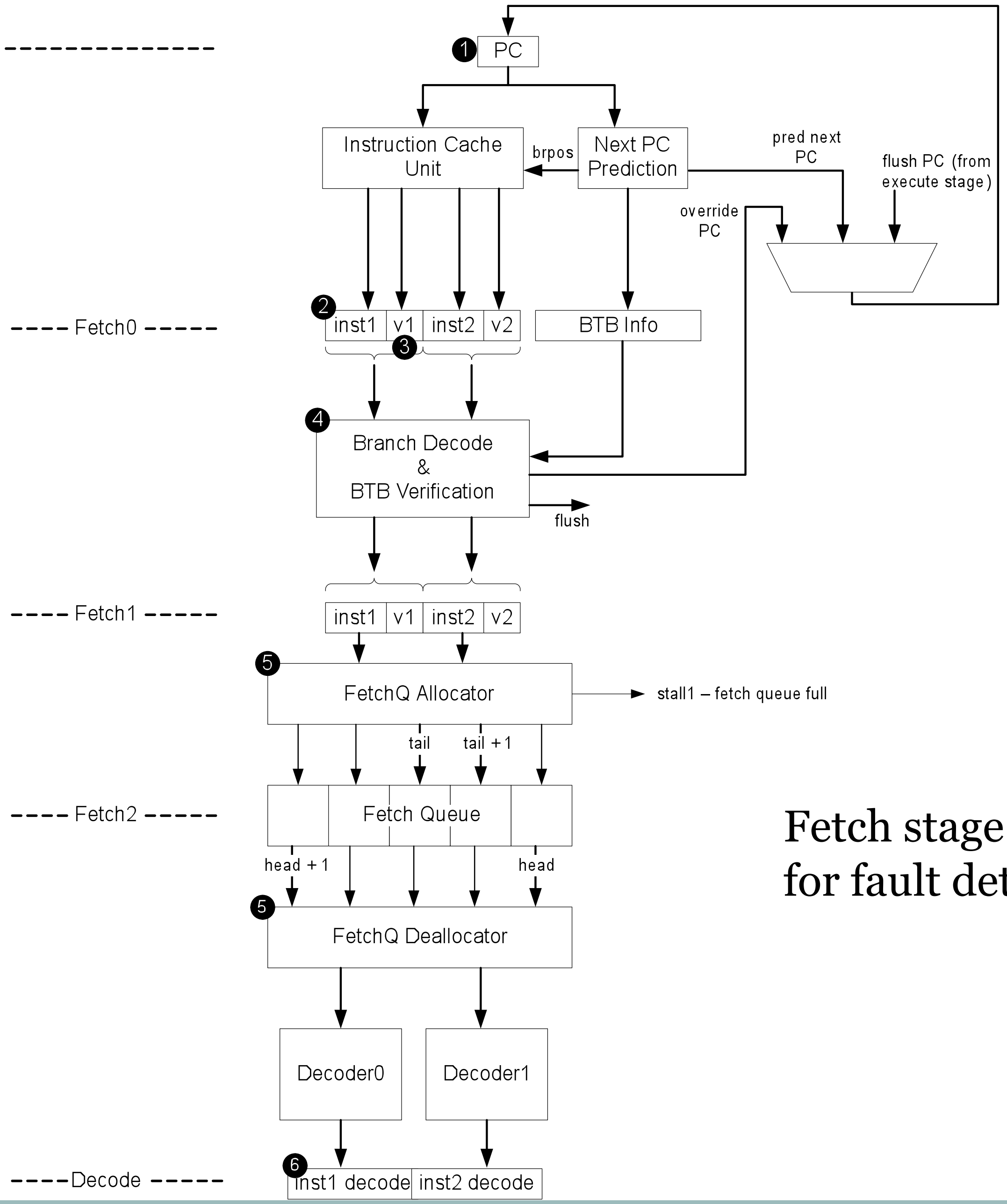
- Detects faults in BTB and decode logic
- Exploits inherent redundancy between the BTB and the decode stage
  - BTB hit produces decode info about branches one cycle earlier than decode stage
  - BTB info should match decode info
  - Mismatch indicates fault in BTB logic (false hit, BTB fault, etc.) or decode stage
  - BTB aliasing mismatches are handled in the same manner (flush the instruction and instructions after it, don’t trust the decoder)

RFT: Simulation Approach

- Evaluation Using Fault Injection, goals:
  - Measure processor fault coverage of a μarch-level fault-check regimen
  - Leverage C/C++ cycle-level μarch. simulators
    - Cost and time efficient
  - Ensure high fault modeling coverage
- Fault Injection Approach
  - Analyze high-level (μarch-level) effects of faults in each pipeline stage
  - Randomly inject μarch-level faults in simulator
  - Example: fetch stage (IF)

IF	ID	REN	DISP	IS	RR	EX	WB	RE	(a)

IF	ID	REN	DISP	IS	RR	AGEN	M	WB	RE	(b)



Fetch stage fault analysis for fault detection

RFT: Simulation Approach

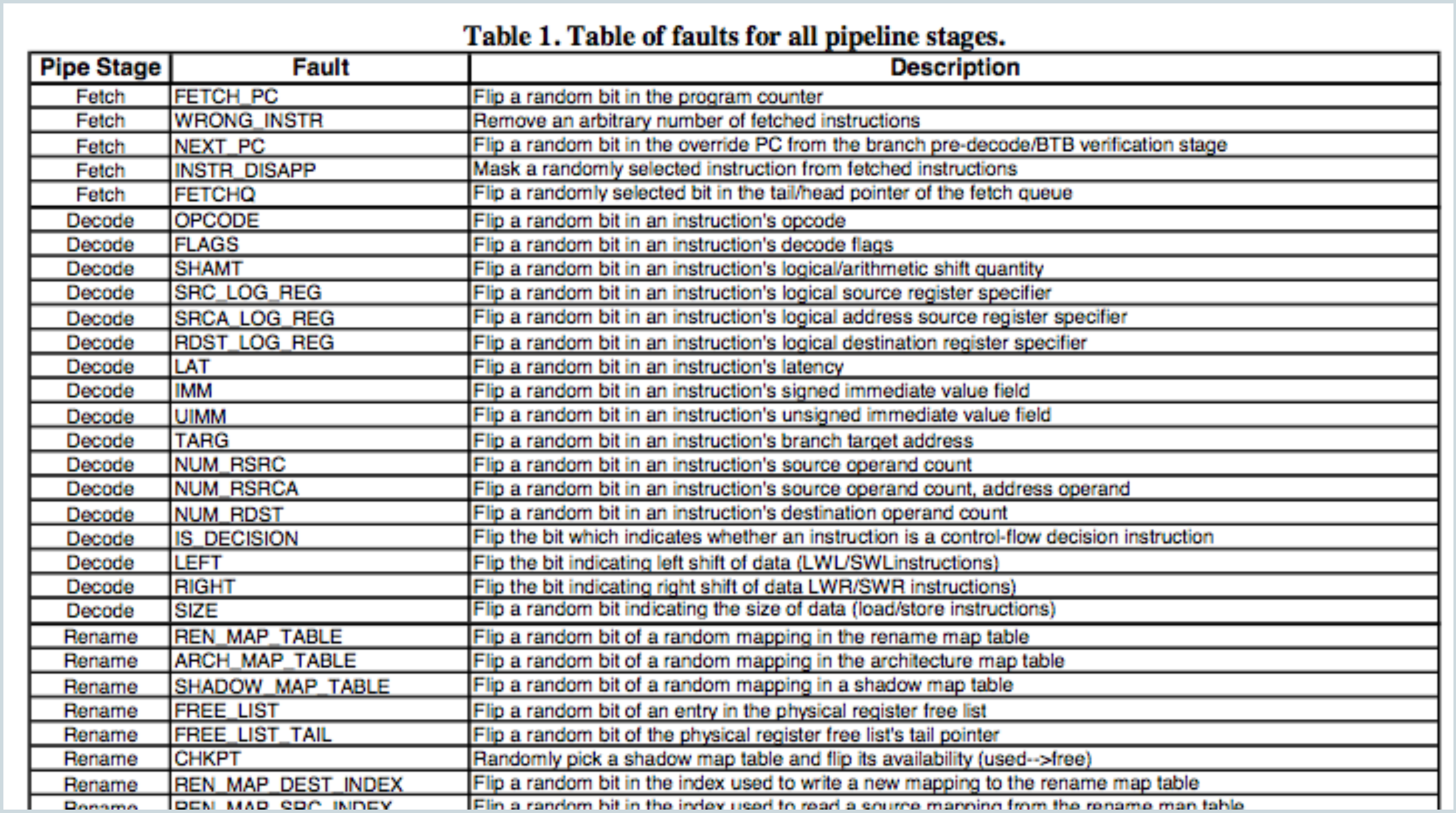
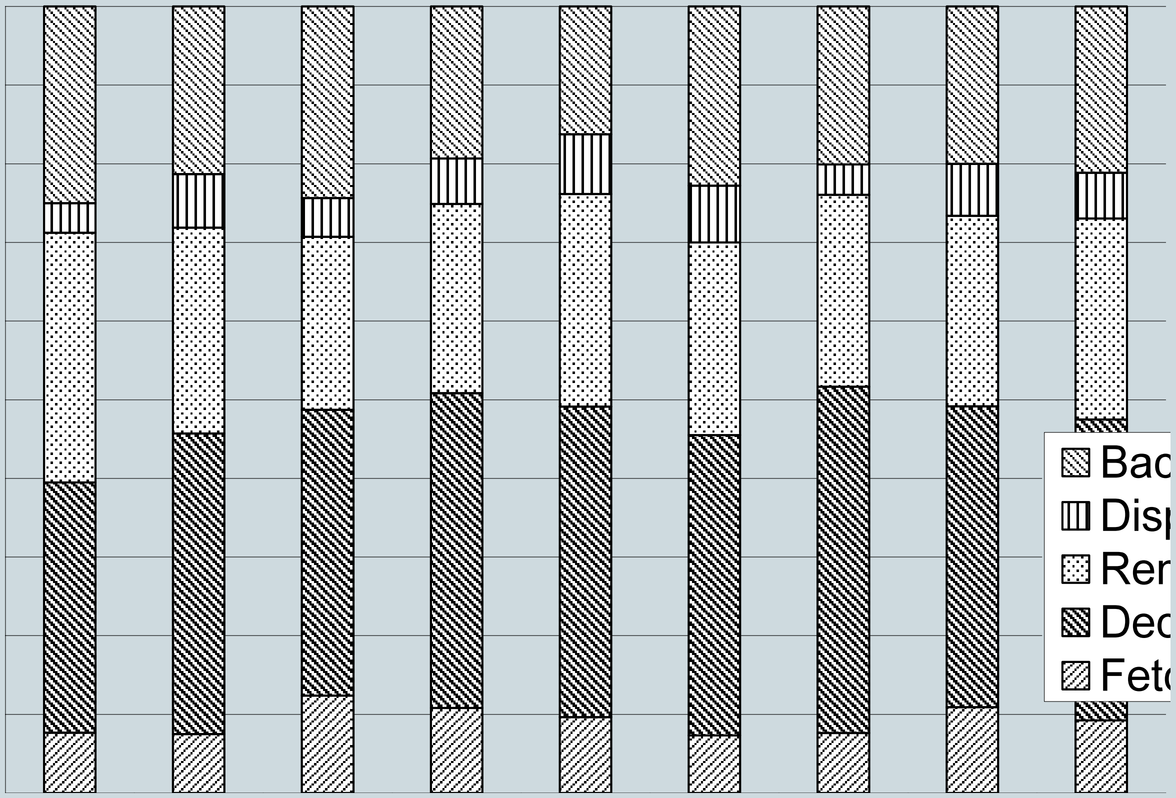


Table 1. Table of faults for all pipeline stages.

Pipe Stage	Fault	Description
Fetch	FETCH_PC	Flip a random bit in the program counter
Fetch	WRONG_INSTR	Remove an arbitrary number of fetched instructions
Fetch	NEXT_PC	Flip a random bit in the override PC from the branch pre-decode/BTB verification stage
Fetch	INSTR_DISAPP	Mask a randomly selected instruction from fetched instructions
Fetch	FETCHQ	Flip a randomly selected bit in the tail/head pointer of the fetch queue
Decode	OPCODE	Flip a random bit in an instruction's opcode
Decode	FLAGS	Flip a random bit in an instruction's decode flags
Decode	SHAMT	Flip a random bit in an instruction's logical/arithmetic shift quantity
Decode	SRC_LOG_REG	Flip a random bit in an instruction's logical source register specifier
Decode	SRCA_LOG_REG	Flip a random bit in an instruction's logical address source register specifier
Decode	RDST_LOG_REG	Flip a random bit in an instruction's logical destination register specifier
Decode	LAT	Flip a random bit in an instruction's latency
Decode	IMM	Flip a random bit in an instruction's signed immediate value field
Decode	UIMM	Flip a random bit in an instruction's unsigned immediate value field
Decode	TARG	Flip a random bit in an instruction's branch target address
Decode	NUM_RSRC	Flip a random bit in an instruction's source operand count
Decode	NUM_RSRCA	Flip a random bit in an instruction's source operand count, address operand
Decode	NUM_RDST	Flip a random bit in an instruction's destination operand count
Decode	IS_DECISION	Flip the bit which indicates whether an instruction is a control-flow decision instruction
Decode	LEFT	Flip the bit indicating left shift of data (LWL/SWLinstructions)
Decode	RIGHT	Flip the bit indicating right shift of data LWR/SWR instructions)
Decode	SIZE	Flip a random bit indicating the size of data (load/store instructions)
Rename	REN_MAP_TABLE	Flip a random bit of a random mapping in the rename map table
Rename	ARCH_MAP_TABLE	Flip a random bit of a random mapping in the architecture map table
Rename	SHADOW_MAP_TABLE	Flip a random bit of a random mapping in a shadow map table
Rename	FREE_LIST	Flip a random bit of an entry in the physical register free list
Rename	FREE_LIST_TAIL	Flip a random bit of the physical register free list's tail pointer
Rename	CHKPT	Randomly pick a shadow map table and flip its availability (used-->free)
Rename	REN_MAP_DEST_INDEX	Flip a random bit in the index used to write a new mapping to the rename map table
Rename	REN_MAP_SRC_INDEX	Flip a random bit in the index used to read a source mapping from the rename map table

RFT: Results – Fault Locations

Fetch – 9%
Decode – 39%
Rename – 24%
Dispatch – 7%
Backend – 21%

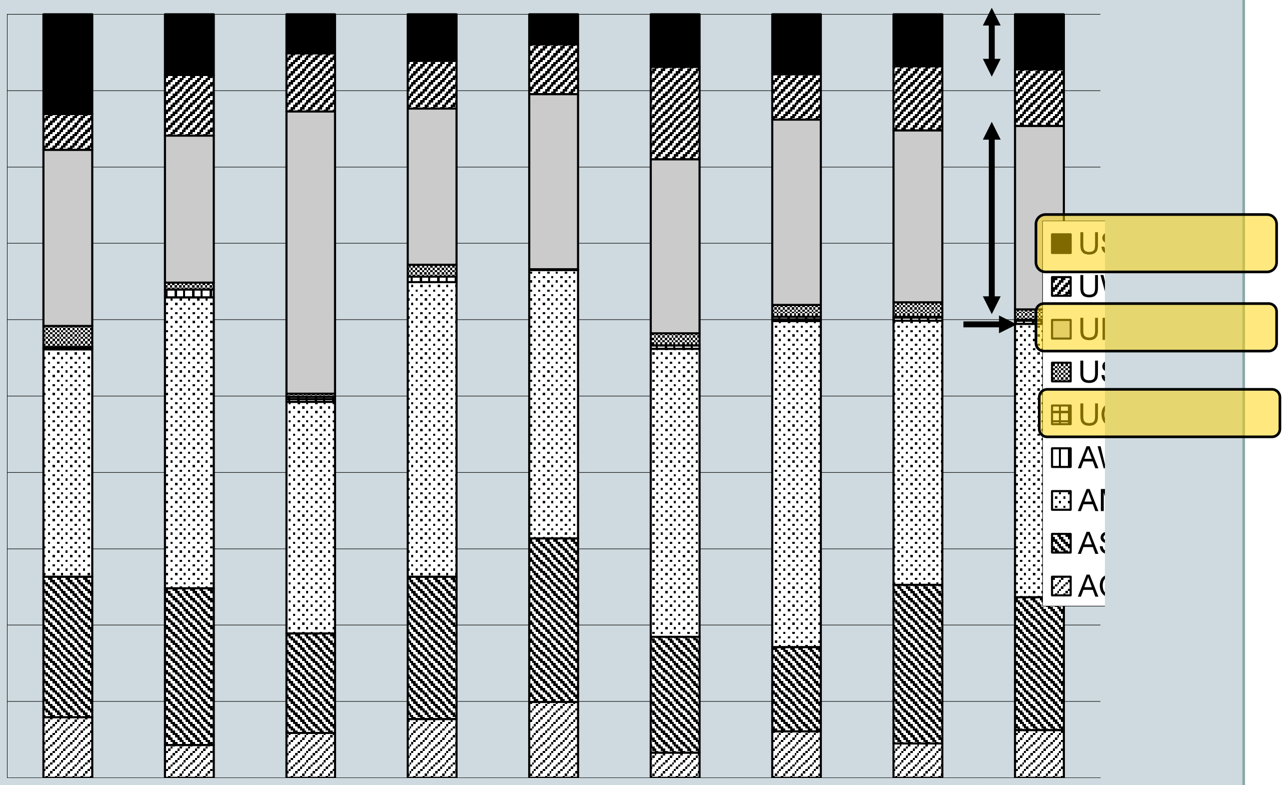


RFT: Results – Fault Outcomes

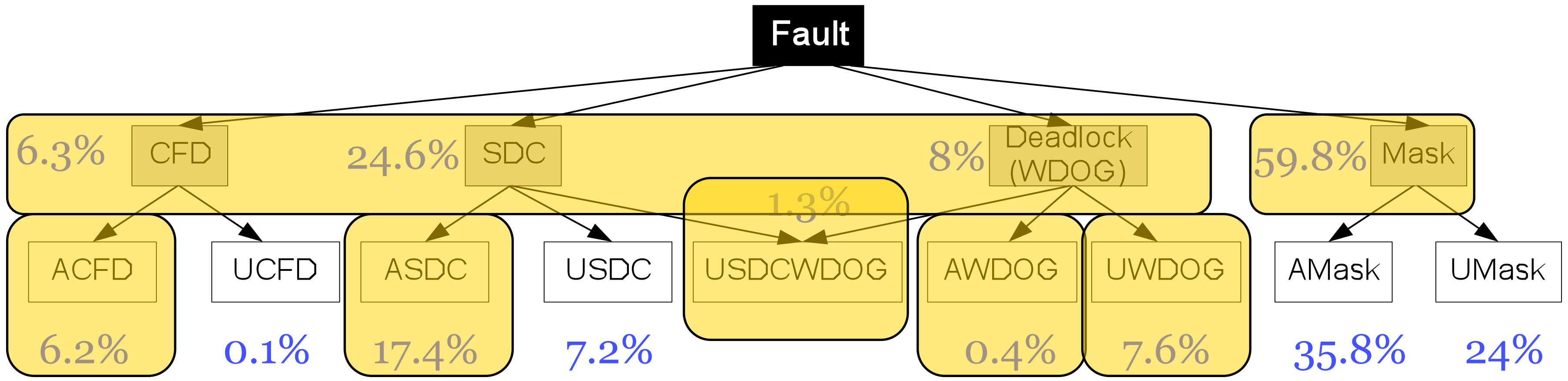
Faults detected by the regimen – 60%

Faults detected by watchdog – 9%

Faults undetected – 31%



RFT: Results (Cont.)



Non-masked faults = 40.2%

Non-masked faults detected by regimen = 24% (60% reduction in vulnerability)

Non-masked faults detected by watchdog = 9% (23% reduction in vulnerability)

Non-masked faults detected by regimen + watchdog = 33% (~83% of non-masked faults get detected)

Summary

- RFT presented a regimen of μarch-level fault checks to protect a superscalar processor
- Injected a broad spectrum of fault types across all pipeline stages
- Regimen-based approach provides substantial fault protection (detects ~83% of non-masked faults)

Thank You!