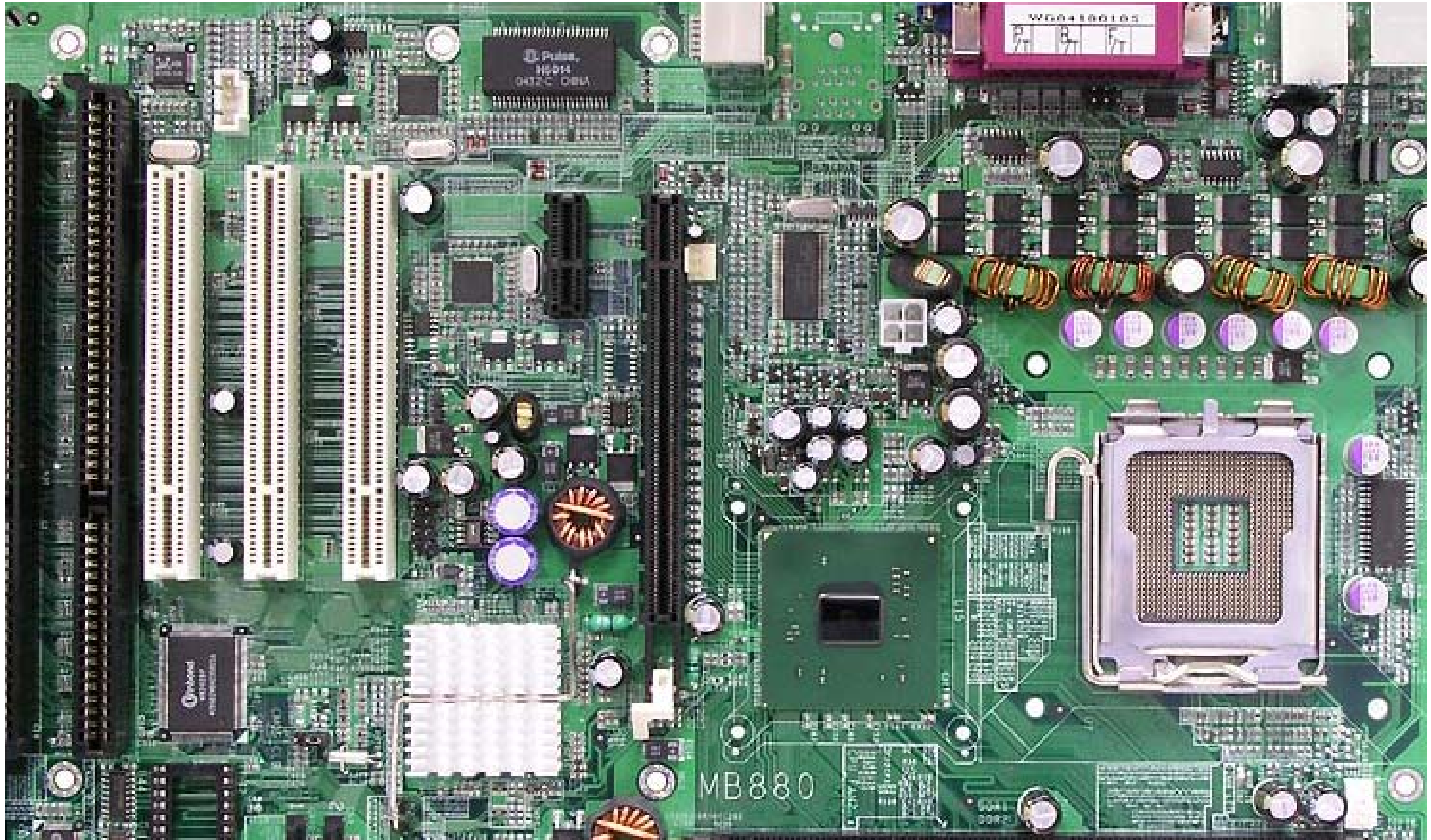
# Fault-Tolerant Computing

# About This Presentation

This presentation has been prepared for the graduate course ECE 257A (Fault-Tolerant Computing) by Behrooz Parhami, Professor of Electrical and Computer Engineering at University of California, Santa Barbara. The material contained herein can be used freely in classroom teaching or any other educational setting. Unauthorized uses are prohibited. © Behrooz Parhami

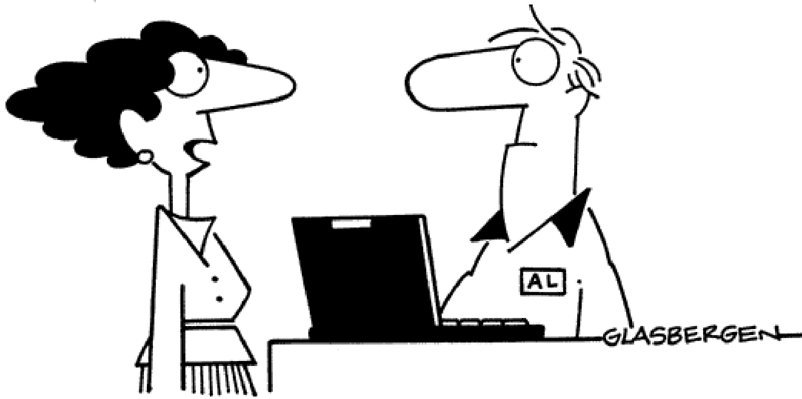
| Edition | Released | Revised | Revised |
|---|---|---|---|
| First | Oct. 2006 | | |

# Hardware Implementation Strategies



## Computer Repairs

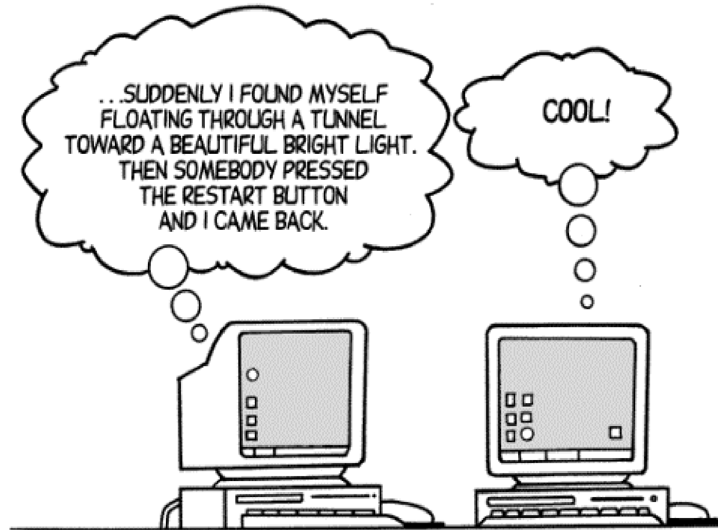
© 2000 Randy Glasbergen.
www.glasbergen.com



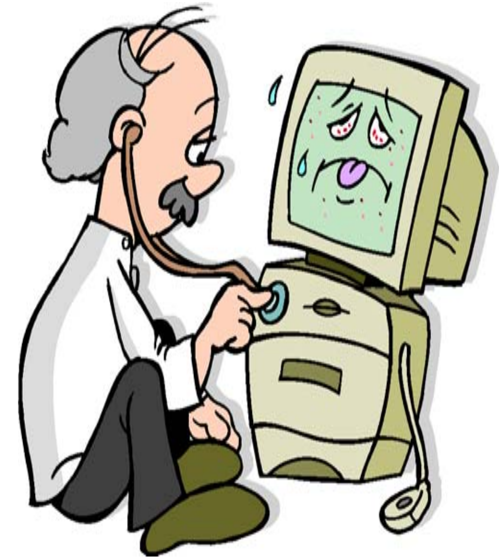

"When I'm away from my desk it goes into sleep mode...but the snoring annoys my coworkers!"

"Crashing is an expression of hostility against your network administrator. Though you appear to be uncooperative, it's actually a desperate cry for help."

www.generalcomics.com




© 1998 Randy Glasbergen. www.glasbergen.com E-mail: randy@glasbergen.com



# Multilevel Model of Dependable Computing

**Level →** Component | Logic | Information | System | Service | Result

Ideal
Defective
Faulty
Erroneous
Malfunctioning
Degraded
Failed

Unimpaired
Low-Level Impaired
Mid-Level Impaired
High-Level Impaired

# Hardware-Based Tolerance/Recovery Methods

**Data path methods:**
Replication in space (costly)
- Duplicate and compare
- Triplicate and vote
- Pair-and-spare
- NMR/hybrid

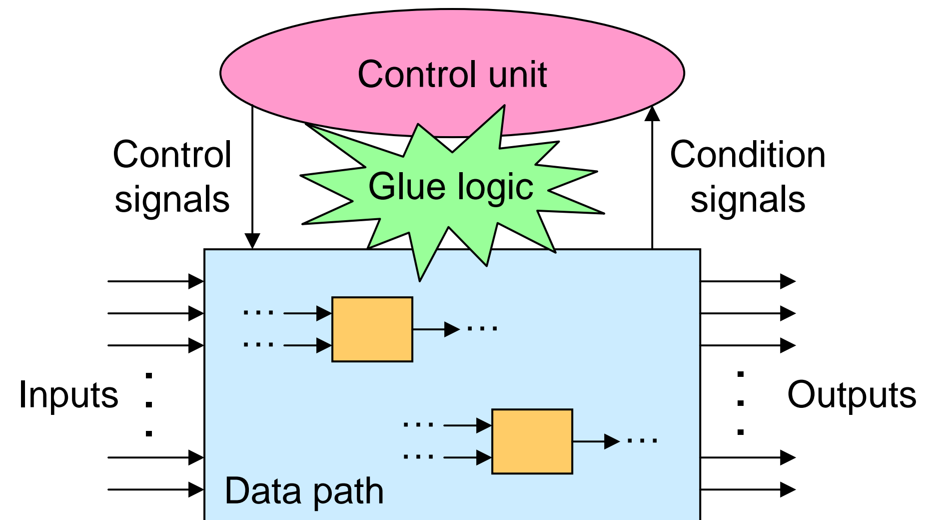
Replication in time (slow?)
- Recompute and compare
- Recompute and vote
- Alternating logic
- Recompute after shift
- Recompute after swap
- Replicate operand segments

Mixed space-time replication
Monitoring (imperfect coverage)
- Watchdog timer
- Activity monitor

Low-redundancy coding
- Parity prediction
- Residue checking
- Self-checking design

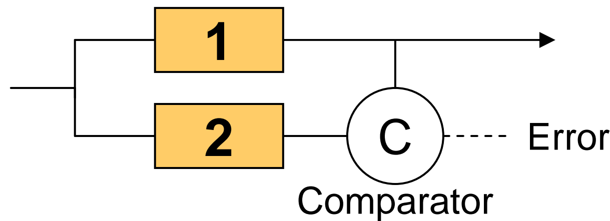
**Control unit methods:**
Coding of control signals
Control-flow watchdog
Self-checking design

**Glue logic methods:**
Self-checking design

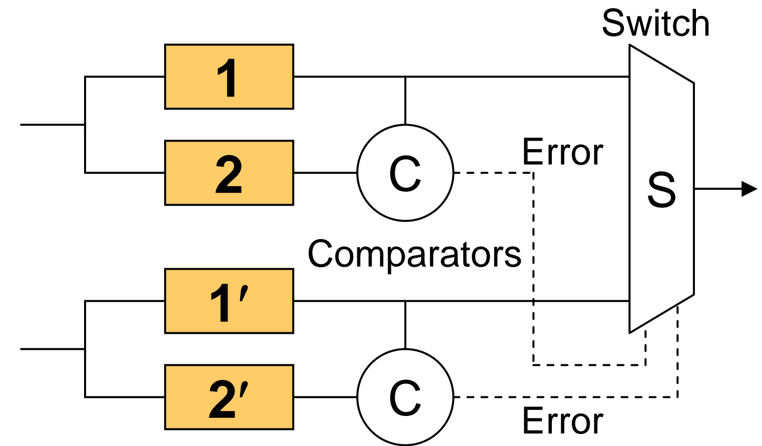


# Replication of Data-Path Elements in Space

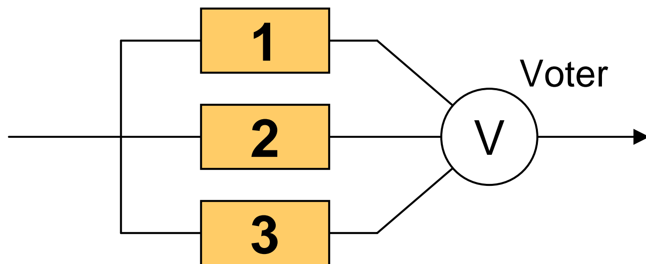
The following schemes have already been discussed in connection with fault tolerance



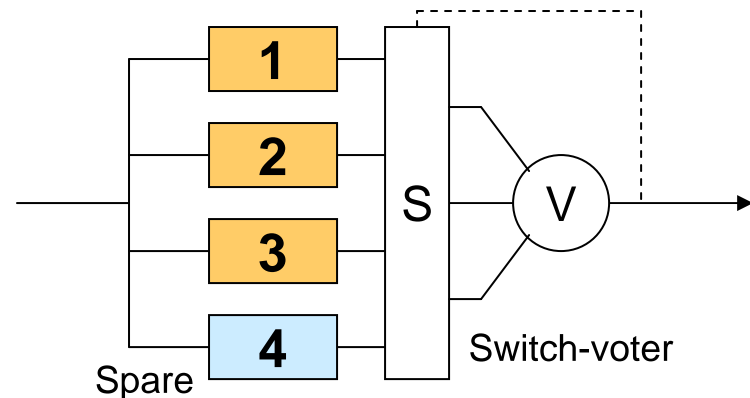

Duplicate and compare




Pair-and-spare




Triplicate and vote

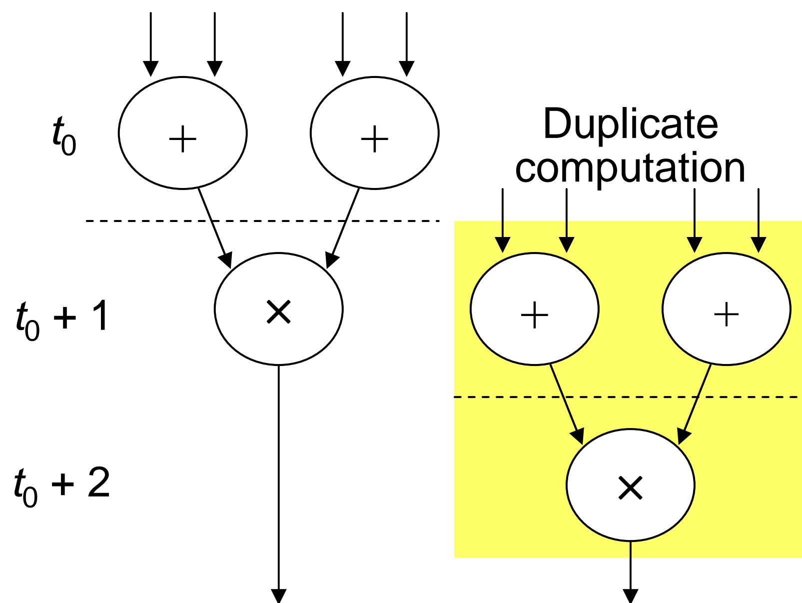



NMR/Hybrid

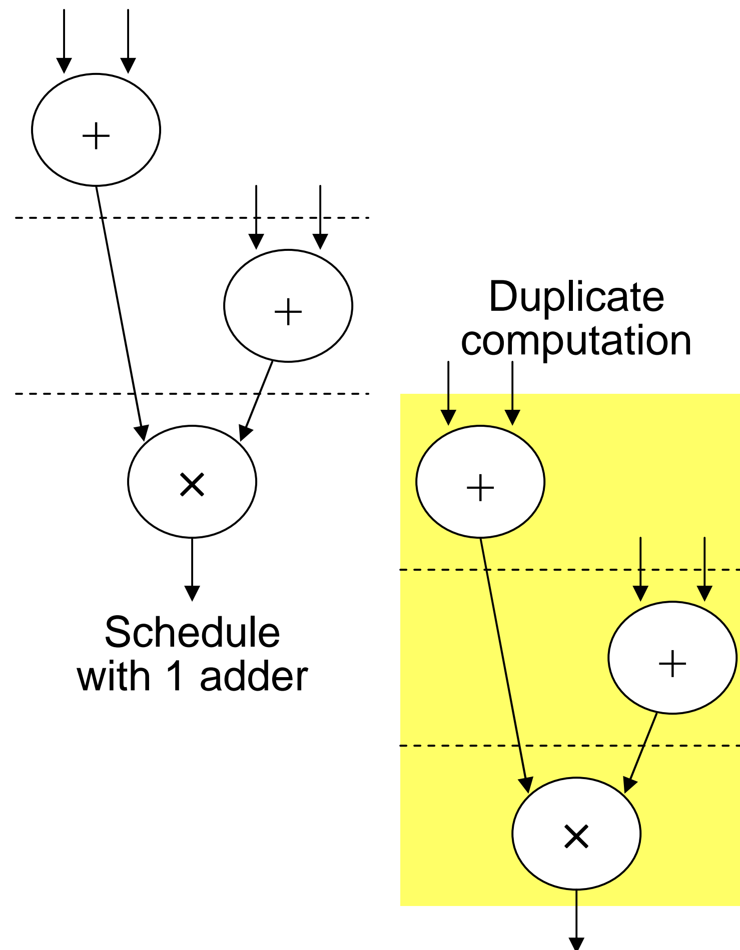
# Main Drawback of Replication in Time

Can be slow, but in many control applications, extra time is available

Interleaving of the primary and duplicate computations saves time



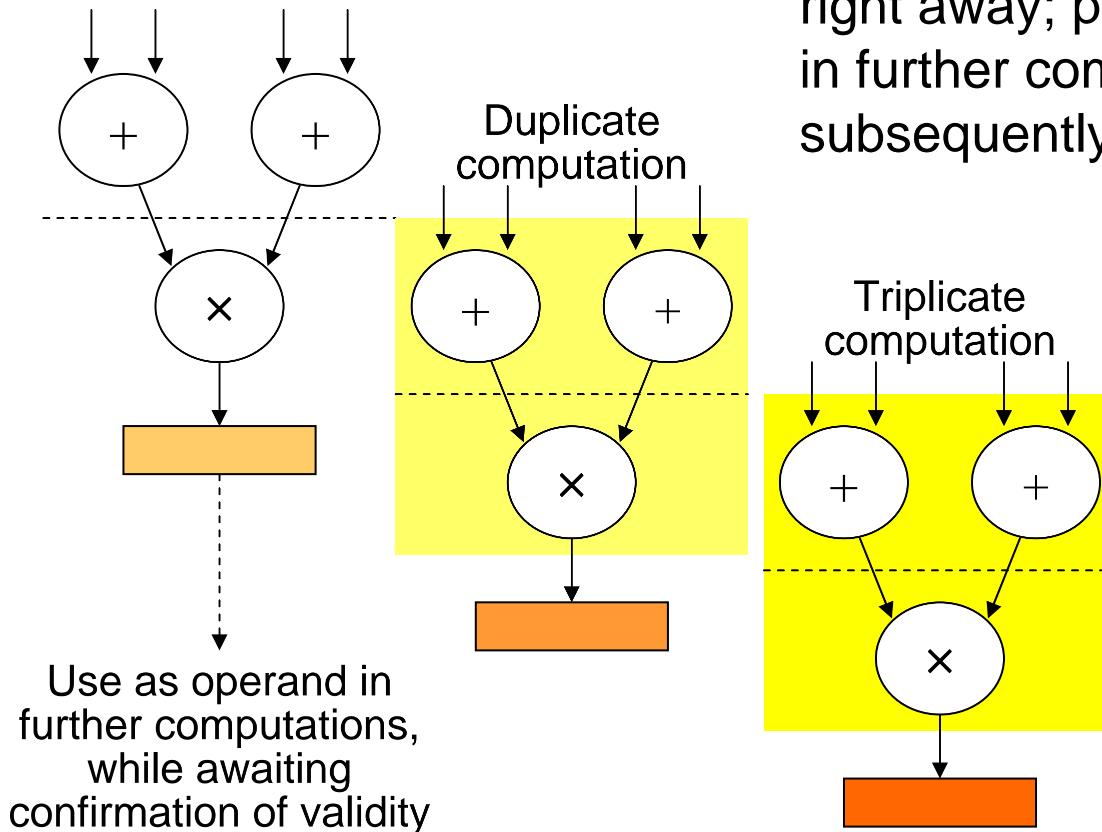

Computation flowgraph, and schedule with 2 adders




Schedule with 1 adder

# Recompute and Compare/Vote

Repeat computation and store the results for comparison or voting




Comparison or voting need not be done right away; primary result may be used in further computations, with the result subsequently validated, if appropriate

On a simultaneous multithreading architecture, two instruction streams may be interspersed

Some Cray machines take advantage of extensive hardware resources to execute instructions twice

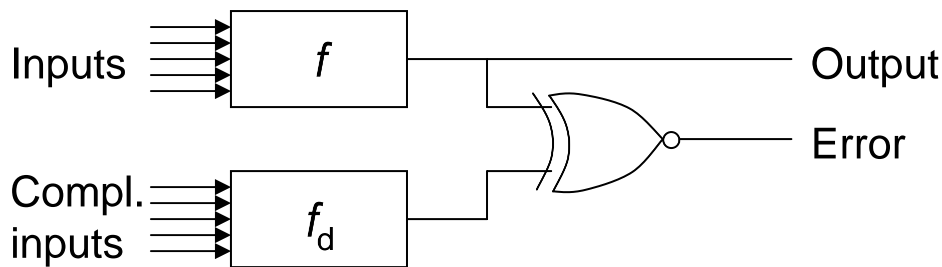
# Alternating Logic: Basic Ideas

Transmission of data over unreliable wires or buses

- Send data; store at receiving end
- Send bitwise complement of data
- Compare the two versions
- Detects wires s-a-0 or s-a-1, as well as many transients

The *dual* of a Boolean function $f(x_1, x_2, \ldots, x_n)$ is another function $f_d(x_1, x_2, \ldots, x_n)$ such that $f_d(x_1', x_2', \ldots, x_n') = f'(x_1, x_2, \ldots, x_n)$

Fact: Obtain the dual of $f$ by exchanging AND and OR operators in its logical expression. For example, the dual of $f = ab \vee c$ is $f_d = (a \vee b)c$

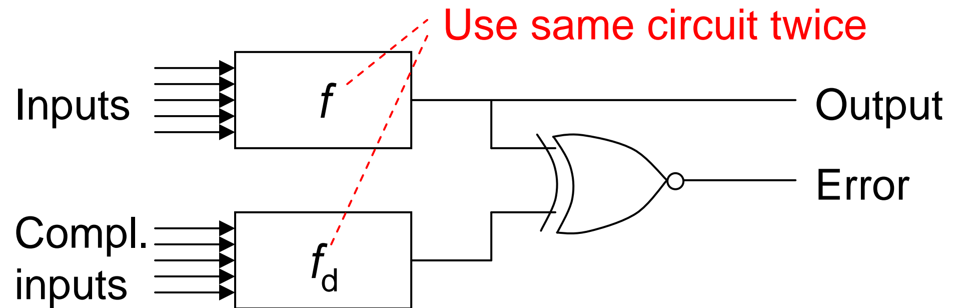



Advantages of this approach compared to duplication include a smaller probability of common errors

# Alternating Logic: Self-Dual Functions

A function $f$ is self-dual if $f(x_1, x_2, \ldots, x_n) = f_d(x_1, x_2, \ldots, x_n)$

For example, both the sum $a \oplus b \oplus c$ and carry $ab \vee bc \vee ca$ outputs of a full-adder are self-dual functions




With a self-dual function $f$, the functions $f$ and $f_d$ in the diagram above can be computed by using the same circuit twice (time redundancy)

Many functions of practical interest are self-dual

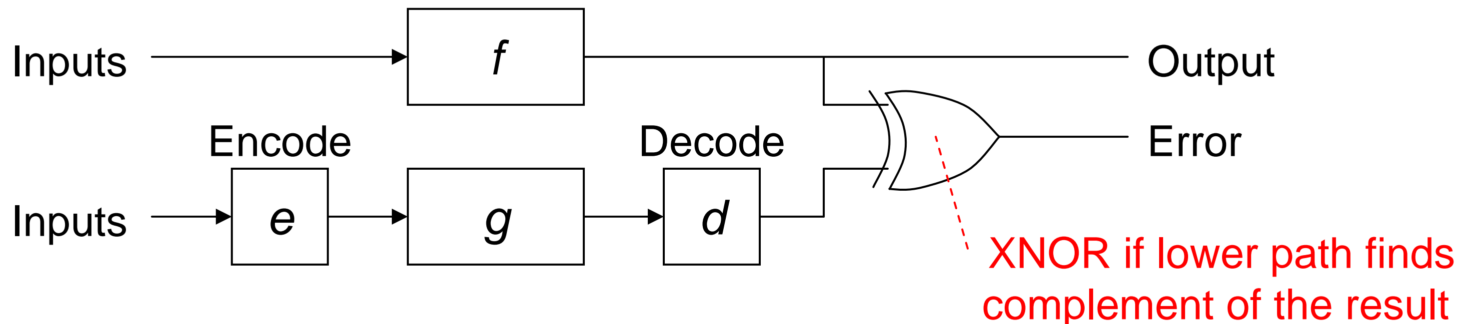
**Examples** (proofs left as exercise)

A $k$-bit binary adder, with $2k + 1$ inputs and $k + 1$ outputs, is self-dual

So are 1's-complement and 2's-complement versions of such an adder

# Recomputing with Transformed Operands

Alternating logic is a special case of the following general scheme, with its encoding and decoding functions being bitwise complementation




**Recompute after shift**

When *f* is binary addition, we can use shifts for encoding and decoding

Shifting causes the adder circuits to be exercised differently each time

Originally proposed for ALUs with bit-slice organization

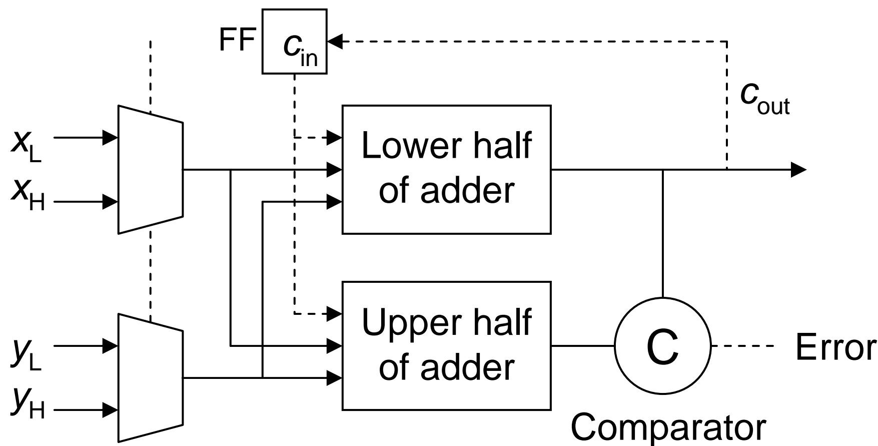
**Recompute after swap**

When *f* is binary addition, we can use swaps for encoding and decoding

- Swap the two operands; e.g., compute $b + a$ instead of $a + b$
- Swap upper and lower halves of the two operands (modified adder)

# Time-Redundant, Segmented Addition

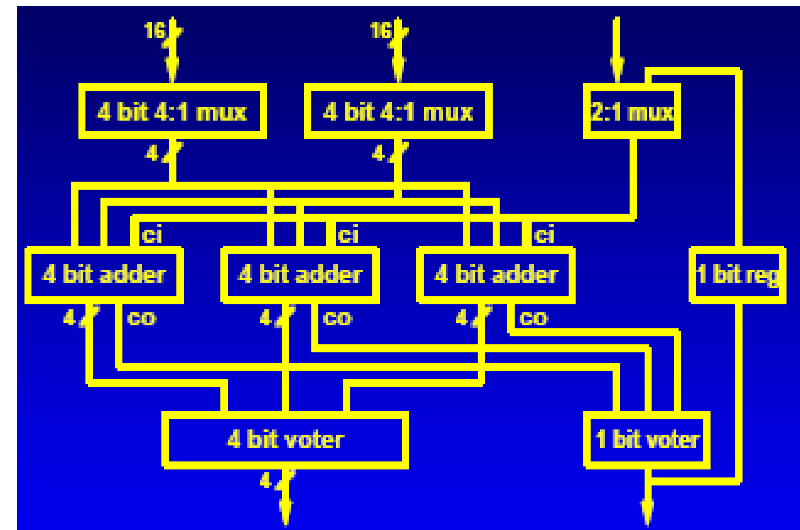
Instead of using a $k$-bit adder twice for error detection or 3 times for error correction, one can segment the operands into 2 or 3 parts and similarly segment the adder; perform replicated addition on operand segments and use comparison/voting to detect/correct error




Sum computed in two cycles: The lower half in cycle 1, and the upper half in cycle 2

Various other segmentation schemes have been suggested

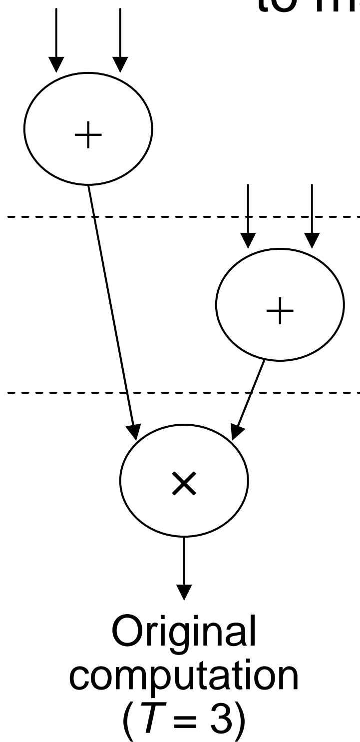
**Example:** 16-bit adder with 4-way segmentation and voting




Townsend, Abraham, and Swartzlander, 2003

# Mixed Space-Time Replication

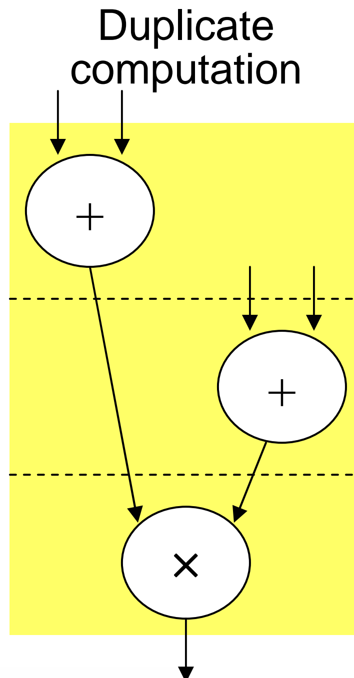
Instead of duplicating the computation with no hardware change (slow) or duplicating the entire hardware (costly), we can add some hardware to make the interleaved recomputations more efficient



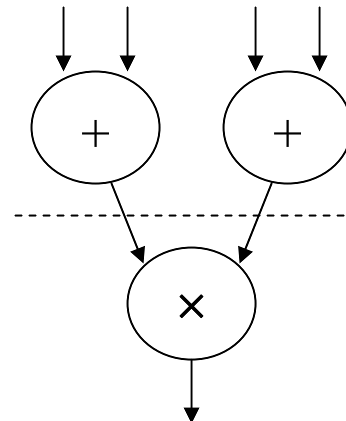

Original computation ($T = 3$)

Recomputation with same hardware resources ($T = 5$, excluding compare time)

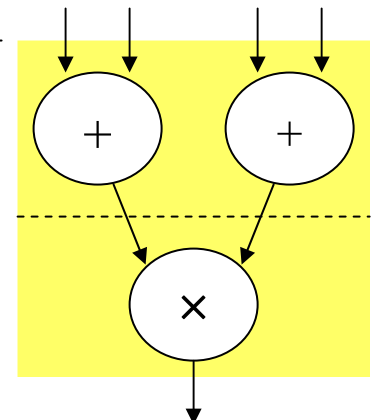
Duplicate computation




Consider the effect of including a second adder



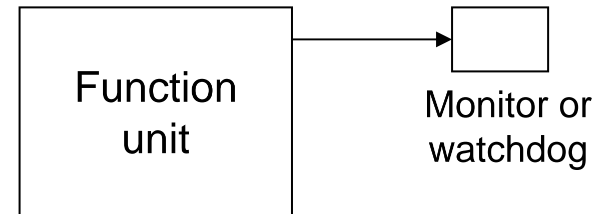

Recomputation with the inclusion of an extra adder ($T = 3$, excluding compare time)



# Monitoring via Watchdog Timers

Monitor or watchdog is a hardware unit that checks on the activities of a function unit

Watchdog is usually much simpler, and thus more reliable, than the unit it monitors




Watchdog timer counts down, beginning from a preset number
It expects to be preset periodically by the unit that it monitors
If the count reaches 0, the watchdog timer raises an exception flag

Watchdog timer can also help in monitoring unit interactions
When one unit sends a request or message, it sets a watchdog timer
If no response arrives within the allotted time, a failure is assumed

Watchdog timer obviously does not detect all problems
It verifies “liveness” of the unit it monitors (good with fail-silent units)
Often used in conjunction with other tolerance/recovery methods

# Activity Monitor

Watchdog unit monitors events occurring in, and activities performed by, the function unit (e.g., event frequency and relative timing)

Observed behavior is compared against expected behavior

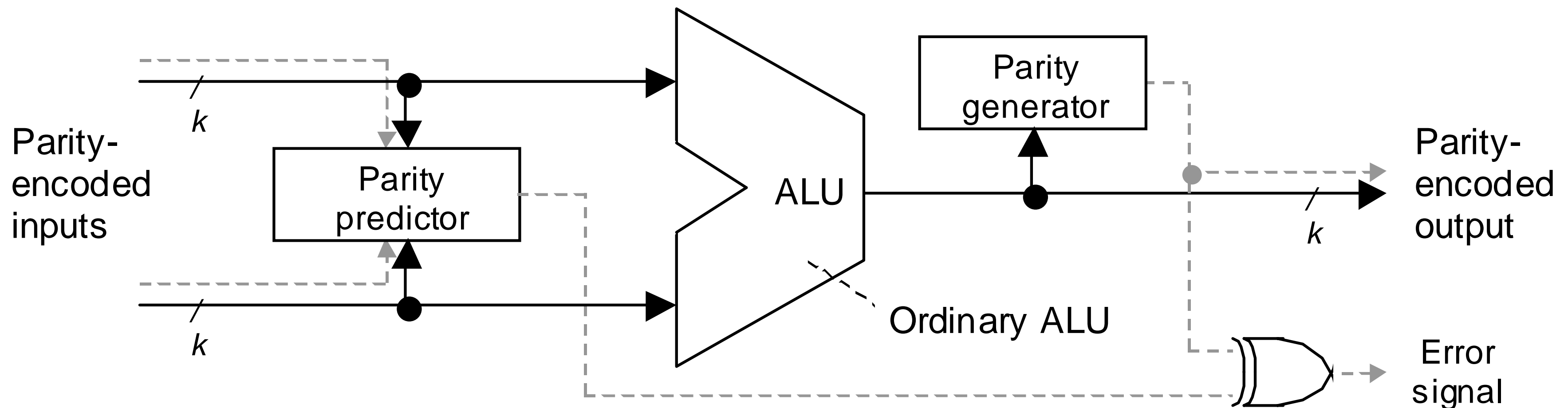
The type of monitoring is highly application-dependent

# Design with Parity Codes and Parity Prediction

Operands and results are parity-encoded
Parity is not preserved over arithmetic and logic operations

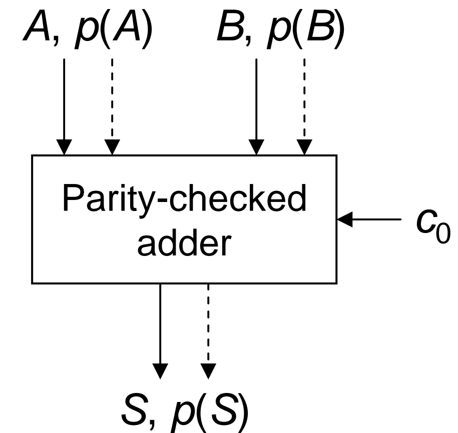
Parity prediction is an alternative to duplication

Compared to duplication:
Parity prediction often involves less overhead in time and space
The protection offered by parity prediction is not as comprehensive



# Parity Prediction for an Adder

| | | |
|---|---|---|
| Operand *A*: | 1 0 1 1 0 0 0 1 | Parity 0 |
| Operand *B*: | 0 0 1 1 1 0 1 1 | Parity 1 |
| $A \oplus B$ | 1 0 0 0 1 0 1 0 | |
| Carries: | 0 0 1 1 0 0 1 1 | Parity 0 |
| Sum *S*: | 1 1 1 0 1 1 0 0 | Parity 1 |



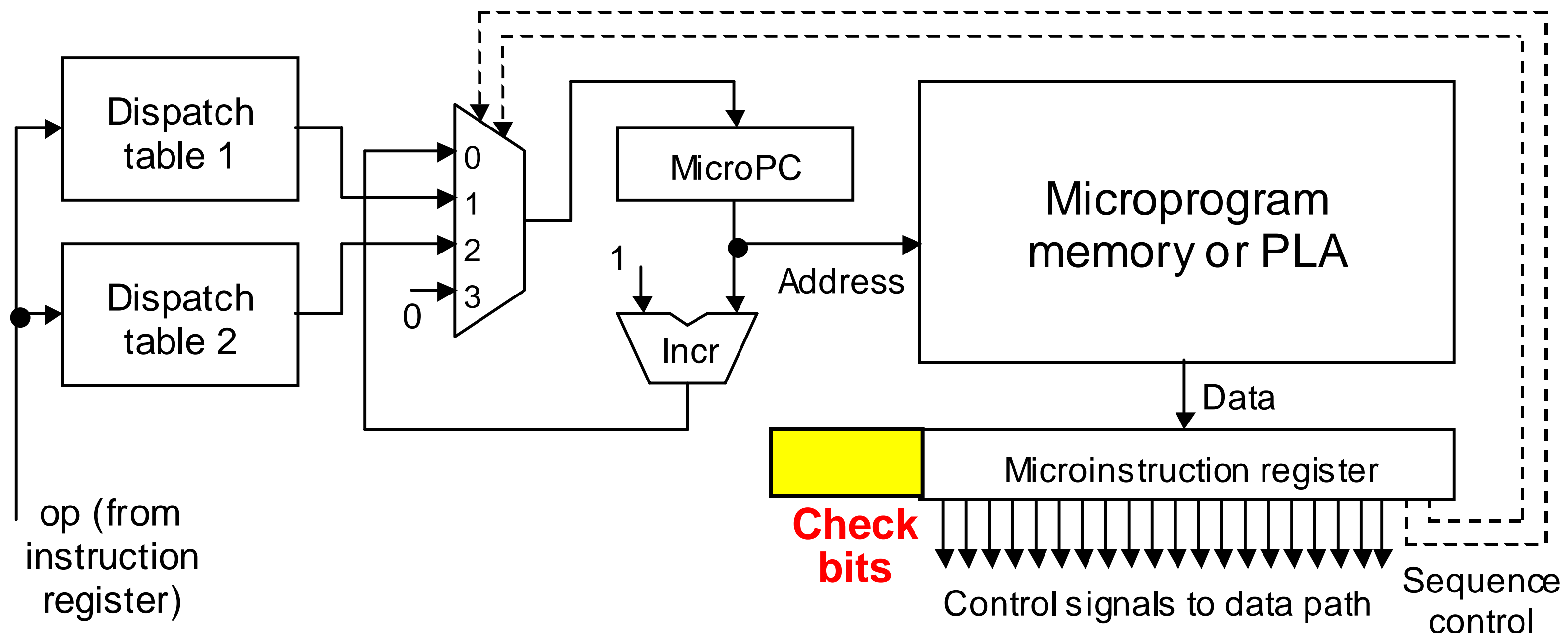

$$p(S) = \underbrace{p(A) \oplus p(B) \oplus c_0}_{\text{Inputs}} \oplus \underbrace{c_1 \oplus c_2 \oplus \ldots \oplus c_k}_{\text{Must compute second versions of these carries to ensure independence}}$$

Parity predictor for our adder consists of a duplicate carry network and an XOR tree

# Coding of Control Signals

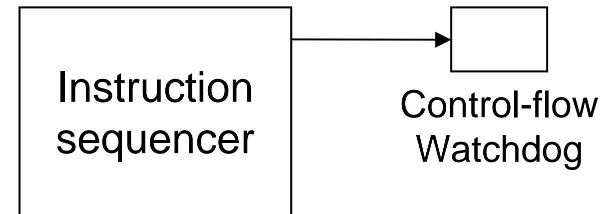
Encode the control signals using a separable code (e.g., Berger code)
Either check in every cycle, or form a signature over multiple cycles

In a microprogrammed control unit, store the microinstruction address and compare against MicroPC contents to detect sequencing errors



# Control-Flow Watchdog

Watchdog unit monitors the instructions executed and their addresses (for example, by snooping on the bus)




The watchdog unit may have certain info about program behavior
- Control flow graph (valid branches and procedure calls)
- Signatures of branch-free intervals (consecutive instructions)
- Valid memory addresses and required access privileges

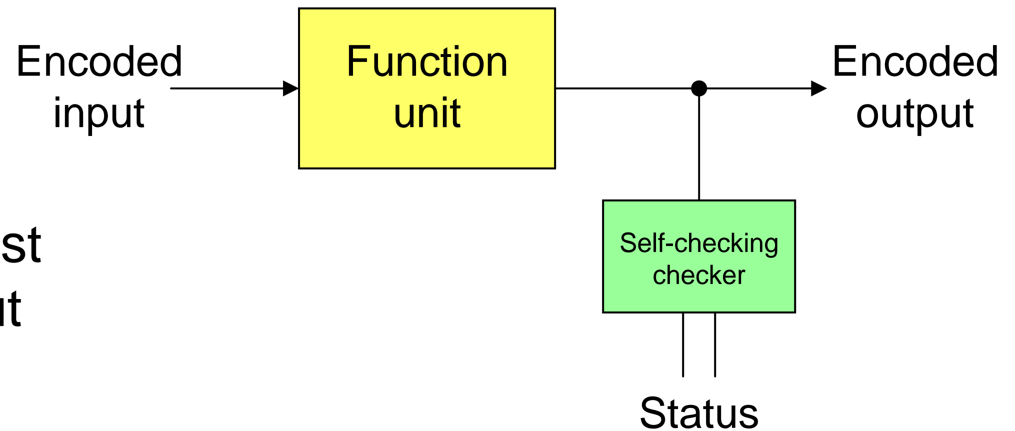
In an application-specific system, watchdog info is preloaded in it
For a GP system, compiler can insert special watchdog directives

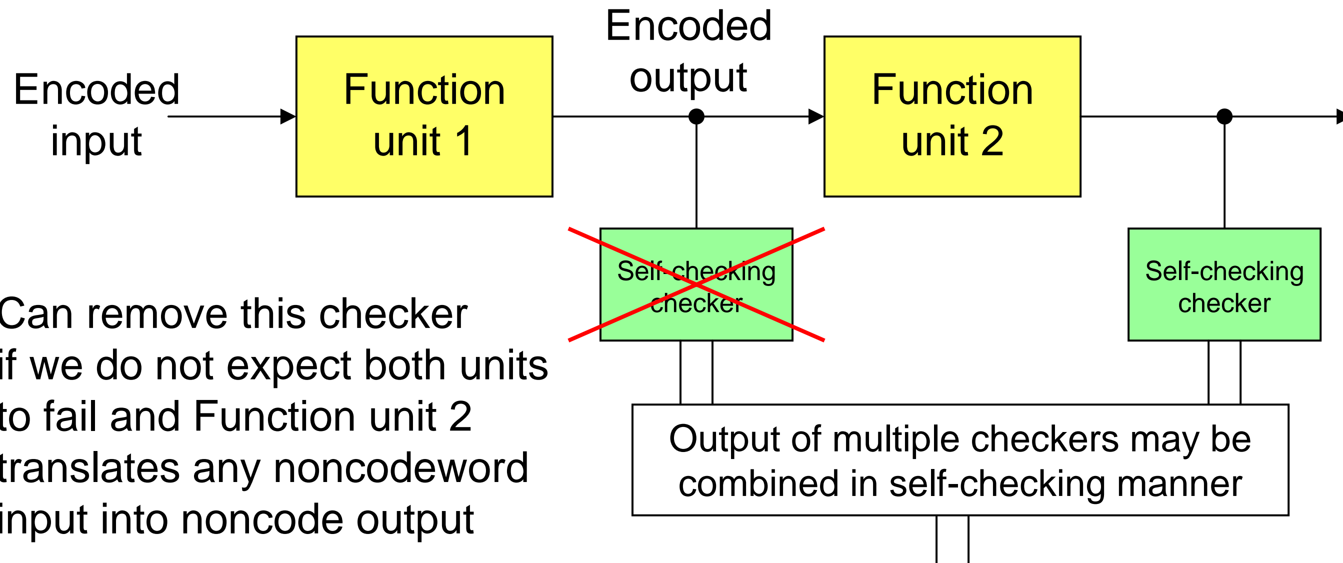
Overheads of control-flow checking
- Wider memory due to the need for tag bits to distinguish word types
- Additional memory to store signatures and other watchdog info
- Stolen processor/bus cycles by the watchdog unit

# Preview of Self-Checking Design

Covered in next lecture

Function unit designed such that internal faults manifest themselves as an invalid output




Can remove this checker if we do not expect both units to fail and Function unit 2 translates any noncodeword input into noncode output