EECS150 - Digital Design

Lecture 24 - High-Level Design (Part 3) + ECC

April 12, 2012

John Wawrzynek

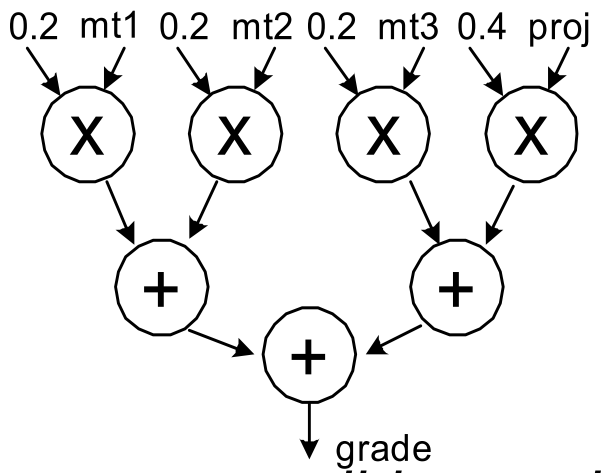
Parallelism

Parallelism is the act of ***doing more than one thing at a time****.*
Optimization in hardware design often involves using parallelism to trade between cost and performance.

- Example, Student final grade calculation:

```
read mt1, mt2, mt3, project;
grade = 0.2 × mt1 + 0.2 × mt2
            + 0.2 × mt3 + 0.4 × project;
write grade;
```

- High performance hardware implementation:



As many operations as possible are done in parallel.

Optimizing Iterative Computations

Types of loops:

1) Looping over input data (streaming):
   - ex: MP3 player, video compressor, music synthesizer.

2) Looping over memory data
   - ex: vector inner product, matrix multiply, list-processing

- 1) & 2) are really very similar. 1) is often turned into 2) by buffering up input data, and processing “offline”. Even for “online” processing, buffers are used to smooth out temporary rate mismatches.

3) CPUs are one big loop.
   - Instruction fetch ⇒ execute ⇒ Instruction fetch ⇒ execute ⇒ …
   - but change their personality with each iteration.

4) Others?

Loops offer opportunity for parallelism
by executing more than one iteration at once,
using parallel iteration execution &/or pipelining

Pipelining Principle

- With looping usually we are less interested in the latency of one iteration and more in the loop execution rate, or throughput.
- These can be different due to *parallel iteration execution &/or pipelining.*
- Pipelining review from CS61C:

  Analog to washing clothes:

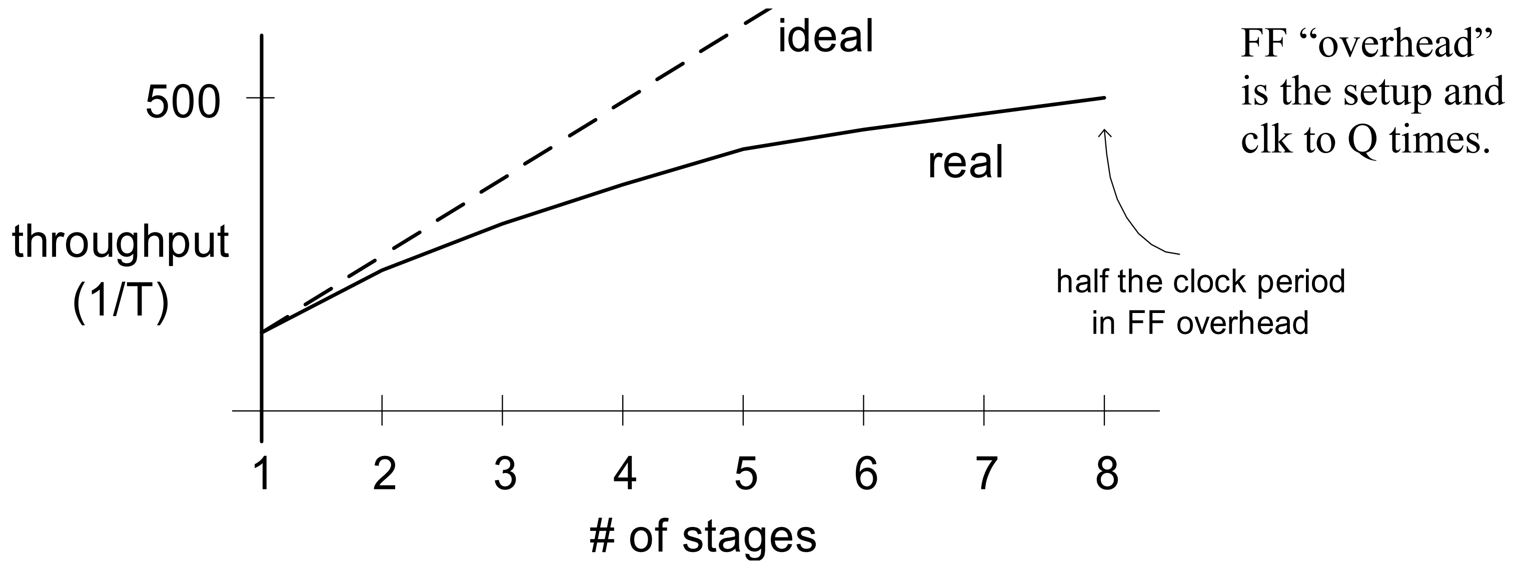
step 1:	wash	(20 minutes)
step 2:	dry	(20 minutes)
step 3:	fold	(20 minutes)
		60 minutes x 4 loads ⇒ 4 hours

wash	load1	load2	load3	load4		
dry		load1	load2	load3	load4	
fold			load1	load2	load3	load4
	20 min					

overlapped ⇒ 2 hours

Limits on Pipelining

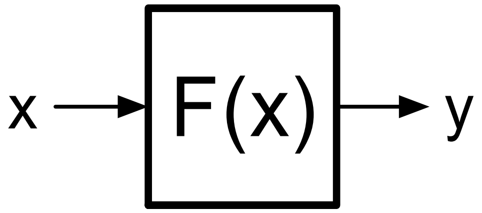
- Without FF overhead, throughput improvement α # of stages.
- After many stages are added FF overhead begins to dominate:



- Other limiters to effective pipelining:
  - clock skew contributes to clock overhead
  - unequal stages
  - FFs dominate *cost*
  - clock distribution power consumption
  - feedback (dependencies between loop iterations)

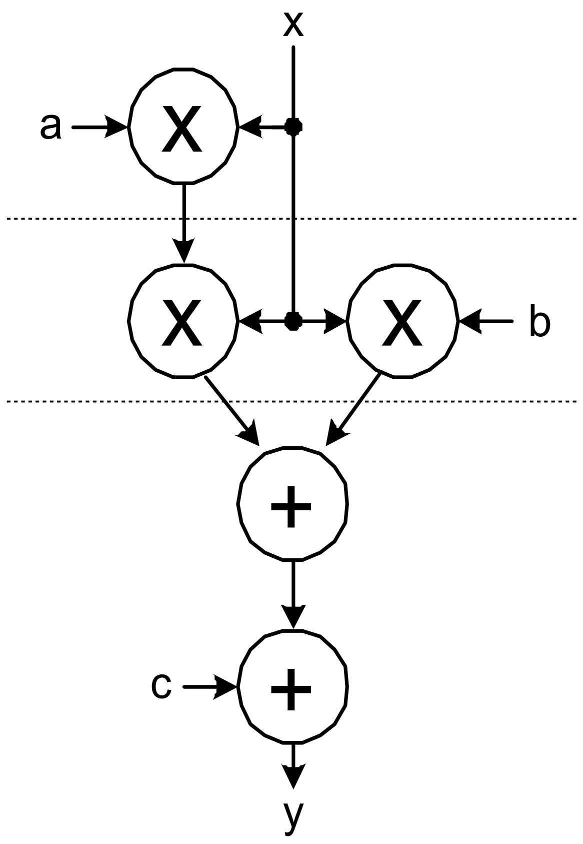
Pipelining Example

- $F(x) = y_i = a\, x_i^2 + b\, x_i + c$



- x and y are assumed to be "streams"
- Divide into 3 (nearly) equal stages.
- Insert pipeline registers at dashed lines.
- Can we pipeline basic operators?

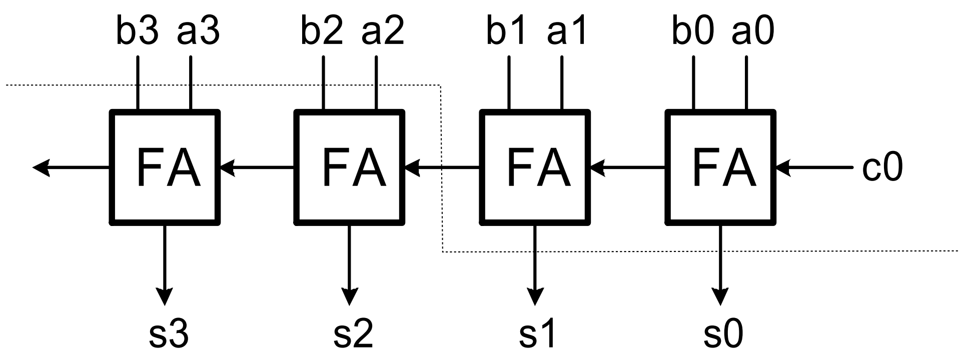
- Computation graph:

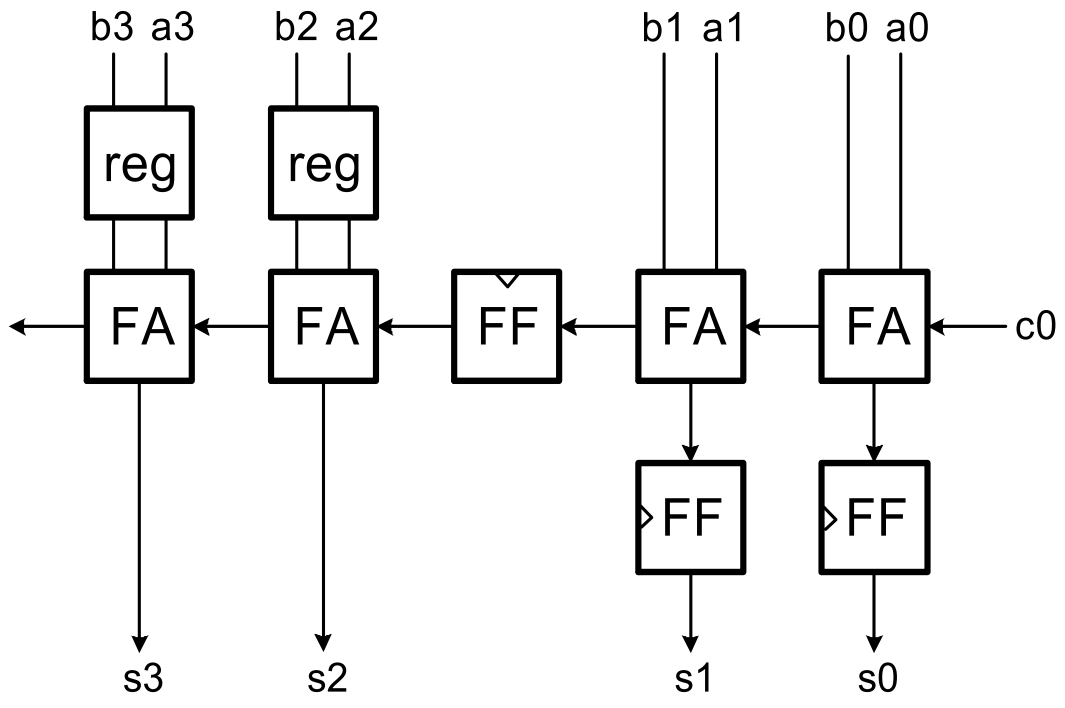


Example: Pipelined Adder

- Possible, but usually not done.

(arithmetic units can often be made sufficiently fast without internal pipelining)

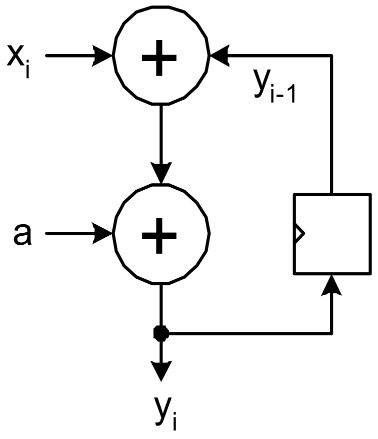




Pipelining Loops with Feedback

"Loop carry dependency"

- Example 1: $\mathbf{y_i = y_{i-1} + x_i + a}$



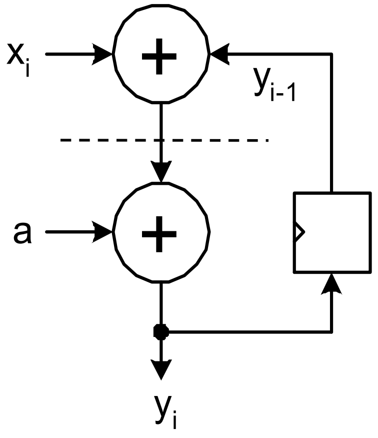
unpipelined version:

add_1	x_i+y_{i-1}	$x_{i+1}+y_i$
add_2	y_i	y_{i+1}

Can we "cut" the feedback and overlap iterations?

Try putting a register after add1:

add_1	x_i+y_{i-1}		$x_{i+1}+y_i$	
add_2		y_i		y_{i+1}

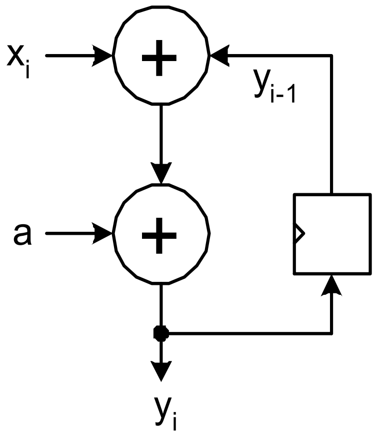


- Can't overlap the iterations because of the dependency.
- The extra register doesn't help the situation (actually hurts).
- In general, can't pipeline feedback loops.

Pipelining Loops with Feedback

"Loop carry dependency"

However, we can overlap the "non-feedback" part of the iterations:

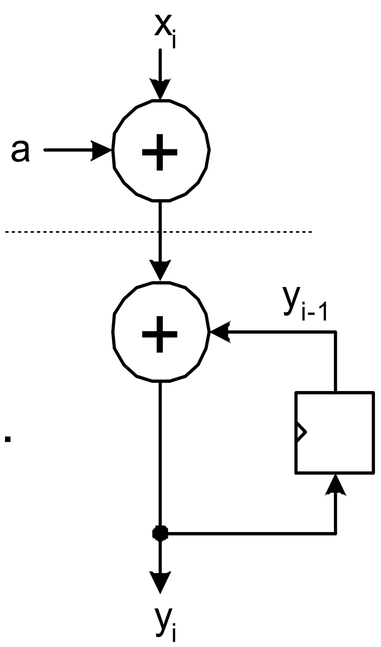


Add is associative and communitive. Therefore we can reorder the computation to shorten the delay of the feedback path:

$$\mathbf{y_i = (y_{i-1} + x_i) + a = (a + x_i) + y_{i-1}}$$

add_1	x_i+a	x_{i+1}+a	x_{i+2}+a	
add_2		y_i	y_{i+1}	y_{i+2}

"Shorten" the feedback path.

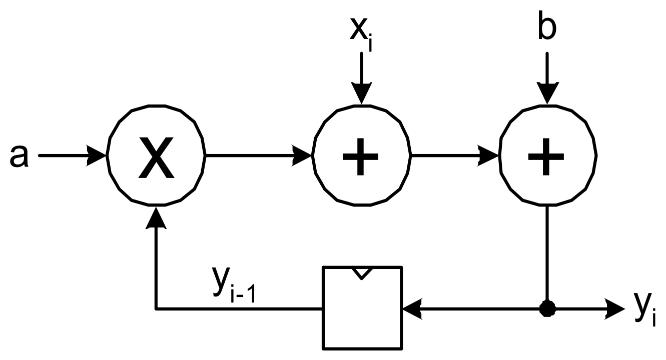


- Pipelining is limited to 2 stages.

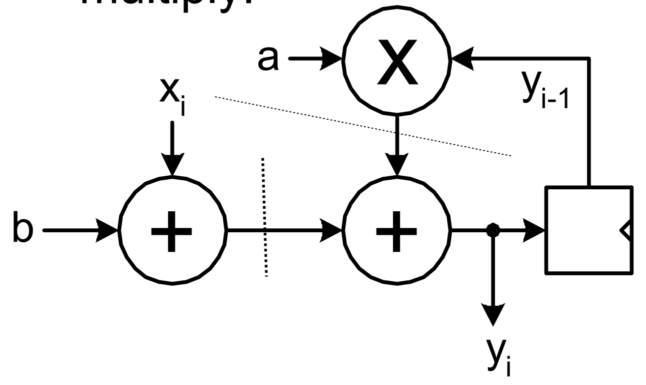
Pipelining Loops with Feedback

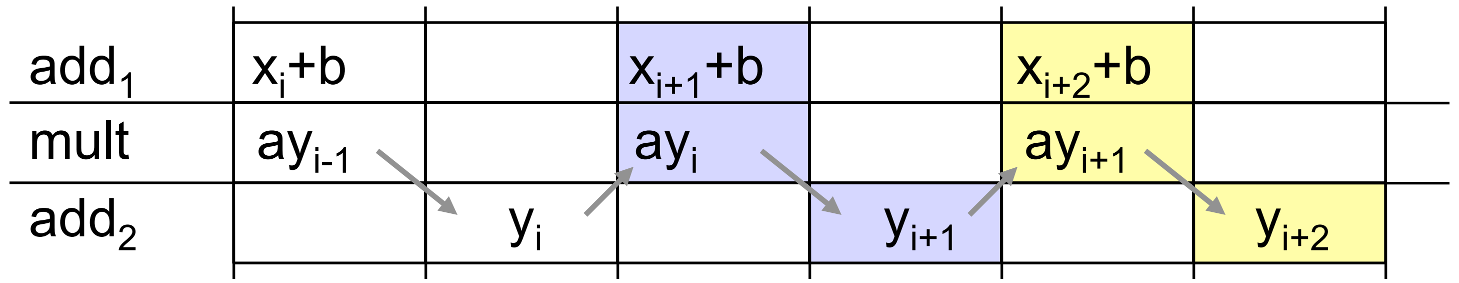
- Example 2:

  $y_i = a\, y_{i-1} + x_i + b$



- Reorder to shorten the feedback loop and try putting register after multiply:





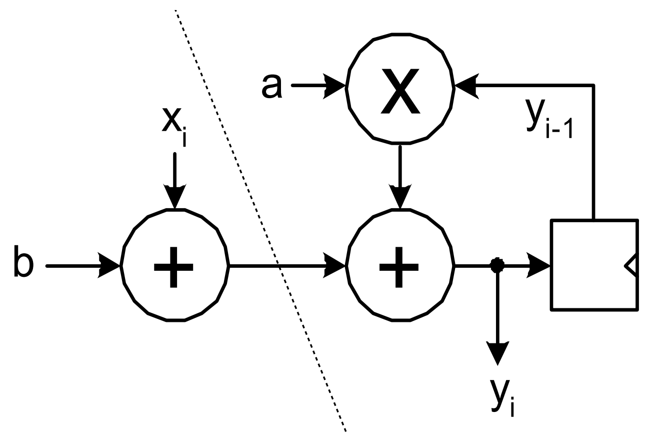
add_1	x_i+b		x_{i+1}+b		x_{i+2}+b	
mult	ay_{i-1}		ay_i		ay_{i+1}	
add_2		y_i		y_{i+1}		y_{i+2}

- Still need 2 cycles/iteration

Pipelining Loops with Feedback

- Example 2:

  $y_i = a\, y_{i-1} + x_i + b$



add_1	x_i+b	x_{i+1}+b	x_{i+2}+b	
mult		ay_{i-1}	ay_i	ay_{i+1}
add_2		y_i	y_{i+1}	y_{i+2}

- Once again, adding register doesn't help. Best solution is to overlap non-feedback part with feedback part.
- Therefore critical path includes a multiply in series with add.
- Can overlap first add with multiply/add operation.
- Only 1 cycle/iteration. Higher performance solution (than 2 cycle version).

- Alternative is to move register to after multiple, but same critical path.

"C-slow" Technique

- Another approach to increasing throughput in the presence of feedback: try to fill in "holes" in the chart with another (independent) computation:

add_1	x_i+b		$x_{i+1}+b$		$x_{i+2}+b$	
mult	ay_{i-1}		ay_i		ay_{i+1}	
add_2		y_i		y_{i+1}		y_{i+2}

If we have a second similar computation, can interleave it with the first:

$$x^1 \rightarrow \boxed{F^1} \rightarrow y^1 = a^1\, y^1_{i-1} + x^1_i + b^1$$

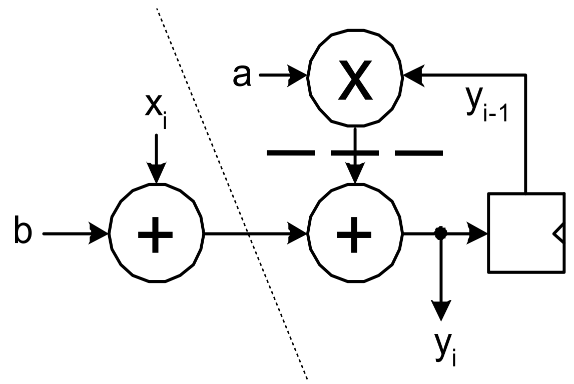
$$x^2 \rightarrow \boxed{F^2} \rightarrow y^2 = a^2\, y^2_{i-1} + x^2_i + b^2$$

Use muxes to direct each stream.
Time multiplex one piece of HW for both stream.
Each produces 1 result / 2 cycles.

- Here the feedback depth=2 cycles (we say C=2).
- Each loop has throughput of F_{clk}/C. But the aggregate throughput is F_{clk}.
- With this technique we could pipeline even deeper, assuming we could supply C independent streams.

"C-slow" Technique

- Essentially this means we go ahead and cut feedback path:



- This makes operations in adjacent pipeline stages independent and allows full cycle for each:

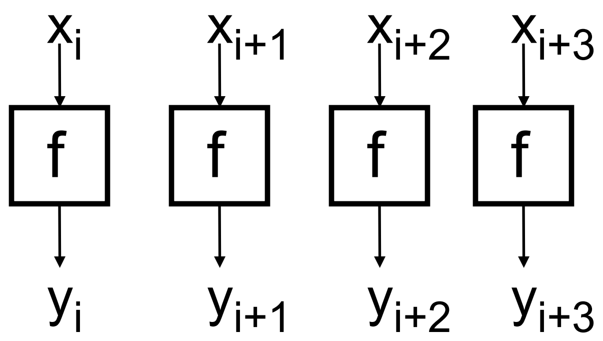
- C computations (in this case C=2) can use the pipeline simultaneously.
- Must be independent.
- Input MUX interleaves input streams.
- Each stream runs at half the pipeline frequency.
- Pipeline achieves full throughput.

Multithreaded Processors use this.

add_1	x+b	x+b	x+b	x+b	x+b	x+b
mult	ay	ay	ay	ay	ay	ay
add_2	y	y	y	y	y	y

Beyond Pipelining - SIMD Parallelism

- An obvious way to exploit more parallelism from loops is to make multiple instances of the loop execution data-path and run them in parallel, sharing the some controller.
- For P instances, throughput improves by a factor of P.
- example: $\mathbf{y_i = f(x_i)}$



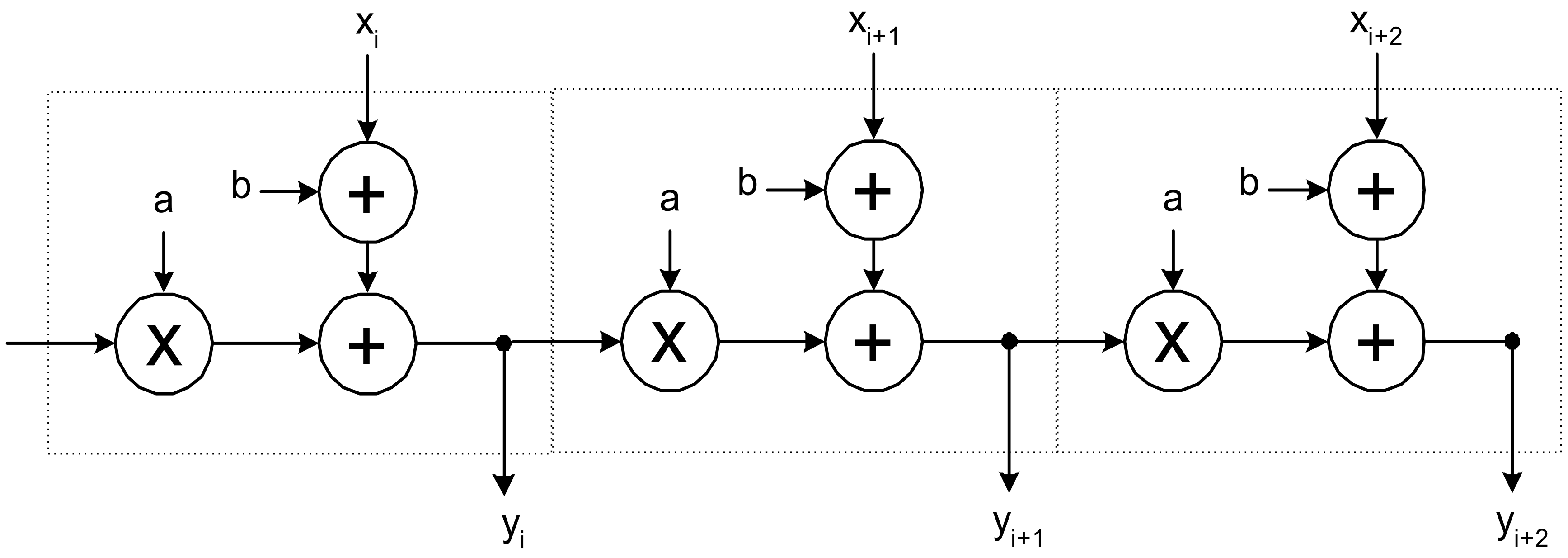
Usually called SIMD parallelism. Single Instruction Multiple Data

-
- Assumes the next 4 x values available at once. The validity of this assumption depends on the ratio of f repeat rate to input rate (or memory bandwidth).
- Cost α P. Usually, much higher than for pipelining. However, potentially provides a high speedup. Often applied after pipelining.
- Limited, once again, by loop carry dependencies. Feedback translates to dependencies between parallel data-paths.
- Vector processors use this technique.

SIMD Parallelism with Feedback

- Example, from earlier:

  $y_i = a\ y_{i-1} + x_i + b$



- In this example end up with "carry ripple" situation.
- Could employ look-ahead / parallel-prefix optimization techniques to speed up propagation.
- As with pipelining, this technique is most effective in the absence of a loop carry dependence.

Error Correction Codes

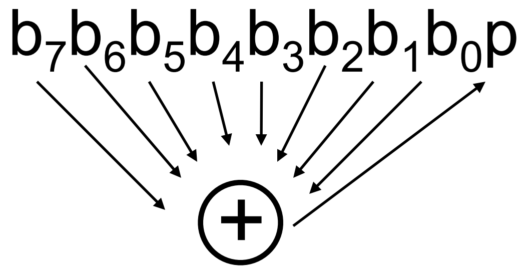
Error Correction Codes (ECC)

- Memory systems generate errors (accidentally flipped-bits)
  - DRAMs store very little charge per bit
  - "Soft" errors occur occasionally when cells are struck by alpha particles or other environmental upsets.
  - Less frequently, "hard" errors can occur when chips permanently fail.
- Where "perfect" memory is required
  - servers, spacecraft/military computers, …
- Memories are protected against failures with ECCs
- Extra bits are added to each data-word
  - extra bits are used to detect and/or correct faults in the memory system
  - in general, each possible data word value is mapped to a unique "code word". A fault changes a valid code word to an invalid one - which can be detected.

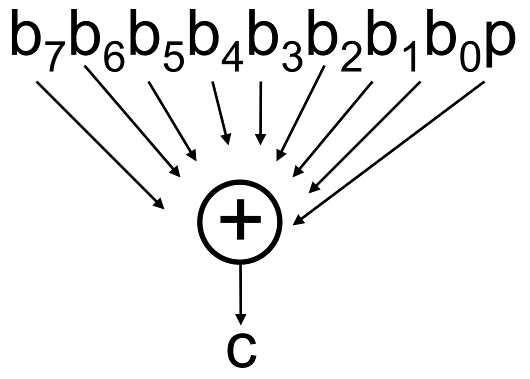
Simple Error Detection Coding

Parity Bit

- Each data value, before it is written to memory is "tagged" with an extra bit to force the stored word to have *even parity*:



- Each word, as it is read from memory is "checked" by finding its parity (including the parity bit).



- A non-zero parity indicates an error occurred:
  - two errors (on different bits) is not detected (nor any even number of errors)
  - odd numbers of errors are detected.

Hamming Error Correcting Code

- Use more parity bits to pinpoint bit(s) in error, so they can be corrected.
- Example: Single error correction (SEC) on 4-bit data
  - use 3 parity bits, with 4-data bits results in 7-bit code word
  - 3 parity bits sufficient to identify any one of 7 code word bits
  - overlap the assignment of parity bits so that a single error in the 7-bit word can be corrected
- **Procedure**: group parity bits so they correspond to subsets of the 7 bits:
  - p_1 protects bits 1,3,5,7
  - p_2 protects bits 2,3,6,7
  - p_3 protects bits 4,5,6,7

1	2	3	4	5	6	7
p_1	p_2	d_1	p_3	d_2	d_3	d_4

Note: number bits from left to right.

Bit position number

$$\left.\begin{array}{l} 001 = 1_{10} \\ 011 = 3_{10} \\ 101 = 5_{10} \\ 111 = 7_{10} \end{array}\right\} p_1$$

$$\left.\begin{array}{l} 010 = 2_{10} \\ 011 = 3_{10} \\ 110 = 6_{10} \\ 111 = 7_{10} \end{array}\right\} p_2$$

$$\left.\begin{array}{l} 100 = 4_{10} \\ 101 = 5_{10} \\ 110 = 6_{10} \\ 111 = 7_{10} \end{array}\right\} p_3$$

Hamming Code Example

1	2	3	4	5	6	7
p_1	p_2	d_1	p_3	d_2	d_3	d_4

- Note: parity bits occupy power-of-two bit positions in code-word.
- On writing to memory:
  - parity bits are assigned to force even parity over their respective groups.
- On reading from memory:
  - check bits (c_3,c_2,c_1) are generated by finding the parity of the group and its parity bit. If an error occurred in a group, the corresponding check bit will be 1, if no error the check bit will be 0.
  - check bits (c_3,c_2,c_1) form the position of the bit in error.

- Example: $c = c_3c_2c_1 = 101$
  - error in 4,5,6, or 7 (by c_3=1)
  - error in 1,3,5, or 7 (by c_1=1)
  - no error in 2, 3, 6, or 7 (by c_2=0)
- Therefore error must be in bit 5.
- *Note the check bits point to 5*

- By our clever positioning and assignment of parity bits, the check bits always address the position of the error!

- c=000 indicates no error

Hamming Error Correcting Code

- Overhead involved in single error correction code:
  - let *p* be the total number of parity bits and *d* the number of data bits in a *p* + *d* bit word.
  - If p error correction bits are to point to the error bit (*p* + *d* cases) plus indicate that no error exists (1 case), we need:

    $$2^p >= p + d + 1,$$

    thus $p >= \log(p + d + 1)$

    for large *d*, *p* approaches log(*d*)

- Adding on extra parity bit covering the entire word can provide double error detection

1	2	3	4	5	6	7	8
p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4

- On reading the C bits are computed (as usual) plus the parity over the entire word, P:

  C=0 P=0, no error

  C!=0 P=1, correctable single error

  C!=0 P=0, a double error occurred

  C=0 P=1, an error occurred in p_4 bit

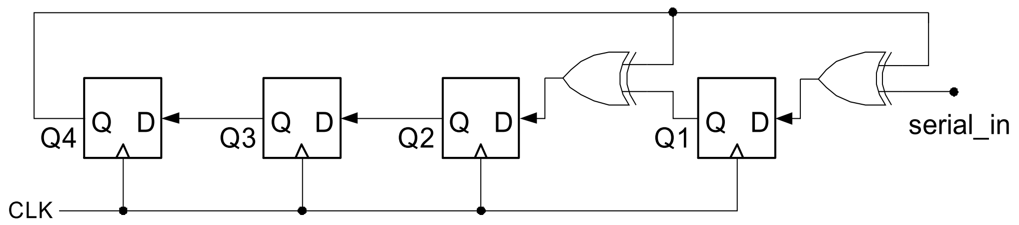
Typical modern codes in DRAM memory systems:
64-bit data blocks (8 bytes) with 72-bit code words (9 bytes), results in SEC, DED.

Error Correction with LFSRs

```
                      11 message bits    4 check bits
bit sequence:  1 1 0 0 1 0 0 0 1 1 1 0 0 0 0

        0 0 0 0 1
    xor 0 0 0 0 0
        0 0 0 0 1 1
      xor 0 0 0 0 0
          0 0 0 1 1 0
        xor 0 0 0 0 0
            0 0 1 1 0 0
          xor 0 0 0 0 0
              0 1 1 0 0 1
            xor 1 0 0 1 1
                0 1 0 1 0 0
              xor 1 0 0 1 1
                  0 0 1 1 1

                            . . . . . . . . . . .
                                      1 0 1 0
```



Error Correction with LFSRs

- XOR Q4 with incoming bit sequence. Now values of shift-register don’t follow a fixed pattern. Dependent on input sequence.
- Look at the value of the register after 15 cycles: “1010”
- Note the length of the input sequence is 2^4-1 = 15 (same as the number of different nonzero patters for the original LFSR)
- Binary message occupies only 11 bits, the remaining 4 bits are “0000”.
  - They would be replaced by the final result of our LFSR: “1010”
  - If we run the sequence back through the LFSR with the replaced bits, we would get “0000” for the final result.
  - 4 parity bits “neutralize” the sequence with respect to the LFSR.

    1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 $\Rightarrow$ 1 0 1 0

    1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 $\Rightarrow$ 0 0 0 0
- If parity bits not all zero, an error occurred in transmission.
- If number of parity bits = log total number of bits, then single bit errors can be corrected.
- Using more parity bits allows more errors to be detected.
- Ethernet uses 32 parity bits per frame (packet) with 16-bit LFSR.