EECS 151/251A

Spring 2021

Digital Design and Integrated Circuits

Instructor:
John Wawrzynek

Lecture 19: Parallelism

Announcements

- Virtual Front Row for today 4/1:
  - *Bernard Chen*
  - *Matthew Tran*
  - *Jennifer Zhou*
  - *Suphakorn Lertruchtkul*
  - *Rahul Arya*
- **Please ask question or make comments!**
- Homework assignment 7 (power & memory) posted due Monday.

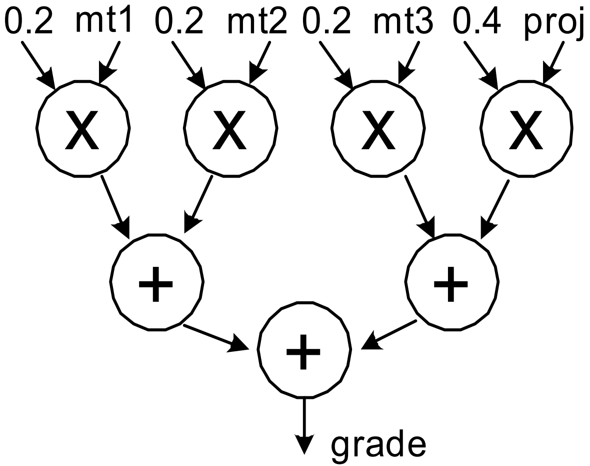
Parallelism

*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. Parallelism can often also be used to improve energy efficiency.*

- 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;
```

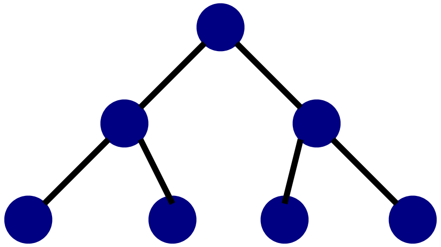
- High performance hardware implementation:



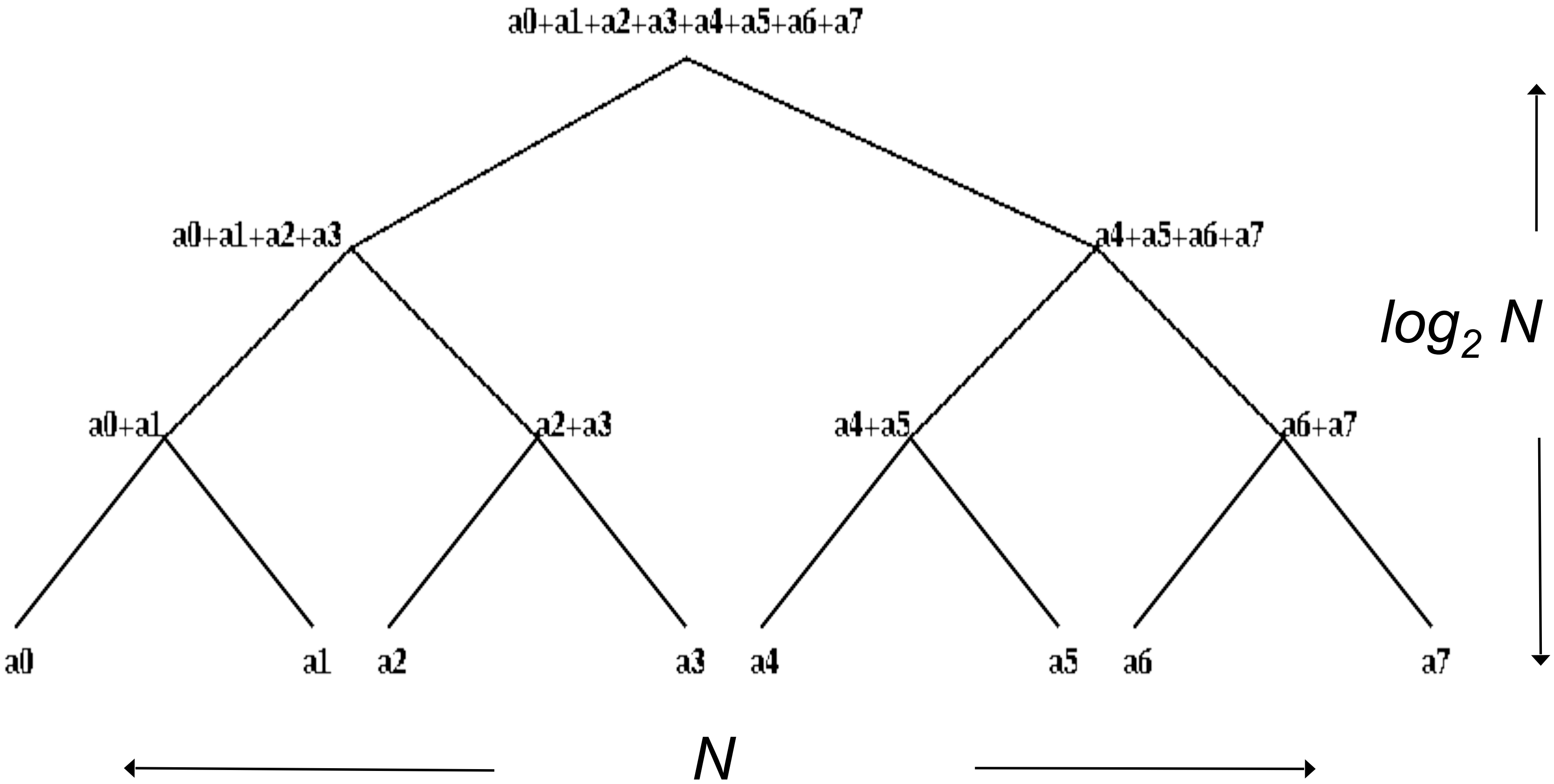
As many operations as possible are done in parallel.

A log(n) lower (time) bound to compute any function of n variables

- Assume we can only use binary operations, each taking unit time
- After 1 time unit, an output can only depend on two inputs
- Use induction to show that after k time units, an output can only depend on 2^k inputs
  - After $\log_2$ n time units, output depends on at most n inputs
- A binary tree performs such a computation

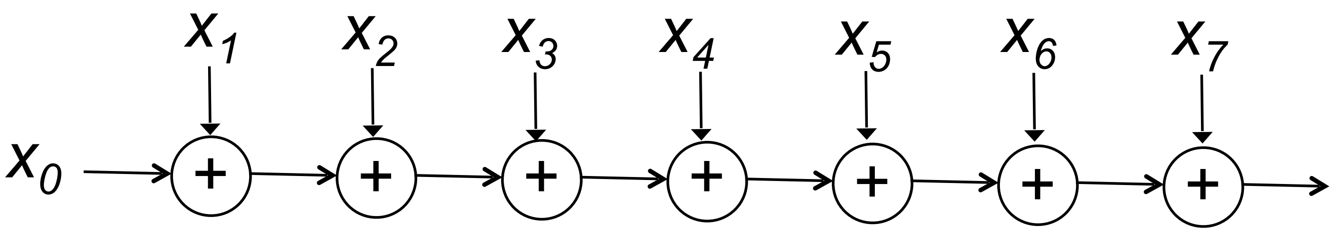


Example: Reductions with Trees



If each node (operator) is k-ary instead of binary, what is the delay?

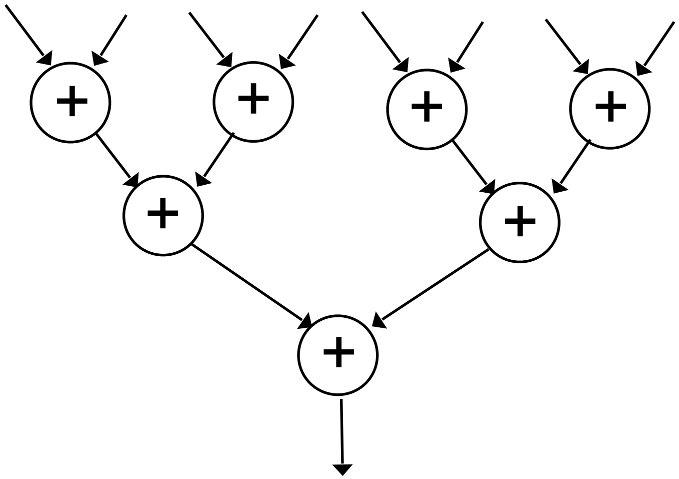
Trees for optimization



$T = O(N)$

$((((((x_0 + x_1) + x_2) + x_3) + x_4) + x_5) + x_6) + x_7$

Same number of operations (N-1)



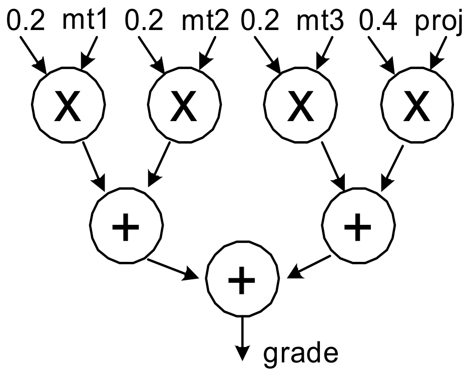
$T = O(\log N)$

$((x_0 + x_1) + (x_2 + x_3)) + ((x_4 + x_5) + (x_6 + x_7))$

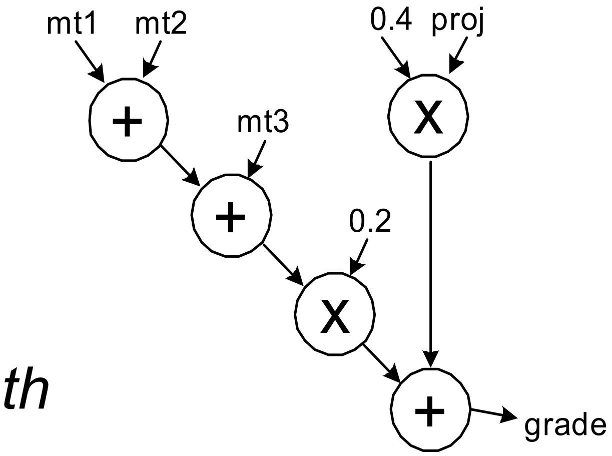
- ❑ *What property of "+" are we exploiting?*
- ❑ *Other associate operators? Boolean operations? Division? Min/Max?*

Parallelism

- Is there a lower cost hardware implementation? Different tree organization?
- `grade = ((0.2 × mt1)+(0.2 × mt2)) +((0.2 × mt3)+(0.4 × proj));`



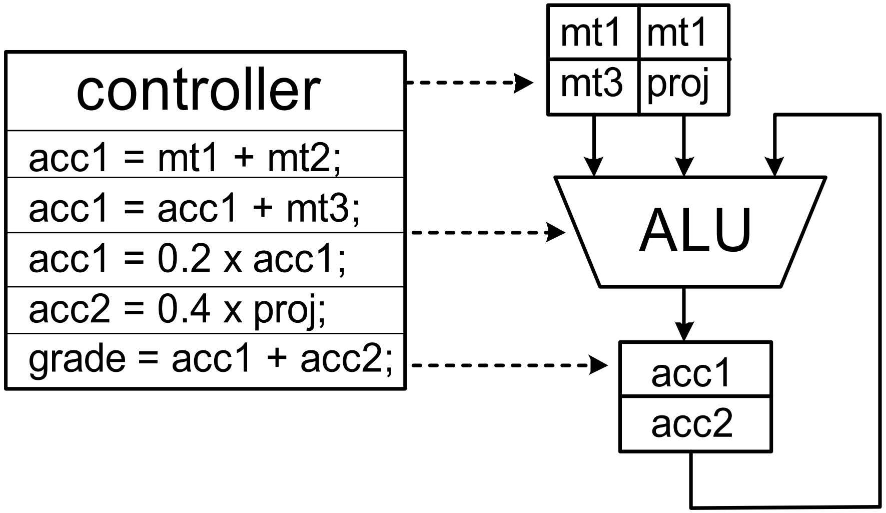
- Can factor out multiply by 0.2 (use factoring and associativity):



- `grade =(0.2 × ((mt1 + mt2) + mt3))) + (0.4 × proj);`
- *Compare the cost and critical path in both implementations.*
- How about sharing operators (multipliers and adders)?

Time-Multiplexing

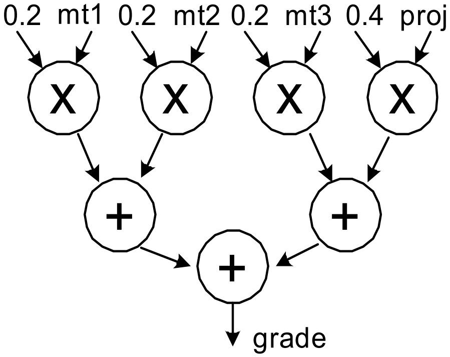
- *Time multiplex* single ALU for all adds and multiplies:
- Attempts to minimize cost at the expense of time.
  - Need to add extra register, muxes, control.



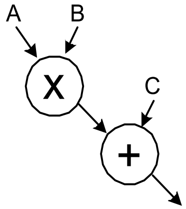
controller

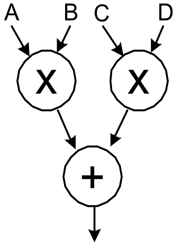
```
acc1 = mt1 + mt2;
acc1 = acc1 + mt3;
acc1 = 0.2 x acc1;
acc2 = 0.4 x proj;
grade = acc1 + acc2;
```

- If we adopt above approach, we can then consider the combinational hardware circuit diagram as an *abstract computation-graph*.



Using other primitives, other coverings are possible.





- This time-multiplexing "covers" the computation graph by performing the action of each node one at a time. (Sort of *emulates* it.)

HW versus SW

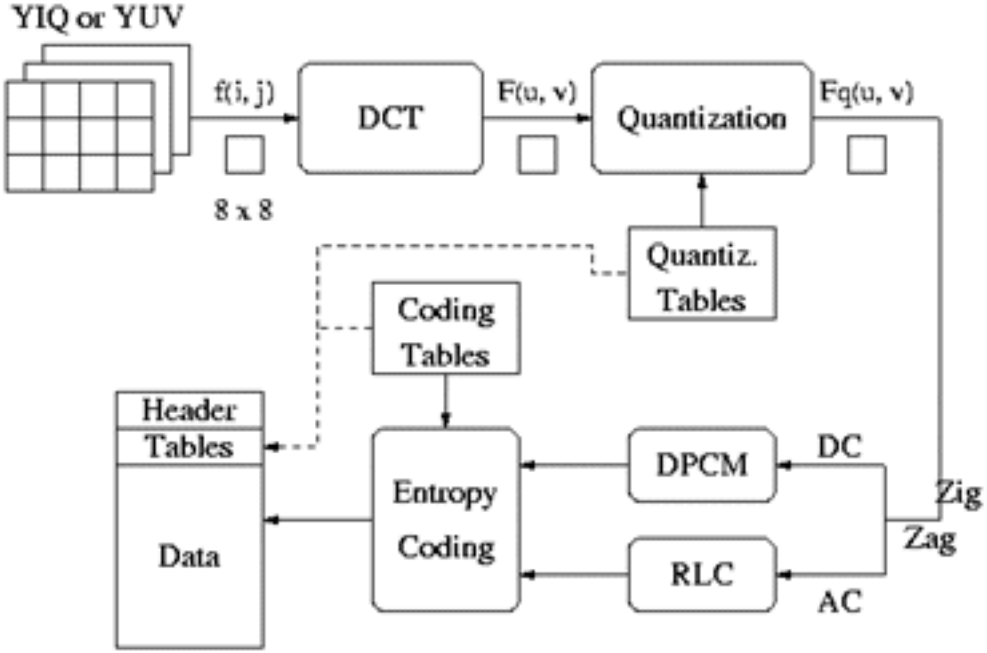
- This **time-multiplexed ALU** approach is very similar to what a conventional software version would accomplish:
- CPUs time-multiplex function units (ALUs, etc.)

```
add r2,r1,r3
add r2,r2,r4
mult r2,r4,r5
       .
       .
       .
```

- This model matches our tendency to express computation sequentially - even though most computations naturally contain parallelism.
- Our programming languages also strengthen a sequential tendency.
- In hardware we have the ability to exploit problem parallelism - gives us a “knob” to tradeoff performance & cost.
- Maybe best to express computations as abstract computations graphs (rather than “programs”) - should lead to wider range of implementations.
- *Note: modern high-performance processors spend much of their cost budget attempting to* restore *execution parallelism: “super-scalar execution”.*

Exploiting Parallelism in HW

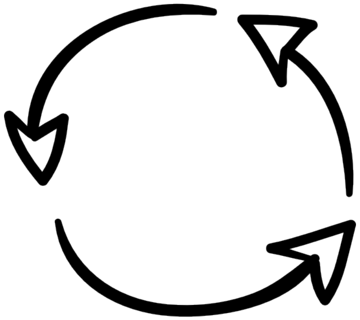
- Example: Video Codec



- Separate algorithm blocks implemented in separate HW blocks, or HW is time-multiplexed.
- Entire operation is pipelined (with possible pipelining within the blocks).
- “Loop unrolling used within blocks” or for entire computation.

Optimizing Iterative Computations

- Hardware implementations of computations almost always involves looping. Why?
- Is this true with software?
- Are there programs without loops?
  - Maybe in “through away” code.



- We probably would not bother building such a thing into hardware, would we?
  - (FPGA could change this.)
- Fact is, our computations are closely tied to loops. Almost all our HW includes some looping mechanism.
- What do we use looping for?

Optimizing Iterative Computations

Types of loops:

1) Looping over input data (streaming):

   – ex: MP3 player, video compressor

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

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

Pipelining

wash	load1	load2	load3	load4		
dry		load1	load2	load3	load4	
fold			load1	load2	load3	load4

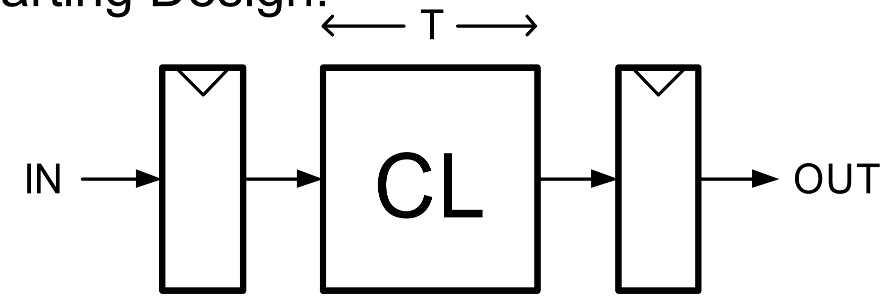
- In the limit, as we increase the number of loads, the average time per load approaches 20 minutes (1 load completed every 20 minutes)

- The latency (time from start to end) for one load = 60 min.
- The throughput = 3 loads/hour

- The pipelined throughput ≈ # of pipe stages x un-pipelined throughput.

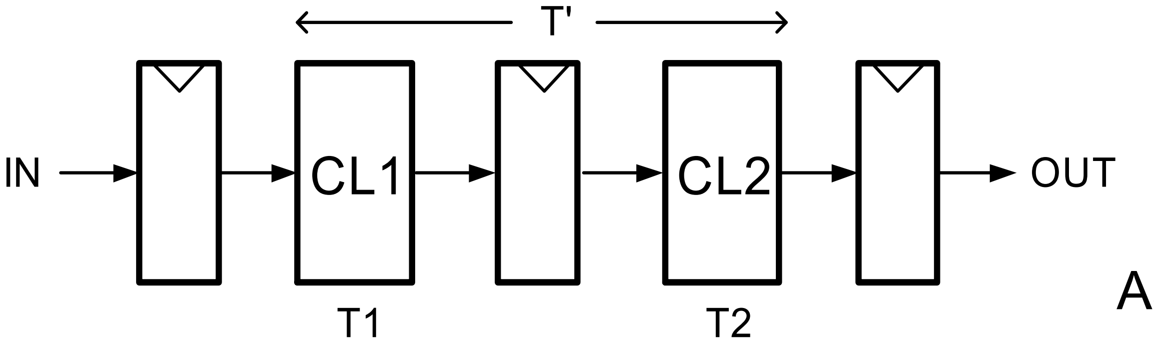
Hardware Pipelining Example

- Starting Design:



Assume T=8ns
T_{FF}(setup +clk→q)=1ns
F = 1/9ns = 111MHz

- Cut the CL block into pieces (stages) and separate with registers:



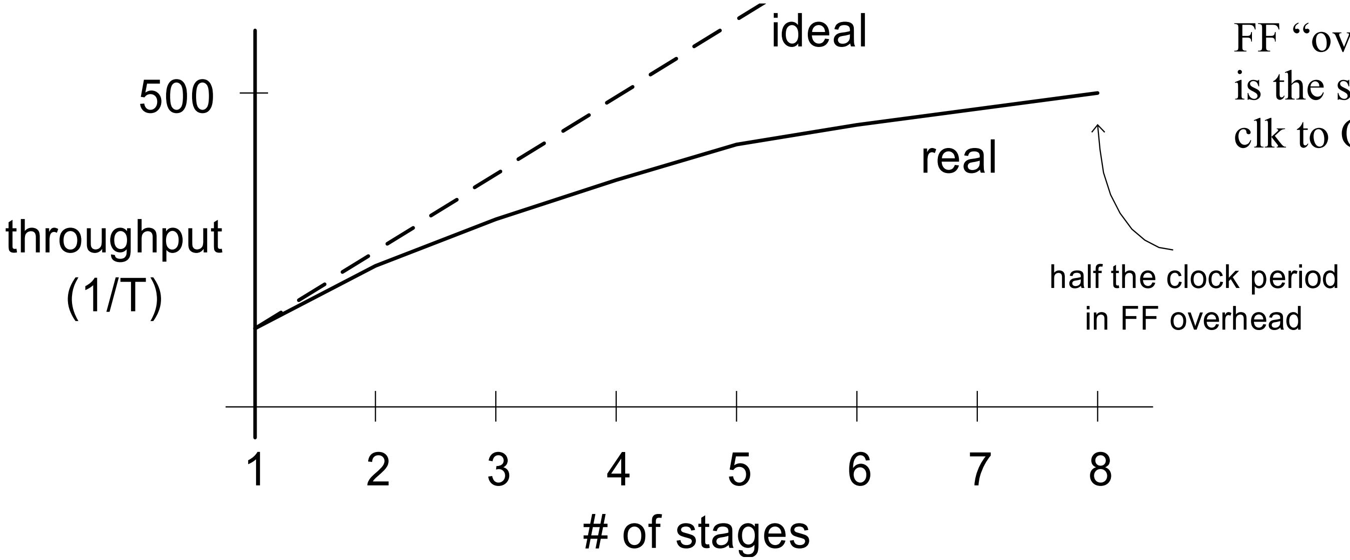
Assume T1 = T2 = 4ns

T’ = 4ns + 1ns + 4ns +1ns = 10ns
F = 1/(4ns +1ns) = 200MHz

- CL block produces a new result every 5ns instead of every 9ns.

Limits on Pipelining

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

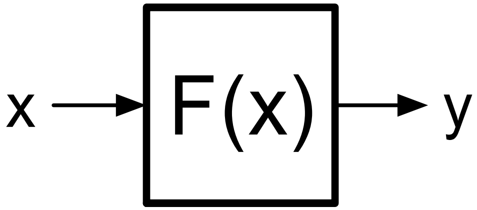


FF “overhead” is the setup and clk to Q times.

- 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) - in CPUs, we these *data hazards*

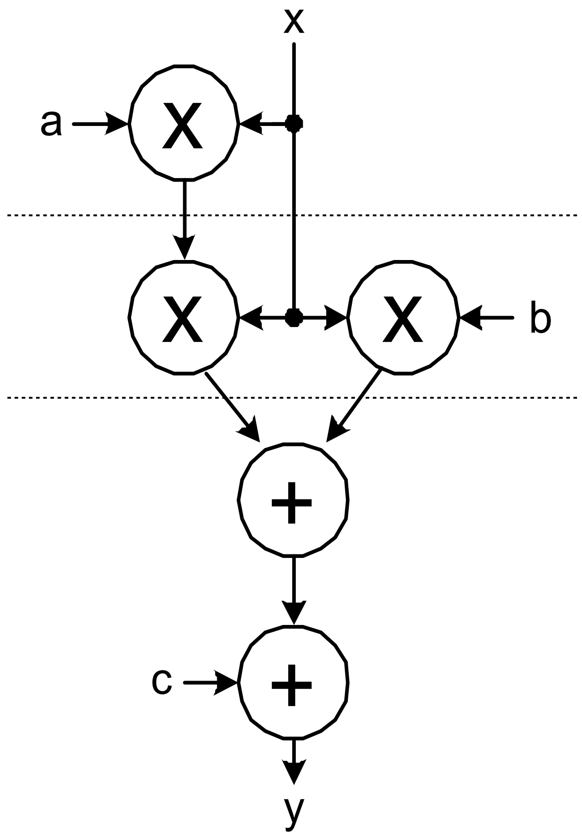
Pipelining Example

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



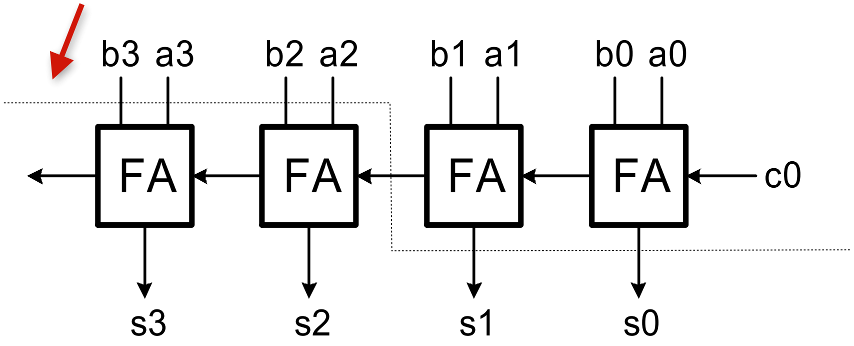
- x and y are assumed to be "streams" of integers (or floats)
- Divide into 3 (nearly) equal stages.
- Insert pipeline registers at dashed lines.
- Can we pipeline basic operators?

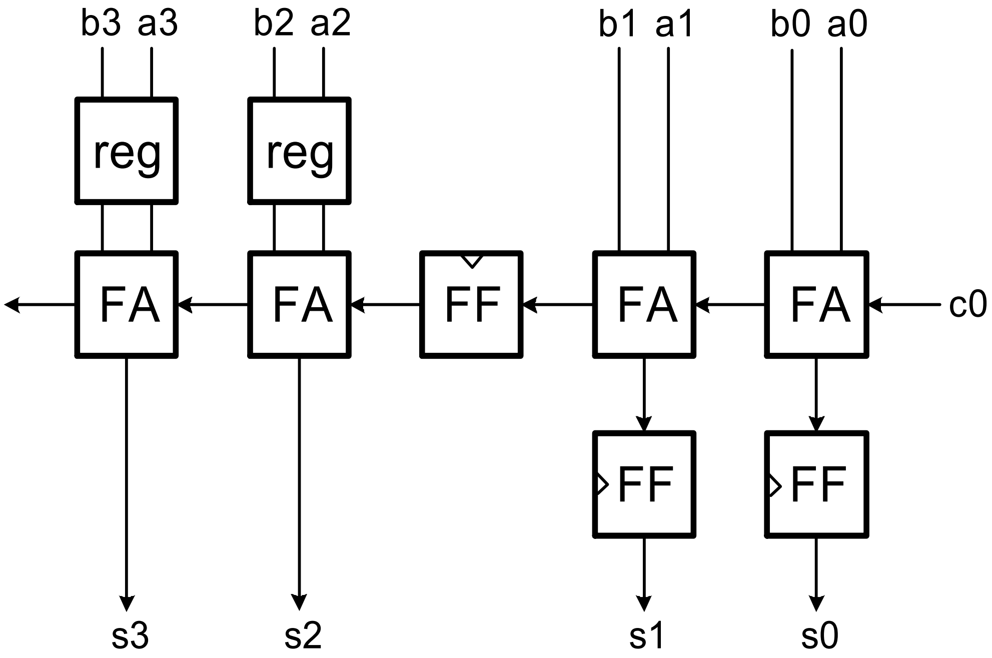
- Computation graph:



Example: Pipelined Ripple Adder

Insert pipeline register





- Critical path cut in half
- Latency now two cycles
- Cost and energy increases by adding registers

- Possible, but usually not done.

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

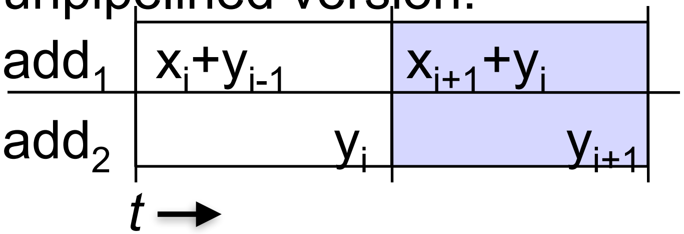
More common to pipeline multiplication.

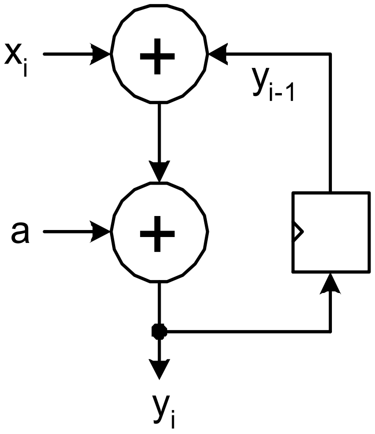
Pipelining Loops with Feedback

"Loop carry dependency"

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

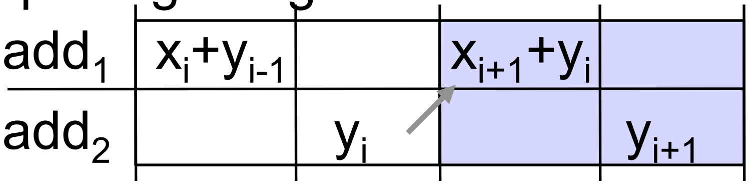
unpipelined version:

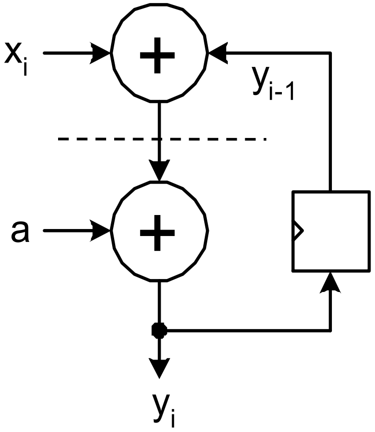




Can we "cut" the feedback and overlap iterations?

Try putting a register after add1:



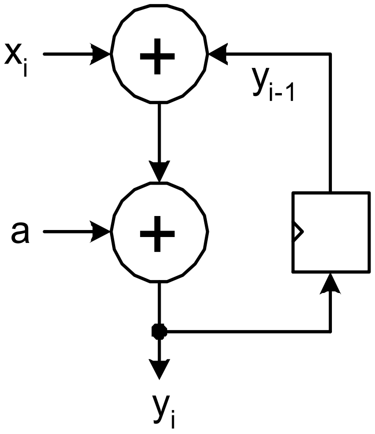


- Can't overlap the iterations because of the dependency.
- The extra register doesn't help the situation (actually hurts).
- In general, can't effectively pipeline feedback loops.

Pipelining Loops with Feedback

"Loop carry dependency"

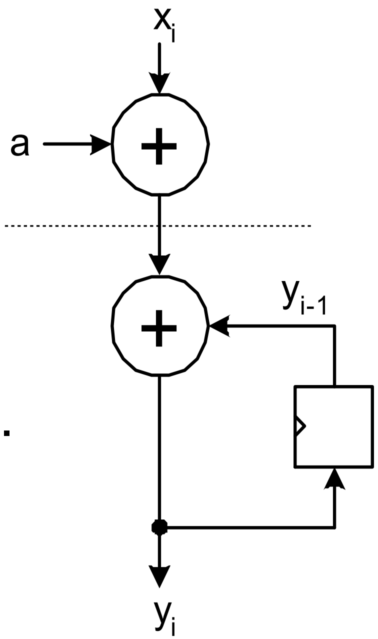
However, we can overlap the "non-feedback" part of the iterations:



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}



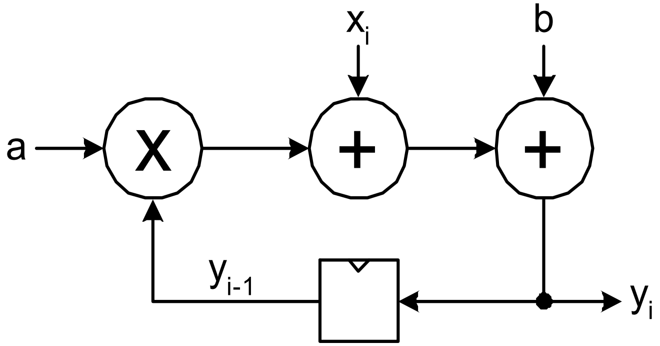
"Shorten" the feedback path.

- Pipelining is limited to 2 stages.

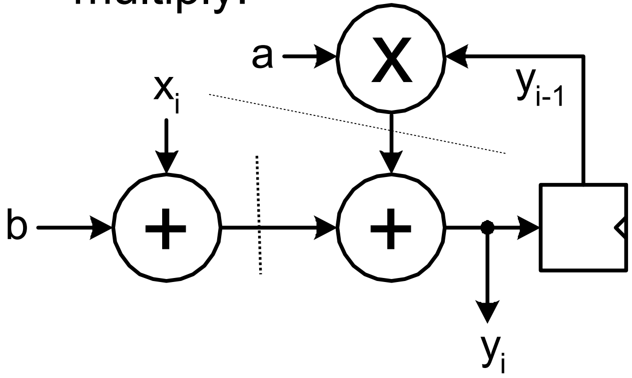
Pipelining Loops with Feedback

- Example 2:

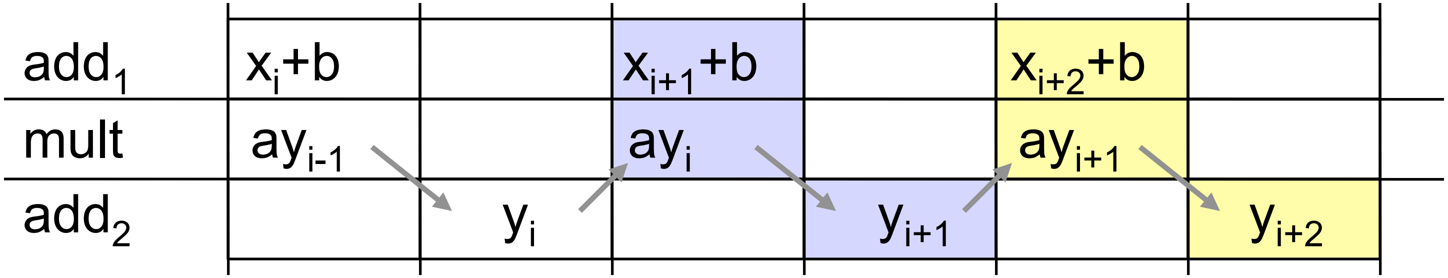
$$y_i = a\, y_{i-1} + x_i + b$$



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



- Just said we can't - but let's anyway.



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}

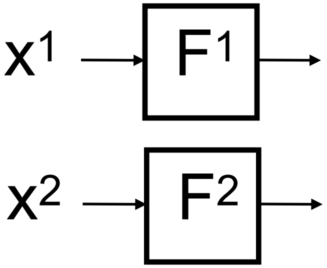
- Still need 2 cycles/iteration

“C-slow” Technique

- An 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:



$y^1 = a^1 y^1_{i-1} + x^1_i + b^1$

$y^2 = a^2 y^2_{i-1} + x^2_i + b^2$

Use muxes to direct each stream.

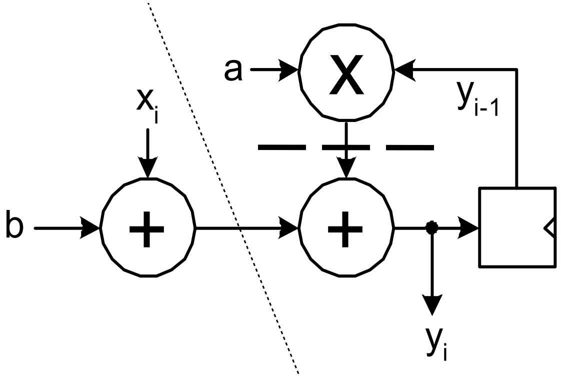
Time multiplex one piece of HW for both stream.

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:



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

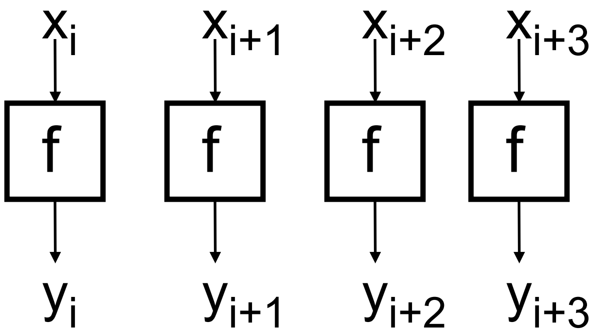
- C computations (in this case C=2) can use the pipeline simultaneously.
- Must be independent.
- Input MUX interleaves input streams.
- Each stream runs at half the pipeline frequency.
- Pipeline achieves full throughput.

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: $\mathbf{y_i = f(x_i)}$



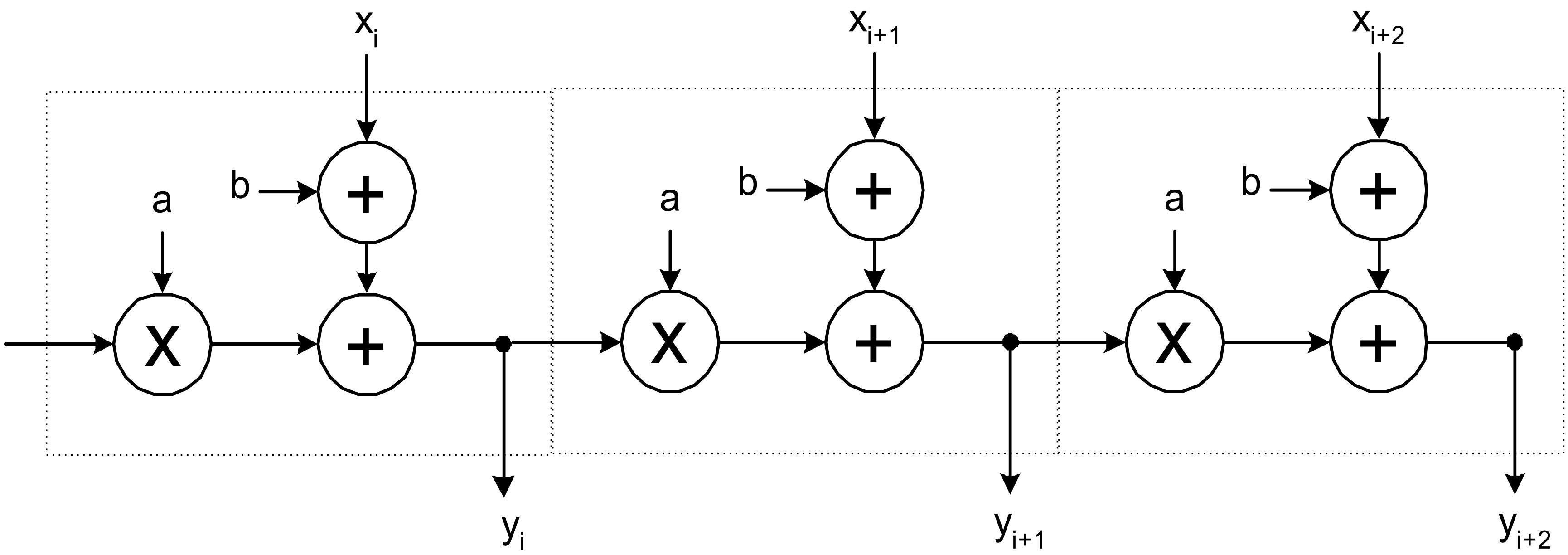
Usually called SIMD parallelism. Single Instruction Multiple Data

- 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.
- Vector processors use this technique.
- Limited, once again, by loop carry dependencies. Feedback translates to dependencies between parallel data-paths.

SIMD Parallelism with Feedback

- Example, from earlier:

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



- As with pipelining, this technique is most effective in the absence of a loop carry dependence.
- With loop carry dependence, end up with “carry ripple” situation.
- For associative operations we can employ look-ahead / parallel-prefix optimization techniques to speed up propagation (coming soon!)