EECS150 - Digital Design

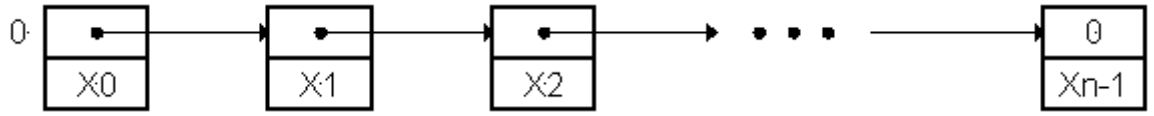
Lecture 23 - High-Level Design (Part 2)

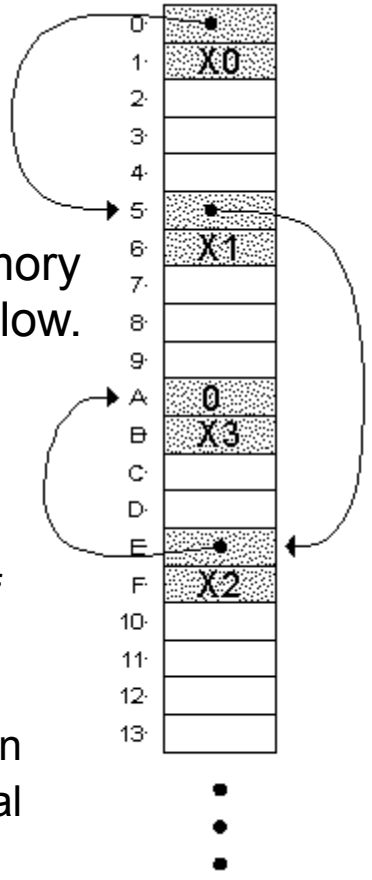
April 10, 2012

John Wawrzynek

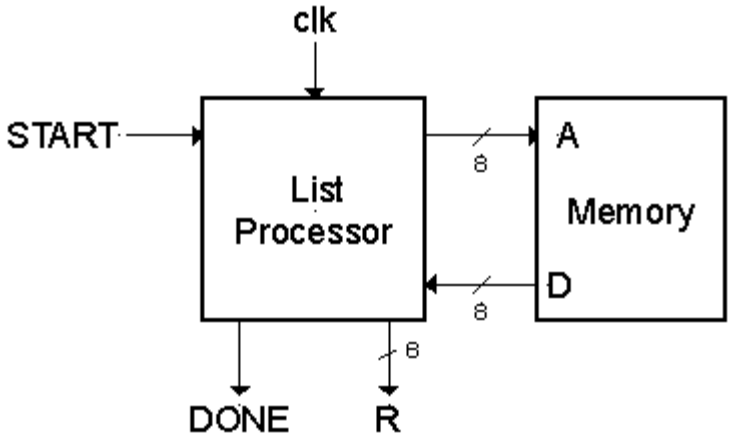
List Processor Example

- Design a circuit that forms the sum of all the 2's complements integers stored in a linked-list structure starting at memory address 0:





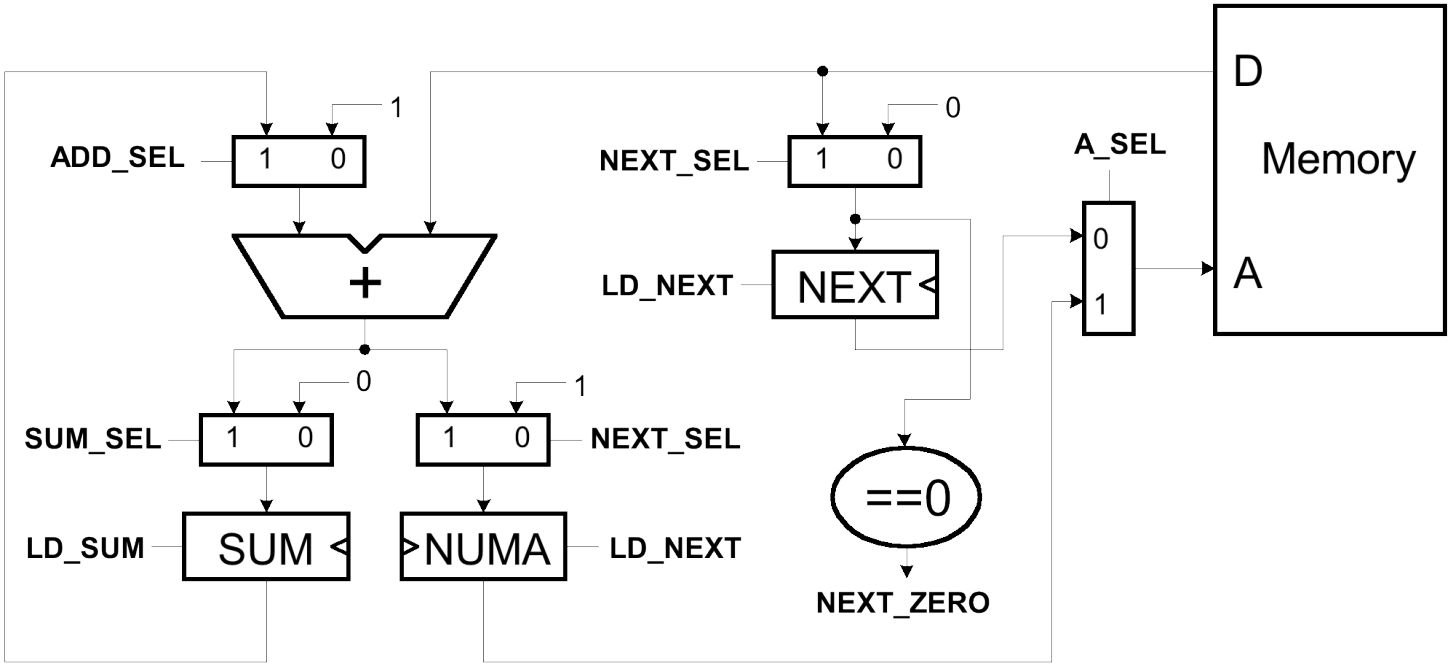
- All integers and pointers are 8-bit. The link-list is stored in a memory block with an 8-bit address port and 8-bit data port, as shown below. The pointer from the last element in the list is 0.



I/Os:

- START resets to head of list and starts addition process.
- DONE signals completion
- R, Bus that holds the final result

5. Optimization, Architecture #3




```
If (START==1) NEXT←0, SUM←0, NUMA←1;
   repeat {
      SUM←SUM + Memory[NUMA];
      NUMA←Memory[NEXT] + 1, NEXT←Memory[NEXT] ;
      } until (NEXT==0);
R←SUM, DONE←1;
```

- Performance:
  - T > 23ns, F < 43Mhz

Resource Utilization Charts

- One way to visualize these (and other possible) optimizations is through the use of a *resource utilization charts.*
- These are used in high-level design to help schedule operations on shared resources.
- Resources are listed on the y-axis. Time (in cycles) on the x-axis.
- Example:

memory	fetch A_1	fetch A_2	fetch A_3				
bus		fetch A_1	fetch A_2	fetch A_3			
register-file		read B_1	read B_2	read B_3			
ALU			A_1+B_1	A_2+B_2	A_3+B_3		
cycle	1	2	3	4	5	6	7

- Our list processor has two shared resources: memory and adder

List Example Resource Scheduling

- Unoptimized solution: 1. SUM←SUM + Memory[NEXT+1]; 2. NEXT←Memory[NEXT];

memory	fetch x	fetch next	fetch x	fetch next
adder1	next+1		next+1	
adder2	sum		sum	
	1	2	1	2

- Optimized solution: 1. SUM←SUM + Memory[NUMA];
  2. NEXT←Memory[NEXT], NUMA←Memory[NEXT]+1;

memory	fetch x	fetch next	fetch x	fetch next
adder	sum	numa	sum	numa

- How about the other combination: add x register

memory	fetch x	fetch next	fetch x	fetch next
adder	numa	sum	numa	sum

1. X←Memory[NUMA], NUMA←NEXT+1;
2. NEXT←Memory[NEXT], SUM←SUM+X;

- Does this work? If so, a very short clock period. Each cycle could have *independent* fetch and add. T = max(T_{mem}, T_{add}) instead of T_{mem}+ T_{add}.

List Example Resource Scheduling

- Schedule one loop iteration followed by the next:

Memory	$next_1$		x_1		$next_2$		x_2		
adder		$numa_1$		sum_1		$numa_2$		sum_2	

- How can we overlap iterations? $next_2$ depends on $next_1$.
  - "slide" second iteration into first (4 cycles per result):

Memory	$next_1$		x_1	$next_2$		x_2		
adder		$numa_1$		sum_1	$numa_2$		sum_2	

  - or further:

Memory	$next_1$	$next_2$	x_1	x_2	$next_3$	$next_4$	x_3	x_4	
adder		$numa_1$	$numa_2$	sum_1	sum_2	$numa_3$	$numa_4$	sum_3	sum_4

The repeating pattern is 4 cycles. Not exactly the pattern what we were looking for. But does it work correctly?

List Example Resource Scheduling

- In this case, first spread out, then pack.

Memory	$next_1$			x_1		
adder		$numa_1$			sum_1	

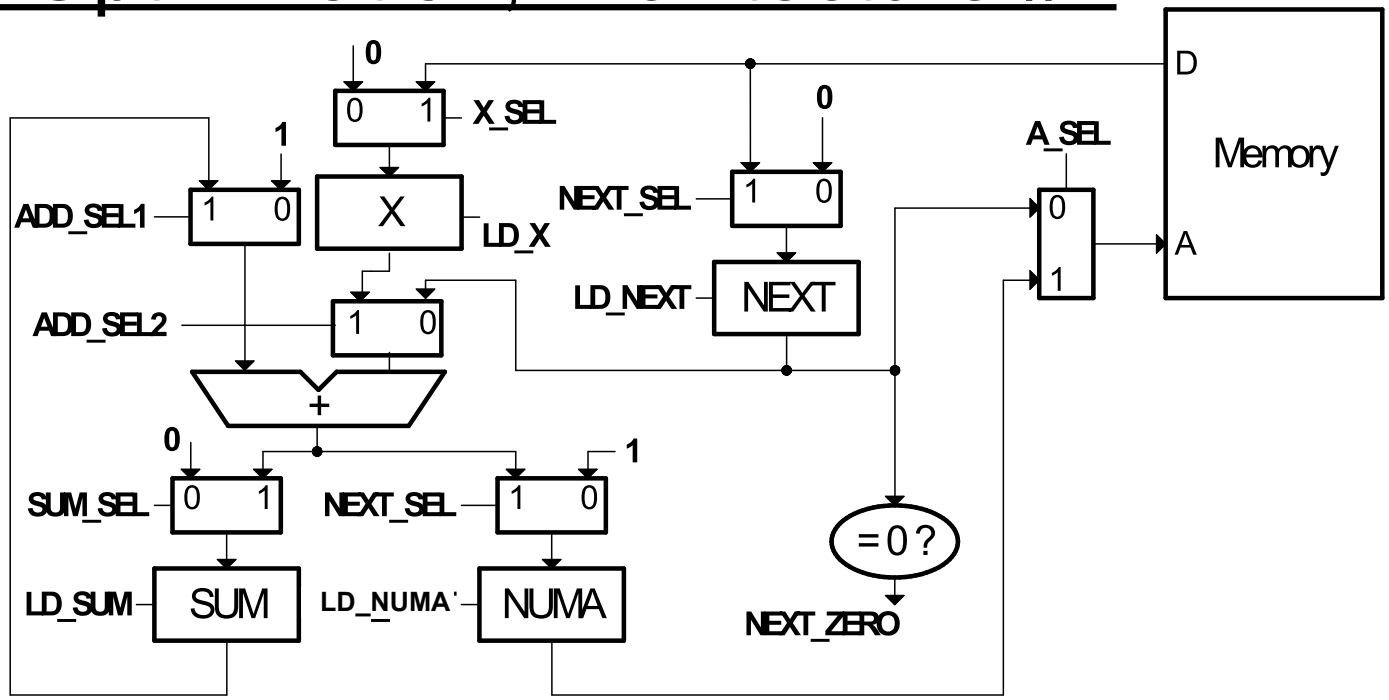
Memory	$next_1$		$next_2$	x_1	$next_3$	x_2	$next_4$	x_3	
adder		$numa_1$		$numa_2$	sum_1	$numa_3$	sum_2	$numa_4$	sum_3

1. X←Memory[NUMA], NUMA←NEXT+1;
2. NEXT←Memory[NEXT], SUM←SUM+X;

- Three different loop iterations active at once.
- Short cycle time (no dependencies within a cycle)
- full utilization (only 2 cycles per result)
- Initialization: x=0, numa=1, sum=0, next=memory[0]
- Extra control states (out of the loop)
  - one to initialize next, clear sum, set numa
  - one to finish off. 2 cycles after next==0.

5. Optimization, Architecture #4

- Datapath:



- Incremental cost:
  - Addition of another register & mux, adder mux, and control.
- Performance: find max time of the four actions

1. X←Memory[NUMA], NUMA←NEXT+1; 0.5+1+10+1+0.5 = 13ns
   same for all ⇒ T>13ns, F<77MHz
2. NEXT←Memory[NEXT], SUM←SUM+X;

Other Optimizations

- Node alignment restriction:
  - If the application of the list processor allows us to restrict the placement of nodes in memory so that they are aligned on even multiples of 2 bytes.
    - NUMA addition can be eliminated.
    - Controller supplies "0" for low-bit of memory address for NEXT, and "1" for X.
  - Furthermore, if we could use a memory with a 16-bit wide output, then could fetch entire node in one cycle:

    {NEXT, X} ← Memory[NEXT], SUM ← SUM + X;

    ⇒ execution time cut in half (half as many cycles)

List Processor Conclusions

- Through careful optimization:
  - clock frequency increased from 32MHz to 77MHz
  - little cost increase.
- "Scheduling" was used to overlap and to maximize use of resources.
- Questions:
  - Consider the design process we went through:
  - Could a computer program go from RTL description to circuits automatically?
  - Could a computer program derive the optimizations that we did?
  - It is the goal of "High-Level Synthesis" to do similar transformations and automatic mappings. "C-to-gates" compilers are an example.

Modulo Scheduling

- Review of list processor scheduling:

Memory	$next_1$			x_1		
adder		$numa_1$			sum_1	

- How did we know to "spread" out the schedule of one iteration to allow efficient packing?

Memory	$next_1$		$next_2$	x_1	$next_3$	x_2	$next_4$	x_3	
adder		$numa_1$		$numa_2$	sum_1	$numa_3$	sum_2	$numa_4$	sum_3

- The goal of *modulo scheduling* is to find the schedule for one *characteristic section* of the computation. This is the part the control loops over.
- The entire schedule can then be derived, by repeating the characteristic section or repeating it with some pieces omitted.

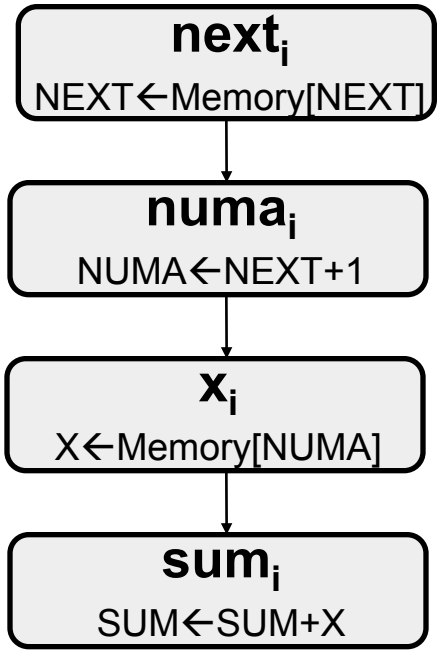
Modulo Scheduling Procedure

1. Calculate *minimal length of characteristic section*.

   The maximum number of cycles that any one resource is used during one iteration of the computation (assuming a resource can only be used once per cycle).

2. Schedule one iteration of the computation on the characteristic section wrapping around when necessary. Each time the computation wraps around, decrease the iteration subscript by one.
3. If iteration will not fit on minimal length section, increase section by one and try again.

Modulo Scheduling List Processor



- Assuming a single adder and a single ported memory. Minimal schedule section length = 2.

  Because both memory and adder are used for 2 cycles during one iteration.

memory	$next_i$	
adder		$numa_i$

memory	$next_i$	X_{i-1}
adder		$numa_i$

wrap-around, decrease subscript

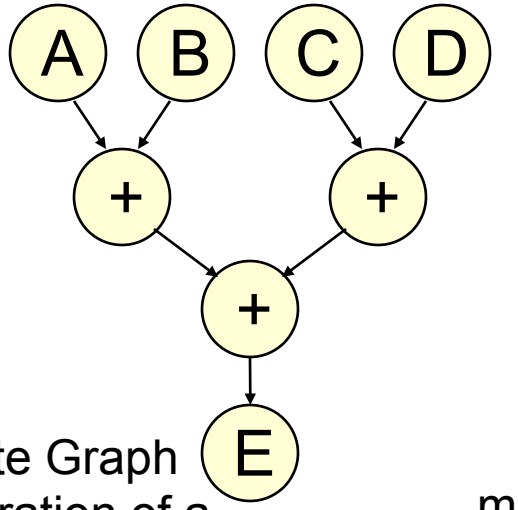
memory	$next_i$	X_{i-1}
adder	sum_{i-2}	$numa_i$

wrap-around, decrease subscript

- Finished schedule for 4 iterations:

Memory	$next_1$		$next_2$	x_1	$next_3$	x_2	$next_4$	x_3	
adder		$numa_1$		$numa_2$	sum_1	$numa_3$	sum_2	$numa_4$	sum_3

Another Scheduling Example



Compute Graph (one iteration of a repeating calculation)

- Assume A, B, C, D, E stored in a dual port memory.
- Assume a single adder.
- Minimal schedule section length = 3. (Both memory and adder are used for 3 cycles during one iteration.)

memory port 1	load A	load C	
memory port 2	load B	load D	store E
adder	E =	A + B	C + D

Repeating schedule:

load A	load C		load A	load C		load A	load C	
load B	load D	store E	load B	load D	store E	load B	load D	store E
E =	A + B	C + D	E =	A + B	C + D	E =	A + B	C + D

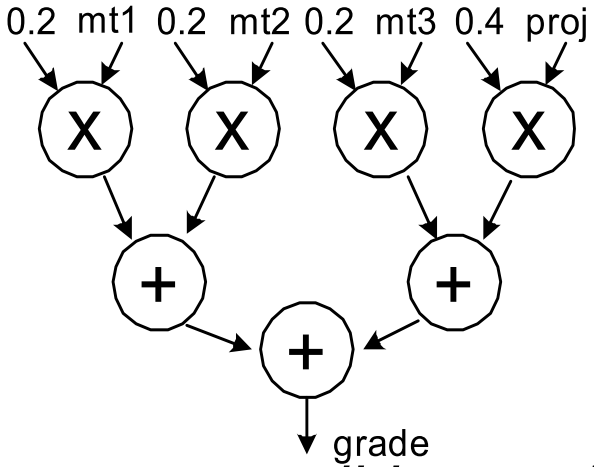
Parallelism

*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;
```

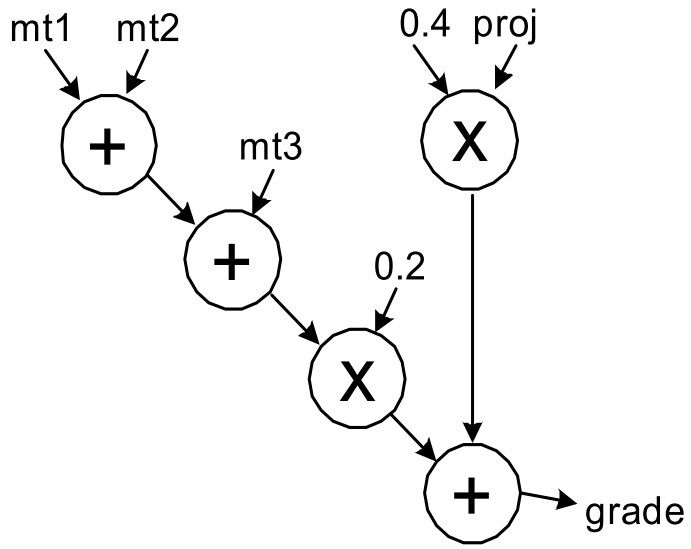
- High performance hardware implementation:



As many operations as possible are done in parallel.

Parallelism

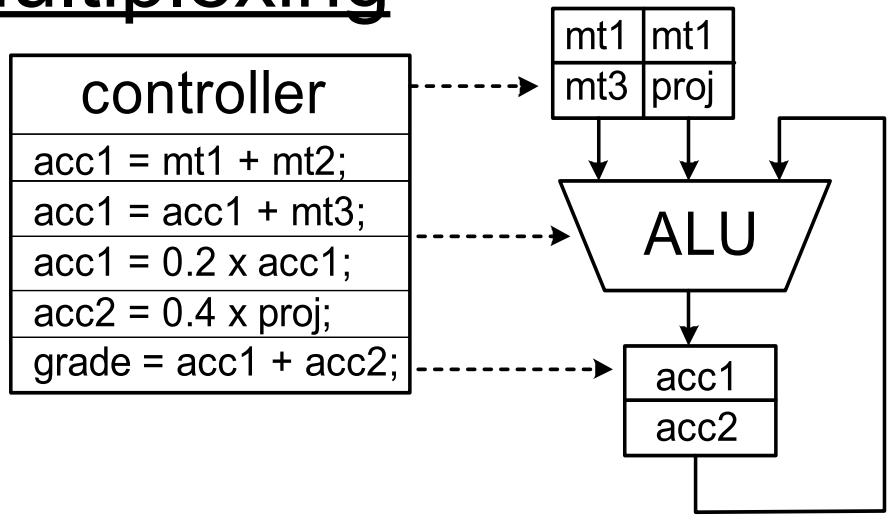
- Is there a lower cost hardware implementation? Different tree organization?
- Can factor out multiply by 0.2:



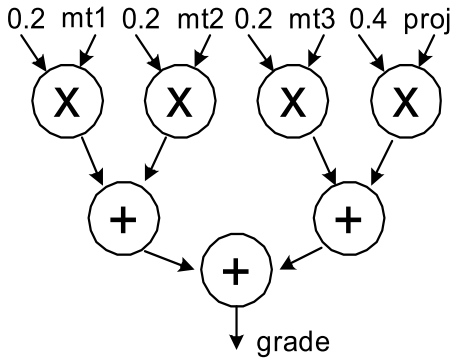
- How about sharing operators (multipliers and adders)?

Time-Multiplexing

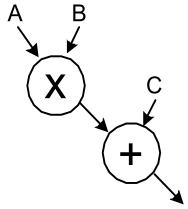
- *Time multiplex* single ALU for all adds and multiplies:
- Attempts to minimize cost at the expense of time.
  - Need to add extra register, muxes, control.

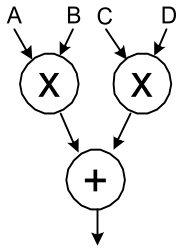


- If we adopt above approach, we can then consider the combinational hardware circuit diagram as an *abstract computation-graph*.



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

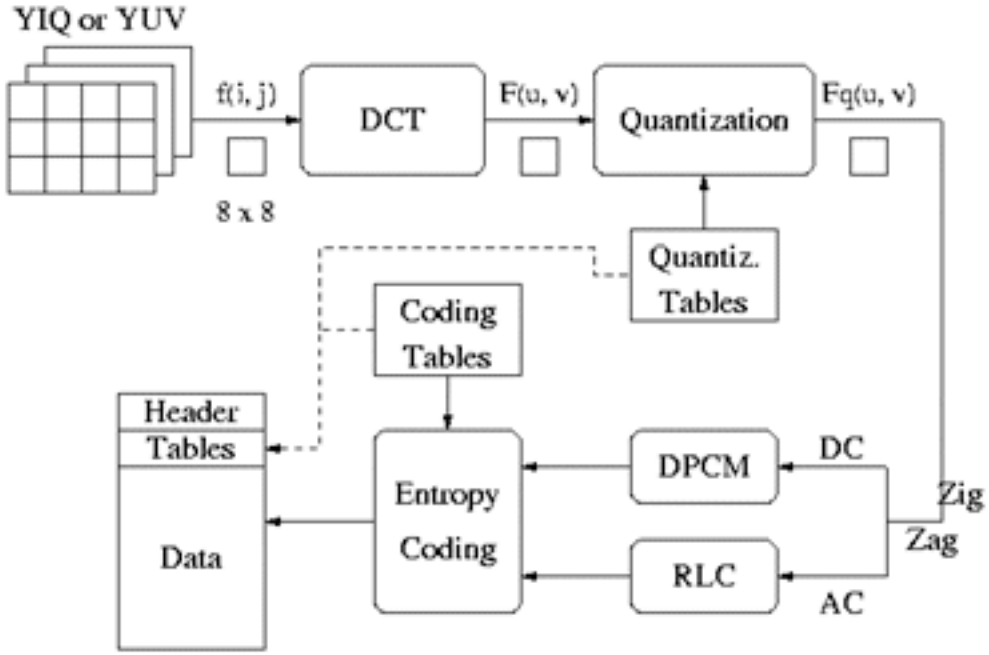
- This **time-multiplexed ALU** approach is very similar to what a conventional software version would accomplish:
- CPUs time-multiplex function units (ALUs, etc.)

```
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 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 may 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, 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?

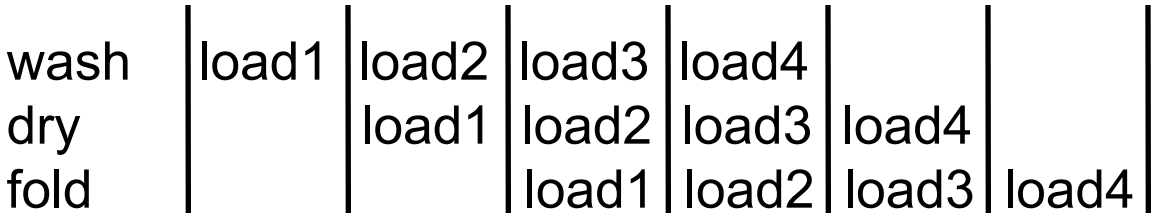
Loops offer opportunity for parallelism
by executing more than one iteration at once,
using parallel iteration execution &/or pipelining

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



overlapped ⇒ 2 hours

Pipelining

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

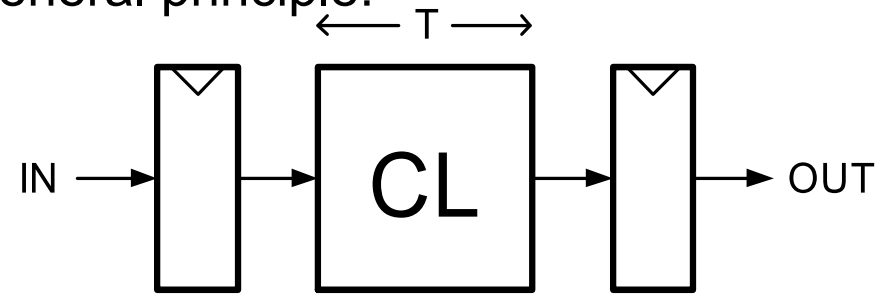
- In the limit, as we increase the number of loads, the average time per load approaches 20 minutes.

- 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.

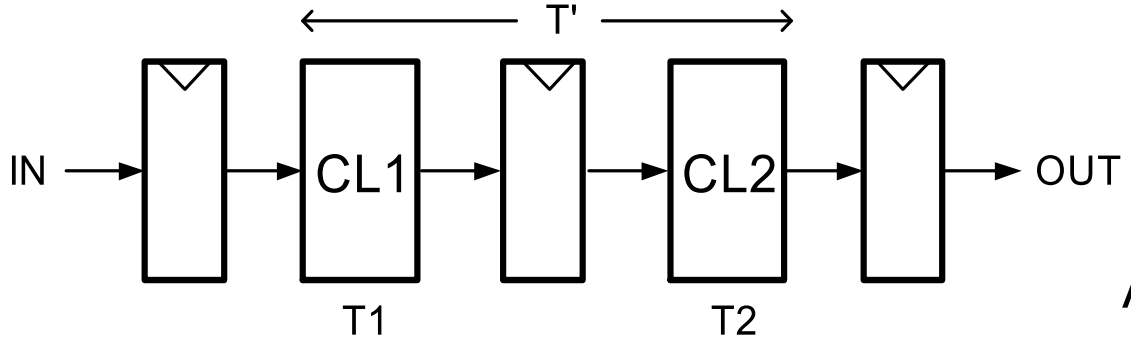
Pipelining

- General principle:



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

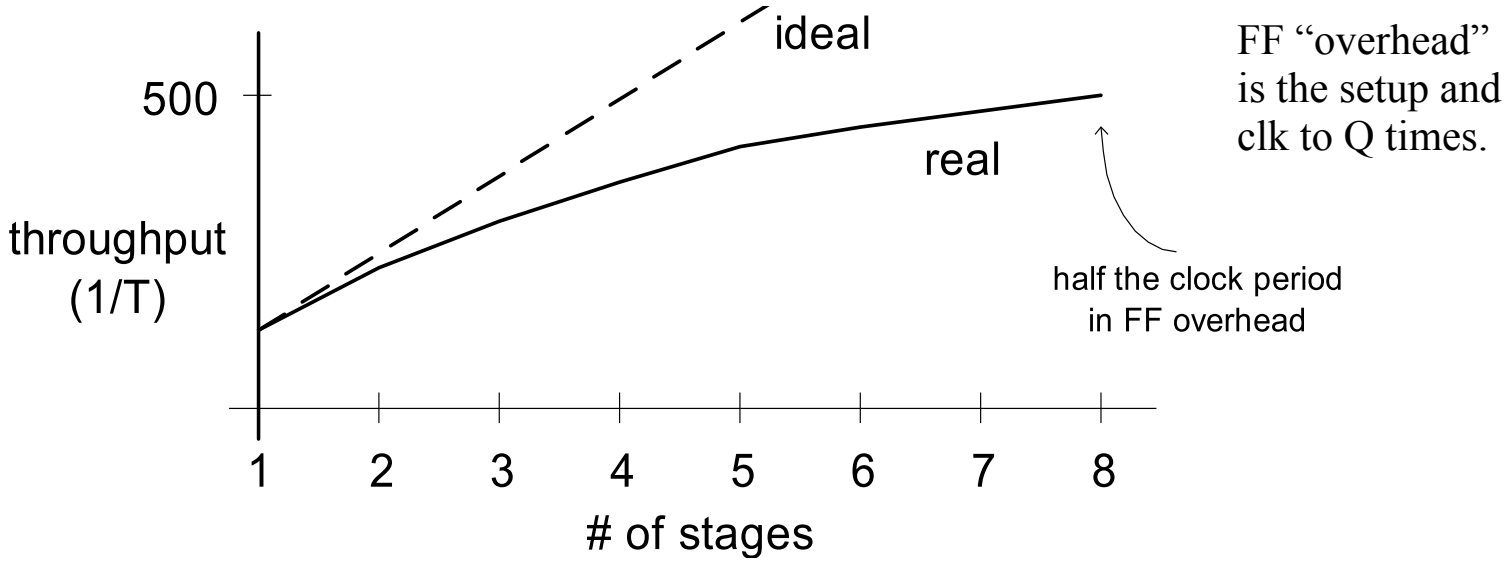
T' = 4ns + 1ns + 4ns +1ns = 10ns

F = 1/(4ns +1ns) = 200MHz

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

Limits on Pipelining

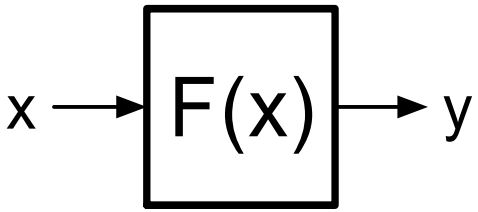
- Without FF overhead, throughput improvement α # of stages.
- After many stages are added FF overhead begins to dominate:



- 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)

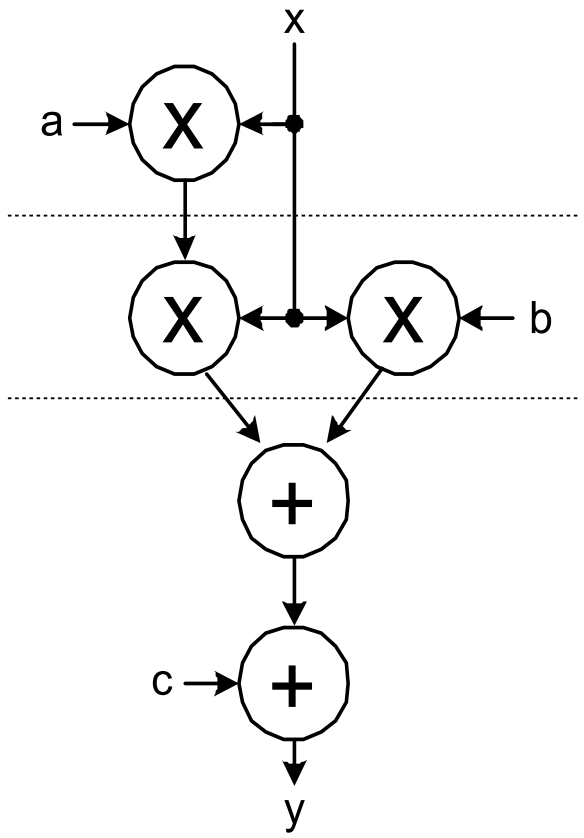
Pipelining Example

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



- 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?

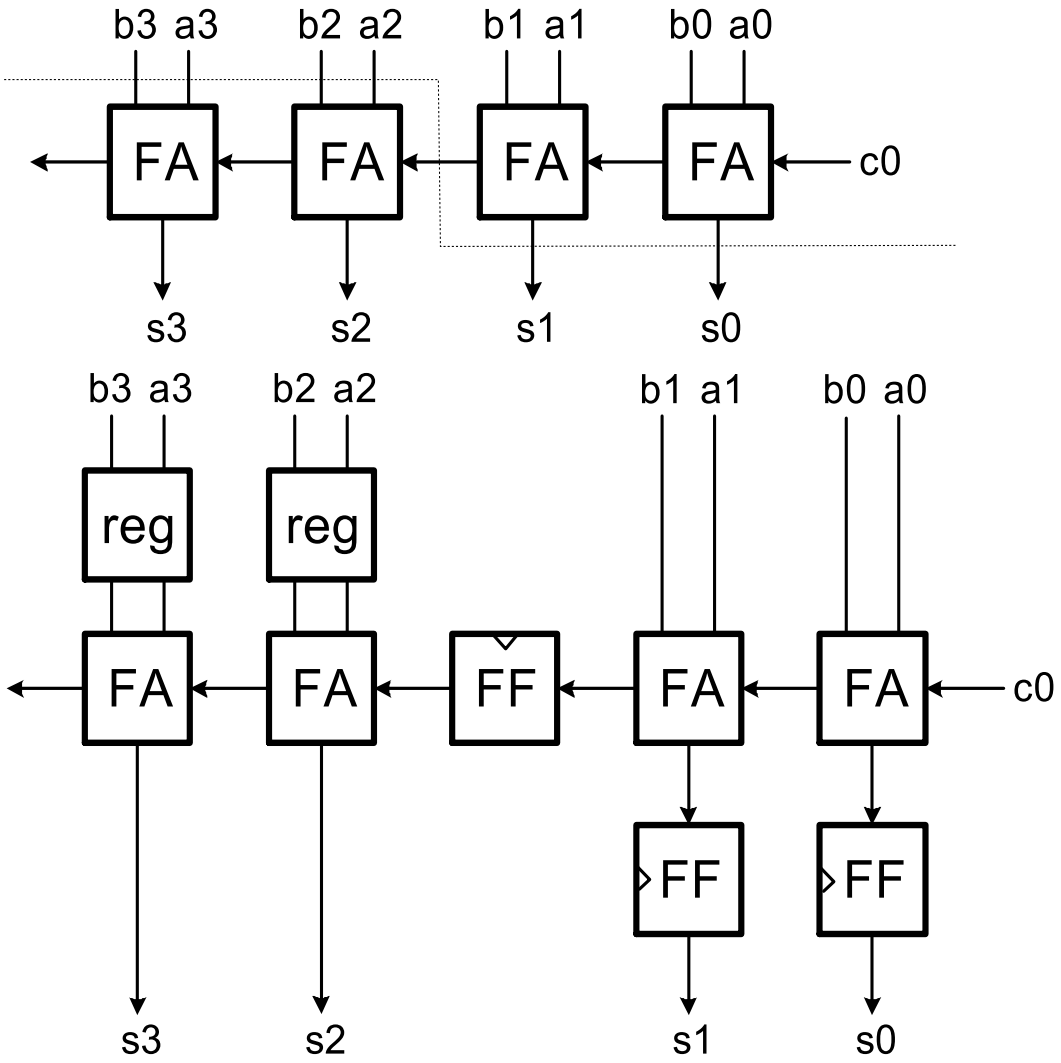
- Computation graph:



Example: Pipelined Adder

- Possible, but usually not done.

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



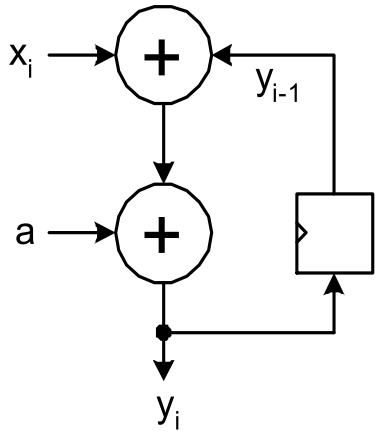
Pipelining Loops with Feedback

"Loop carry dependency"

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

unpipelined version:

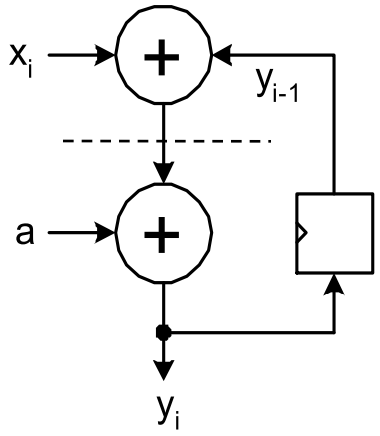
add_1	x_i+y_{i-1}	$x_{i+1}+y_i$
add_2	y_i	y_{i+1}



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}

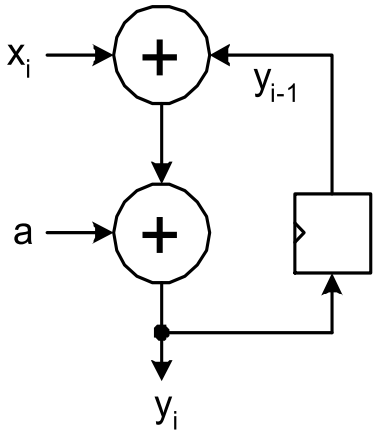


- Can't overlap the iterations because of the dependency.
- The extra register doesn't help the situation (actually hurts).
- In general, can't pipeline feedback loops.

Pipelining Loops with Feedback

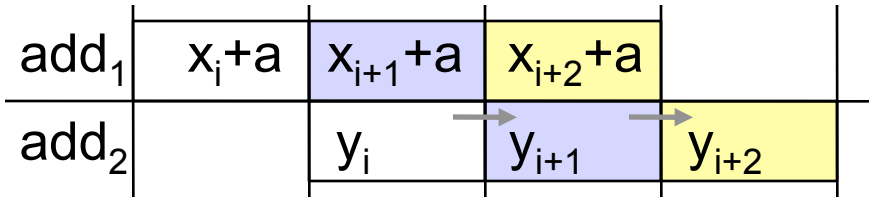
"Loop carry dependency"

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

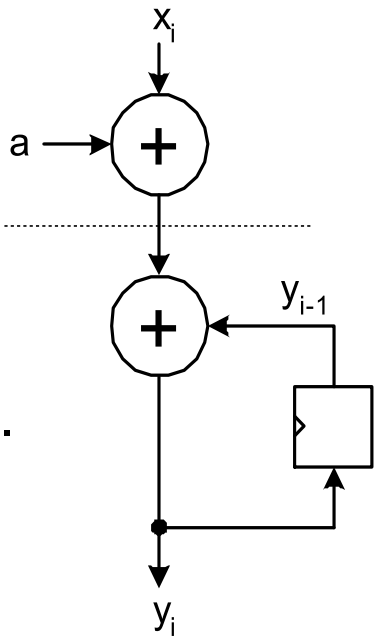


Add is associative and communitive. Therefore we can reorder the computation to shorten the delay of the feedback path:

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



"Shorten" the feedback path.

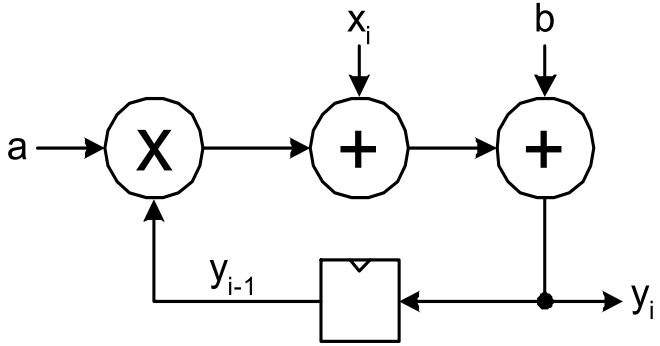


- Pipelining is limited to 2 stages.

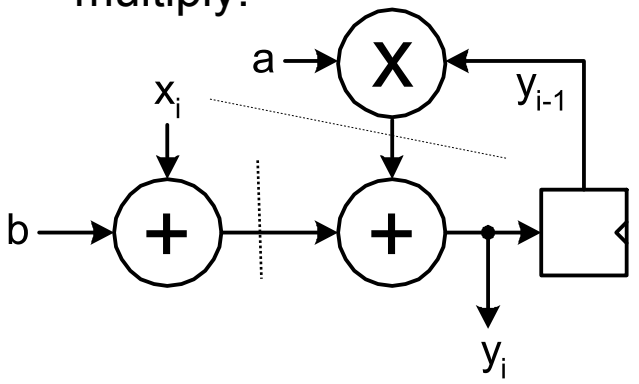
Pipelining Loops with Feedback

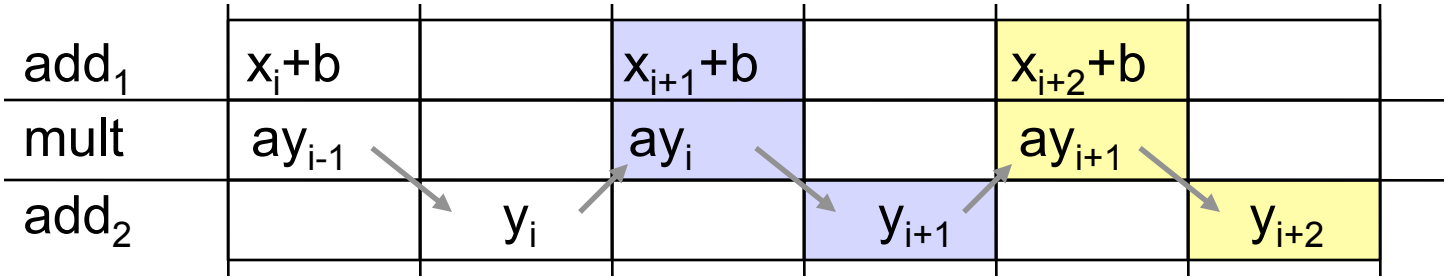
- Example 2:

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



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



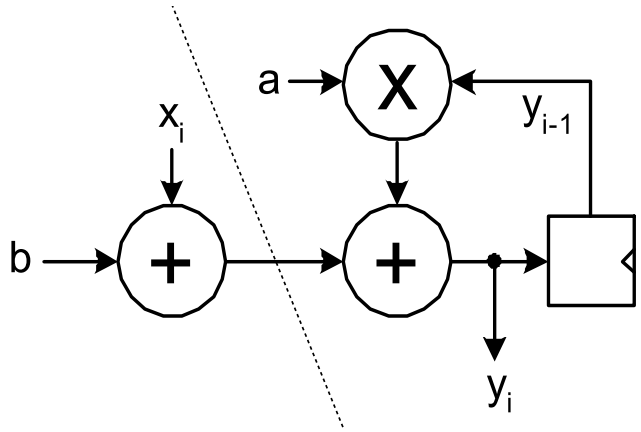


- Still need 2 cycles/iteration

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:

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

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

Use muxes to direct each stream.

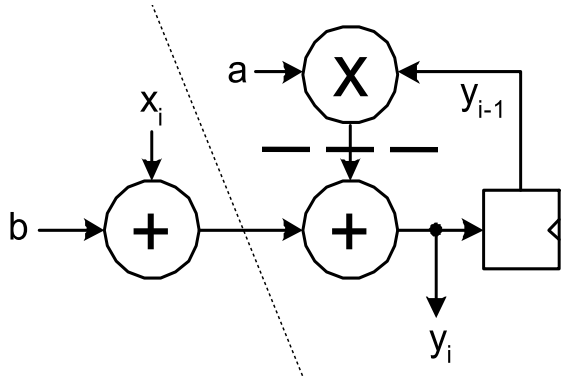
Time multiplex one piece of HW for both stream.

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:

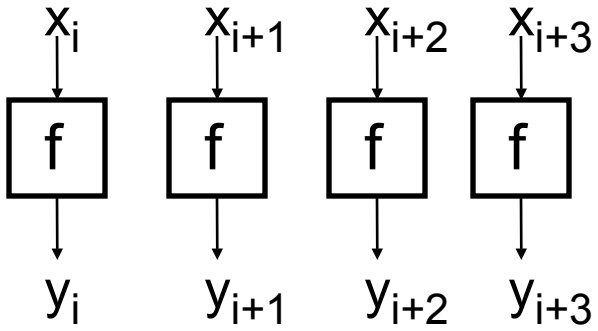
- C computations (in this case C=2) can use the pipeline simultaneously.
- Must be independent.
- Input MUX interleaves input streams.
- Each stream runs at half the pipeline frequency.
- Pipeline achieves full throughput.

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



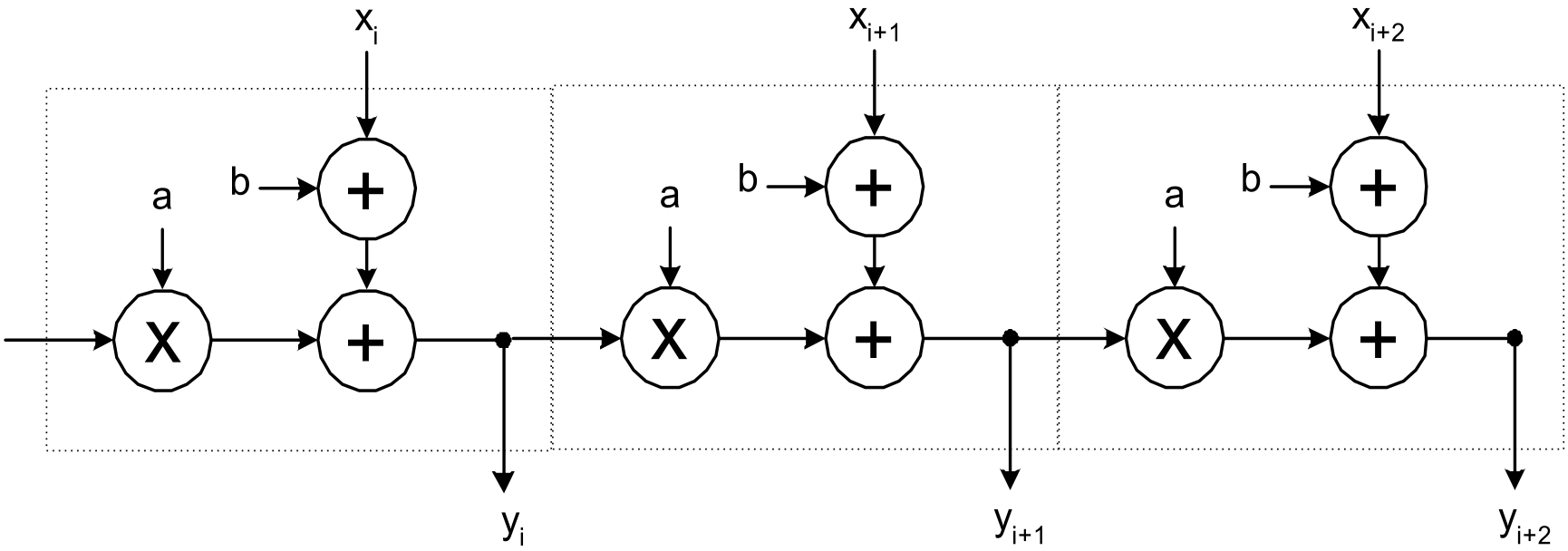
Usually called SIMD parallelism. Single Instruction Multiple Data

- 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.