Retiming

Nachiket Kapre
nachiket@uwaterloo.ca

Outline

- Need for automated pipelining
- Step-by-step look at retiming
- Invalid retiming scenarios

Retiming

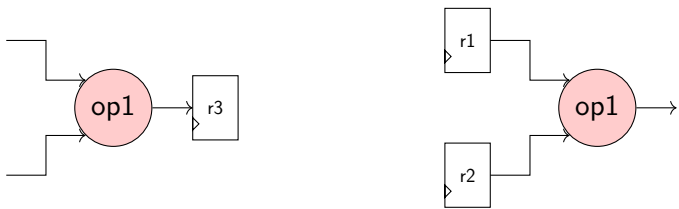
- Manual pipelining is tedious and error-prone
- Retiming is the automated way to pipeline your design to boost clock frequency
  - CAD tool will automatically move registers around for you
  - Constraint: From the outside the circuit input and outputs behave exactly as before
  - Same throughput and latency as the design at the start

Retiming Optimization

- A retiming algorithm will iteratively move registers around
- (1) Register move from input to output: take one register from each input, and add one register to each output

Retiming Optimization

- A retiming algorithm will iteratively move registers around
- (2) Register move from output to input: take one register from each output, and add one register to each input



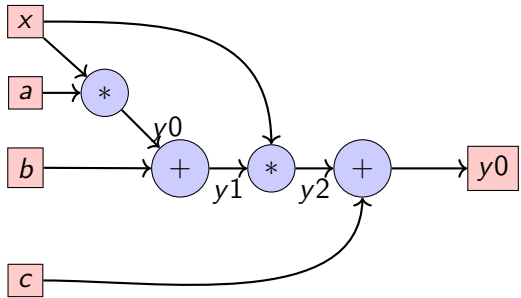
Retiming Optimization

- A retiming algorithm will iteratively move registers around
- Move register from input to/from output in a manner that preserves input $\rightarrow$ output latency.
- If an input is a constant, that input need not be registered after a move
- Your `operation` may be a register itself!
- Goal is to improve clock frequency

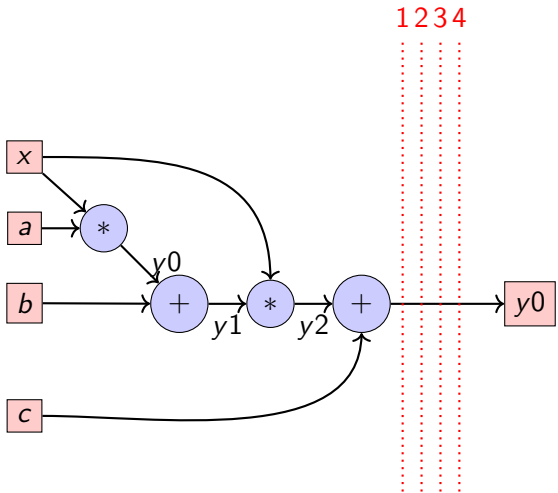
Latency tolerant circuits

- (1) If you start from an unpipelined circuit, there are no registers to move!
- (2) For some pipelined circuits, the system-level constraints may allow additional latency.
- Add a bunch of registers back-to-back to **either** input or output of the circuit.
- Make sure you add the exact same number of registers to **all** outputs (or inputs).
- End-to-end latency will increase $\rightarrow$ must know from the system-level constraints, how much latency you can tolerate
- Benefit of retiming = automatic pipelining

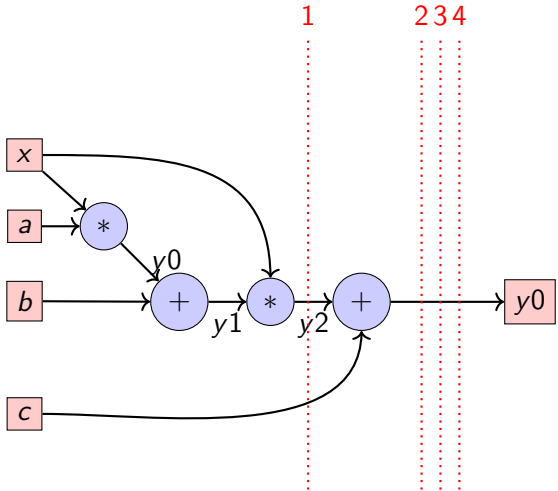
Retiming poly circuit view



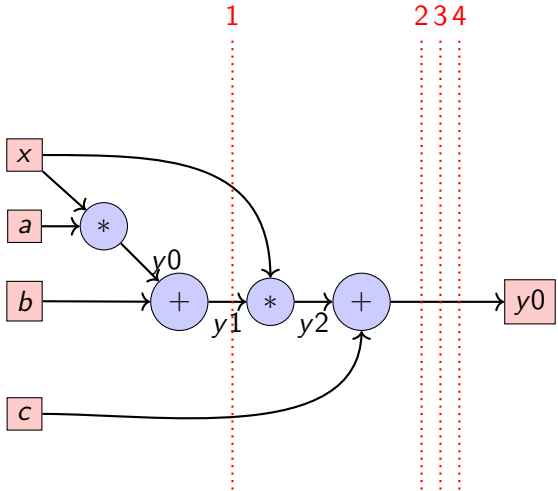
Retiming poly circuit view



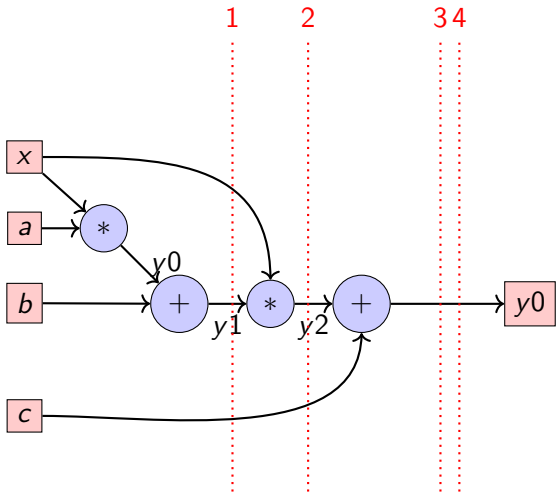
Retiming poly circuit view



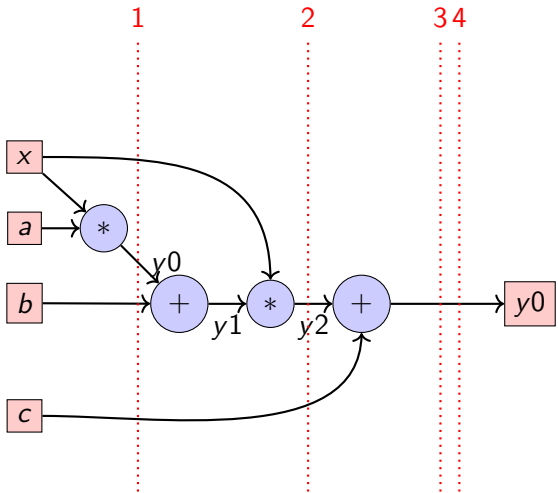
Retiming poly circuit view



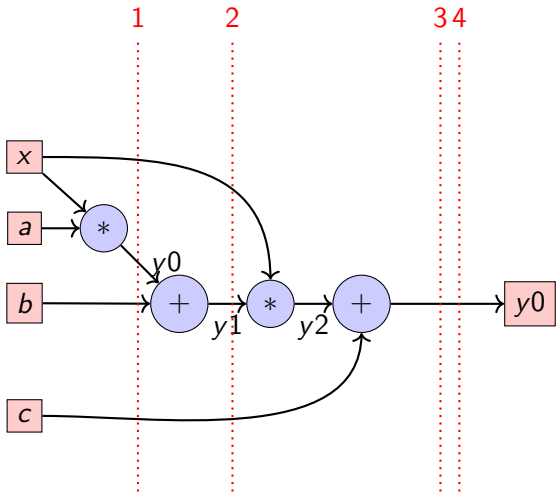
Retiming poly circuit view



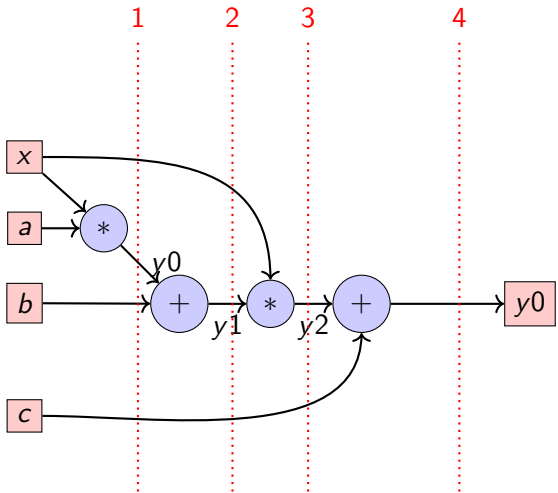
Retiming poly circuit view



Retiming poly circuit view



Retiming poly circuit view

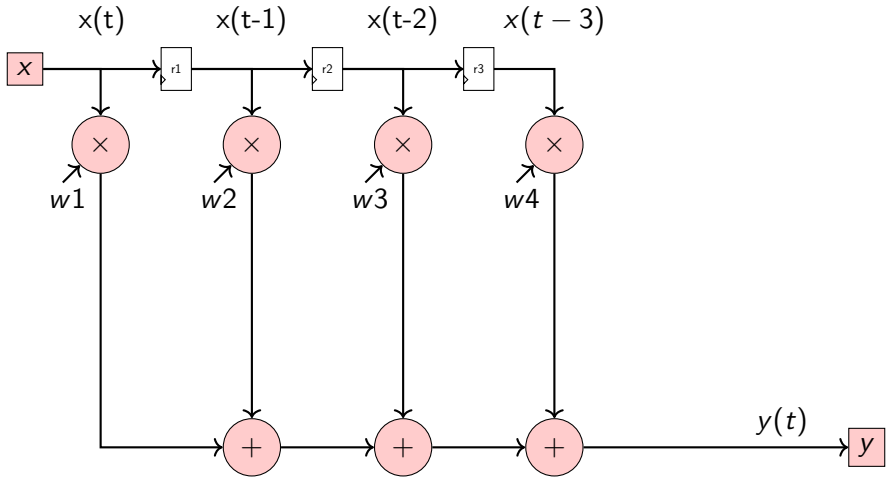


FIR filter circuit view

- FIR filter is a signal-processing computation used in audio/video and other multimedia tasks
- 1-D version with 4 taps performs the following computation:
  $y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$
- Here, t is time, and $t-1$ is one cycle before t, $t-2$ is two cycles before, and so on.
- Thus, we get $x(t-1)$ from $x(t)$ by adding a register on the input x.
- The values of x are streamed in cycle-by-cycle. Thus the throughput of the FIR circuit must be 1.
- You are free to increase the latency of the computation.

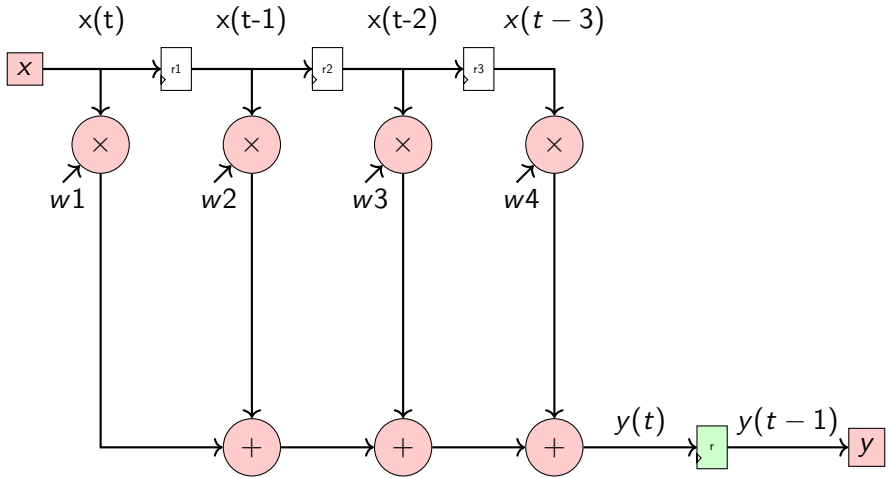
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



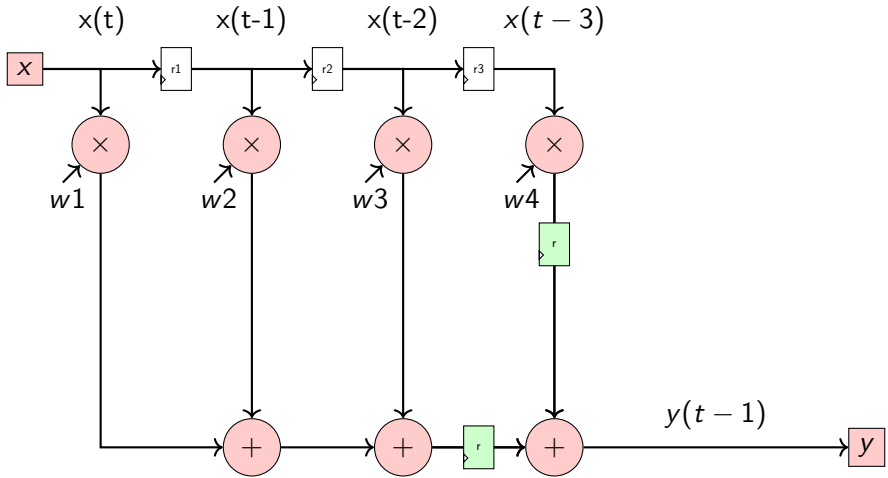
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



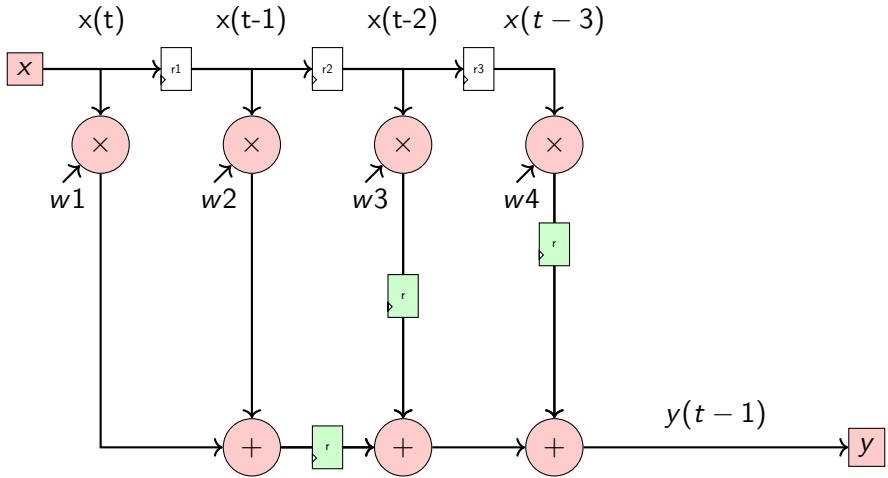
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



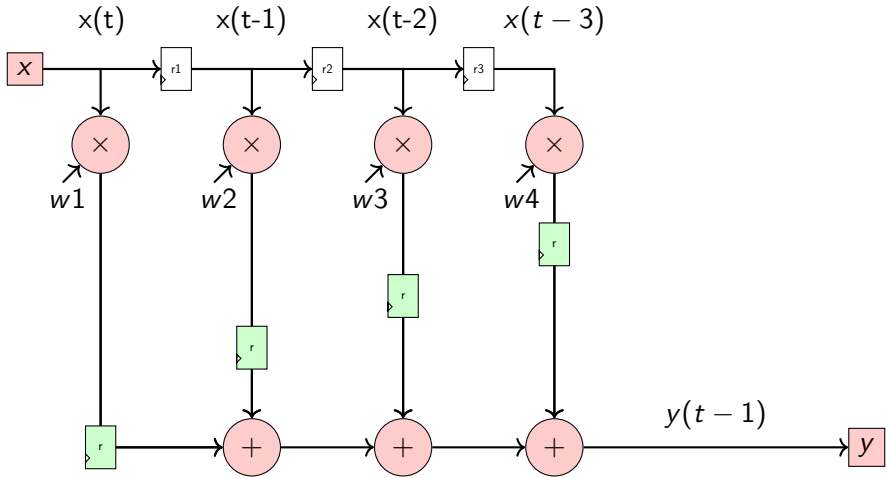
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



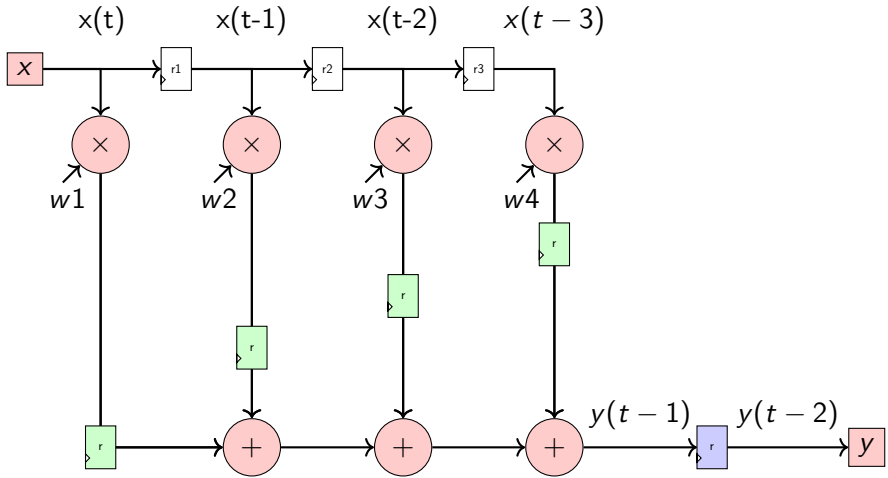
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



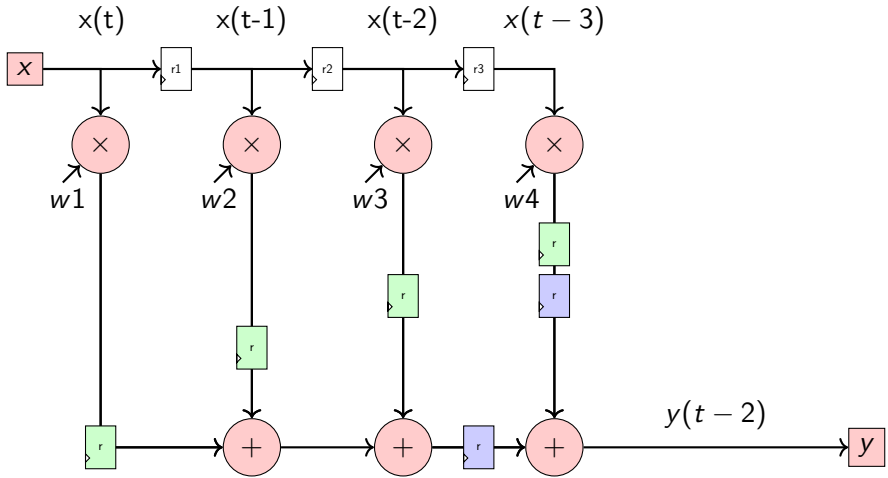
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



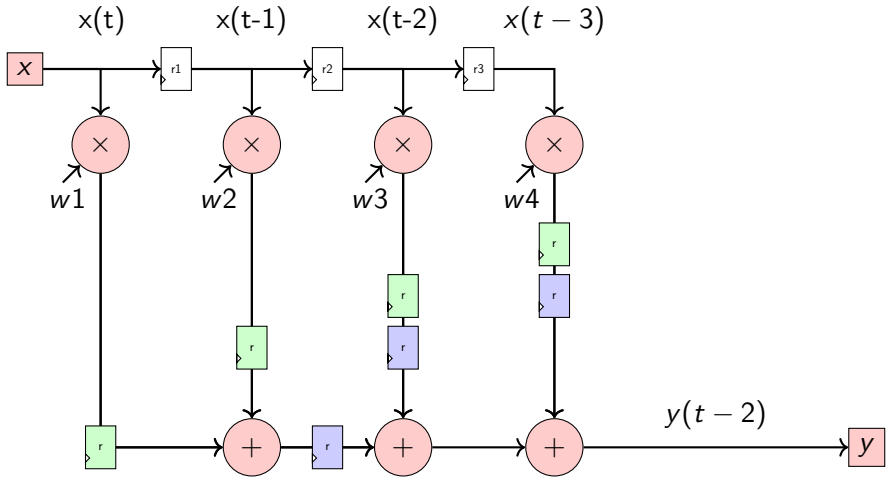
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



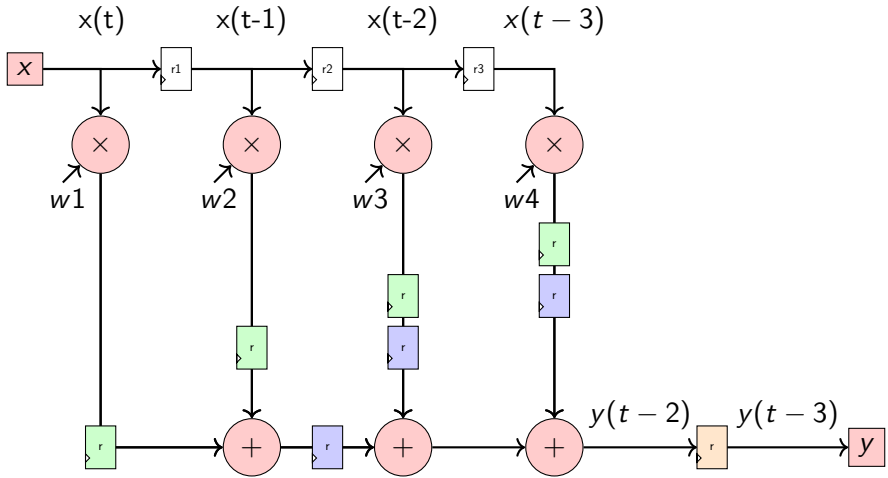
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



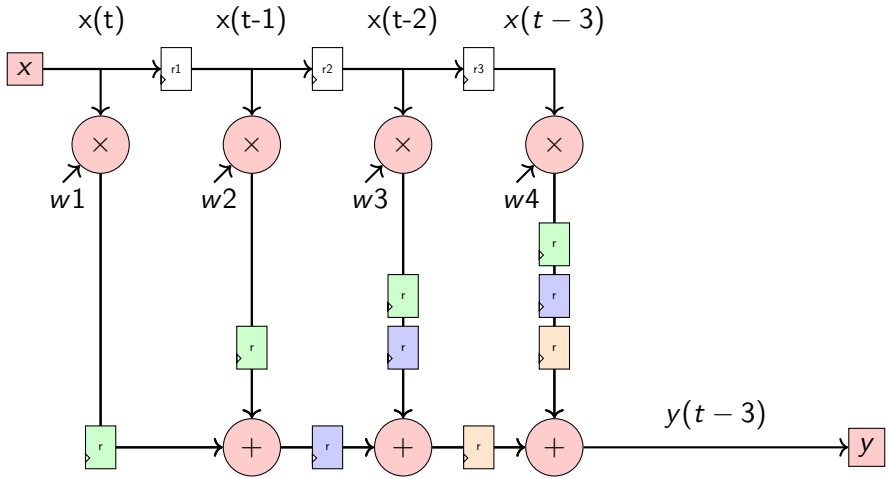
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



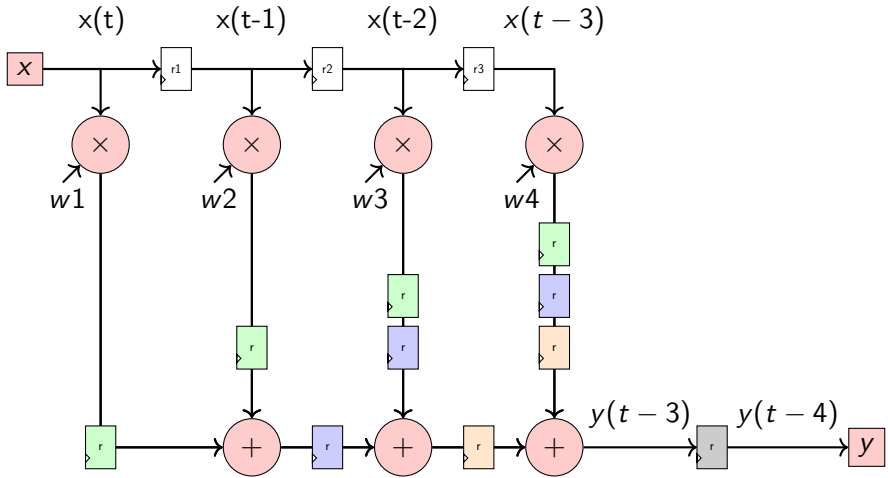
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



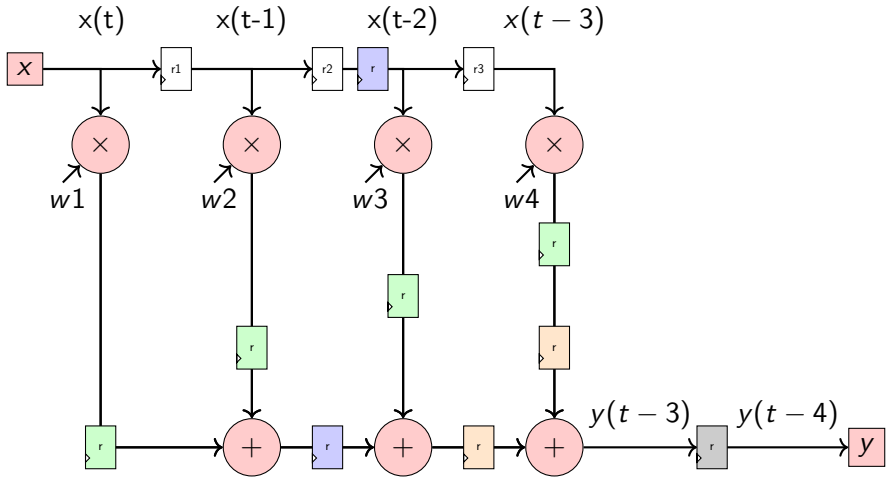
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$



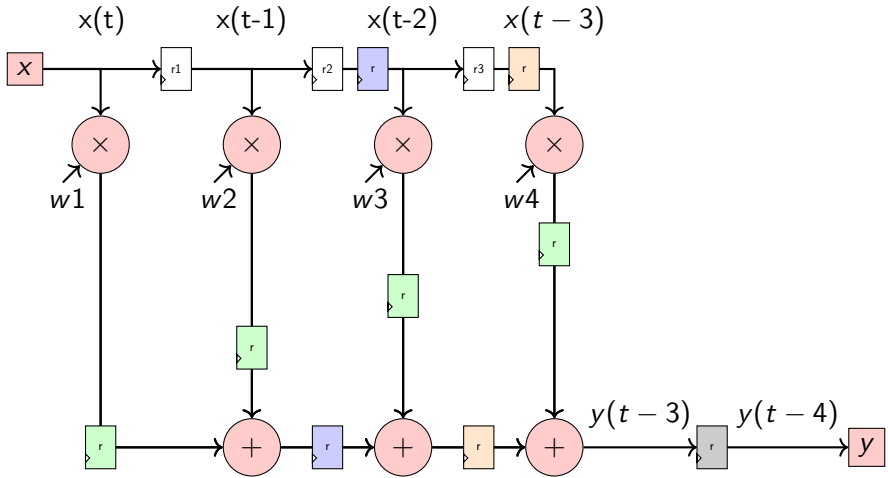
4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$

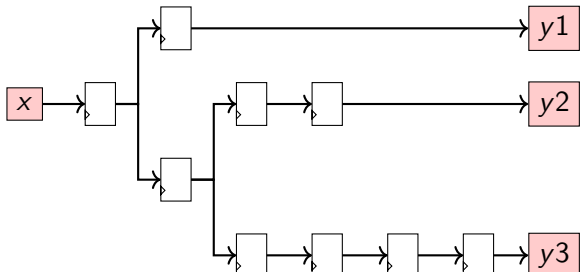


4-tap FIR filter

$$y(t) = x(t) \cdot w1 + x(t-1) \cdot w2 + x(t-2) \cdot w3 + x(t-3) \cdot w4$$

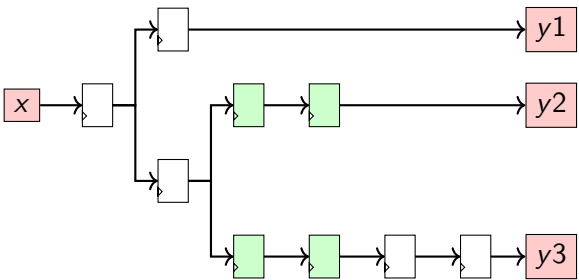


Register Fanout

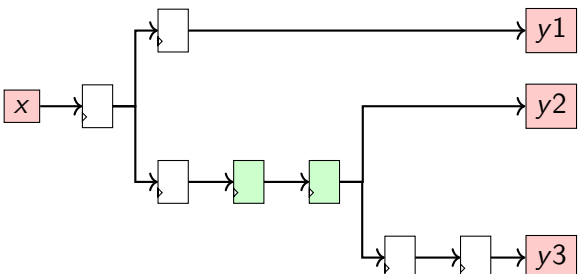


- Measure latency from $x \to y1$, $x \to y2$, and $x \to y3$.
- Any register moves have to preserve correctness + end-to-end latency between all input-output pairs

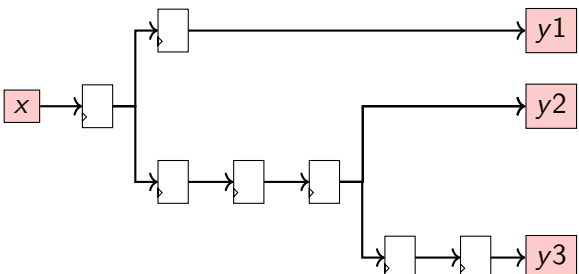
Register Moves



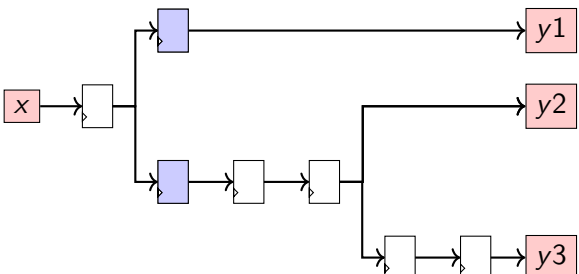
Register Moves



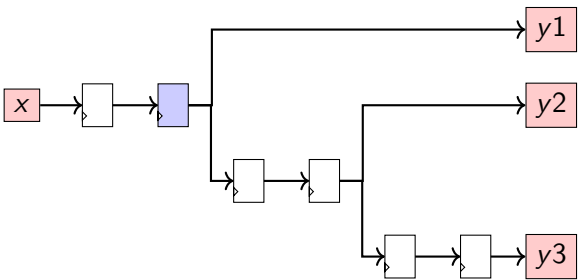
Register Moves



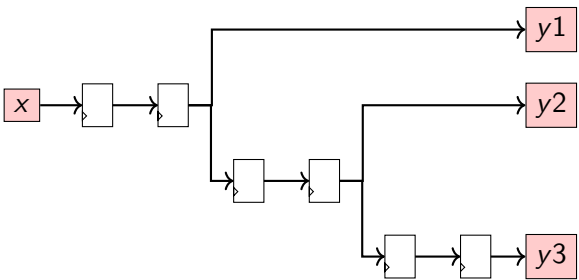
Register Moves



Register Moves



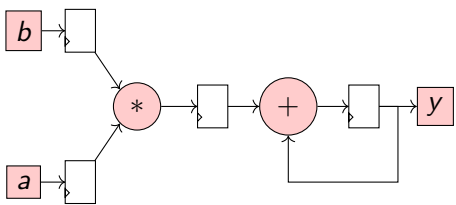
Register Moves



Retiming Heuristic

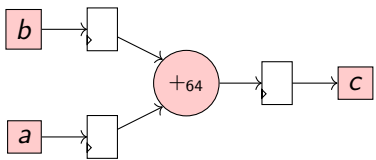
- Formal retiming algorithms are able to exactly compute locations of retiming registers
- Manual heuristic for full retiming is simple, less optimal:
  - Push a register down as far as possible until you hit the inputs or another register
  - Stop adding registers if there are no more locations to push
  - Move things across operators if it helps improve frequency

Circuits with Feedback



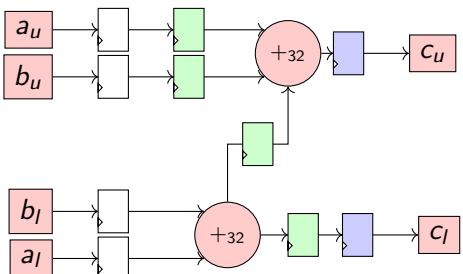
- Feedback loops in circuits are tricky for retiming
  - $y+ = a * b$
- Clock Period $= \max(T_*, T_+)$
- Lets say, $T_* < T_+$
  - Recall Google TPU 32b accumulators for 8b multiplications,
  - For FPGA experiments, lets set 4b multiplier, and 64b accumulator.
- How to improve performance?

Pipeline the Critical Add



- If 64b adder is the slowest component, split into two 32b adds!
- Connect *c_out* from output of lower 32b add to *c_in* of the upper 32b add
- Add pipeline stage between them → increase latency to 3, reduces clock period!

Pipeline the Critical Add



- If 64b adder is the slowest component, split into two 32b adds!
- Connect *c_out* from output of lower 32b add to *c_in* of the upper 32b add
- Add pipeline stage between them $\rightarrow$ increase latency to 3, reduces clock period!

Retiming for Feedback

- 2-cycle split adder cannot work with the $y+ = a * b$ circuit at full throughput=1!
- Why? $\rightarrow$ feedback is produced a cycle too late
  - Cycle 0: a(0) and b(0) were supplied to multiplier
  - Cycle 1: a(0)*b(0) is supplied to 2-cycle adder. a(1) and b(1) are supplied to multiplier, produced anything!
  - Cycle 2: a(1)*b(1) is supplied to 2-cycle adder. a(2) and b(2) are supplied to multiplier. 2-cycle adder hasn't produced valid output yet!
  - Cycle 3: a(2)*b(2) is supplied to 2-cycle adder. a(0)*b(0)+0 is now available as second input to adder! a(3) and b(3) are supplied to multiplier!
  - Here a(0)*b(0) will be added to a(2)*b(2)! Wrong result!
- C-slowing is one way to solve this problem (out of scope of 327)

Wrapup

- Retiming is a powerful technique to pipeline a datapath
- Modern FPGA/ASIC tools allow automated retiming $\rightarrow$ add a constant number of registers to your design output (or input)
- Retiming algorithm involves moving registers across operators while preserving semantics of the computation
- Retiming improves Clock Frequency, Throughput properties are unchanged, Latency can increase (if you add extra registers)
- Straight-forward retiming is not allowed for feedback loops $\rightarrow$ c-slow retiming is one technique to retime feedback loops