EECS150 - Digital Design

Lecture 24 - High-level Design and Optimization 2, Parallelism and Pipelining

April 23, 2002

revised April 30

John Wawrzynek

Parallelism

- Example, Student final grade calculation:

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

- These two 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 more opportunity for parallelism by executing more than one iteration at once, through parallel iteration execution &/or pipelining

Pipelining

- 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

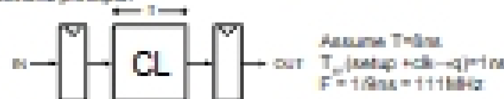
overlapped -> 2 hours

Pipelining

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

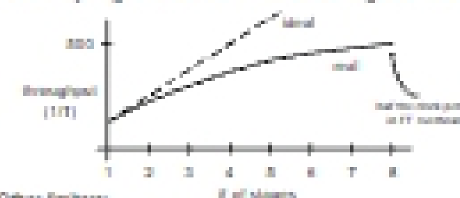
Assume T1 = T2 = 4ns

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 ∝ # of stages.
- After many stages are added, FF overhead begins to dominate:

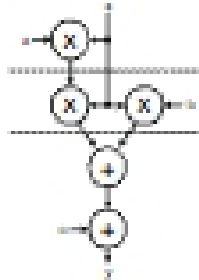


- Other limiters:
  - clock skew contributes to clock overhead
  - unequal stages
  - FFs dominate cost
  - clock distribution power consumption
  - feedback (dependencies between loop iterations)

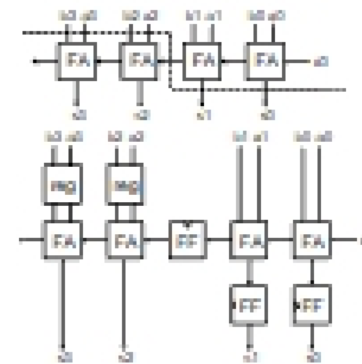
Example

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

- 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:



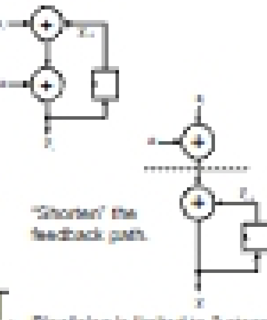
Pipelined Adder



Pipelining Loops with Feedback

Loop carry dependency

- Example 1:
  $y_i = y_{i-1} + x_i + a$
  Can we "cut" the feedback?



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

Add is associative and communitive.

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

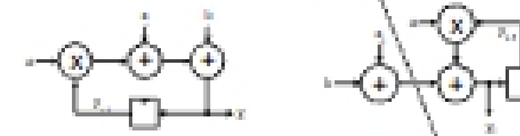
"Shorten" the feedback path.

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

- Pipelining is limited to 2 stages.

Pipelining Loops with Feedback

- Example 2:
  $y_i = a\,y_{i-1} + x_i + b$
- Shorten the feedback loop:

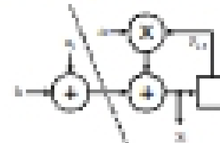


add_1	x_i+b		$x_{i+1}+b$		$x_{i+2}+b$	
mult	ay_{i-1}		ay_i		ay_{i+1}	
add_2		y_i		y_{i+1}		y_{i+2}

- Still need 2 cycles/iteration

Pipelining Loops with Feedback

- Example 2:
  $y_i = a\,y_{i-1} + x_i + b$

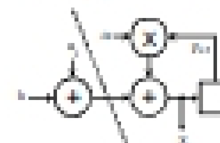


- The chart as drawn does not match the graph.
- This chart can't work. Why?
- The chart implies a delay (flip-flop) after the ay_{i-1} multiply, plus the original after the 2nd add.
- Therefore $y_i = f(y_{i-2})$ instead of y_{i-1}.

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}

Pipelining Loops with Feedback

- Example 2 **Corrected Version:**
  $y_i = a\,y_{i-1} + x_i + b$



- Impossible to "cut" the feedback loop. It's already cut.
- 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 anyway (as noted earlier).

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}

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