EECS150 - Digital Design

Lecture 09 - Parallelism

Feb 19, 2013

John Wawrzynek

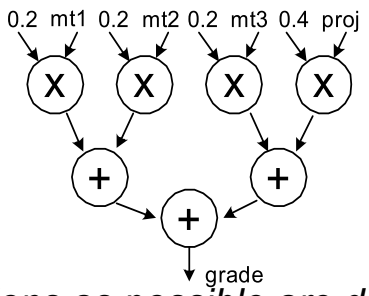
Parallelism

*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;
```

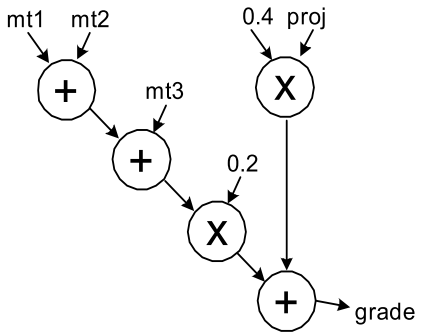
- High performance hardware implementation:



As many operations as possible are done in parallel.

Parallelism

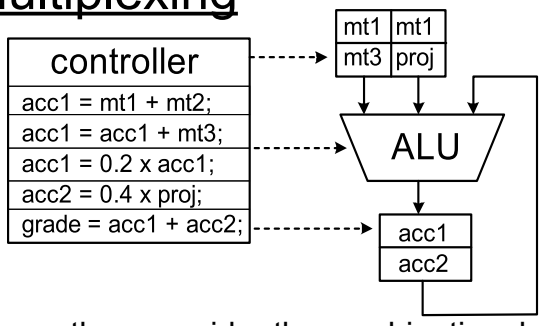
- Is there a lower cost hardware implementation? Different tree organization?
- Can factor out multiply by 0.2:



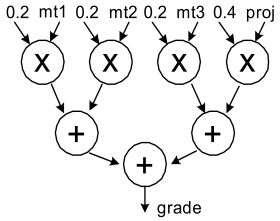
- How about sharing operators (multipliers and adders)?

Time-Multiplexing

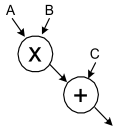
- *Time multiplex* single ALU for all adds and multiplies:
- Attempts to minimize cost at the expense of time.
  - Need to add extra register, muxes, control.

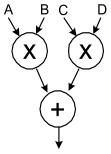


- If we adopt above approach, we can then consider the combinational hardware circuit diagram as an *abstract computation-graph*.



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

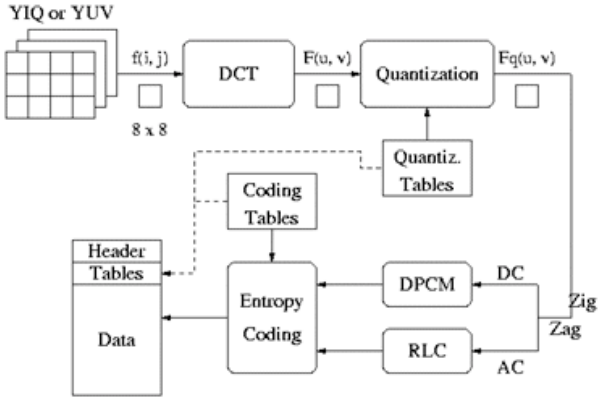
- This **time-multiplexed ALU** approach is very similar to what a conventional software version would accomplish:
- CPUs time-multiplex function units (ALUs, etc.)

```
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?

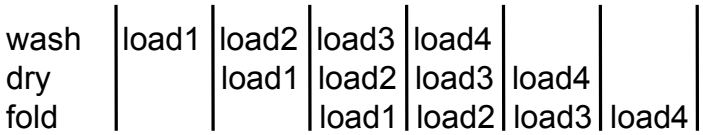
Loops offer opportunity for parallelism
by executing more than one iteration at once,
using parallel iteration execution &/or pipelining

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:

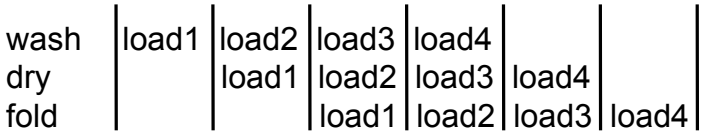
  Analog to washing clothes:

  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



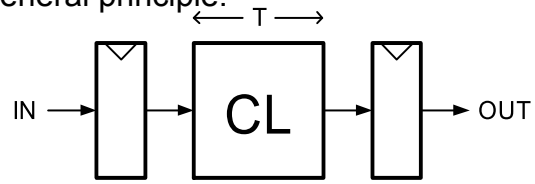
- In the limit, as we increase the number of loads, the average time per load approaches 20 minutes.

- 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.

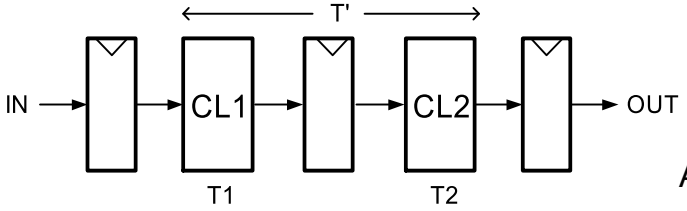
Pipelining

- 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:



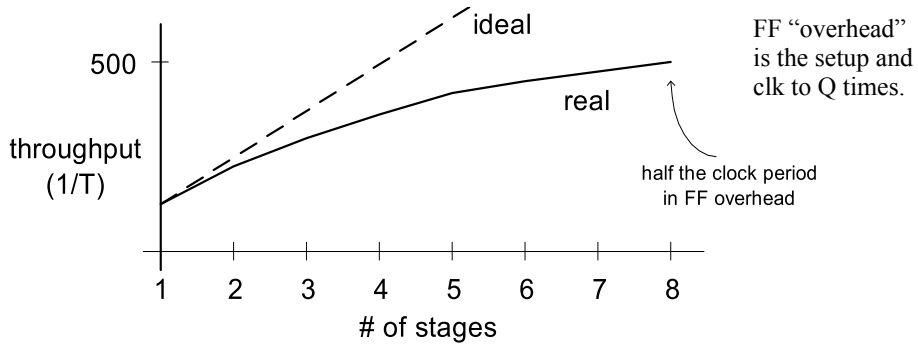
Assume T1 = T2 = 4ns

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:

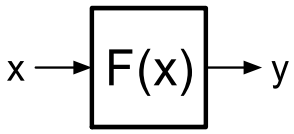


FF "overhead" is the setup and clk to Q times.

- 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)

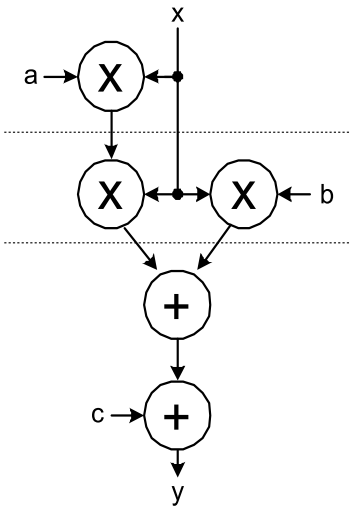
Pipelining Example

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



- 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?

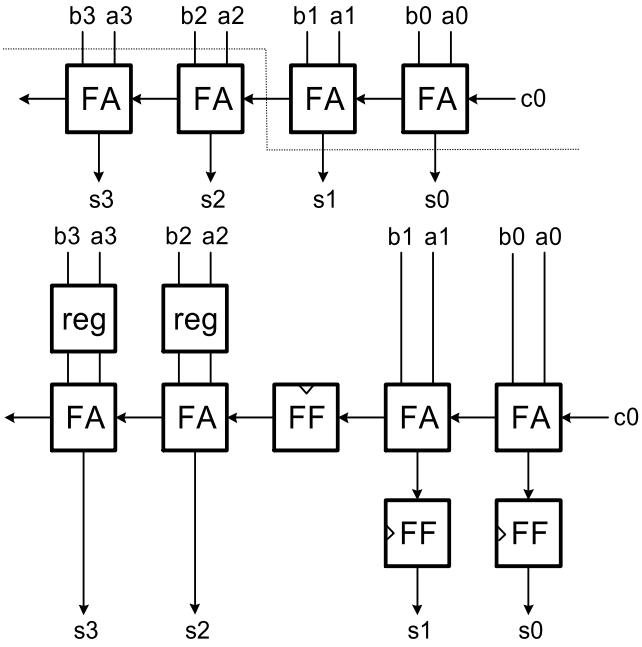
- Computation graph:



Example: Pipelined Adder

- Possible, but usually not done.

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



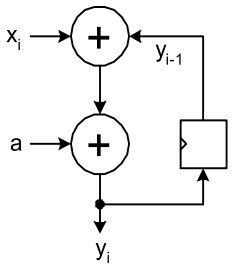
Pipelining Loops with Feedback

"Loop carry dependency"

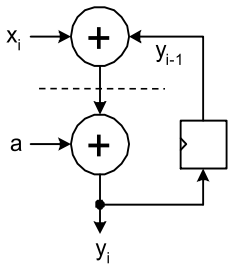
- Example 1: $y_i = y_{i-1} + x_i + a$

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?



- 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.

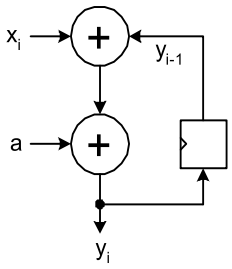
Try putting a register after add1:

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

Pipelining Loops with Feedback

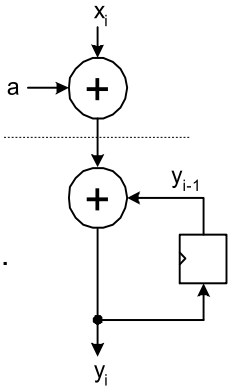
"Loop carry dependency"

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:

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



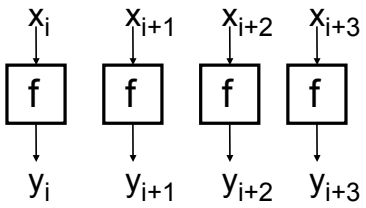
"Shorten" the feedback path.

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.

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



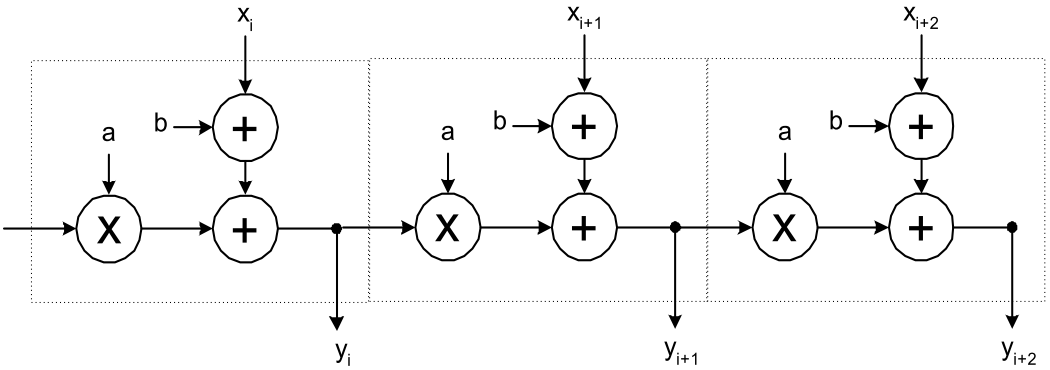
Usually called SIMD parallelism. Single Instruction Multiple Data

- 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.