University of Pennsylvania
Department of Electrical and Systems Engineering
Electronic Design Automation

ESE535, Spring 2009 Assignment #3 Wednesday, February 18

Due: Wednesday, March 4, beginning of class.

Resources: You are free to use any books, articles, notes, or papers as references. Provide citations in your writeup as appropriate.

Collaboration: Please work independently on this assignment. You may discuss general algorithmic strategies and help each other with the compiler, build environment, and debugging, but each student should develop his or her own solution. If you do discuss strategy or get debugging help, please acknowledge in your writeup.

Writeup: Writeup should be in an electronically readable format (HTML or PDF preferred—I do not want to decipher handwriting or hand-drawn figures). State any assumptions you need to make.

Problem 1 (programming) [7pts]

Revise your scheduler to accommodate the intermediate storage strategy suggested below and generate an efficient control FSM for it. This involves several components (each of which is relatively simple):

1. modify provisioning to select operator set and mux population (for this assignment this is no more than omitting unused operators and inputs) [modify provided `your_provision.c`]
2. modify scheduler to respect input limitations (see description below) [likely copy over `your_scheduler.c` and modify it; you will want to look at revisions to `fcfs.c`]
3. write out PLA for controller using sequential state assignment (controller should provide sequencing; use don't-cares where possible) [fill in stub in `your_writers.c`]
4. write out KISS2 for controller with symbolic states (controller should provide sequencing; use don't-cares where possible) [fill in stub in `your_writers.c`]
5. optimize PLA with `espresso`
   (we provide make rules and default options; you may want to refine options used—modifications to `test/Makefile`)
6. optimize KISS2 using `nova` and `espresso` (we provide make rules and default options; you may want to select and refine options used—modifications to `test/Makefile`)

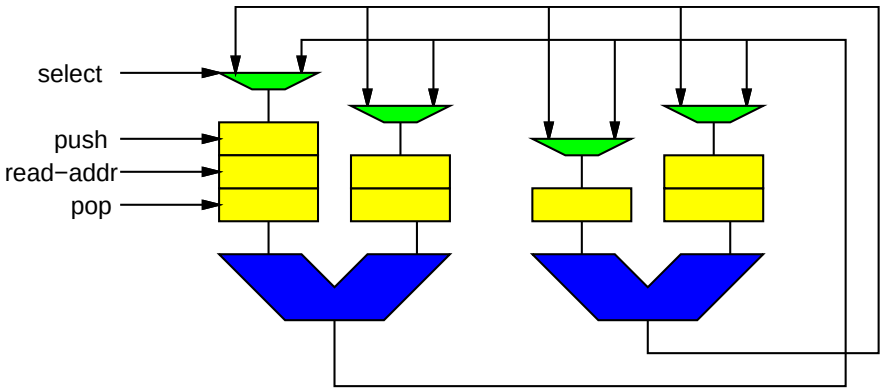
Provided starting point is available in `~ese535/spring2009/assign3.tar` on `eniac`. This builds on the assign2 framework. You will want to merge in your scheduler from assignment 2, making the modification noted above.

You can find `espresso` and `nova` in `~ese535/bin`. Documentation on the PLA format for `espresso` exists in `~ese535/man/man5/` (see both `pla.5` and `espresso.5`). Documentation for `espresso` and `nova` are in `~ese535/man/man1/`. To format the document in the `man` subdirectories, use `nroff -man` *filename*. The KISS2 file format is described in appendix A.8 of `~ese535/doc/sis_memo.pdf`.

Turnin:

1. Your code (a tar file as on previous assignments that can be unpacked and built)
2. Explanation of changes you made
3. Summary of your results across the provided benchmark set including:
   - mux requirements
   - comparison of controllers (pterm count or memory words)
     - memory controller case
     - unoptimized PLA
     - optimized PLA for sequential assignment
     - optimized PLA for `nova` assignment
4. Explanation of any tuning, benchmarking, and profiling you performed to arrive at your current solution. This includes selection of options for `nova` and `espresso`.
5. If you add any options or otherwise change how the code is run, provide an explanation of how to run your scheduler.

Storage Strategy



Consider each operator as having a FIFO queue on each of its inputs. When an output is produced, it is selected and stored into the queue for all of its successors. Note that this means each of the successor input queues can only absorb one value per cycle, a limit that was not imposed on assignment 2. Reads do not have to be to the front of the FIFO; rather, they

must specify which value in the queue they need to select. Provided code already calculates these offsets and stores them in the `read_addr` array slot of the `platform` structure.

We are suggesting this FIFO strategy to limit the details we have to worry about during this assignment. In practice, we would likely not use push/pop but more explicit memory stores. Using push/pop allows us not to deal with memory assignment and to only specify addresses on the read. Similarly, we put the FIFOs on the inputs to minimize the impact on scheduling; an alternate strategy is to put the FIFOs on the outputs of operators, but this will require some additional care in architecture or scheduling (what happens when multiple operators want values produced on different cycles form the same operator?).

Each operator input will therefore require the following controls on each cycle (shown on one of the memory columns in the figure above):

- input mux select (possibly multiple bits)
- push (one bit) – write the selected value to memory
- pop (one bit) – remove the oldest value from memory
- memory select (possibly multiple bits) – select the nth oldest value in memory.

Issue a pop on the cycle **after** the oldest value in memory is used the last time; assume the pop occurs at the beginning of the cycle issued, so select addresses are with respect to memory contents after the pop. Similarly, if there is a push and a pop in the same cycle, the memory capacity required remains unchanged (the push can go into the slot freed by the pop). Provided code already calculates the pop/pop controls and stores them in the `push_pop_action` array slot of the `platform` structure.

Mux Inputs

We special case output and input operators, essentially assuming there is an external fully populated interconnect so you can bring a value in from any input and produce values to any output.

- You only need a single mux input for each "INPUT".
- Output of your computations can go to any "OUTPUT"
- There is no memory buffering on the OUTPUTs; you will see in the code (`fcfs.c`) that we co-schedule an output immediately following the production.

In part we do this so you don't have to worry about scheduling INPUT operators based on where they are used. This also simplifies the muxes so they do not end up being dominated by the wide range of inputs.

Problem 2 [3pts]

Consider how you would minimize the multiplexing costs for a larger design.

1. Devise and describe a strategy for performing this optimization, including formulating any subproblems in your strategy.

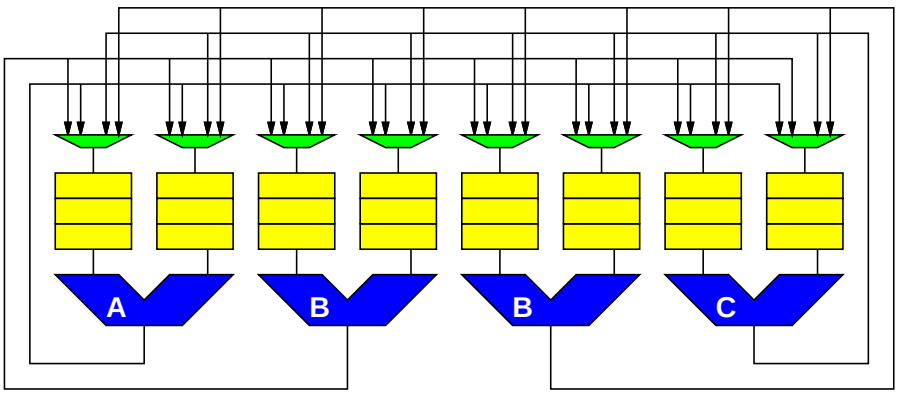
2. Provide psuedocode for your algorithm.
   (You may describe the complete algorithm(s); you may use algorithms you've seen in class as subroutines, but you must carefully specify how they are used (*e.g.* what cost functions do you use, how are nodes and edges weighted).

Details

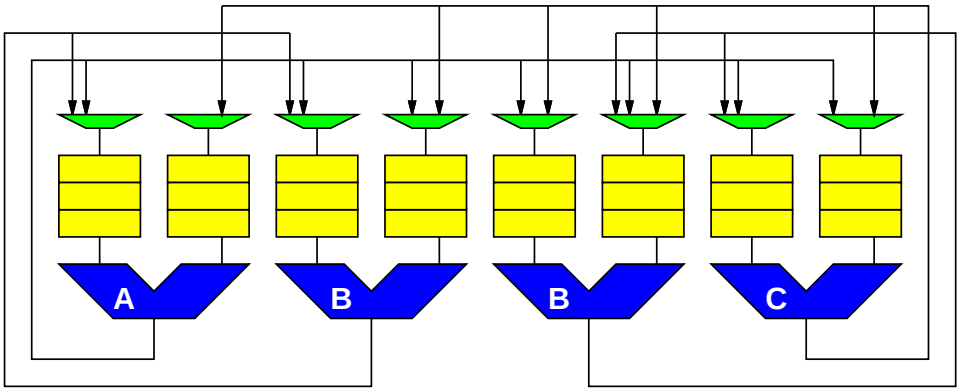
- Basic model follows the one above.
- Here we consider provisioning more than one of each operator. To contain the scope of this problem, assume we have at most two of each operator type.
- Can we arrange for the schedule to reduce multiplexing cost? (*e.g.* When we had one operator of Type A which consumed a result from the one operator of type B, then we had to have a mux input to the A operator from the B operator. However, when we have two operators of type B, it may not be necessary for A to have an input from both B operators. Particularly, if we can manage to schedule every B that feeds an A onto one of the B operators, then we only need the one B as an input; if we schedule one such B onto the other B operator, we must have both inputs.)
- You may consider single-level multiplexing (every input queue has a single multiplexer selecting from each operator output) and two-level multiplexing (there is one or more multiplexers that select a subset of signals for input to other multiplexers).

Multiplexer Optimization Examples

Unoptimized Single Level



Optimized Single Level



Cluster Optimized Only, Two Level