Lecture 15:

Datapath Functional Units

Overview

Reading

W&E 8.2.8 Shifters

W&E 8.2.7 Multiplication

Introduction

This lecture continues to discuss how to build functions that are used in datapaths. It begins by talking about shifters, and then briefly describes multiplication. After describing these two functions, we take a step back, and look at how datapaths are put together, and how the control for the datapath is generated.

Shifters

There are many different kinds of shifters.

- Simple Shifter

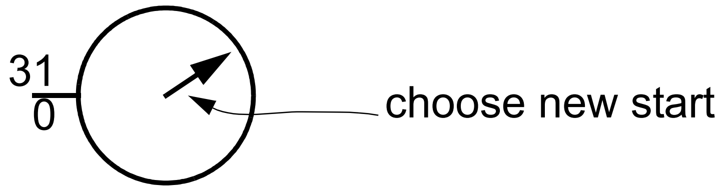
  Shift the number to the right or left and fill-in with zeros

- Arithmetic Shifter

  Left shifts are the same as simple shifter, but on right shifts use the sign bit to fill-in the new blank spaces (-2 shifted right by 1 gives -1)[1]

- Barrel Shifter

  Wrap the number onto a circle. The shift amount indicates where the new MSB will be. (Useful for rotating the bytes in a 32-bit word)

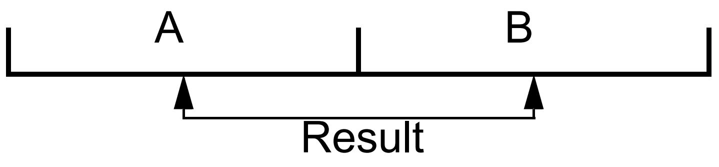


1. Need to be careful about arithmetic right shifts. ‘1’ right shifted by 1 is 0. ‘-1’ right shifted by 1 is -1

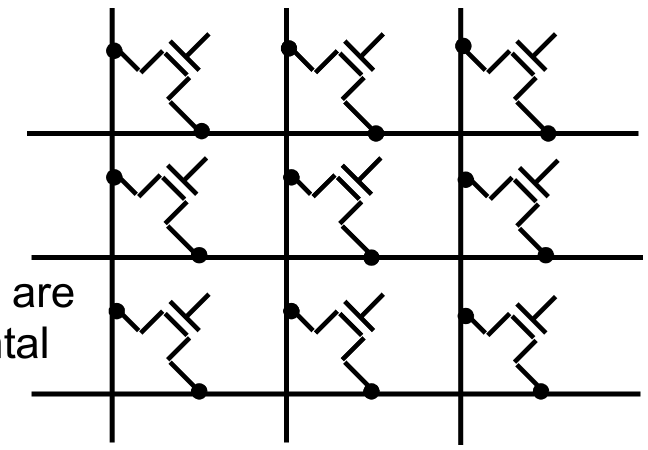
Funnel Shifter

Is the most general kind of shifter

- Can do all the other shifts.
- Concatenates two n-bit words together and then selects any contiguous n-bit subfield.



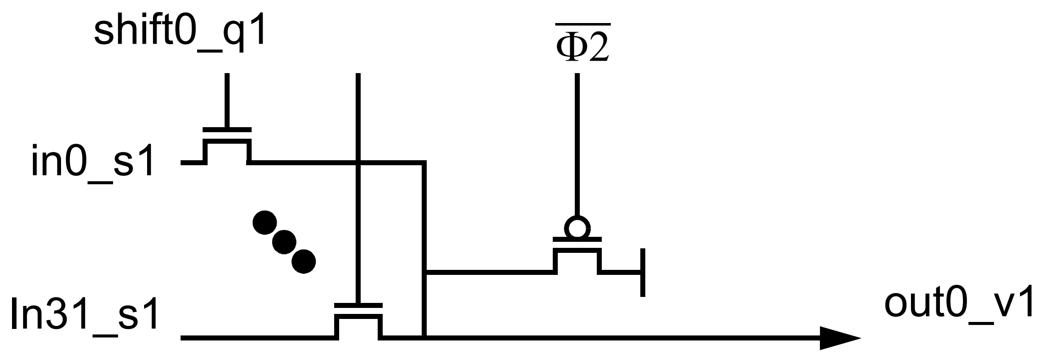
- If A=B get a barrel shifter
- If A = sign bit, get arithmetic shifts
- And it does byte inserts too.
- Can implement this shifter using a cross-bar switch, where the inputs are vertical and the output are horizontal



Shifter Design

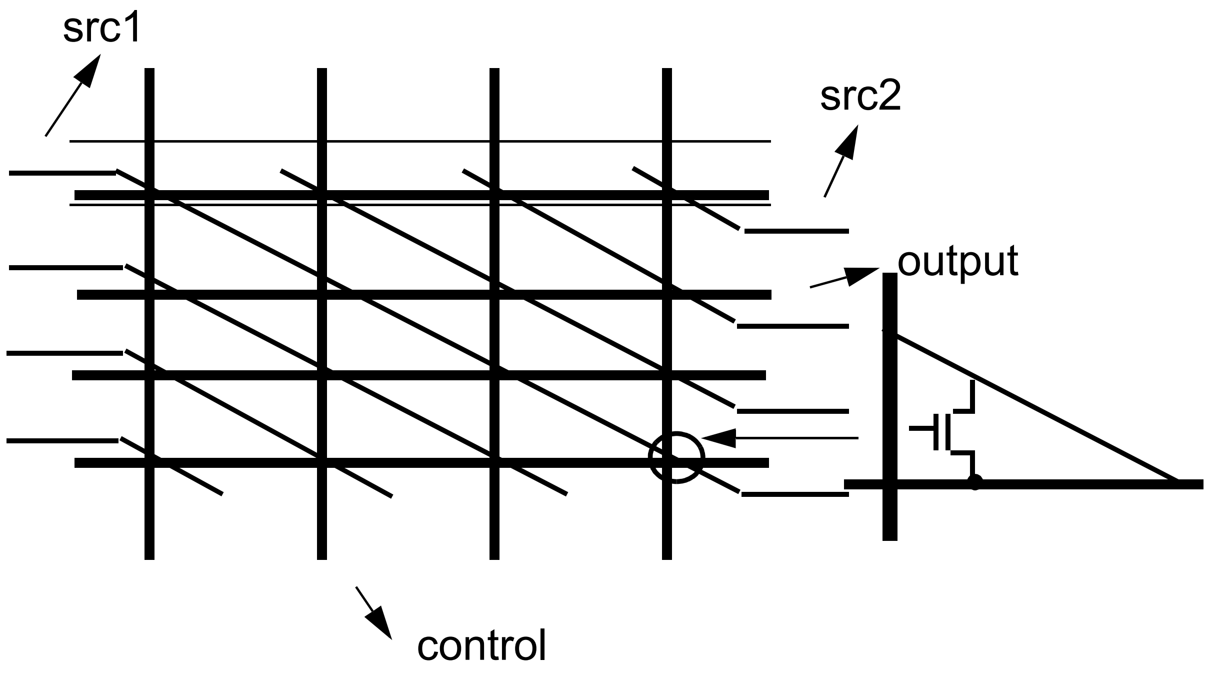
Can think of a shifter as a large fanin multiplexer

- Connect the correct input to the output
  - For a completely general Funnel Shifter there are 2n inputs
  - This is 64 inputs for a 32 bit machine
- Clearly, we don't want full CMOS transmission gates
  - Reduce the number of control lines by using nMOS only gates
  - Use precharging - the input will selectively discharge the output.



Shifter Layout

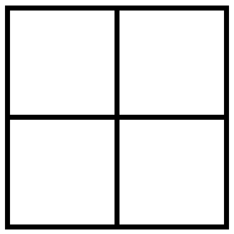
It is a little tricky, but is not very bad:

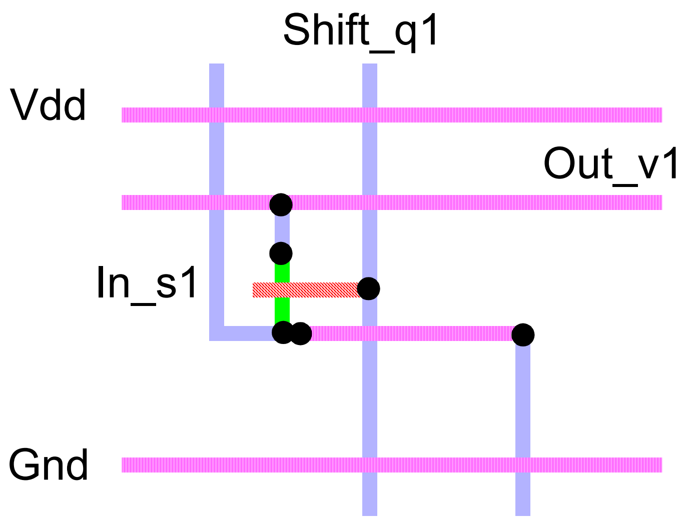


Inputs are driven from both sides, and then run diagonally through the array. Control runs vertically, and the output runs horizontally.

Bitslice Shifter Works Well

Even though there are many wires that flow between bits, this data is regular and can easily be embedded in the cell. In our system we do not have diagonal wires, so we make them stair steps



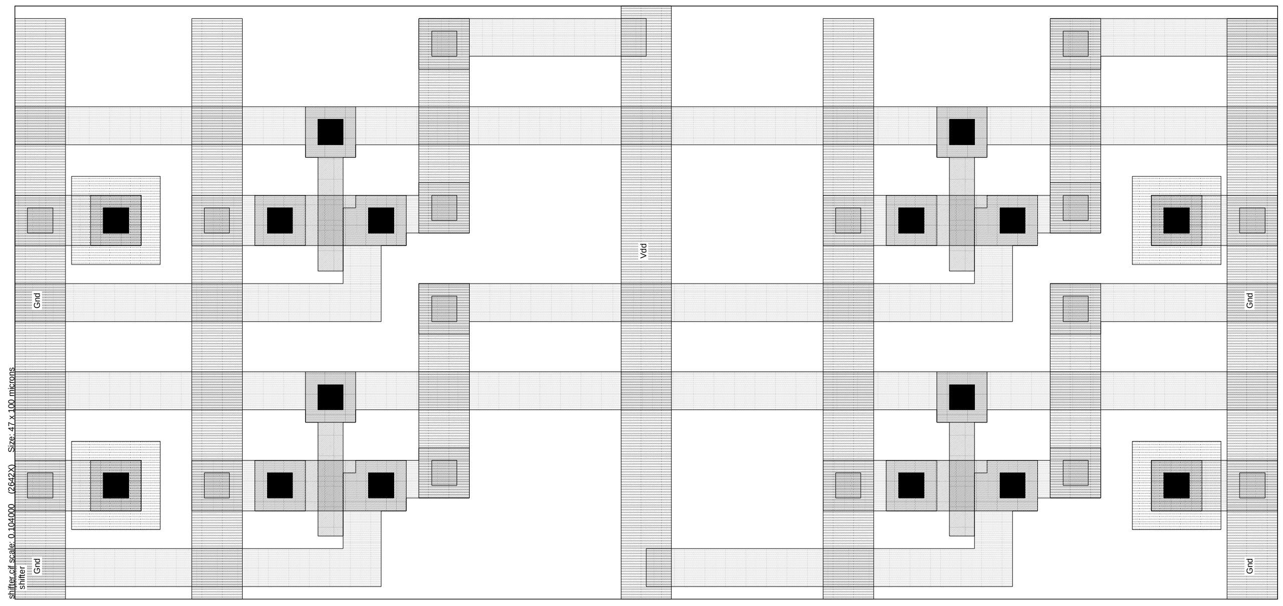


Cells don't use Vdd, Gnd, but need to route them for the other cells. The shift cells should not be mirrored (all nMOS so there is no need), but can fit in with a mirrored datapath (just change Vdd Gnd names)

Shifter Layout

Diagonal dataflow mapped into a regular manhattan layout.

Using M2 data, M1 control. (cell is rotated so data flows vertically)



Multiplication

In binary it is pretty simple.

- The bit multiplication is just an AND gate
- All you need to do is add up the partial products

$$
\begin{array}{lccccccccl}
\textbf{multiplicand:} & & & & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & & \mathbf{12} \\
\textbf{multiplier:} & & & & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & & \mathbf{5} \\
\cline{5-8}
 & & & & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & & \\
 & & & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & & & \textbf{4 partial products} \\
 & & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & & & & \\
 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & & & & & \\
\cline{2-8}
 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & & \mathbf{60}
\end{array}
$$

repeat n times:

compute partial product; shift; add

Multiplier Implementation

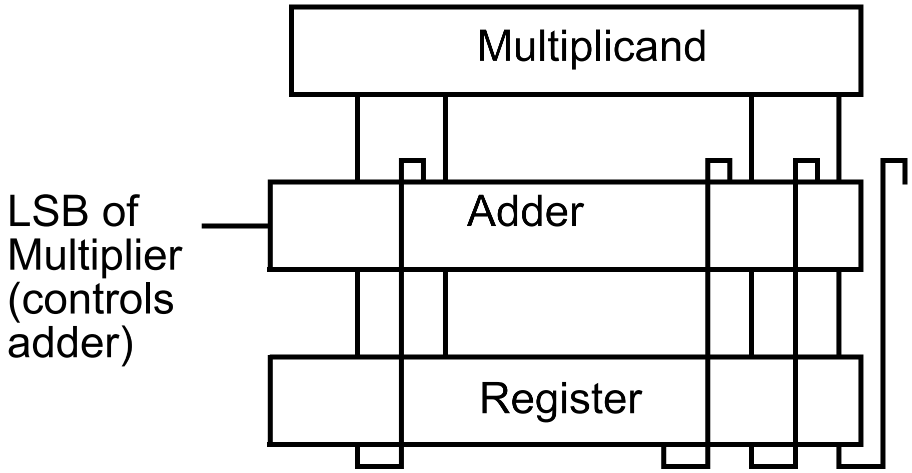
Two approaches

- Shift and Add
- Parallel Multipliers

Shift and Add

- Use a standard adder, and multiple cycles to add up all the partial products.
- Use the LSB of the multiplier to decide whether to add the multiplicand this cycle
- Then shift both the multiplier and the partial result right by one, and repeat.

Serial Multiplier

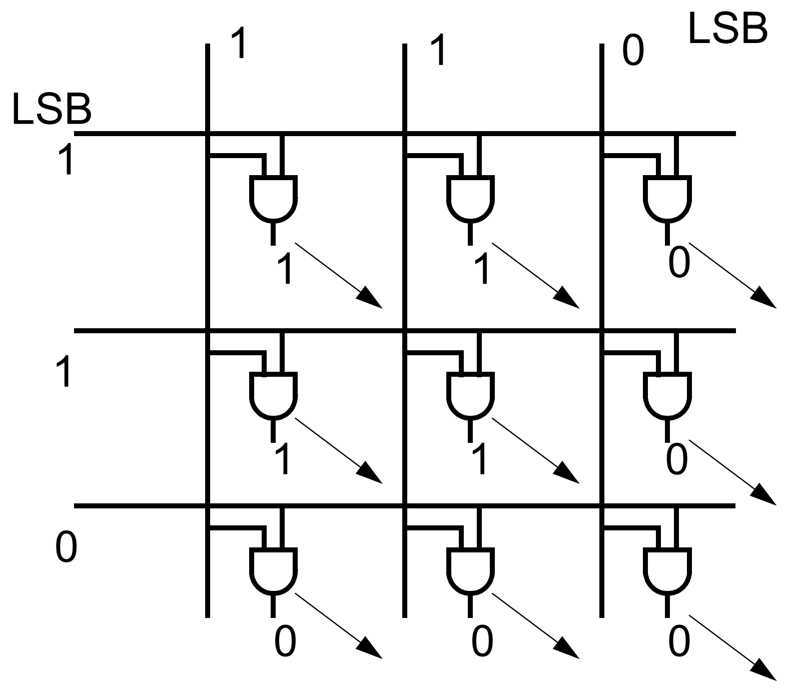


This hardware generates a 2n bit result, so you need to catch the bits that shift off the end of the register. If you only wanted to generate an n bit result, then it would be better to shift the multiplicand to the left.[1]

1. There are many techniques to make this faster. One is to have some hardware find the ‘1’s in the multiplier since, the zeros don’t do much. For this we need to shift the result by the number of zeros skipped plus one. There is also a way to recode the multiplier (Booth Coding) to reduce the number of steps, but it is outside the scope of this class.

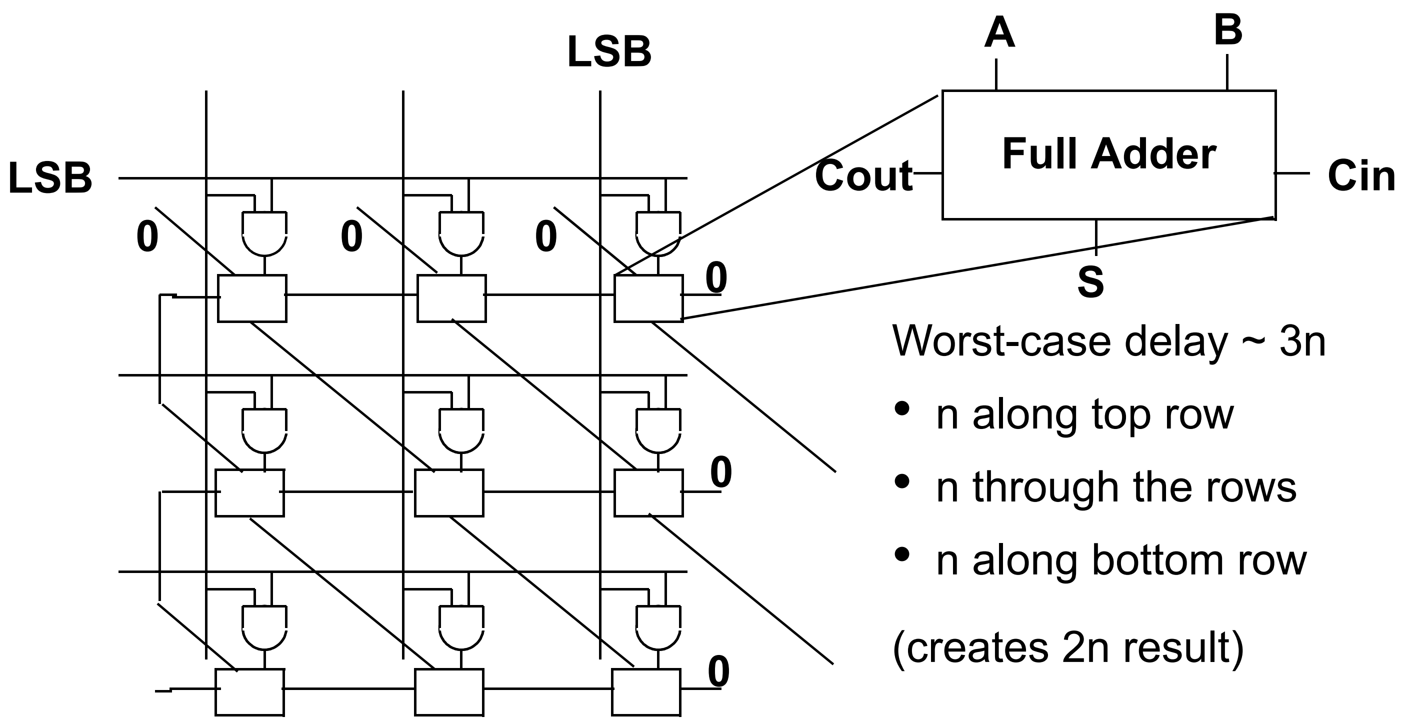
Parallel Multipliers

Idea is to generate all the partial products in parallel, and then add them together by using lots of adders, rather than lots of cycles:



Adding the Partial Products

- Use a ripple adder in each row (carry ripple right to left)
- Sums ripple diagonally (down and to the right)



Worst-case delay ~ 3n

- n along top row
- n through the rows
- n along bottom row

(creates 2n result)

A Better Way

There are a number of better ways to do parallel multiplication.

- Look at a full adder again

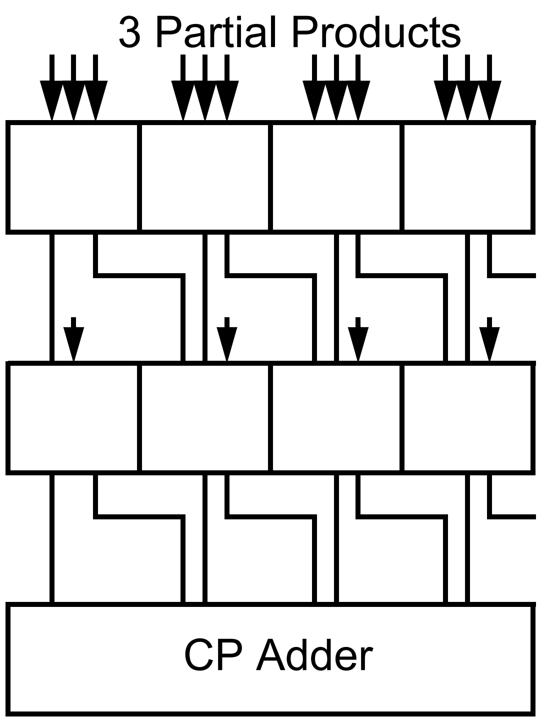
  Takes three inputs of equal weight, and adds them together

  Produces two outputs of different weights

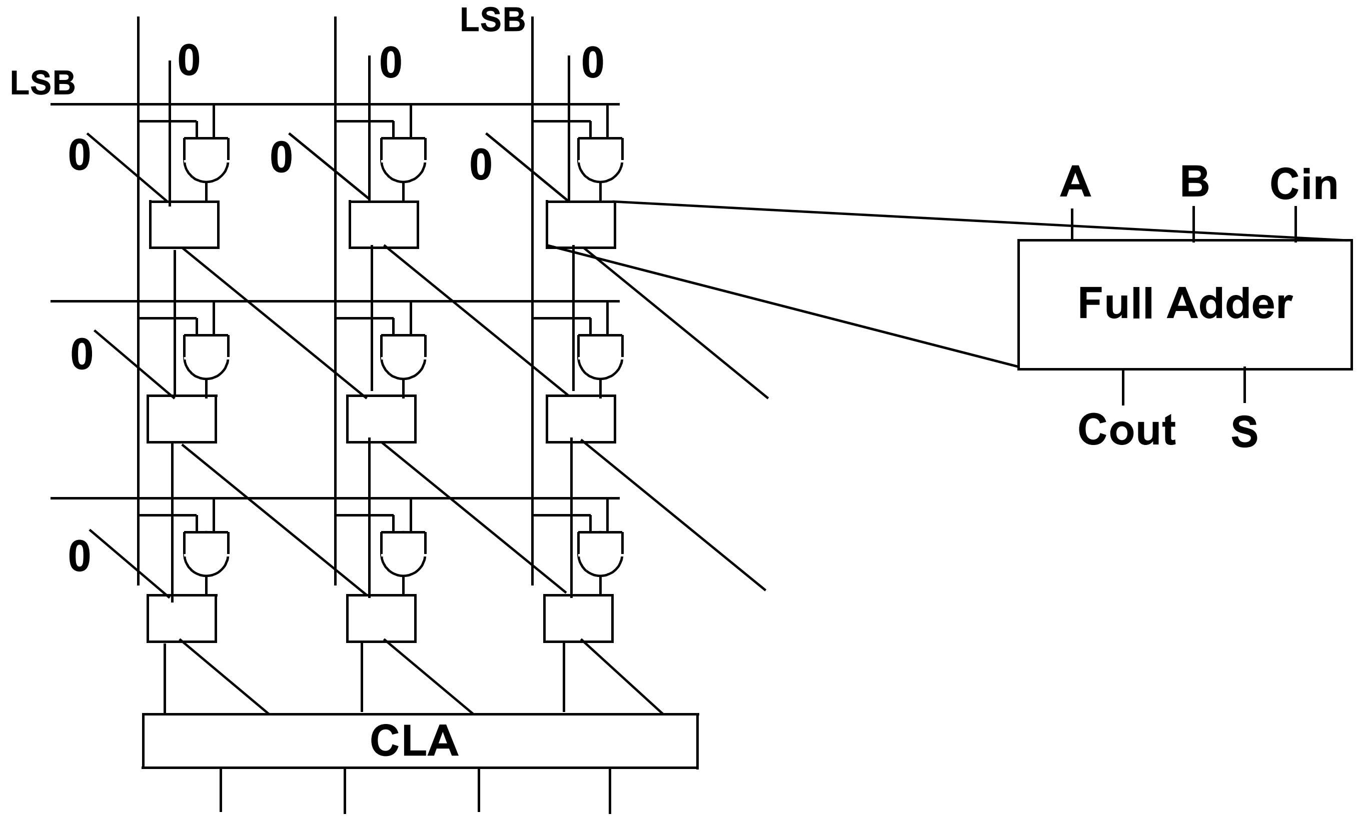
  So a row of FAs could add three partial products and produce two outputs (one is shifted by one)

  Next row could add one more partial product

  Need Carry Propagate Adder at the end to reduce the final two partial products into a single sum. This can be a simple ripple adder, or a more complex Carry Look Ahead Adder for better performance.

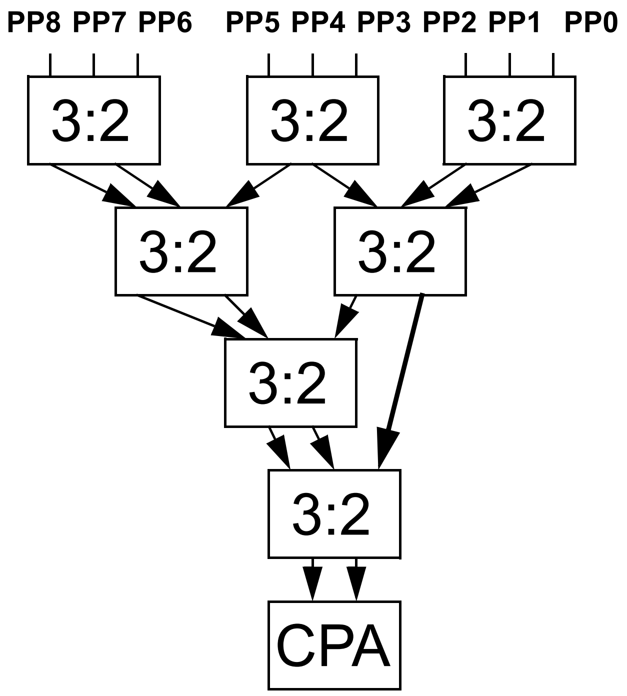


Carry Save Adder Array



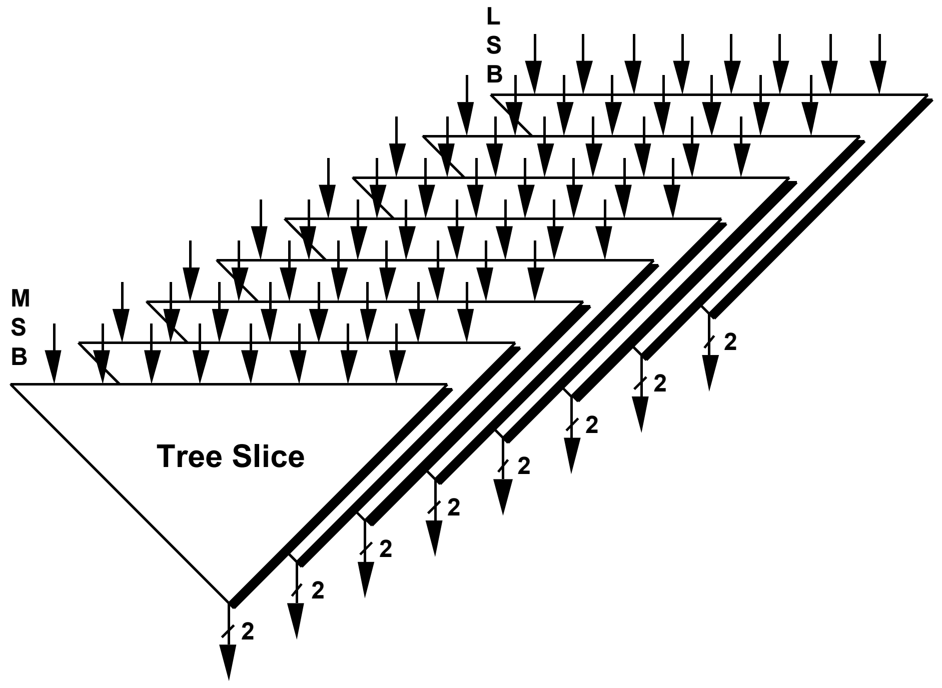
+ Multiplier Trees

Can reduce the number of series carry-save adders, by adding more of the partial products in parallel. Use a tree of adders. First proposed by Wallace (Wallace Trees)



+ Multiplier Tree Slices

An n X n bit multiplier requires 2n slices to produce the complete product. Some bits from one slice must go to the adjacent slice further complicating the wiring.



+ Multiplier Trees

- Multiplier trees are pretty fast.
- Time is proportional to the Log(n)
- Can pipeline the levels in the tree to increase the data rate

BUT

- Takes a lot of hardware
- Wiring is a mess

  While you draw a tree, the adders must be layed out in a linear array (Each adder is n-bits wide already).

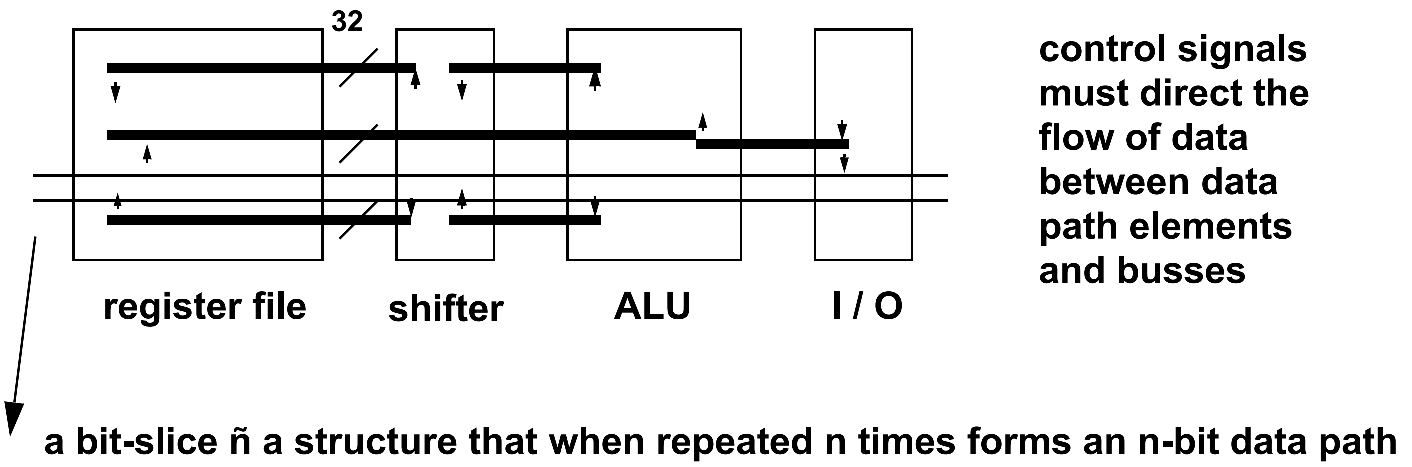
  This means that there are number of wiring tracks that need to allocated to each bit slice.

  And unlike arrays, some wires in a tree are not very short.

Use only when you need the speed (or have a program to layout the tree.

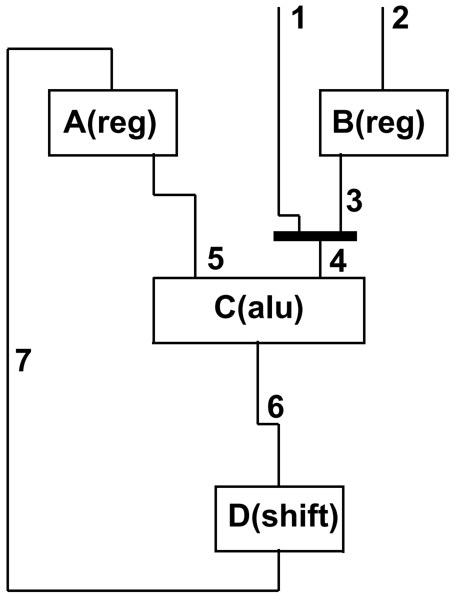
Datapath Summary

- Composed of register files, latches, ALUs, latches, Shifter, latches, other functional units, latches
- Uses a bitsliced layout organization. Wordslices often pipelined.
- Uses a bused based communication protocol
- Wiring in the datapath is regular.


32
control signals must direct the flow of data between data path elements and busses
register file
shifter
ALU
I / O
a bit-slice ñ a structure that when repeated n times forms an n-bit data path

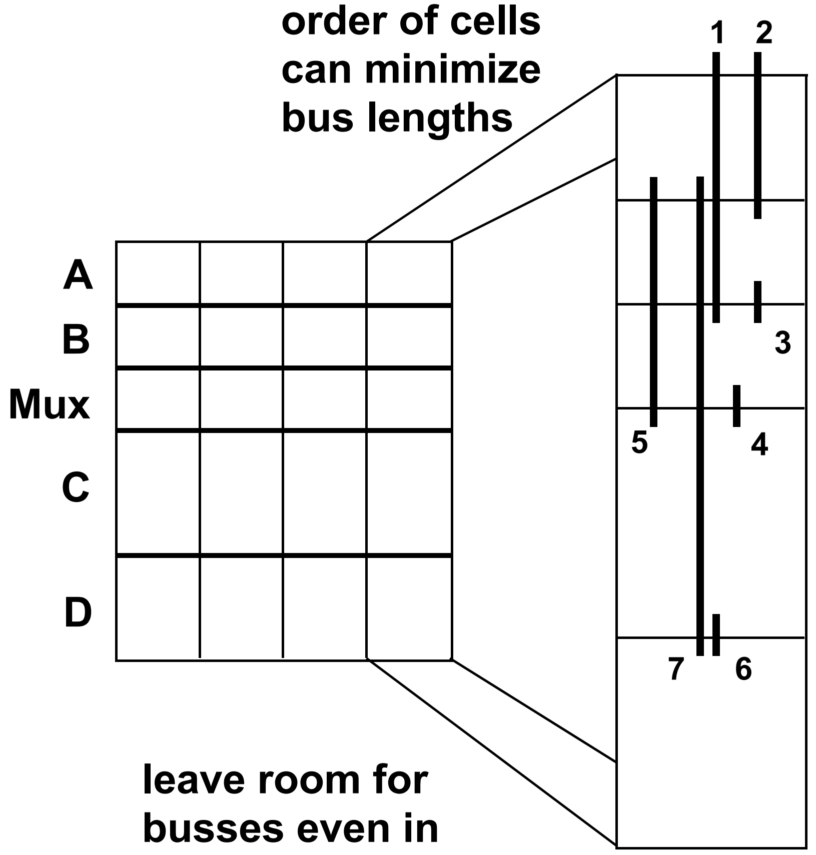
- Problem is that all cells must have the same height -- means you need to estimate which cell is the tallest, and use that to set bit pitch.

Bit Slice Design



if all busses are 32 bits this is a nightmare to interconnect (note: ALU inputs need to be interleaved)

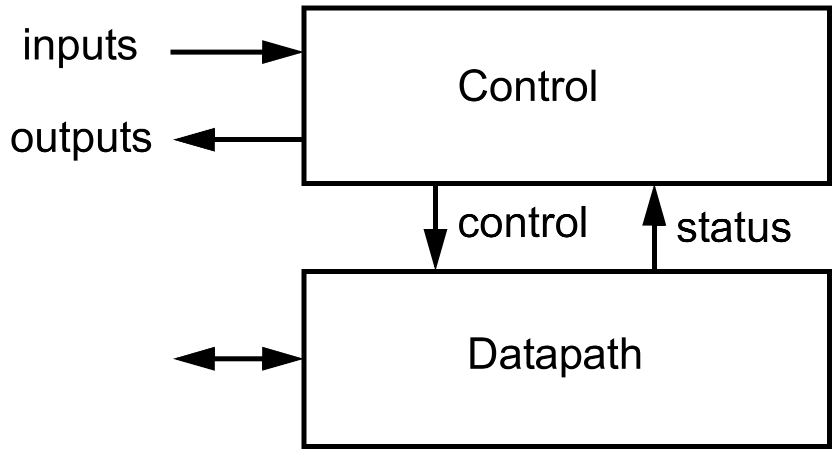
order of cells can minimize bus lengths



leave room for busses even in cells that don't need them

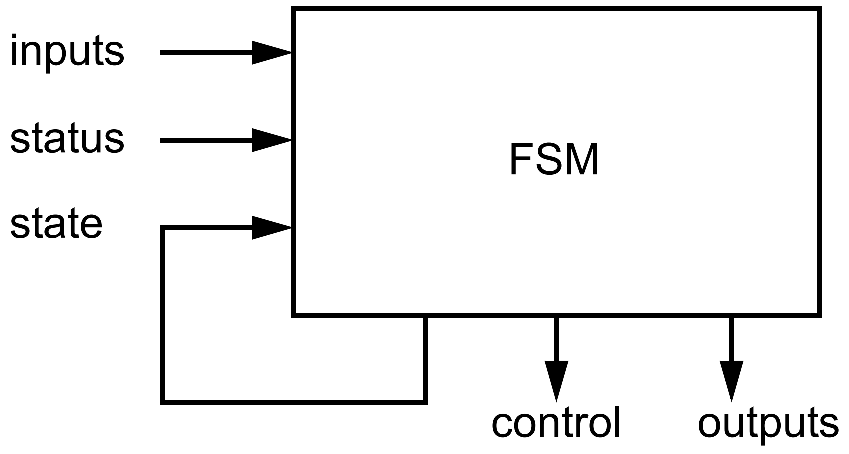
Datapath Control

So far we have talked about what goes into the datapath. These function units / latches / muxes manipulate the data itself. But this logic is only part of the whole chip. Something needs to tell the datapath elements what to do. And this is the function of the control.



When the datapath is pipelined (multiple parts of operations in progress simultaneously), the control keeps track of the status of all of the pieces.

Control



It is usually an FSM since some operations that the chip performs take multiple clock cycles, and the controller must know where it is in the instruction.

In pipelined microprocessors, each instruction may take n cycles to complete, but the next instructions are started before previous ones have finished. The controller must track which instruction is at each function in each cycle. Complexity explodes with pipeline depth.

Control Implementation

In an ideal world:

- Write HDL (Verilog) of the control

  Define the inputs / outputs / states of the controller

- Use synthesis tools to convert it into logic
- Use place and routing tools to convert logic into layout

But the world is never so ideal:

- The automated-tool-only approach is both too big and too slow
- Need to help the tools along

Real Controller Design Flow

1. Write a description of the controller in Verilog

   Simulate it to get it to work

   Think up hard cases to try to break it

2. Use synthesis tools to generate the logic

   Look at the critical paths

   If (where) they are too slow,

   - Try to figure out why the tool generated the logic it did
   - Change the section in the Verilog that generated the slow logic
   - Goto 1.

3. Use standard cell place and route system to generate the layout

   Look at area, make sure aspect ratio is ok, and total area is ok

   Look at speed. If slow, try to change the placement. If that doesn't work then change the Verilog and goto 1.

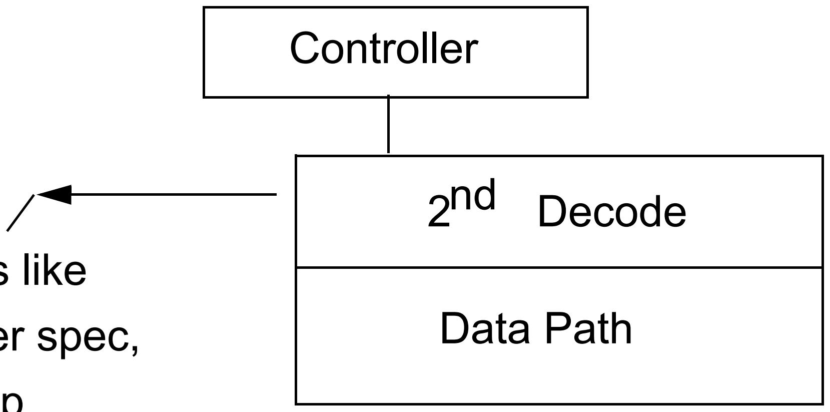
Local Control

Don’t want to generate all the control signals in the controller.

- Too many wires needed between sections
- Poor control of datapath timing

Use a pitch-matched local control section aligned on top of datapath

- Generate true/complement pairs, buffered-up versions, and clock-qualification here.



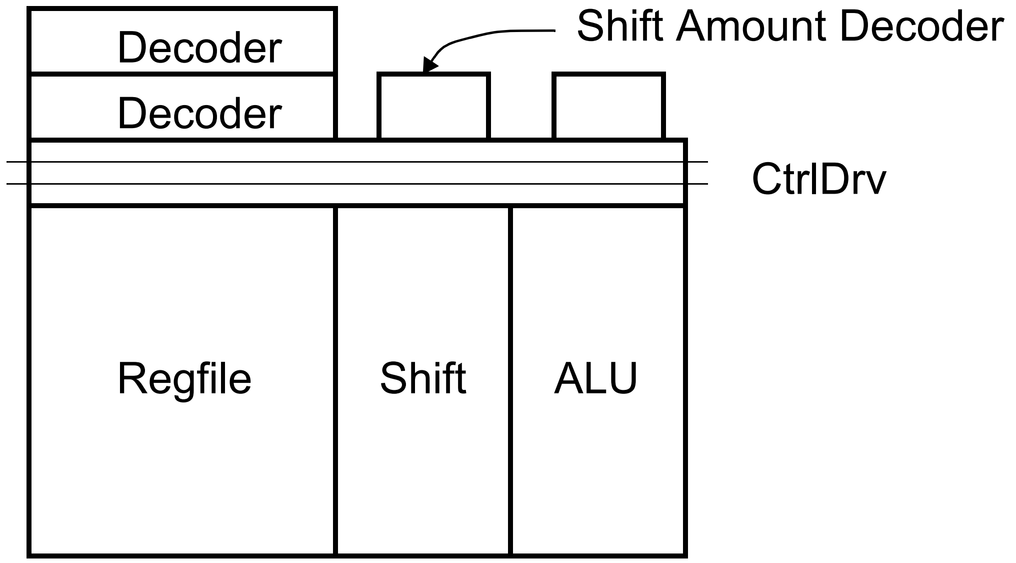
Things like
register spec,
ALU op
and other things that don't effect the sequencer.

Local Control

Run a track on top of the datapath for control drivers for the data path.

- Create true and complement control when needed.
- Do buffering and clock qualification (when needed)

On top of this track place local decoders to reduce # of control wires.


Decoder
Decoder
Shift Amount Decoder
CtrlDrv
Regfile
Shift
ALU