Lecture 8:

Synthesis, Implementation Constraints and High-Level Planning

Overview

Reading

Synopsys Verilog Guide

WE 6.3.5-6.3.6 (gate array, standard cells)

Introduction

We have been talking about how to represent a design using Verilog. Verilog is a good design description since there are a number of tools that will convert your Verilog description to logic gates. While the synthesis programs do some optimization, it really only works at the gate level. To create an efficient solution, you will need to write your verilog so it can be efficiently implemented in silicon. This means worrying about the regularity of your solution, the amount of communication needed between modules, and reusing existing designs or design styles.

Tools often have obvious limitations, things they should do but don't. We won't talk much about these issues. Rather this lecture will look at some of the fundamental constraints of VLSI, and how that affects high-level design.

Hardware Description Languages

Can use this level as the starting point of our implementation

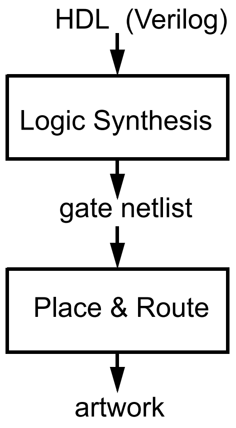
We need this level of design anyway

- Need an executable specification of our design
  - Need to check that you described what you wanted
  - Need to completely describe the operation of what you want
- Systems getting more complex
  - Designing 100K gates one at a time can be a pain
  - Making sure you have them all right is not fun either
  - Abstract away most of the details -- specify the functionality

Once you have this level of the design can use it to generate the logic level implementation using synthesis.

Logic Synthesis

Logic synthesis tools read in a verilog description and create a netlist which can be used by a placement and routing system to create a finished layout.



Translate HDL

Optimize Logic

Map to available gate library

Place cell (gate) instances

Connect them together

Optimize area /speed

Often takes replacement

Optimization

Using CAD tools helps the optimization process

- You work at higher level of abstraction
- Can quickly see the implication of a design decision
- Give feedback about what part of the design needs work
  - Many design sections might meet performance specification
  - Leave these sections alone, and work on the broken stuff

But they don't solve the problem for you. You still need to think.

- Usually the first design that is generated is not good enough
- Need to change the design to improve the performance
- Can tweak it in many different ways. Run the tools many times

Area, Delay, and Power

Area

- Sum of cell area and wire area
- Although each wire is different, can depend on average wire lengths
- Good estimate of area is K * cell area

  K ranges from 1 (datapath layout) to around 2 (standard cells)

Delay

- Set by critical path
- Delay of elements are affected by wire capacitance, which depends on wire length which is harder for the tools to estimate
- Often timing after placement is not fast enough
- *Need to iterate the design to get the right performance.*

Power

- Set by the total capacitance that is switched

Helping the Tools

In an ideal world it shouldn't matter how you write the Verilog

Optimization in the CAD tools will find the best solution

But the world is not ideal, yet (and progress is slow)

1. Tools work best on smaller problems

   Need to partition real problem into pieces for you and the tools

   Tools tend to use your module decomposition

2. Tools use your code as a starting point

   Your structure is not completely eliminated (probably good)

3. Structure of problem is often important

   Finding a "good" way to think about the problem is key

Like optimizing compliers for C, tools are good for local optimizations, but don't expect them to rewrite your code and change your algorithm

What is Important in Physical Design

Wires

- Chip is mostly wires
  - Transistor gate area is around 4% of total area
- Can be cheaper (in area and speed) to duplicate functions
  - Save on routing the wires
- Power, Gnd, and Clocks need to be wired too
  - These wires have more constraints on them

Complexity

- There are millions of transistor chips (~10^7 rectangles)
- Don't want to draw each one
- Need to partition the problem into smaller problems
- Need to find a way to reuse design and layout
  - Regularity is key

Structure is Good

Wires are bad in two ways:

- They take up a lot of area
- They have a lot of capacitance, making the part go slower

Using a solution that has regular wiring is a big win:

- Smaller and faster circuits
- Usually also reuses cells too

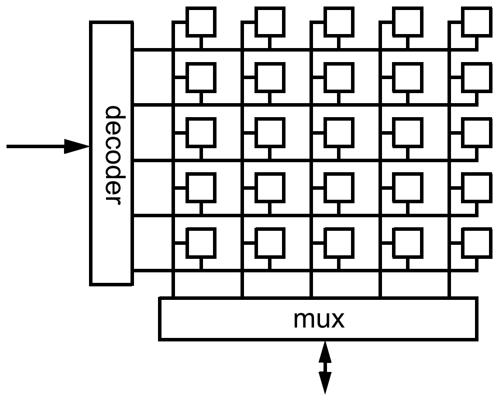
Even if the wiring is random, cell reuse is important

- Don't need to design each gate/module. Design a set and use them
- Customizing each gate is expensive (and dangerous[1])

1. You need to make sure that each cell you create works.

Best Case for Structured Wiring

Best example is a memory array:



It has N^2 elements and only 2N wires. It is an easy way to use 1M transistors. The layout is quite dense. Most of the transistors in the million transistor chips are in memory arrays

Arrays

Best target for silicon implementations:

- Very simple wiring

  Structured in both the X and Y directions

  Control in one direction, data in the other
- Large reuse of cells

  Array cell used N^2 times

  I/O circuits are used roughly N times
- Easy to test

  Access to all the cells to figure out if they work

True of most arrays, not just memory arrays.

Arrays Uses

It is mostly for memory:

- RAMs
- Registerfiles
- ROM
- Structured logic (PLA)

Regular Computation:

- SIMD machines
- Multiplication

But not a large application space. This much structure is too restrictive. Need to have a structured style that is more flexible.

Datapath Design

Datapath design is another structured wire design style.

- Forms a one dimensional array
- Useful for operating on multi-bit data

  Design for one bit, and then replicate
- Assumption is that each bit only needs to communication with other cells in the same bit position.
- The resulting wiring is the same in all the bit positions[1]
- Control to the function unit (all the bit cells) is perpendicular to bit wires

Advantage of this style is that the wires can be embedded in the cells, and then the wires are replicated when the cells are replicated. The whole system can be constructed without needing random wiring

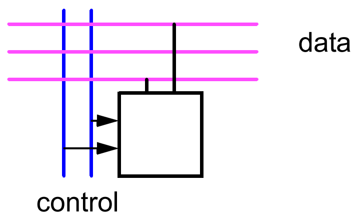
1. The cells in each bit position need not be exactly the same, but usually are. Sometimes the cells are slightly different to make the n-bit functional unit more efficient (for example in an n-bit adder), or to handle limited communication between bits.

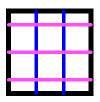
Datapath Cells

- Data flows horizontally

  Data flows between functional units
- Control flows vertically

  Control goes to all the bits in this function unit
- These wires need to be built into the cell





wires can run over cell

Datapath Uses

Datapaths are useful for implementing the dataflow portion of a design

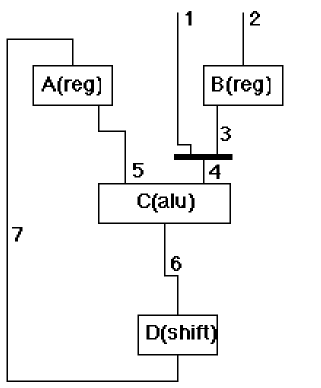
- Each functional unit can be constructed as a stack of bit cells
- Functional units can connect to each other using the bus lines

Efficient implementation style

- Wires are in the cell, so it is dense
- Layout is regular
  - Easier to estimate size, wire lengths, performance tune design

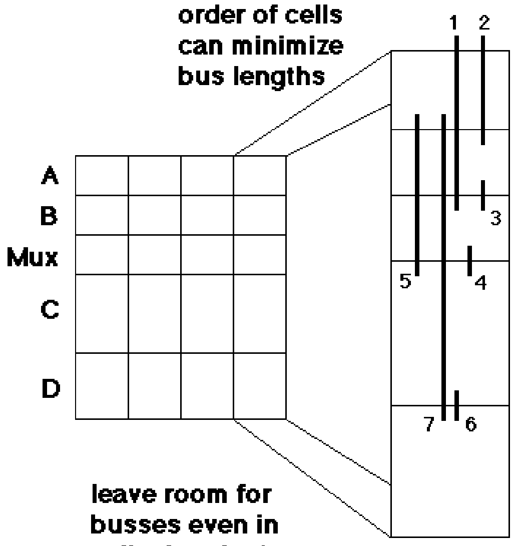
Try to convert your design to this style

Datapath Example



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

Unstructured Wiring

Sometimes there is no structure to the communication pattern

- It is just a pile of gates
- Interconnect wires will be random, but

To reduce the design effort, use a set of predefined cells

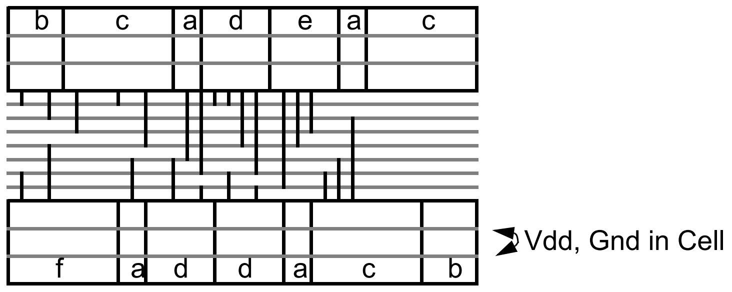
- Cells all have standard height
- Cells have power, gnd, and sometimes clock running through them

  These critical signals are connected by the cells
- Cells are pre-tested to ensure that they work

To reduce work further get a computer to place and route the cells. The general algorithm generates rows of cells. Called standard cell layout.

Standard Cells

- All cells are the same height with abutting power and gnd connections
- Cells tiled into rows
- Rows of cells are separated by routing
- Wiring area set after the routing, so the wires always fit
- Connection between rows is done by routing over the cells



Other Constraints

It is hard to think about 1M gates at one time

- Use hierarchy to logically partition the problem into smaller pieces
- Connect together these smaller pieces to get whole solution

Tools have similar problems

- Will only work on problems up to a certain size
- Handle large problems by partitioning and solving the smaller problems
- Generally use the partition that is already there
- So top-level logical and physical designs are the same
  - Top-level blocks become real physical layouts
  - Little optimization done between modules
  - Real wires are needed to connect these blocks

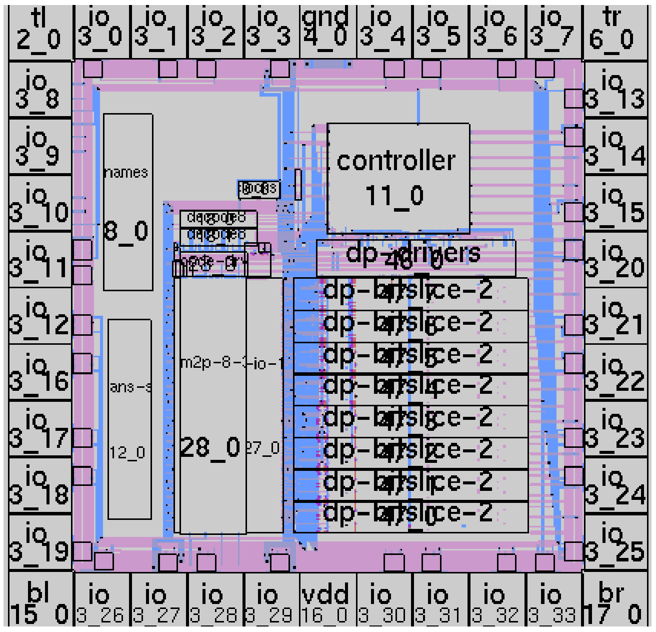
Chip-Level Layout

Macro-block place and route

At the top level all chips use roughly the same technique. There are a number of large macro-cells that are placed on the chip and then routed together. This collection of macro-cells is then surrounded by the padframe, which is used to connect the internal signals to the external world.

While it is possible for these macro-cells to be comprised of smaller macro-blocks that are placed and routed, each nesting level adds some inefficiency. So since there are tools to help in this process, all the macro cells can be brought up to the top level and placed and routed at one time. This is one of the reasons that the layout hierarchy is not deep.

Chip Layout



Testing

Two kinds of testing:

- Is the design correct
- Does the chip work

Need to create test cases (sometimes called test vectors) for both

- To test design

  Add test scaffolding to verilog model

    Generates tests (sometimes reads from file)

    Checks the answers (or make self-checking tests)

    Often check internal nodes to be sure stuff is working

- To test chip

  Need to ensure that the chip is the same as the design (no faults)

  All tests must be from the pins of the chip. No access to internal nodes

Modeling Dilemma

- Writing a model is easier if you don't worry about implementation
  - No need to worry about structure, wires, resources
- Usually first goal is to see if the idea works
  - Do I understand the problem
  - Is this function the function I want?
- Thinking about the physical structure will slow you down

What do you do?

- Depends
  - Write model twice

    Good when you have little idea what you are doing, lots of time

    Never works in industry (never have lots of time)
  - Write one model, and then fix it up

    Have a basic idea, so you take your best shot at its structure

Implementation Constraints on Verilog Description

Top-level partition should be well thought out - create chip-plan early

- Need to match physical partition
  - Signals between blocks will be real wires
  - Blocks have to fit together in the layout
- Units with different implementation methods should be separate
  - Units with regular wiring should be separated out
  - Use specific tools for these units
- Estimate long wires between blocks

Need to think about testing early on

- Have a plan on how to test design
- Write the needed testing support in the Verilog

Experience

Experience is very useful in writing models

- Provides a handle on what works
- And what blew up in your face

Hard to short-circuit this learning process

- I will give you some suggestions, warnings
- But give yourself some time to learn this stuff
- Hard to have intuition from the start

Helpful to understand synthesis process

- Understand what it does, and does not do

I hope this class will give you some experience for your NEXT design

What Goes Well in VLSI

Best way to find this out, is look at what other people have placed on chips

- Memory, memory and more memory
  - static memory, dynamic memory, registers,
  - fifos, multiported memory
  - specialized memories like content addressable memory
- Computational blocks
  - Adders, multiplier, divider, shifters, field insert/extract
  - Duplication is easy, it is good to have parallelism to exploit
- Data rearrangement
  - Crossbar, more memory
- Random Logic

VLSI Microarchitecture

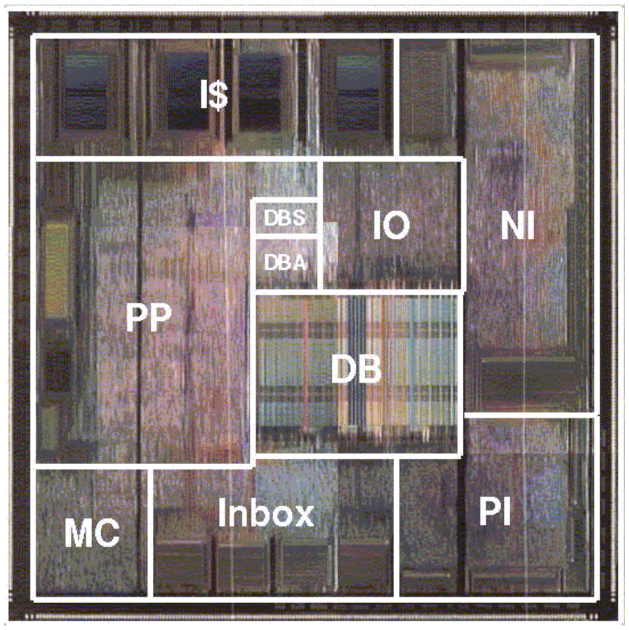
Goal is to leverage regular structures like memories

- Good method is to make a processing engine
  - Have a (many) regular computation blocks

    Denser, implementation of these blocks
  - Memory stores program and data
  - Also have less regular logic areas
- Look at Magic chip
  - Chip is the core of a multiprocessor
  - Connects processor to memory, network and I/O
  - Partitioned into 8 main blocks

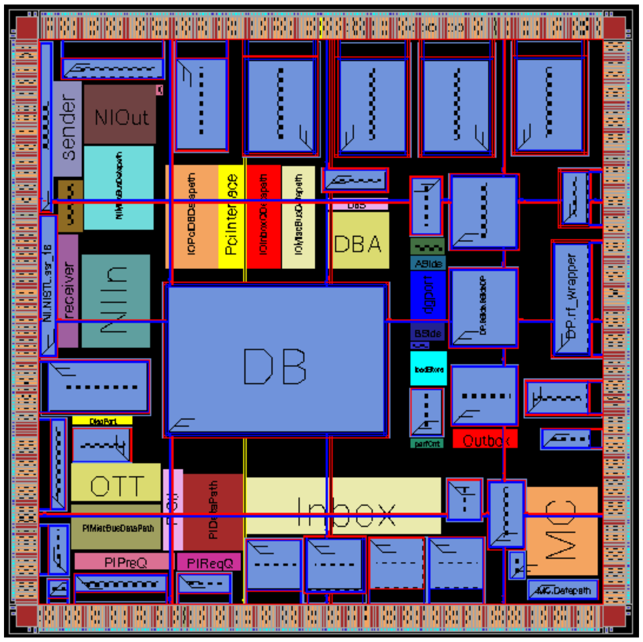
    processor interface (PI), network interface (NI), PCI IO, memory controller (MC), databuffers (DB), protocol processor (PP), instruction cache for the PP, inbox (dispatch table for PP)

Magic Chip



Magic Floorplan

- Boxes show all the high-level objects in the magic chip
- Blue boxes are mostly memory, but there are some data path elements
- More boxes on high-level to account for different implementation styles
- Datapaths were separated out from control logic



+ Improving Synthesis Results

There are many ways to change Verilog to improve synthesis

- Fall into two general classes:

  Algorithmic changes

  - Think of a better solution to the problem

  Local changes

  - Use different verilog constructs
  - Restructure your verilog to guide the synthesis
  - Give the tool information it can't figure out
- For applications where there is a clear (to you) good solution
  - Doing it yourself is always an option
  - Often for datapath stuff, it is easier to write structural code
  - Put in explicit adders, latches etc. in the Verilog

+ Structural Code Example

```
module ALU (...);
    input ...
    input ...

    not     Not(inB_b_s1, InB_s1);
    mux2    AdderInB(inB_s1, InB_s1, inB_b_s1);
    adder   ADDER(add_v1, InA_s1, InB_s1);
    or      OR(or_v1, InA_s1, InB_s1);
    and     AND(and_v1, InA_s1, InB_s1);
    xor     XOR(xor_v1, InA_s1, InB_s1);
    mux5    OutMux(out_v1, inB_b_s1, add_v1, or_v1, and_v1, xor_v1);
    latch   OutLatch(Out_s2, out_v1, Phi1);
endmodule
```

+ Algorithmic Changes

This is just cleverness in thinking about the problem

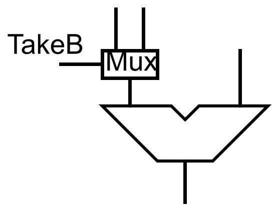
- Hard to teach or explain, since each situation is different
- Need to allow yourself to think of really crazy ideas
  - Filter them later to see if any have promise

- Solutions generally:
  - Take a global view of the design
  - Try to fix the global problem rather then the specific problem given
  - Often use ideas that have been used in a different problem domain
  - Are clearly good solutions when explained

+ Branches in Computers

In a computer, the address of the next instruction can depend on the result of a previous branch instruction. Assume that the preceding branch has a compare in it, and this signal is called *TakeBranch*. Your job is to speed up the PC adder, since it is on the critical path after the branch comparison.

```
if (TakeBranch) then
        PCBus_s2 = PCreg_s2 + Disp_s2;
else
        PCBus_s2 = PCreg_s2 +1;
```



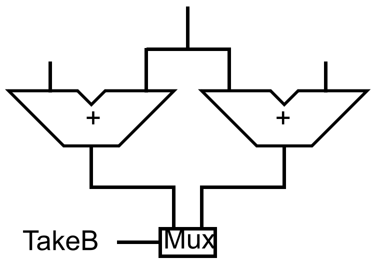
Since TakeBranch arrives late, the add must be fast

+ Faster Branches

Change the code to do the adds and compares in parallel

- Then use the comparison to do a “late select”
- Takes two adders, but is much faster

```
PCDisp_s2 = PCreg_s2 + Disp_s2;
PCNext_s2 = PCreg_s2 + 1;
if (TakeBranch) then
        PCBus_s2 = PCDisp_s2;
else
        PCBus_s2 = PCNext_s2;
```



+ Local Changes

- Watch ordering of variables and parenthesis

  How you order your computation will set the actually logic

  a + b + c + d implies 3 serial adders

  (a+b) + (c+d) implies a tree adder

  Need to make sure you choose which you want.
- This goes for more than just adders

  If you have an input signal that comes in late (slow input)

  You want to write your logic so that this signal has little logic in its path

  If *AEqualB* is late

  TakeBranch = (bunch of logic) & AEqualB;

  The optimization will generally do this for you, but sometimes the signals cross cell boundaries, and reordering the variable becomes harder to do

Synthesis

- Synthesis tools are like compilers
  - Allow the user to work at a higher level
  - Show you what the details look like (maybe)
- Use tools to understand the parts that need extra work
  - Like a profile of a program
  - Optimize the parts that don’t meet the constraints
  - Don’t improve what is not broken
- Tools leverage your creativity
  - Not a substitute for thinking
  - Might need to be more clever since your competitors are using tools on their projects too.