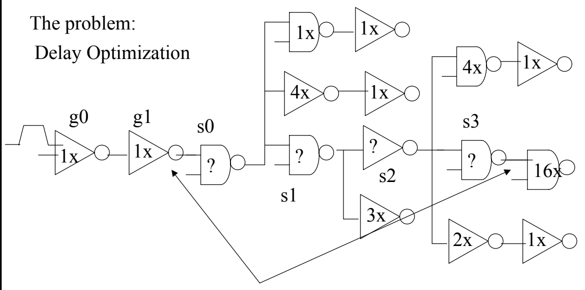
The problem:

Delay Optimization



Size the gates marked with '?' to minimize the delay between the points indicated. Any gates with known gate sizes are FIXED sizes and cannot be changed.

Methodology: Tilos

- Perl script used for Tilos method
- Sizes incremented in 0.5X steps
- Sizes started at 1.0X
  - delay measured from input of G0 to output of S3
  - initial delay through all 1.0X sized gates used for first comparison
- Question: what is acceptable lack of improvement before we move on?
  - if old path = 600 ps, and new path is 598 ps, is this worth the extra 0.5x gate size?
  - Ran two cases: +/- 1.0% difference (+/- 6 ps for 600 ps path), absolute difference (accept any improvement at all).

Tilos Absolute vs +/- 1.0% Results

Technology,		path delay (ps)	diff (ps)	inv FO4 delay (ps)	diff in FO4 delays
0.35 2/1	absolute	624			
	+/1.0 %	690	+66	97	0.68
0.35 1.4/1	absolute	635			
	+/1.0 %	701	+66	100	0.66
0.18 2/1	absolute	300			
	+/- 1.0%	326	+27	38	0.71
0.18 1.4/1	absolute	300			
	+/- 1.0%	327	+27	35	0.77

FO4 delay = inverter fanout 4 delay = 1x inverter driving 4X inv load.

Tilos Comments

- Assuming both absolute and +/1% solutions are believable, what is an acceptable degradation in path delay because of less than optimal solution?
- If we want the best performance, then tweak for the best possible solution if this is the critical path and nothing less than the fastest is acceptable.
- If we express the delay difference in FO4 delays, then greater than 1 FO4 is unacceptable
  - levels of logic are precious
  - much work expended to reduce # of FO4 delays between registers, each FO4 delay is a logic level
  - giving up one entire logic level worth of delay because of poor timing optimization is not acceptable
- Less than 1 FO4 delay and greater than 0.5 FO4 delay is a grey area
- Within 0.5 FO4 delay of best known solution is good.

Methodology: Leffort

- Leffort optimization only requires N (number of stages), logical effort values, and load values
  - perl script used to compute values
- Load values expressed in transistor widths
  - for 2/1 inverter, Load = 3; for 2/2 nand2, Load = 4
  - for 1.4/1 inverter, load = 2.4, for 1.4/2 nand2, Load = 3.4
- 'g' value for nand2 measured as discussed in notes
- Number of stages = 5
- Two iterative methods used
  - #1: Ignore branching effort for first iteration (this under-sizes gates initially, and assumes smallest possible branch effort)
  - #2: Assume gate sizes initially = 1X, compute branching effort based on this (this over-sizes gates initially, and assumes largest possible branch effort)
  - Both approaches gave same results.
- After gates were sized, rounded to nearest 0.5X size.

Tilos (absolute) vs Leffort

Technology, method leffort (g nand2)		Gate Sizes				delay (ps)	diff in FO4s
		s0	s1	s2	s3		
0.35 2/1	tilos (abs)	3.0	4.0	6.0	4.0	624	
	leffort (1.27)	1.5	1.5	3.5	5.0	699	0.77
0.35 1.4/1	tilos	3.5	4.0	6.0	5.0	635	
	leffort (1.3)	1.5	1.5	3.5	5.0	724	0.89
0.18 2/1	tilos	2.0	3.5	6.0	5.5	300	
	leffort (1.24)	1.5	1.5	3.5	5.0	328	0.74
0.18 1.4/1	tilos	3.0	4.0	6.0	5.0	300	
	leffort (1.3)	1.5	1.5	3.5	5.0	328	0.80

Tilos (+/- 1%) vs. Leffort

Technology, method leffort (g nand2)		Gate Sizes				delay (ps)	diff in FO4s
		s0	s1	s2	s3		
0.35 2/1	tilos	2.0	2.0	3.0	2.5	690	
	leffort (1.27)	1.5	1.5	3.5	5.0	699	0.09
0.35 1.4/1	tilos	2.5	2.0	3.0	2.5	701	
	leffort (1.3)	1.5	1.5	3.5	5.0	724	0.23
0.18 2/1	tilos	2.0	2.0	3.0	2.5	326	
	leffort (1.24)	1.5	1.5	3.5	5.0	328	~0
0.18 1.4/1	tilos	2.0	2.0	3.0	2.5	327	
	leffort (1.3)	1.5	1.5	3.5	5.0	328	~0

Comments

- Tilos required about 30 iterations for absolute solution, about 15 iterations for +/1% path difference solution, significant simulation time
  - +/- 1% solution did not use any gate sizes > 3.5 -- this is not good, why have the larger gates?
- Leffort required approximately 5 iterations, negligible computation time
- Tilos results were consistently better than Leffort (approx 10%) at absolute solution, about the same for +/1% path difference solution
  - Tilos always sized stage S3 (inverter) larger than S4
  - Leffort always sized stage S3 (inverter) smaller than S4

Improving Leffort?

- The main problem with Leffort seemed be that it missed sizing S3 larger than S4
  - Either under-estimating load seen by S3 or over-estimating drive capability of S3
- Should not be under-estimating load since load calculation is consistent between stages
- Must be over-estimating drive capability of S3 – Why?
- Leffort is a linear model applied to a non-linear process
  - Electrical effort (Cout/Cin) is supposed to be proportional to the delay associated with an external load – this is a linear factor
  - However, diminishing returns on delay improvement as you increase a gate size
  - This should be reflected in the Leffort model in someway – perhaps a non-linear scaling factor for electrical effort based on load size.

Combining Leffort with Tilos

- Use final gate sizes of Leffort as initial gate sizes of Tilos instead of using minimum gate sizes as starting part
- For 0.35u, inv ratio = 2/1, and using absolute differences with Tilos, the result was basically the same as starting with minimum sized gates, but only took 15 iterations instead of 30 iterations!
  - Similar results with other cases
  - Reduced Tilos run time by 50%

See detailed numbers/calculations from a student solution at the end of this presentation.

A Plea --- Learn Perl or Something Similar!!

- I was able to do all four cases and investigate other issues (such as absolute vs +/- 1% in Tilos) because I used Perl.
- Occasionally, an Engineer will have to write a program!
  - Not all problems can be solved by spreadsheets
- What types of problems might require programming?
  - Generate complex data streams as input to another problem
  - Parse/collect information out of large data files
  - Write a program that runs other programs in a regression test
  - Convert data files from one format to another
- Many programs are throw away code – use once to complete a task, then forget about it.
- Usually an Engineer is under time pressure
  - Need to become very familiar with a 'favorite' programming language, and use it enough to become time efficient.

How did I use Perl?

Had a file called *run_params.sp* that defined the gate sizes for each stage, the model file ('tscm_0_35.model'), and the inverter ratio. This file was included by my main spice file (*tilos.sp*).

The Perl script had internal variables for gate sizes – the basic iteration loop in the script was:

1. Generate the file "run_params.sp".
2. Run hspice
3. Read the delay from the hspice output file.
4. Calculate new gate sizes based on delay, goto '1' and continue until tilos algorithm complete.

Also wrote Perl script for Leffort as well – calculated sizes based on leffort model, then ran Hspice at end to get delay with Leffort sizes.

Some Assumptions

- Programming is not the main task in your engineering workday
- Most programs you write are small, throw away programs
  - less than 100 lines
  - Use a program one or twice, then forget about it
- Computer environment is either Unix-based or Windows based
- Many work environments use both Windows and Unix
  - Windows for productivity tools (Spreadsheets, Word processing, Powerpoint)
  - UNIX workstations/servers for compute intensive jobs

Desirable Features of a Programming Language for 'throw-away' code

- Powerful – get a lot done with a little code
- Flexible – be able to do many different types of tasks
  - GUIs, string processing, program control, number crunching, etc.
- Well documented
- Large user base so external libraries, examples readily available
- Portable – be able to run on different systems
  - carry your favorite code with you when you change jobs
- Readily available ("free" is the best!)

Compiled Programming Languages

- C, C++, Fortran are traditional compiled languages
- Pros
  - High performance code
  - Portable between systems
  - Free C, C++, Fortran compilers from the Free Software Foundation
- Cons
  - Usually have to write a lot of code to get even simple tasks done
  - Non-standard extension libraries which means you have to move your favorite library from system to system
  - Must compile source code first on target system before execution.
  - GUI interfaces are Operating System dependent
  - Support for 'scripting' (i.e, controlled execution of other programs) is minimal and Operating System dependent
- Best for large, complex tasks -- but may not be best choice for simple tasks (i.e, < 100 lines of code)

UNIX Shell Scripting Languages

- All UNIX shells (csh, bash, ksh, etc) support a scripting language
- Pros
  - Builtin to shell, always available for use
  - Ideal for scripting duties (control of other programs)
- Cons
  - Fairly primitive features, no powerful operators or data handling features
  - Very slow – at least 100x slower than compiled code
  - No GUI capabilities
  - Only useful for Unix applications

Visual Basic

- Visual Basic is the most common scripting/tool extension language on Windows platforms
- Pros
  - Integrated with Windows productivity tools (spreadsheets, etc) – can be used to extend their base capabilities
  - Very nice GUI building capabilities
  - Complex data types, powerful library functions
  - Has scripting capabilities
  - Decent performance (about 10x less than compiled C/C++)
- Cons
  - only works under Windows OS
  - development environment costs $$$
- Might be the best choice for throw-away code if you never touch a Unix system

Java

- Portable object-oriented programming language
- Pros
  - Powerful data structures, functions
  - Portable between Unix/Windows
  - Free development environment
  - Powerful GUI building
  - Useful for Web page enhancement
  - Decent performance (about 10x less than compiled C/C++)
- Cons
  - Limited scripting, string processing
  - Object-oriented programming model is possible overkill for simple throw-away programs

Perl

- Scripting language that combines the best of Unix shell languages plus powerful string handling and builtin functions
- Pros
  - Powerful data structures, functions
  - Portable between Unix/Windows
  - Free development environment
  - Decent performance (about 10x less than compiled C/C++)
  - Many public libraries available for tasks such as HTML processing and data base access
  - Extremely large user base – the scripting language of choice under Unix
- Cons
  - No GUI building capabilities
  - Code may be unreadable after you write it!
- In my opinion, best choice for throw away code development under Unix environment, and perhaps a combined Windows/Unix environment.

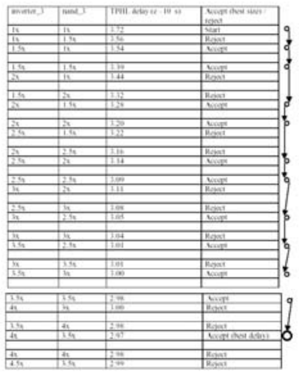
Others

- Tcl/Tk
  - Portable scripting (TCL) + GUI development (TK) environment
  - A better choice than Perl if you need both sophisticated scripting and GUI development in the same programming language
  - Free implementations for both Windows and Unix
- Python
  - Best described as an object oriented scripting language
  - Has powerful GUI building features
  - Free implementations for both Windows and Unix
  - Better choice than Perl if you need sophisticated scripting + GUI development, and you like the object-oriented programming model

Where to learn more

- http://www.activestate.com
  - Free implementations of Perl, TCL/TK, Python for Win32
- Java – http://java.sun.com
- Visual Basic
  - Limited form of VB comes with Excel
  - Start Excel, go to Tools→Macro →Visual Basic Editor
  - Access help function under Visual Basic Editor
  - Full version requires purchase of Visual Basic Studio (http://msdn.microsoft.com/vbasic/default.asp)
- UWIN comes with Gnu compilers for C, C++, Fortran

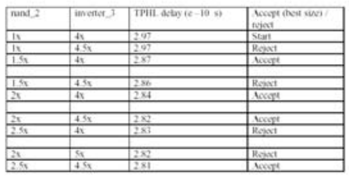
Tilos Detailed Steps



inverter_3	nand_3	TPHL delay (e -10 s)	Accept (best size) / reject
1x	1x	3.72	Start
1x	1.5x	3.56	Reject
1.5x	1x	3.54	Accept
1.5x	1.5x	3.39	Accept
2x	1x	3.44	Reject
1.5x	2x	3.32	Reject
2x	1.5x	3.28	Accept
2x	2x	3.20	Accept
2.5x	1.5x	3.22	Reject
2x	2.5x	3.16	Reject
2.5x	2x	3.14	Accept
2.5x	2.5x	3.09	Accept
3x	2x	3.11	Reject
2.5x	3x	3.08	Reject
3x	2.5x	3.05	Accept
3x	3x	3.04	Reject
3.5x	2.5x	3.01	Accept
3x	3.5x	3.01	Reject
3.5x	3x	3.00	Accept

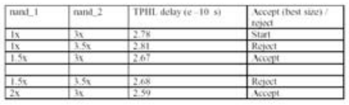
3.5x	3.5x	2.98	Accept
4x	3x	3.00	Reject
3.5x	4x	2.98	Reject
4x	3.5x	2.97	Accept (best delay)
4x	4x	2.98	Reject
4.5x	3.5x	2.99	Reject

Sizing of last 2 gates in path.



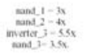
nand_2	inverter_3	TPHL delay (e -10 s)	Accept (best size) / reject
1x	4x	2.97	Start
1x	4.5x	2.97	Reject
1.5x	4x	2.87	Accept
1.5x	4.5x	2.86	Reject
2x	4x	2.84	Accept
2x	4.5x	2.82	Accept
2.5x	4x	2.83	Reject
2x	5x	2.82	Reject
2.5x	4.5x	2.81	Accept

Next step, notice that inverter_3 started where it left off in previous step.



nand_1	nand_2	TPHL delay (e -10 s)	Accept (best size) / reject
1x	3x	2.78	Start
1x	3.5x	2.81	Reject
1.5x	3x	2.67	Accept
1.5x	3.5x	2.68	Reject
2x	3x	2.59	Accept

Final sizing



nand_1 = 3x
nand_2 = 4x
inverter_3 = 5.5x
nand_3= 3.5x

Final Gate sizes