Exploring the Use of Sub-Threshold Transistors Using Genetic Algorithms

David Lewis, Dr. Matthew R. Guthaus
2007 UCSC SURF-IT
NSF Grant #CCF-0552699

Current circuit design techniques are constrained by physical properties. Component size, trace width, and supply voltage all limit layout possibilities. This project will examine how varying supply voltage can affect layout design while still achieving the desired output. Because of the number of layouts possible for a given number of transistors, it is necessary to select an efficient method of evaluating a circuit layout. For this project, Dr. Guthaus choose to implement genetic algorithms as a key component of the evaluation of circuit layouts.

Genetic algorithm implementations have distinct steps that must be in. A population of ***n*** binary strings is randomly generated. These binary strings can be referred to as chromosomes, individuals, or genotypes. Each individual is evaluated with respect to a particular specification, which is usually the function to be optimized. A measure of fitness, or survivability is then assigned to each genotype. Based on this fitness score, the population undergoes the process of selection. A set of ***n*** individuals is then chosen in such way that a higher fitness score confers higher chances of selection to an individual. This way, the selection process is biased towards fitter individuals. After selection, pairs of individuals are randomly chosen from the selected pool, and with a particular probability, undergo the recombination process, also called crossover. The crossover operation splices the contents of each pair of individuals, creating two offspring through mixing the genetic contents present in their parents' strings. In case the two parents do not undergo the crossover operation, they are copied unchanged to the new pool. The mutation operator is then applied to the new pool of individuals produced after the application of crossover. Mutation is typically applied with a very low rate to all bit positions that constitute a given individual. If successful, the value of the particular bit is negated. Otherwise, the bit is left unchanged. After mutation, the production of a new generation of individuals is complete. This new generation then goes through the process as described, from evaluation to mutation. This cycle repeats until a stop criterion is met, such as when a maximum number of generations is reached or a desired solution is found.

There are two distinct types of transistors, termed N-type and P-type. The type of substrate that the transistor is built on classifies these. Transistors' minimum required operating voltage is currently 1.5v-1.8v depending on manufacturing and reliability guidelines. Input voltages below these values provide unreliable outputs. Transistors operating below 1.5v have produced analog-like outputs. It is unknown what sort of circuits could be developed to take advantage of this analog-like behavior. This is the target of our research.

The use of simulation in circuit design is well established. Using simulation, it is possible to test circuits for proper behavior before taking the costly steps of manufacture. When exploring new techniques, simulation allows multiple variables to be assessed in a reasonable manner. The simulator that is interfaced with accepts a circuit definition file such as this NAND gate.

```
* first line must always be a comment or blank since it is ignored
simulator lang=spice lookup=spice

* include the transistor models
.lib "tsmc18dN" NMOS
.lib "tsmc18dP" PMOS
.usim_opt mos_method=s postl=1

* gnd! is always 0
.global 0 vdd! gnd!

* set the supply voltage
VCC vdd! gnd! dc 1.8

* list of transistors and interconnects
M1 (net6 a gnd! gnd!) tsmc18dN w=270.0n l=180.0n
M0 (z b net6 gnd!) tsmc18dN w=270.0n l=180.0n
M3 (z b vdd! vdd!) tsmc18dP w=270.0n l=180.0n
M2 (z a vdd! vdd!) tsmc18dP w=270.0n l=180.0n

* put an output load on the circuit
C0 z gnd! 100f
```

```
* define the input stimulii for the above circuit
VINA  a  gnd!   PULSE(0 1.8 0 5n 5n   10n   50n)
VINB  b  gnd!   PULSE(0 1.8 0 5n 5n   20n  100n)

* save the output to a text file
.print V(z) V(a) V(b)

* specify to run a transient simulation for 700ns with 250ps resolution
.tran 250p 700n

.end
```

The list of transistors and interconnects is the information needed for the genetic algorithm. Each time that an individual is tested for fitness, a circuit definition file, as above, is generated for the simulator. This is the Ruby code that was written to extract the circuit layout from a circuit definition file.

```
def parse_genome( lines )
 net_list = Array.new
 nodes = Array.new
 node_lookup = Array.new
 genome = String.new
 #get transistor network description
 lines.each { |w| net_list << w.scan( /\w\d+\s+\(\w+\!?\s+\w+\!?\s+\w+\!?\s+\w+\!?/ ) }
 #get list of nodes in the transistor network
 node_line = net_list.uniq
 node_line.each do |nodes|
   nodes.each do |node_string|
     nodes << node_string[4..-1].split
   end
 end
 #generate node lookup array
 nodes.each do |connections|
   connections.each do |sub_node|
     if( node_lookup.count_of( sub_node ) == 0 ) then
       node_lookup << sub_node
     end
   end
 end
 #generate binary string based on node_lookup index
 nodes.each do |connections|
   connections.each do |sub_node|
     genome += sprintf( "%b", node_lookup.index( sub_node ) )
   end
 end
 genome
end
```

Once the binary string is produced, the genetic algorithm uses it to produce the population of individuals. As each individual is tested for fitness, a circuit definition file is created with the new interconnect layout from the individual's converted bit string. The circuit definition file is submitted to the simulator for processing. When the simulator is finished, an output file is created showing the performance of the circuit. The genetic algorithm reads this output file to determine if the circuit produced output that falls within parameters. If the output is good, then the originating binary string is added to the evolution pool for the next generation. Otherwise the bit string is discarded. This is the Ruby code that was written to extract the raw data from the simulator output file.

```
def read_output_file
  output = Array.new
  return_array = Array.new
  lines = read_file( get_file_name + OUT )
  lines.each { |w| output << w.scan( /-?\d+[.]\d+[ayzfpnum]?/ ) }
  output.each { |out_line|
    temp = Array.new
    out_line.each { |column|
      last_char = column[-1]
      lp = column.to_f
      multiplier = case last_char
        when 121 then 1.0e-24
        when 122 then 1.0e-21
        when 97  then 1.0e-18
        when 102 then 1.0e-15
        when 112 then 1.0e-12
        when 110 then 1.0e-9
        when 117 then 1.0e-6
        when 109 then 1.0e-3
        else 1
      end
      temp << lp * multiplier
    }
    return_array << temp
  }
  return_array
end
```

Work is currently ongoing in this area. Items to be addressed include:

- Generation of circuit definition file
- Determination of simulator output pass/fail
- Support for Sun Grid Engine

Some possible implementations of this research include super-low voltage processors, new novel designs of standard circuit models, and eventually, a way to expand on the binary nature of computers. As work on this project will be continuing, expect to see more information in the future.