Transistor re-ordering algorithm.

Step 1:

1. Compute delay in each gate.
2. Backward compute slack based on clock cycle.
3. If negative slack, then find optimal transistor order.
   Reorder it.

The latest arrival signal connected to the input with smallest delay.

4. Compute forward to update the delay.
5. Repeat 2 → 4 until no negative slack.

Step 2:

1. Backward trace.
2. Trade positive slack for less power consumption by transistor re-ordering.
3. Compute slack, make sure no non slack.
Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>delay reduction</th>
<th>power reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>U92</td>
<td>9.5%</td>
<td>7.3%</td>
</tr>
<tr>
<td>rds4</td>
<td>6.9%</td>
<td>6.5%</td>
</tr>
<tr>
<td>alu4</td>
<td>8.2%</td>
<td>4.3%</td>
</tr>
</tbody>
</table>

Slack calculation

\[ t = 9 \]

\[ \text{delay} = 4 \quad \text{slack} = 0 \]

\[ \text{delay} = 3 \quad \text{slack} = 3 \]

\[ \text{delay} = 3 \quad t = 3 \]

\[ \text{delay} = 2 \quad t = 2 \]

\[ \text{delay} = 3 \quad t = 6 \]
• After device sizing, the next target is interconnect.

• In deep submicron design, the interconnect capacitance dominates the gate capacitance.
  • Interconnect design is crucial.

• Discuss the interconnect issue on circuit partitioning, node clustering, placement, and routing.

• Discuss wiresizing for delay and power optimization.

Circuit partitioning:

Smaller and manageable blocks.

Complex & large design:

Objective: minimize the number of critical nets across cuts or simply the cut size.
For low-power design, average switching frequency of the nets are important.

Example:

Both partitions are balanced (contains equal # of nodes).

(b) has smaller capacitive power dissipation

lower switching activity across the two blocks.

1. First, estimate switching activity for each net.
2. Then, apply a partitioning method with the objective of minimizing switching capacities.
3. Partitioning will be further discussed in design automation of VLSI.
- Map the gates in a circuit onto positions on a layout surface.

- Conventional method: try to minimize the total net length of the final placement, meeting various delay constraints.

- For low-power placement: try to minimize the total switching capacitance, instead of total wirelength (capacity), subject to timing constraints.

Objective function:

\[ L_2 = \sum_{\text{net } N} \sum_{\text{cells } i, j \in N} \left\{ (x_i - x_j)^2 + (y_i - y_j)^2 \right\} \]

\[ \uparrow \]

Quadratic net length model.
\[ f_N \left[ (x_i-x_j)^2 + (y_i-y_j)^2 \right] + f_N \left[ (x_i-x_k)^2 + (y_i-y_k)^2 \right] \]

- The problem can be even more complex when timing delay is added as a constraint.

- Either the lumped capacitance model, the pathlength delay model or the Elmore delay model can be used to formulate the delay model.

- An average of 10% power reduction obtained when compared to the minimum net length solution.

\textit{Routing}.

- Routing completes the electrical connection between cells.

- Contains global routing and detailed routing.
Global Routing: Assign each net to a set of routing regions

Objective: Minimize the total wirelength.

Detailed Routing: Determine the actual wiring geometries and layer assignments within each region.

Objective: Minimize the channel height and trace wirelength.

Low-power routing: Minimize the total effective switching capacitance of all nets while meeting the delay constraints.

Just concentrate on global routing.
To minimize the total effective switching capacitance, we must route high switching activity nets with minimum wirelength, and allow detour for low activity nets to avoid channel congestion.

- Should avoid detour for critical nets.

- It was reported that only marginal improvement in power dissipation is observed for global routing.

- In deep submicron design, power dissipation due to coupling effect (cross talk) has become more significant.

- Try to find proper spacing and wire segment assignment/ordering

______________________
↑ Keep wide spacing for high activity nets.
↓ and assign them to different layers.
- partitioning, placement and routing mainly concentrate on interconnect length and switching activity.

- There is room for optimization for a fixed-length interconnect by sizing.

- Very important for deep submicron technology, since the delay of interconnect dominates.

- Definitions:

  - Rooting Tree: \( T \)
  - Sink (\( T \)) : The set of sinks in \( T \) (A, B, C)
  - \( W \): the wire sizing solution (wire width assignment for each segment of \( T \)).
$t_i(W)$: the Elmore delay from the source to sink $S_i$ under $W$.

Objective function for wiresizing:

$$t(W) = \sum_{S_i \in \text{Sink}(T)} \lambda_i t_i(W)$$

$\lambda_i$: measures the criticality of sink $S_i$.

E.g. large $\lambda_i$ for timing critical sinks.

Tradeoff between routing area (also interconnect capacitance and power dissipation) and performance.

$$\alpha \cdot \text{Area}(W) + \beta \cdot t(W)$$

$\alpha, \beta$ are weights to indicate the relative importance of area and performance.

- Consider either $t(W)$ or $\alpha \cdot \text{Area}(W) + \beta \cdot t(W)$, there are three important properties for wiresizing:
  - Monotone,
  - Separability,
  - and dominance
There exists an optimal wiring solution which is:

Monotone: Given any pair of wire segments $E$ and $E'$ such that $E'$ is the downstream of $E$, the width of $E$ is no smaller than that of $E'$.

\[ E \quad E' \]

A very natural result since $E$ drives more devices, so should have larger width, in general.

Separability: If the width assignment of a path $P$ originated from the source is given, the optimal width assignment for each tree branching off $P$ can be carried out independently.

\[ T_1, T_2, T_3, T_{m-1}, T_m \]
• In the above figure, based on the separability property, the optimal width assignment for \( T_1, \ldots, T_m \) can be computed independently, if the width assignment of \( p \) is given.

• This property is extremely important in simplifying the wire sizing problem.

Ref. Given two wire width assignments \( W \) and \( W' \), \( W \) dominates \( W' \) if for any segment \( E \), the width assignment of \( E \) in \( W \) is greater than or equal to that of \( E \) in \( W' \).

Ref. Given a routing tree \( T \), a wire width assignment \( W \) on \( T \), and any particular segment \( E \) of \( T \), a local refinement of \( W \) with respect to \( E \) is the operation of optimizing the width of \( E \) while keeping the wire width assignment of other segments in \( W \) unchanged.
Then: let \( w^* \) be an optimal width assignment. If a width assignment \( W \) dominates \( w^* \), then any local refinement of \( W \) still dominates \( w^* \). Similarly, if a width assignment \( W \) is dominated by \( w^* \), then any local refinement of \( W \) is dominated by \( w^* \).

\[
\begin{array}{c}
\text{\( w^* \)} \\
\text{\( \downarrow \)} \\
\text{\( W \)}
\end{array}
\]

Start from max width.

Can be used to find the upper bound of the optimal wire width.

\[
\begin{array}{c}
\text{\( w^* \)} \\
\text{\( \downarrow \)} \\
\end{array}
\]

Start from min width.

Local refinement here is still dominated by \( w^* \).

It can be used to find the lower bound of the optimal wire width.
Based on monotone and separability properties:

**Optimal Wiring Algorithm:**

![Diagram showing tree structure with sub-trees and labels](image)

**EC1** **EC2**

How to determine the width of \( E \)?

Possible widths: \( w_1, w_2, \ldots, w_r \).

For each possible width assignment \( w_k \) of \( E \) (1 \( \leq k \leq r \)), we determine the optimal assignment for each subtree \( EC_1, \ldots, EC_r \) independently by recursively applying the same procedure to each \( EC_i \) with \( \{w_1, w_2, \ldots, w_r\} \) as the set of possible widths. The optimal assignment for \( E \) is the one which gives the smallest total delay.

(Use of E|more delay model)
A possible optimal wiring layout.

3 wires

H.M.S.
Without this technique, designers generally use conservative design: End up using oversized, uniform-width wires.

- Waste too much area & power, causes longer signal delay.

- Optimal wiring can be used to "taper" wires for area and power dissipation without performance degradation.

[Diagram showing tapering of wires]

- A computer simulation result demonstrates:
  - Area reduction: 10% compared with manually tapered design, reduced by
  - Clock power reduced by 6%.
  - Average delay is delay is reduced by 3%.
Software Design for Low Power

- Major emphasis:
  1. Minimize memory accesses
  2. Optimize selection and sequencing of machine instructions
  3. Exploit the low power features of some processors

- Sources of software power dissipation:
  - Control logic
  - Clock distribution
  - Memory system
  - System busses
  - Data path

Each execution of instruction involves:

- Important especially if operational units are active all the time (e.g., pipeline or superscalar)
- **Architectural-level power estimation.**

  ![Cpu with controller, datapath, and register](image)

  **Model:**
  1. Power dissipation at each major component
  2. Which components will be activated by the program.

  Less precise, but much faster than gate-level estimation.

- **Bus switching activity**

  - Use bus activity to represent the overall switching activity in a processor.

  Requires knowledge of:
  * Bus architecture, *op-codes* for the instruction set,
  * Representative input data to a program,
  * Program code and data mapping to *RAM*,
  * Simulation fir switching at different buses.

- **Instruction-level power analysis.**
- Use an empirical method for power characterization of short instruction sequences.
- Then, use the results to estimate the power dissipation of a program.

**Basic idea**

![Diagram](image)

- Choice of instruction sequences for characterization is critical.
- Try to estimate the **base cost** of each individual instruction.

- **Base cost**: the portion of power that is independent of the prior state of the processor.

  - Avoid estimating pipeline stalls, cache misses, bus switching... due to consecutive instructions...
- How?
  - Current meter
  - CPU
    - Loop head
    - Add
    - Add
    - Loop tail
  - Give several instances of the instruction and execute an infinite loop.

- The loop should be as long as possible to minimize estimation error due to loop overhead.
- The loop cannot be too long. Avoid cache misses.

- Power
  - Energy: Power supply voltage \times \text{average current draw}

- The effect of prior processor state should also be considered.
  - Examples: pipeline stall, buffer stall, cache misses.
  - Called Circuit State Effect.

\[
\begin{align*}
\text{ADD} & \quad A \leftarrow B, C \\
\text{MULT} & \quad C \leftarrow A, B
\end{align*}
\]
How to measure circuit state effect?

Diagram:

- Current meter
- Loop
- ADD
- MUT
- ADD
- MUT

Problem: We are measuring the effect of ADD to MUT. But, it is impossible to separate

\[
\text{ADD} \rightarrow \text{MUT} \\
\text{MUT} \rightarrow \text{ADD}
\]

Instruction-level power estimate

\[
E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (C_{i,j} \times N_{i,j}) + \sum_k E_k
\]

- Overall energy cost of a program
- Base cost for instructions of type i
- Circuit state effect of type i instruction followed by type j

Example:

- DLOAD \( A \leftarrow x, B \leftarrow y \)
- LOAD \( C \leftarrow z \), MULT \( D \leftarrow A, B \)
- ADD \( A \leftarrow C, D \)
<table>
<thead>
<tr>
<th></th>
<th>Base Cost</th>
<th>Circuit State Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLoad</td>
<td>2.37</td>
<td>1.19</td>
</tr>
<tr>
<td>Load, MULT</td>
<td>2.25</td>
<td>1.06</td>
</tr>
<tr>
<td>ADD</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 1 of pp. 439

Each instruction requires 25 ns.

\[
\text{Total Energy} = 8.65 \text{ PJ}.
\]

\[
\text{Average power} = 8.65 \text{ PJ/25 ns} = 118 \text{ MW}.
\]

- Automatic analysis process

1. Basic blocks are extracted from a program.
2. Power for each block is totalled (Base + Circuit state effect).
3. The frequency and cost of stalls for each block are estimated and added into the block.
4. The program profiler determines the # of time each block executes to accumulate total cost.
A cache simulator is used to estimate the frequency of cache misses, and add the power to the total program power.

- Software power optimizations

- By Instruction selection & ordering

- Cache performance
  - Large code and small cache $\rightarrow$ frequent cache misses
  - $\rightarrow$ high power penalty
  - Should maximize code density for embedded processors

- Instruction packing
  - Example: Fujitsu DSP

```
+-----------------+-----------------+
| ALU             | Memory          |
| instruction     | data transfer   |
```

* 50% of energy reduced if instruction fetch overhead is not duplicated
This method is also used in VLIW & Superscalar architectures.

Instruction ordering

![Diagram showing instruction ordering and circuit state effect]

Reorder instructions to minimize the circuit state power.

*This technique is more significant for DSPs than general-purpose architectures.*

Operand swapping

![Diagram showing operand swapping]

Latch

ALU

Bus

\[ x + 7 \]

\[ y + 7 \]

"7" should be placed in the latch.
Another example: 

```
multiplier
```

```
 multiplicand
```

```
00 -- 01 1
```

* Only very limited # of add & shift operations are required.

* Reorder small weight # to the second operand (multiplier)

```
    multiplicand
```

```
    Booth multiplier
```

```
                multiplicand
```

```
    x
```

```
    00000
```

* Different bit pattern determines the # of additions & subtractions

* The # of additions & subtractions for each bit
  is called recoding weight

* Place the value with low recoding weight to the 2nd position.
Minimizing Memory Access Cost

- Memory is both a power and performance bottleneck.
  - It is slow & power hungry.

- Try to
  - Minimize the # of memory accesses required by an algorithm.
  - Make memory accesses as close to the processor as possible.
    register → cache → RAM
  - Minimize the total memory required by an algorithm
  - Make the most efficient use of available memory bandwidth.
    e.g., try to use multiple word, parallel loads.

Example:

For $i = 1$ to $N$ do

$$B[i] = f(A[i]);$$

for $i = 1$ to $N$ do

$$C[i] = g(B[i]);$$

If $B$ array is large,

register $\rightarrow$ memory transfer

$2N$ memory wasted per $i$, $N$ transfers

For $i = 1$ to $N$ do

begin

$$B[i] = f(A[i]);$$

$$C[i] = g(B[i]);$$

end

Compiler can use a register to store $B[i]$, $N$ memory transfers only.
Exploiting Low Power Features of Hardware.

- Software control over power management (instead of gated clock).
- Software designers can use instructions to power-down some components.

* Example:

  Standby mode: CPU core is stopped, but other operations are maintained.
  Sleep mode: All operations except the real-time clock stops.

- Advantage:
  - Software designer and compiler can better determine when to remove or slow down a clock.
  - Pure hardware power down may make incorrect decision (based on processor activity) → hurt performance & power, system restoring cost.

- Disadvantage:
  - Needs program execution cycles, more costly than hardware-controlled power-down.