Digital System Synthesis

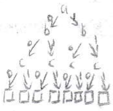
Boolean satisfiability (SAT)

Find an assignment to the variables $x_1, \dots, x_n$ such that the Boolean function $f(x_1, \dots, x_n) = 1$, or prove that no such assignment exists

Ordered Binary Decision Digrams (OBDDs)

OBDDs provide a representation of a Boolean function in the form of a binary tree. In contrast to truth tables or algebric representation OBDDs provide an efficient means of recursive analysis that can tackle the solution of complex SAT problems.

In high-level synthesis OBDDs are applied in:
- Digital circuit design and verification
- Sensitivity analysis of digital circuits
- Finite-state system analysis.



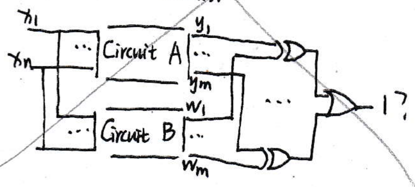
[Uses of OBDDs and SAT]

EDA: { Testing and Verification, Logic Synthesis, FPGA routing, Path delay analysis

AI: { Knowledge base deduction, Automatic theorem problem

[Example: SAT → solve the logic equivalence problem]

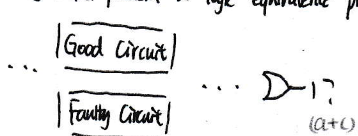
Combinational circuit C_A, with n inputs and m outputs
Combinational circuit C_B, with n inputs and m outputs
Are the two circuit equivalent?
- A complicated problem formulation:
  Are the outputs equivalent for all input values?
- A simpler alternative:
  Are there (at least one) input values that distinguish outputs of two circuits?



[SAT → ATPG Problem in Testing]

ATPG: Automatic Test Pattern Generation
Digital integrated circuits can exhibit defects
Physical defects are modeled as logical faults
Most often used fault mode: single stuck-at fault model
- Circuit lines stuck-at a fixed logic value
  - Sa-0: fixed at 0
  - Sa-1: fixed at 1

Reducing ATPG problem to logic equivalence problem



$(a+c)(\bar{a}+c) = a\bar{a}+ac+\bar{a}c+c = c$

[Conjunctive normal form (CNF)] 合取范式

A Boolean formula is in CNF if it is a conjunction of clauses [ ], where a clause is a disjunction of literals. $F = \bar{a}b\bar{c} + ab\bar{c} = (\bar{a}+a)(\bar{a}+b)(\bar{a}+\bar{c})$
$(b+a)(b+b)(b+\bar{c})$
$(\bar{c}+a)(\bar{c}+b)(\bar{c}+\bar{c})$
$= b(\bar{a}+\bar{c})(a+c)$ (literal, clause)

In other words, CNF is an AND of ORs

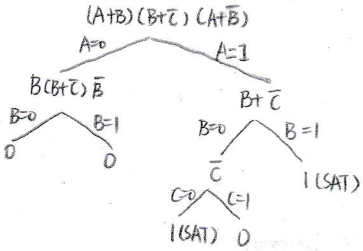
Davis-Putnam (DP) SAT algorithm

Recursive Algorithm that creates a binary search tree by making assignments to the remaining variables at each stage of the algorithm.

[DP Algorithm]

```
Function DP (clause set S)
  ① If S is empty return true
  ② If S contains a null clause return false
  ③ Chose a variable L in S and split the clause set
S recursively for L=0 and L=1
  if DP(S(L=0)) is true (SAT)
     return true
  else if DP(S(L=1)) is true (SAT)
     return true
  else
     return false
```

{EXAM Standard}

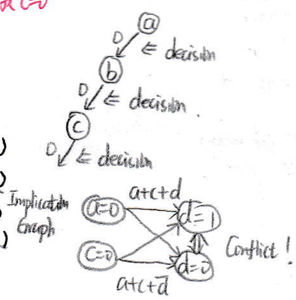


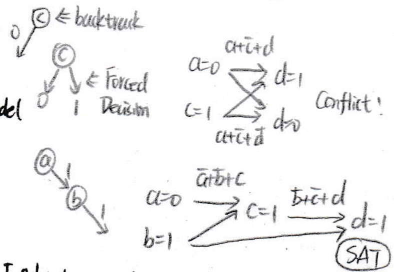
[DLL Algorithm – an improvement over DP]

DLL applies the DP algorithm, using depth first search, plus tree pruning (修剪) through backtracking and implication graphs (隐含图)

选 C=0

$(\bar{a}+b+c)$
$(a+c+d)$
$(a+c+\bar{d})$
$(a+\bar{c}+d)$
$(a+\bar{c}+\bar{d})$
$(\bar{b}+\bar{c}+d)$
$(\bar{a}+b+\bar{c})$
$(\bar{a}+\bar{b}+c)$





[Advantages of DLL over DP]

Eliminates the exponential memory requirements
Exponential time is still a problem
Limited practical applicability
Limited size of problems are allowed.

State how processing time of the DP algorithm is related to the problem size in the worst case. Briefly outline possible modification of DP that may lead to better computational performance?

⇒ In the worst case DP is NP-complete, i.e. of non-polynomial (exponential) time wrt number of variables.

Possible modification include the DLL algorithm with backtracking

Further modification may include conflict driven learning and non-chronological backtracking
(↑ 非时间先后)

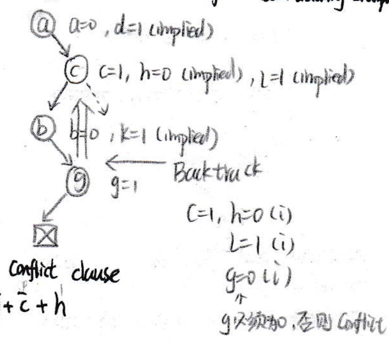
Binary Decision Diagrams {或 0 / 与 1}

Store the Boolean function in a Directed Acyclic Graph (DAG) representation. Compacted form of the binary decision tree.

[Conflict Driven Learning and Non Chronological Backtracking Example]

$a+d$
$a+\bar{c}+\bar{h}$
$a+h+\bar{l}$
$b+k$
$\bar{g}+\bar{c}+i$
$\bar{g}+h+\bar{i}$ } i & $\bar{i}$ Conflict
$g+h+\bar{j}$
$g+j+\bar{l}$

$\bar{g}+\bar{c}+h$ ⇓ Add conflict clause $\bar{g}+\bar{c}+h$



[Conflict-Driven Learning – advantages]

Learned clause is useful forever
Useful in generating future conflict clauses.
Can restart, i.e. abandon the current search tree and reconstruct a new one
- Add robustness in the solver.
- The clauses learned before the restart are still in the Boolean function after the restart and can help pruning the search space

※ BCP is based on unit clauses (clauses are composed of a single literal)

Binary (Boolean) Constraint Propagation (BCP == UP == OLR)

If a set of clauses contains the unit clause U, the other clauses are simplified by the iterative application:
① Every clause (other than clause U itself) containing U is removed
② In every clause that contains the negation of U: $\bar{U}$, the literal $\bar{U}$ is removed.

The application of these two rules leads to a new, simpler set of clauses, that is equivalent to the old one.

Example:

$$F = a(a+b)(\bar{a}+c)(\bar{c}+d)$$

① $U = a$

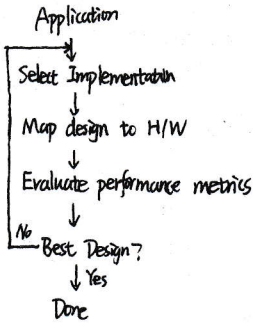
$$F = a(a+b)(\bar{a}+c)(\bar{c}+d) = a \cdot c \cdot (\bar{c}+d)$$

(↑ U itself; ↑ clause that containing U; ↑ negation of U)

② $U = c$

$$F = a \cdot c \cdot (\bar{c}+d) = a \cdot c \cdot d$$

Design space exploration and synthesis



Optimisation goal: find the best design in the design space.

Optimisation often has conflict objectives

Typical trade-offs: speed, power, size, cost

- Evolutionary optimisation (genetic)
- Simulated annealing

Integer programming

Simulated annealing

- At temperature T:
  - Generate a random perturbation x' of the current design point x.
- If the simulated 'energy', i.e. the cost function $f(x')$ has decreased, move to this new design point.
- If the energy $f(x')$ has increased, the new point is accepted according to the probability of Boltzman distribution:

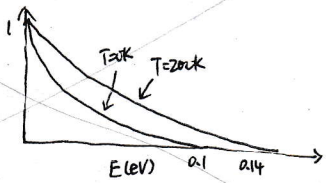
$$P = e^{-\frac{f(x')-f(x)}{kT}}$$

when temperature $T \to 0$, $P \to 0$

k - Boltzman constant

- When system reaches thermal equilibrium at temperature T, decrease T and repeat
- How temperature should change, i.e. the cooling schedule, is important.

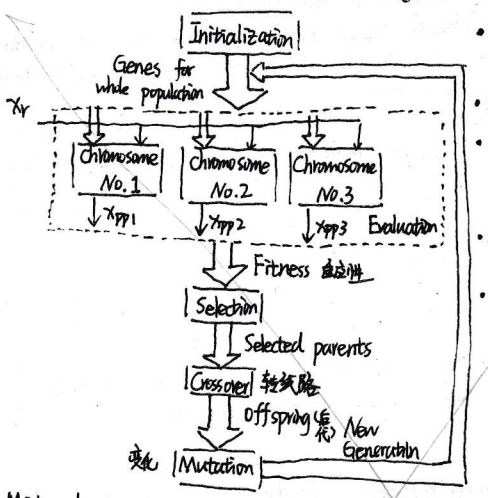
Boltzman distribution probabilities



Main adv: guaranteed to find global optimum in the presence of many local optima provided it run long enough

Main disadv: slow performance, heavily dependent on heuristics (启发法) concerning the cooling schedule and probability of accepting a worse move.

Evolutionary optimisation - basic Genetic Algorithm



- Design ⇒ "Chromsome"
  - Genes are the design parameters
- Optimization Function ⇒ "Fitness"
- Create "Generations" of solutions
  - Generation: a population of acceptable (valid) designs
- Select a subset of "fit" designs to breed (养育) next generation
- Generate next generation by
  - Mutation: modify genes of previous generation
  - Crossover: select two (or more) "parents" and create children by combining their genes.

Main advantages of GA:

① Can be used easily in multi-objective optimisation (biggest advantage)

② Is capable of finding the global minimum

③ Easy application to synthesis, both structural and parametric.

④ Lend itself to easy implementations on parallel platforms

Main disadvantages.

① GA can be slow compared with other stochastic methods.

② GA is very sensitive to the fine tuning 微调 of its parameters (e.g. mutation probabilities 变异概率
elitism percentage 精英主义百分比
fitness normalization

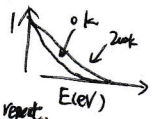
1) Describe how the adaptive body biasing technique is employed in digital designs to provide performance-power trade off.

⇒ Adaptive body biasing applies reverse body bias or forward body bias to transistors to reduce leakage power and reduced performance, or to improve performance with higher leakage power. Despite its practical complexity, the technique is used in commercial designs such as state-of-the-art- Intel processors.

At temperature T,
- generate a random perturbation Δx at the current design point
- If the simulated energy e.g. the cost function $f(x)$ has decreased, move to this new design point
- If the energy $f(x)$ has increased, the new design point is achieved according to the probability of boltzmann distribution

$$P = e^{-\frac{f(x)-f(x)}{kT}}$$

$P \to 0, T \to 0$

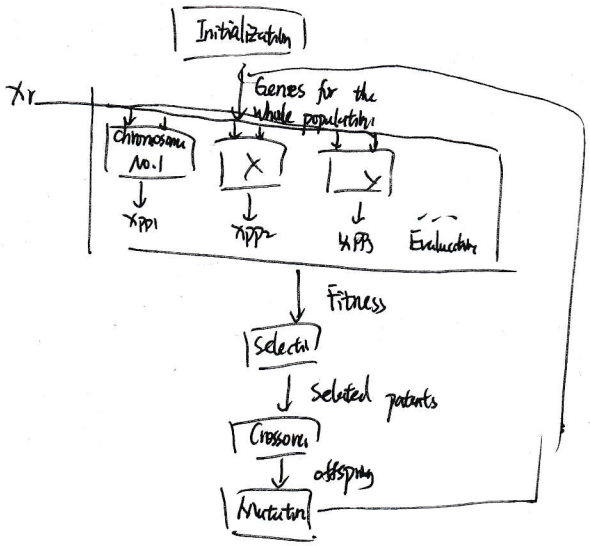


k is the boltzmann constant and repeat, T

When the system reaches thermal equilibrium, reduce temperature

Main adv: guaranteed to find a global optimum in the presence of many local optima provided it runs long enough

Slow performance, highly dependent on the heuristics coming the cool scheduling and probility of accepting a worse run.



- Can be easily used in multi-objective optimization
- Easy to synthesize both structural and parameter
- Is capable of finding a global minimum
- Lend itself easy implementation on parallel platforms

GA is slow compared to other stochastic methods

GA is very sensitive to fine tuning of its par

```
# include "systemc"

SC_Module (Adder ) {

    sc_in <sc_logic> A, B, Cin ;
    sc_out <sc_logic > Sum, Cout;

    void add() {
```

A	B	Cin	Sum	Cout
0	0	0		0
0	0	1		0
0	1	0		0
0	1	1		1
1	0	0		0
1	0	1		1
1	1	0		1
1	1	1		1

~~Cout~~ = BCin

Cout = BCin + AB + ~~A~~Cin

= Cin(A+B) (A+B)

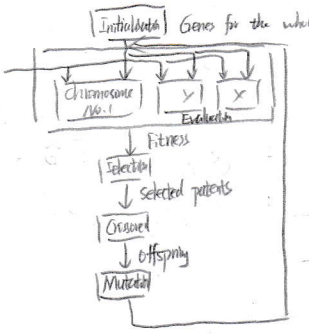
At temperature T

- Generate a random perturbation X' of the current design point X.

If the simulated 'energy' has decreased, move to this new design point. If the energy f(x') has increased the new design point is accepted tentatively & the probability of the boltzman distribution. $p = e^{-\frac{f(x')-f(x)}{kT}}$

When the system reaches the thermal equilibrium at temper T. reduce the temper. and repeat.

How the temper should change. e.g. the cooling scheduling is important find a global optimum in the presence of Adv: guaranteed to many local optima provided it runs long enough. - Disadv. slow performance and highly dependent on the heuristic concerning the cooling schedule and the probability of moving to another or worse point.



optimization

Main adv: ① Can be used easily for multi-objective
② Easy to synthesize both structural and
③ Lend itself easy implementation on parallel platform
④ Is capable of finding a global minimum

DS slow perf compared to stochastic methods
GA is very sensitive to ~~other~~ the fine tuning of its input variables

If S can c depend
It s must change

Choose a variable L in S and split the choice set S recursive for L=0 .

DLL DP

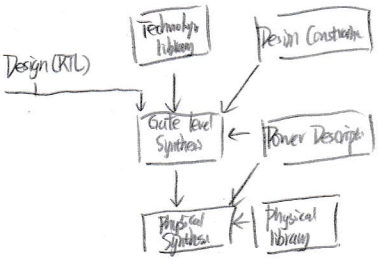
DP(S(L=0)) = two depth for scan and

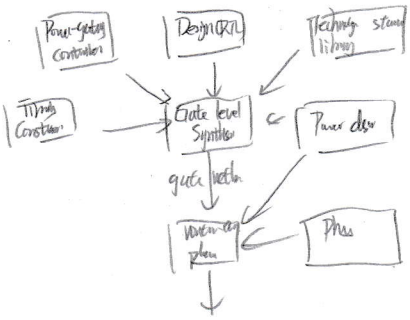
DP(S(L=1)) trace passing through bedroom and implication graph

Eliminates the exponential memory requirement
Limited practical applicability
limited set of problems are solved

~~DP~~ algo. In the worst case, DP is an NP-complete problem i.e. of non-polynomial time with respect to the size of input

DLL is
Fight modified includes conflict clause learning and chronological backtracking





ASAP the ce

Determine the conditions, center is the node whose predecessors have been scheduled

schedule all center nodes one cycle later the max cycle of the predecessors

Schedule before the start cycle of successors

Label each node by the longest path passing through it

Initialize the cycle count L

Delete from U the unlabeled nodes with

Sort S in the

If S

$P_t - N_t$

Longest

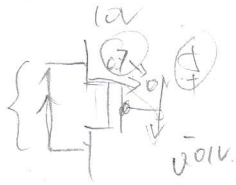
S

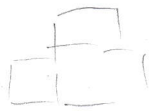
Compute

Schedule

Assign

0.69





High-level synthesis steps - Scheduling

[Automated high-level synthesis]

Determine when to perform each operation
- Scheduling
- Consider alternatives and optimise (e.g. for minimum latency)

Allocate resources for each operation
- Resource allocation
- Consider alternatives and optimise

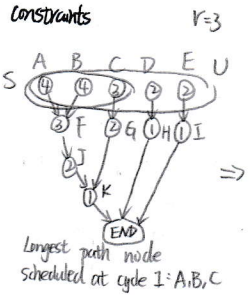
Map operations onto resources
- Binding

Scheduling

Scheduling assigns a start time to each operation in DFG
- Start times must not violate dependencies in DFG
- Start times must meet performance constraints

[ISSUE]

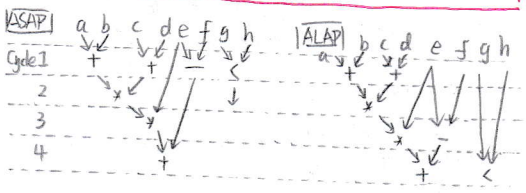
① Unconstrained scheduling
② Minimum latency scheduling
③ Latency constrained scheduling
④ Resource constrained scheduling
⑤ Minimum-latency, resource constrained
⑥ Minimum-resource, latency constrained



Minimum Latency Scheduling

ASAP (as soon as possible) algorithm
- Find a candidate node in DFG
  (candidate node whose predecessors have been scheduled (or there are no predecessors)
- Schedule one node one clock cycle later than max cycle of predecessor.
- Repeat until all nodes scheduled

ALAP (as late as possible) algorithm
- Run ASAP, get minimum latency L
- Find a candidate node in DFG (or there's none
  (candidate is node whose successors are scheduled
- Schedule node one clock cycle before start cycle of successor
  • Node with no successors scheduled to last cycle L
- Repeat until all nodes scheduled.



Latency constrained scheduling

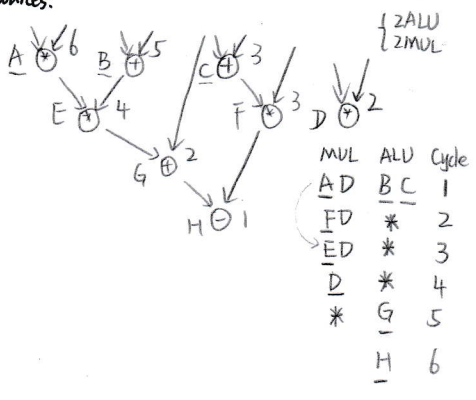
Instead of finding the minimum latency, find latency at most that of a given constraint L.

[Solutions]

Use ASAP, verify that minimum latency is at most L
Use ALAP starting with cycle L instead of minimum latency (no need to run ASAP first, as L is given)

Scheduling with Resource Constraints

Schedule must use less than specified number of resources.



Minimum-Latency, Resource-Constrained Scheduling

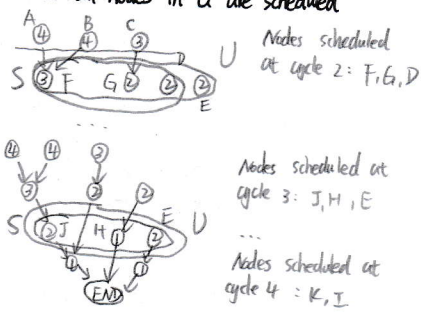
Given resources constraints, find a schedule that has the minimum latency.

[Hu's algorithm] ← Assume that all resources are of one type (number of resources)

Input: Data flow graph G, Resource constraints r

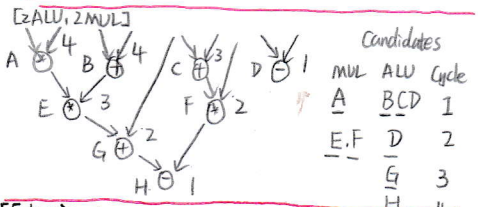
① Label each node in G by the longest path passing through it
② Initialise cycle count: L = 1
③ Repeat: EXAM Standard
  ③.1 Determine the subset U of unscheduled nodes in G whose predecessors have been scheduled or there are no predecessors
  ③.2 Select subset S of U such that |S| ≤ r and labels in S are largest.
  ③.3 Schedule the S nodes at cycle L
  ③.4 L++

Until all nodes in G are scheduled



[Extensions to Hu's scheduling algorithm]
- Multiple resource types
- Multi-cycle operation

[List Scheduling - minimum latency, constrained resource]
- Extension for multiple resource types
- Basic idea - run Hu's algorithm for each resource type
1. Input: graph, set of constraint R for each resource type t
2. Label nodes based on max distance to output
3. For each resource type t
  3.1 Determine candidate nodes, C (no predecessors or scheduled predecessors)
  3.2 Schedule up to Rt operations from C based on priority, to current cycle (Rt is the constraint on resource type)
4. Increase cycle, repeat from 3 until all nodes scheduled.

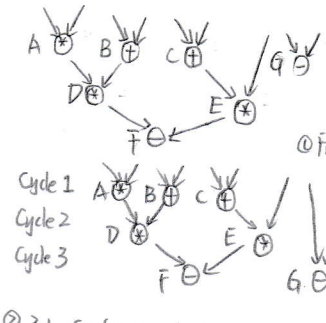


[Extension for multicycle operations]
1. Label the nodes based on max cycle latency (distance to output) taking multi-cycle operations into account.
2. For each resource type t
  2.1 Determine candidate nodes, C (those with no predecessors or with scheduled and completed predecessors)
  2.2 Schedule up to (Rt - Nt) operations from C based on priority, one cycle after the predecessors.
  - Rt is the constraint on resource type t
  - Nt is the number of resources of t in use from previous cycles.
  - Repeat loop 2 until all nodes scheduled.

Scheduling for minimum resources with latency constraint.

Optimise: for all schedules that have latency less than the constraint, find the one that uses the fewest resources.

[List scheduling for minimum resources]
1. Compute last start times for each node using ALAP with specified latency constraint.
  - Latest start times must take into account multicycle operations
2. For each cycle and for each resource type
  2.1 Determine candidate nodes
  2.2 Compute slack for each candidate
    Slack = current cycle - latest possible value
  2.3 Schedule the nodes with 0 slack
    Update required number of resources
    (start with 1 of each at the beginning)
  ☆ 2.4 Schedule the nodes that require no extra resources
3. Repeat loop 2. until all nodes scheduled



② 2.1 C = {A, B, C, G}
2.2
Node	LPC	Slack	
A	1	1-1	0
B	1	1-1	0
C	1	1-1	0
D	2	3-1	2

Cycle 1
2.3 Resource = {1 MUL, 2 ALU}

② 2.1 C = {D, E, G}
2.2
Node	LPC	Slack	
D	2	2-2	0
E	2	2-2	0
G	3	3-2	1

Cycle 2
2.3 Resource = {2 MUL, 2 ALU}
2.4 Schedule the nodes that require no extra resources

② 2.1 C = {F}
2.2 F 3 0
2.3 ...
2.4 ...

Node	LPC	Slack	Cycle
A	1	1-1=0	1
B	1	1-1=0	1
C	1	1-1=0	1
D	2	2-2=0	2
E	2	2-2=0	2
F	3	3-3=0	3
G	3	3-2=1	2

Minimum-resource, latency-constrained schedule

Step 1: Start with an ALAP schedule and calculate the Latest Possible Start for each node

Step 2: (Extended Hu's algorithm) List scheduling
For each cycle and for each resource type
1. Determine candidate nodes
2. Compute slack for each candidate
   Slack = current cycle - latest possible cycle
3. Schedule the nodes with 0 slack
   If necessary, update required number of resources
4. Schedule the nodes that require no extra resources

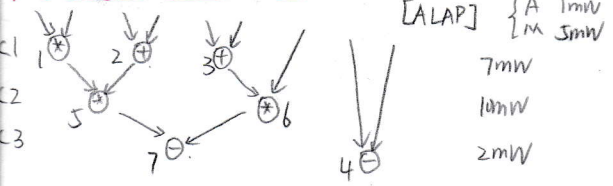
EXAM Standard

Scheduling for minimum power

[Power constrained scheduling = based on scheduling for minimum resources with latency constraints]

Modification of Hu's algorithm for minimum power consumption:

1. Start with only one type of each resource
2. Complete latest start times for each node using ALAP with specified latency constraint.
3. For each cycle and for each resource type
   - 3.1 Determine candidate
   - 3.2 Compute slack for each candidate
     slack = current cycle − latest possible cycle
   - 3.3 If there are nodes with 0 slack, schedule all of them − ask for extra resources if necessary
   - 3.4 If there are no nodes with 0 slack, schedule only one node from those with the lowest slack; arbitrary, but first try a node with the highest power consumption.
4. Repeat loop 3. until all nodes scheduled.



Node	LPC	Slack	Cycle
1	1	1-1 0	1
2	1	1-1 0	1
3	1	1-1 0	1
4	3	2/1/0	3
5	2	2-2 0	2
6	2	2-2 0	2
7	3	3-3 0	3

C={1,2,3,4}, R={2A+1M}
C={4,5,6}, R={2A+2M}
C={4,7}, R={2A+2M}

Binding 封裝

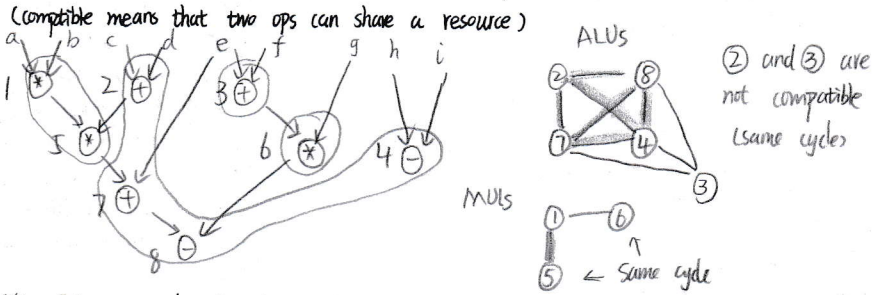
Given a schedule, map operations onto resources such that operations in same cycle do not use same resource

Bad binding may increase resources, require huge steering logic, reduce clock rate.

[Compatible Graphs]

- Each node is an operation
- Edges ~~of~~ represent compatible operations between a pair of nodes

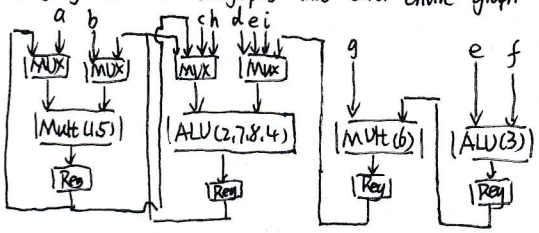
(compatible means that two ops can share a resource)



Note: Fully connected subgraphs can share one resource (all involved nodes are compatible)

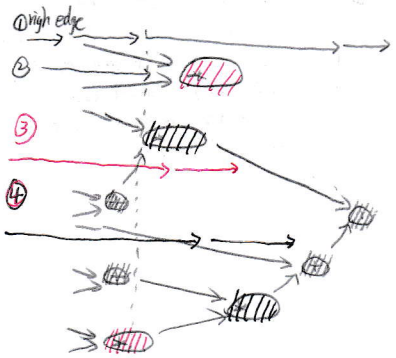
Binding goal: Find minimum number of fully connected subgraphs that cover entire graph

1) Add resources and registers
2) Add mux for each input
3) Add input to left mux for each left input in DFG
4) Do same for right mux
5) If only one input, remove mux



Left Edge Algorithm

1) Initialize right_edge to 0
2) Find a node N whose left edge is >= right_edge
3) Bind N to a particular resource
4) Update right_edge to the right edge of N
5) Repeat from 2) for nodes using the same resource type until right_edge passes all nodes.
6) Repeat from 1) until all nodes bound.



Find a valid binding, but they do not consider amount of steering logic required

[Possible solution]

Use weighted edges/nodes to introduce cost function representing steering logic

Perform clique (派系) partitioning, finding the set of cliques that minimize weight

Low Power

Power of digital circuit: $P = P_{dynamic} + P_{leakage}$

$P_{dynamic} = V_{dd}^2 * F * C_L * E_{sw}$ ← gate switching activity

Dynamic power reduction happens at various design levels
- Physical layout level ⇒ short interconnect ⇒ smaller C_L
- Transistor level ⇒ transistor with small W ⇒ smaller C_L
- Architectural level
  - Switching activity reduction: Clock gating
  - Vdd reduction: Multiple Vdd

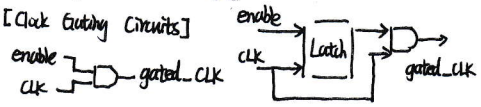
Clock gating (architectural level)

When clocked modules (memory, reg files ...) are inactive (stored values not changing), use clock gating to disenable the clocking feeding module.

[Reason]

FF consumes power due to switching activity even when input is not changing.

[Clock Gating Circuits]



Latch captures the state of the enable signal and hold it until next clock cycle.

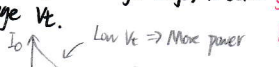
Multi Vdd (architectural level dynamic power reduction)

Modules on critical path use highest Vdd (to meet required timing constraints)
Modules on non-critical paths use lower Vdd (reduce power)

Leakage Power Reduction

Leakage power consumption occurs as long as the circuit is powered on.

Sub-threshold current between source and drain in MOS transistor occurs when gate voltage (V_{gs}) is below transistor threshold voltage V_t.



[Technology level]

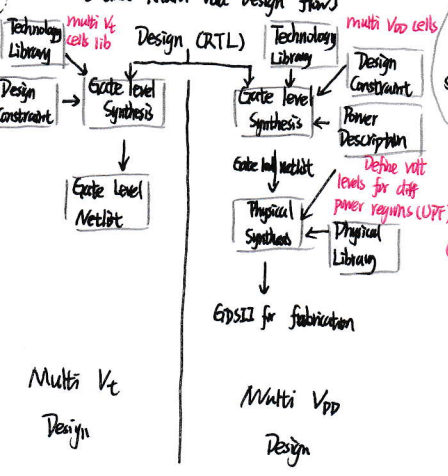
Multiple V_t technology where fabrication process provides both high and low threshold transistor V_t (low: fast and leaky and high: slow and less leaky)

Default design uses high V_t and careful replacing high V_t with low V_t, one can get low V_t performance and yet significantly lower leakage power.

Example: critical path delay determines performance of the design, all other paths have lower delay. Use low V_t transistors in modules on the critical path, and high V_t transistors in modules on non-critical paths

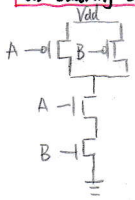
~~Many Vt technology~~
Many libries are required, hard to use multiple libraries

[Multi Vt and Multi Vdd Design flow]

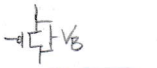


[Stacking Effect]

Sub-threshold voltage current flowing through stacks of series connected transistors reduces when more than one transistor in the stack is turned off. This effect is known as stacking effect.



[Body Bias] Cost too much



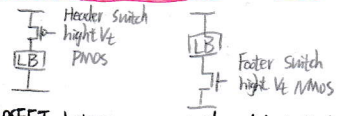
Applying reverse body bias (by) to transistor reduces leakage and can be used to reduce standby leakage power.

Alternatively apply forward body bias, which reduces V_t and improves circuit performance but increases leakage

[Adaptive Body Bias]

Combination of reverse and forward body bias can be applied to reduce leakage power with reduced performance or improved performance with higher leakage power.

Power Gating



① Sleep transistors
- Insert high V_t MOSFET between power rail and logic blocks
- Header transistor off lead to logic block floating to near zero
- Footer transistor off lead to logic block floating to near Vdd

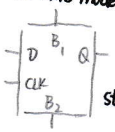
Header switches preferred if multiple power rails since gated block logics float to zero irrespective of supply voltages.

↑ Sleep transistors to decouple the power-gate logic block from its Vdd.

② State rentention register

State rentention registers to restore states after sleep. Typical register will include normal FF with a ballon (latch)

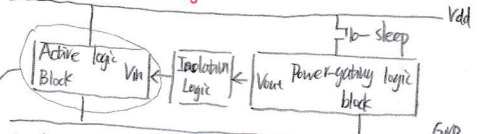
Two modes:
1. Sleep mode: state stored in balloon (latch) which is powered by Vdd
2. Active mode: balloon inactive; do not interfere with normal operation



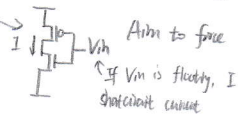
B1, B2 pins control the saving and restoring of state.

③ Output Isolation

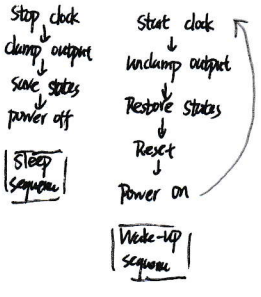
Output isolation to clamp the floating output of the power-gating block and to prevent short circuit current injection into active logic block.



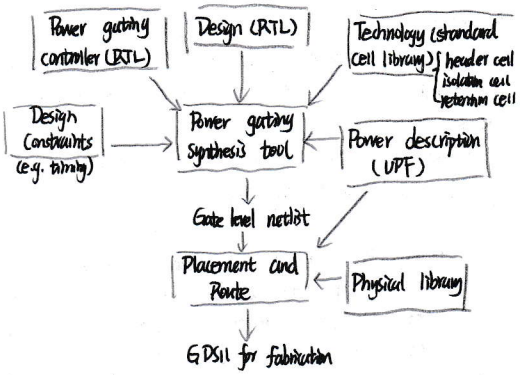
Without output isolation, floating output causes short-circuit current in active logics



④ Power gating Controller (in the form of a state machine)



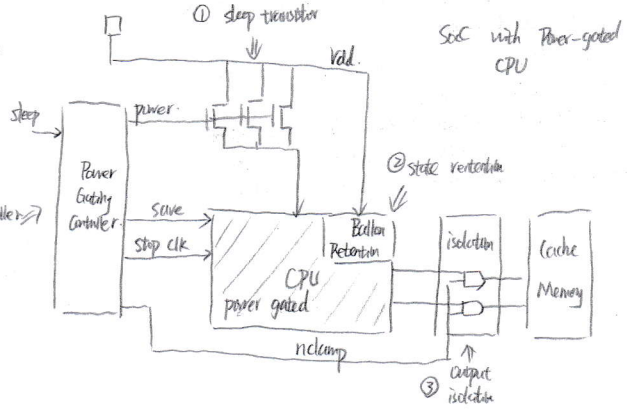
Power gating design synthesis flow



Adaptive body biasing applies reverse body bias or forward body bias to transistors to reduce leakage power and reduced performance, or to improve performance with higher leakage power.

Power gating leads to state short logic value being lost due to shut down. some applications require states when active again. Such states can be stored in state rentain registers.

Such registers have two modes of operations.



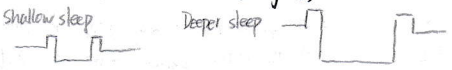
Energy Management

Dynamic power management (DPM)

Shut down processing elements (PE) when inactive.

Different algorithms are used, normally engineered as part of the OS:

- Greedy: Go to sleep as soon as processing is finished
- Timeout: Stay on expecting a new request. After time t of idleness go to sleep.
  - Adaptive Timeout: Vary timeout t depending on activity
  - Shallow sleep states are entered and exited quickly (few clock cycles), deep sleep states cost more in time and energy (1000s of clock cycles)



Dynamic voltage frequency scaling (DVFS)

Adapt PE performance to requirement by dynamically changing PE clock frequency and Vdd

Processing elements have non-uniform workloads which DVFS exploits. The change in frequency or Vdd normally is carried out using energy-efficient operation systems "add-on" such as Linux powersaving governors.

Turning off or reducing Vdd and hence power optimisation is possible due to the presence of idle time and slack time in systems schedule.

In DVFS, power changes cubicly with frequency-voltage scaling, and overall performance is roughly linear with clock frequency.

Vdd	Frequency	Power Saving	Performance-Loss
100%	100%	0	0
95%	95%	14%	5%
90%	90%	27%	10%

Energy Management: System level

Idle time: periods where PEs do not experience any workload

Slack time: amount time of time that a PE is idle during the execution of a function. Results from tasks finishing execution before their deadline.

⇑

Why? Slack = deadline − finish time

{ Not all PEs utilised constantly during run-time
  Performance of PEs can not be perfectly during run-time

Interplay of DPM and DVFS

Energy saved with DVFS come at the cost of increased execution time, implying shortened idle period afor applying DPM, and greater leakage energy consumption.

Ideal DVFS policy must understand this interplay and hence perform V-f setting selection with the objective of reducing the overall system energy consumption.

This depends on the processor workloads characterisation

- DPM, the characterisation is in terms of the distributions of idle period durations
- DVFS, it is in terms of CPU/memory intensiveness of the executing tasks

DPM and DVFS outperform each other under different workloads, no single PM policy fits perfectly all operating conditions, hence hybrid power management is needed.

Describe the necessary interaction between the system software and power-gated and dynamic voltage frequency scaling (DVFS) enabled processors in order to achieve power reduction.

⇒ To activate the power gating, sleep signals need to be generated. Such signals are often enabled by the system software (e.g. OS) resulting from monitoring the activities of the processors, inactive processors are candidates for shutting down.

Similarly to activate DVFS, the processor's performance counters such as CPU utilisation give indication how much processing is taking place. The software through special register Vco will adapt the voltage/frequency of the processors according to tasks

SystemC

{ SC_MODULE
  SC_CTOR
  SC_METHOD

{ sc_in
  sc_out
  <sc_logic>

```
// 1-Bit Adder, Adder.h
# include "systemc.h"
SC_MODULE(Adder) {
    sc_out <sc_logic> Sum, Cout;
    sc_in <sc_logic> A, B, Cin;
    void add() {
        sc_logic tempC, tempD, tempE;
        tempC = A.read() & B.read();
        tempD = A.read() ^ B.read();
        tempE = Cin.read() & tempD;
        Sum.write(tempD ^ Cin.read());
        Cout.write(tempC | tempE);
    }
    SC_CTOR(Adder) {
        SC_METHOD(add);
        sensitive << A << B << Cin;
    }
}

// 8-bit Counter
# include "system.h"
SC_MODULE(Counter8bit) {
    sc_in_clk clock;
    sc_in <bool> enable;
    sc_in <bool> error;
    sc_in <bool> reset;
    sc_out <sc_uint<8>> counter_out;
sc_uint<8> count;
void incr_count() {
    while(true)
    {
        if(enable.read()){
            if(reset.read()){
                count = 0;
                counter_out.write(count);
            }
            else{
                count = count + 1;
                counter_out.write(count);
            }
        }
        wait();
    }
}
SC_CTOR(Counter8bit){
    SC_THREAD(incr_count);
    sensitive << clock.pos();
}
};
```