Design Quality Estimation

Estimation

- Estimates allow
  - Evaluation of design quality
  - Design space exploration

- Design model
  - Represents degree of design detail computed
  - Simple vs. complex models

- Issues for estimation
  - Accuracy
  - Speed
  - Fidelity
Typical estimation model example

<table>
<thead>
<tr>
<th>Design model</th>
<th>Additional tasks</th>
<th>Accuracy</th>
<th>Fidelity</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Low</td>
<td>Low</td>
<td>fast</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU</td>
<td>FU allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg</td>
<td>Lifetime analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg+Mux</td>
<td>FU binding</td>
<td>high</td>
<td>high</td>
<td>slow</td>
</tr>
<tr>
<td>Mem+FU+Reg+Mux+Wiring</td>
<td>Floorplanning</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accuracy vs. Speed

- Accuracy: difference between estimated and actual value
  \[ A = 1 - \frac{|E(D)-M(D)|}{M(D)} \]
  \[ |E(D)|, |M(D)|: estimated & measured value \]
- Speed: computation time spent to obtain estimate
- Simplified estimation models yield fast estimator but result in greater estimation error and less accuracy.
Fidelity

- Estimates must predict quality metrics for different design alternatives
- Fidelity: % of correct predictions for pairs of design implementations
- The higher fidelity of the estimation, the more likely that correct decisions will be made based on estimates.
- Definition of fidelity:
  \[ F = 100 \times \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} u_{i,j} \]

  \[
  \begin{align*}
  (A, B) & = E(A) > E(B), M(A) < M(B) \quad \times \\
  (B, C) & = E(B) < E(C), M(B) > M(C) \quad \times \\
  (A, C) & = E(A) < E(C), M(A) < M(C) \quad \bigcirc \\
  \text{Fidelity} & = 33 \% 
  \end{align*}
  \]

Quality metrics

- Performance Metrics
  - Clock cycle, control steps, execution time, communication rates
- Cost Metrics
  - **Hardware**: manufacturing cost (area), packaging cost (pin)
  - **Software**: program size, data memory size
- Other metrics
  - Power
  - Design for testability: Controllability and Observability
  - Design time
  - Time to market
Clock cycles metric

- Selection of a clock cycle before synthesis will affect the practical execution time and the hardware resources.
- Simple estimation of clock cycle is based on maximum-operator-delay method.

\[
clk(MOD) = \text{Max}_{all_i} (delay(t_i))
\]

- The estimation is simple but may lead to underutilization of the faster functional units.
- Clock slack represents the portion of the clock cycle for which the functional unit is idle.
Clock cycle estimation

- **Slack**: portion of clock cycle for which FU is idle
  \[
  \text{slack}(clk, ti) = \left( \frac{\text{delay}(ti)}{dk} \right) \cdot dk - \text{delay}(ti)
  \]

- **Average slack**: FU slack averaged over all operations
  \[
  \text{ave_slack} = \frac{\sum_i \text{occur}(ti) \cdot \text{slack}(clk, ti)}{\sum_i \text{occur}(ti)}
  \]

- **Clock utilization**: % of clock cycle utilized for computations
  \[
  \text{utilization} = 1 - \frac{\text{ave_slack}(clk)}{clk}
  \]
Clock utilization

\[
\text{ave_slack}(65 \text{ ns}) = 6 + 2 + 2 = 24.4 \text{ ns}
\]

\[
\text{utilization}(65 \text{ ns}) = 1 - \frac{24.4}{65.0} = 62\%
\]

Control steps estimation

- Operations in the specification assigned to control step
- Number of control steps reflects:
  - Execution time of design
  - Complexity of control unit
- Techniques used to estimate the number of control steps in a behavior specified as straight-line code
  - Operator-use method.
  - Scheduling
**Operator-use method**

- Easy to estimate the number of control steps given the resources of its implementation.
- Number of control steps for each node can be calculated:
  \[
  csteps(n_j) = \max_{t_j \in T} \left[ \frac{occ(t_j)}{num(t_j)} \right] \times clocks(t_j)
  \]
- The total number of control steps for behavior B is
  \[
  csteps(B) = \max_{n_j \in N} csteps(n_j)
  \]

**Operator-use method Example**

- Differential-equation example:
**Scheduling**

- A scheduling technique is applied to the behavior description in order to determine the number of controls steps.

- It's quite expensive to obtain the estimate based on scheduling.

- Resource-constrained vs time-constrained scheduling.

---

**Scheduling for DSP algorithms**

- **Scheduling**: assigning nodes of DFG to control times
- **Resource allocations**: assigning nodes of DFG to hardware(functional units)
- **High-level synthesis**
  - Resource-constrained synthesis
  - Time-constrained synthesis
Classification of scheduling algorithms

- Iterative/Constructive Scheduling Algorithms
  - As Soon As Possible Scheduling Algorithm (ASAP)
  - As Late As Possible Scheduling Algorithm (ALAP)
  - List-Scheduling Algorithms

- Transformational Scheduling Algorithms
  - Force Directed Scheduling Algorithm
  - Iterative Loop Based Scheduling Algorithm
  - Other Heuristic Scheduling Algorithms

The DFG in the Example
As Soon As Possible (ASAP) Scheduling Algorithm

- Find minimum start times of each node

```
Input: DFG G = (N, E).
Output: ASAP Schedule.
1. T_{S_0} = 1; /* Set initial time step */
2. While (Unscheduled nodes exist) {
   2.1 Select a node n_j whose predecessors have already
       been scheduled;
   2.2 Schedule node n_j to time step T_{S_j} = \max \{ T_{S_i} + (C_i) \}
       \forall n_i \rightarrow n_j;
}
```

As Soon As Possible (ASAP) Scheduling Algorithm

- The ASAP schedule for the 2nd-order differential equation

![Diagram](image)
As Late As Possible (ALAP) Scheduling Algorithm

- Find maximum start times of each node

**ALAP Schedule for the 2nd-order differential equation**
**List-Scheduling Algorithm (resource-constrained)**
- A simple list-scheduling algorithm that prioritizes nodes by decreasing criticalness (e.g. scheduling range)

```plaintext
Input: DFG G = (N, E), R = (FU).
Output: Final Schedule.
1. T_{S_0} = 1; /* Set initial time step */
2. While (Unscheduled nodes exist) {
   2.1 Locate all nodes whose predecessors have already been scheduled and place into list L;
   2.2 Sort nodes in L by decreasing criticalness;
   2.3 While (L is not empty) {
      2.3.1 Select the first node n_j from L.
      2.3.2 Determine time step T_{S_j} = \max \{ T_{S_i} + (C_i) \}
      \forall n_i \rightarrow n_j;
      2.3.3 If (FUs at T_{S_j} are not full)
         Schedule node n_j to time step T_{S_j}
         else
         Remove node from L.
   }
```

**Force Directed Scheduling Algorithm (time-constrained)**
- Transformation algorithm

```plaintext
Input: DFG G = (N, E), Iteration Period = T.
Output: Final FDS Schedule.
1. While (Unscheduled nodes exist) {
   1.1 Compute the time frames for each node;
   1.2 Build the distribution graph;
   1.3 Compute the self-forces;
   1.4 Compute the predecessor and successor forces;
   1.5 Schedule the node into the time step that
      minimizes the total force;
}
```
Force Directed Scheduling Algorithm

- Figure (a) shows the time frame of the example DFG and the associated probabilities (obtained using ALAP and ASAP).
- Figure (b) shows the DGs for the 2nd-order differential equation.

\[ \text{Self Force}_e(1) = \sum_{i=S_j} L_j [DG(i) \times x(i)] \]

- Example:
  \[ \text{Self Force}_e(1) = \text{Force}_e(1) + \text{Force}_e(2) \]
  \[ = (DG_1(1) \times x_1(1)) + (DG_2(2) \times x_2(2)) \]
  \[ = (2.833 \times (1-0.5)) + (2.333 \times (0-0.5)) \]
  \[ = (2.833 \times +0.5) + (2.333 \times -0.5) \]
  \[ = +0.25 \]
**Force Directed Scheduling Algorithm**

- Example (con’d.):
  
  \[
  \text{Self}_{\text{Force}}4(2) = \text{Force}_4(1) + \text{Force}_4(2) \\
  = (D_G m(1) x_a(1)) + (D_G m(2) x_a(2)) \\
  = (2.833 \times (-0.5)) + (2.333 \times (+0.5)) \\
  = -0.25 \\
  \]

  \[
  \text{Succ}_{\text{Force}}4(2) = \text{Self}_{\text{Force}}4(2) + \text{Self}_{\text{Force}}8(3) \\
  = (D_G m(2) x_a(2)) + (D_G m(3) x_a(3)) \\
  = (2.333 \times (0-0.5)) + (0.833 \times (1-0.5)) \\
  = (2.333 \times (-0.5)) + (0.833 \times (0.5)) \\
  = -0.75 \\
  \]

  \[
  \text{Force}_4(2) = \text{Self}_{\text{Force}}4(2) + \text{Succ}_{\text{Force}}4(2) \\
  = -0.25 - 0.75 \\
  = -1.00 \\
  \]

**Force Directed Scheduling Algorithm (another example)**

A1: $F_{a1}(0) = 0$;  
A2: $F_{a2}(1) = 0$;  
A3: T(1): \( \text{Self}_{F_{a3}}(1) = 1.5 \times 0.5 - 0.5 \times 0.5 = 0.5 \)  
\( \text{Pred}_{F_{m2}}(1) = 0.5 \times 0.5 - 0.5 \times 0.5 = 0 \)  
\( F_{a3}(1) = 0.5 \)  
T(2): \( \text{Self}_{F_{a3}}(2) = -1.5 \times 0.5 + 0.5 \times 0.5 = -0.5 \)  
\( F_{a3}(2) = -0.5 \)  
M1: \( F_{m1}(2) = 0 \);  
M2: T(0): \( \text{Self}_{F_{m2}}(0) = 0.5 \times 0.5 - 0.5 \times 0.5 = 0 \)  
\( F_{m2}(0) = 0 \)  
T(1): \( \text{Self}_{F_{m2}}(1) = -0.5 \times 0.5 + 0.5 \times 0.5 = 0 \)  
\( \text{Succ}_{F_{a3}}(2) = -1.5 \times 0.5 + 0.5 \times 0.5 = -0.5 \)  
\( F_{m2}(0) = -0.5 \)
Scheduler

- Critical path scheduler
  - Based on precedence graph (intra-iteration precedence constraints)
- Overlapping scheduler
  - Allow iterations to overlap
- Block schedule
  - Allow iterations to overlap
  - Allow different iterations to be assigned to different processors.

Overlapping schedule

- Example:
  - Minimum iteration period obtained from critical path scheduler is 8 t.u
Block schedule

• Example:
  – Minimum iteration period obtained from critical path
  scheduler is 20 t.u

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A0</td>
<td>A0</td>
<td>B0</td>
<td>B0</td>
<td>C0</td>
<td>C0</td>
<td>A2</td>
<td>A2</td>
<td>B2</td>
<td>B2</td>
<td>C2</td>
<td>C2</td>
</tr>
<tr>
<td>P2</td>
<td>A1</td>
<td>A1</td>
<td>B1</td>
<td>B1</td>
<td>C1</td>
<td>C1</td>
<td>A3</td>
<td>A3</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D0</td>
<td>D0</td>
<td>E0</td>
<td>D1</td>
<td>D1</td>
<td>E1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branching in behaviors

• Control steps maybe shared across exclusive branches
  – sharing schedule: fewer states, status register
  – non-sharing schedule: more states, no status registers
Execution time estimation

- Average start to finish time of behavior
- **Straight-line code behaviors**
  \[ \text{execution}(B) = \text{csteps}(B) \times \text{clk} \]
- **Behavior with branching**
  - Estimate execution time for each basic block
  - Create control flow graph from basic blocks
  - Determine branching probabilities
  - Formulate equations for node frequencies
  - Solve set of equations
  \[
  \text{execution}(B) = \sum_{b_i \in B} \text{exectime}(b_i) \times \text{freq}(b_i)
  \]

Probability-based flow analysis
Probability-based flow analysis

- Flow equations:
  
  \begin{align*}
  \text{freq}(S) &= 1.0 \\
  \text{freq}(v1) &= 1.0 \times \text{freq}(S) \\
  \text{freq}(v1) &= 1.0 \times \text{freq}(v1) + 0.9 > \text{freq}(v5) \\
  \text{freq}(v1) &= 1.0 \times \text{freq}(v2) \\
  \text{freq}(v1) &= 1.0 \times \text{freq}(v2) \\
  \text{freq}(v1) &= 1.0 \times \text{freq}(v3) + 1.0 > \text{freq}(v4) \\
  \text{freq}(v1) &= 1.0 \times \text{freq}(v5)
  \end{align*}

- Node execution frequencies:
  
  \begin{align*}
  \text{freq}(v1) &= 1.0 \\
  \text{freq}(v2) &= 10.0 \\
  \text{freq}(v3) &= 5.0 \\
  \text{freq}(v4) &= 5.0 \\
  \text{freq}(v5) &= 10.0 \\
  \text{freq}(v6) &= 1.0
  \end{align*}

- Can be used to estimate number of accesses to variables, channels or procedures

Communication rate

- Communication between concurrent behaviors (or processes) is usually represented as messages sent over an abstract channel.
- Communication channel may be either explicitly specified in the description or created after system partitioning.
- Average rate of a channel C, \( \text{avgrate}(C) \), is defined as the rate at which data is sent during the entire channel's lifetime.
- Peak rate of a channel, \( \text{peakrate}(C) \), is defined as the rate at which data is sent in a single message transfer.
Communication rate estimation

- Total behavior execution time consists of
  - Computation time $comptime(B)$
    - Time required for behavior B to perform its internal computation.
    - Obtained by the flow-analysis method.
  - Communication time $commtime(B,C)$
    - Time spent by behavior to transfer data over the channel
      $$commtime(B,C) = access(B,C) \times portdelay(C)$$

- Total bits transferred by the channel,
  $$total \_\_bits(B,C) = access(B,C) \times bits(C)$$

- Channel average rate
  $$average(C) = \frac{Total \_\_bits(B,C)}{comptime(B) + commtime(B,C)}$$

- Channel peak rate
  $$peakrate(C) = \frac{bits(C)}{protocol\_delay(C)}$$

Communication rates

- Average channel rate
  rate of data transfer over lifetime of behavior
  $$\text{averate}(C) = \frac{56\text{bits}}{1000\text{ns}} = 56\text{Mb/s}$$

- Peak channel rate
  rate of data transfer of single message
  $$\text{peakrate}(C) = \frac{8\text{bits}}{100\text{ns}} = 80\text{Mb/s}$$
Area estimation

- Two tasks:
  - Determining number and type of components required
  - Estimating component size for a specific technology (FSMD, gate arrays etc.)
- Behavior implemented as a FSMD (finite state machine with datapath)
  - Datapath components: registers, functional units, multiplexers/buses
  - Control unit: state register, control logic, next-state logic
- Area can be accessed from the following aspects:
  - Datapath component estimation
  - Control unit estimation
  - Layout area for a custom implementation

Clique-partitioning

- Commonly used for determining datapath components
- Let $G = (V, E)$ be a graph, $V$ and $E$ are set of vertices and edges
- Clique is a complete subgraph of $G$
- Clique-partitioning
  - divides the vertices into a minimal number of cliques
  - each vertex in exactly one clique
- One heuristic: maximum number of common neighbors
  - Two nodes with maximum number of common neighbors are merged
  - Edges to two nodes replaced by edges to merged node
  - Process repeated till no more nodes can be merged
Clique-partitioning

Variables used in the behavior are mapped to storage units like registers or memory.

Variables not used concurrently may be mapped to the same storage unit.

Variables with non-overlapping lifetimes have an edge between their vertices.

Lifetime analysis is popularly used in DSP synthesis in order to reduce number of registers required.

Storage unit estimation
**Register Minimization Technique**

- Lifetime analysis is used for register minimization techniques in a DSP hardware.
- A ‘data sample or variable’ is live from the time it is produced through the time it is consumed. After that it is dead.
- Linear lifetime chart: Represents the lifetime of the variables in a linear fashion.
  - Note: Linear lifetime chart uses the convention that the variable is not live during the clock cycle when it is produced but live during the clock cycle when it is consumed.
- Due to the periodic nature of DSP programs the lifetime chart can be drawn for only one iteration to give an indication of the # of registers that are needed.

**Lifetime Chart**

- For DSP programs with iteration period N
  - Let the # of live variables at time partitions $n \geq N$ be the # of live variables due to 0-th iteration at cycles $n-kN$ for $k \geq 0$. In the example, # of live variables at cycle $7 \geq N$ (≥6) is the sum of the # of live variables due to the 0-th iteration at cycles 7 and $(7 - 1 \times 6) = 1$, which is $2 + 1 = 3$. 
Matrix transpose example

To make the system causal a latency of 4 is added to the difference so that $T_{\text{out}}$ is the actual output time.

Circular Lifetime Chart

- Useful to represent the periodic nature of the DSP programs.
- In a circular lifetime chart of periodicity $N$, the point marked $i$ ($0 \leq i \leq N - 1$) represents the time partition $i$ and all time instances $\{Nl + i\}$ where $l$ is any non-negative integer.
- For example: If $N = 8$, then time partition $i = 3$ represents time instances $\{3, 11, 19, \ldots\}$.
  - Note: Variable produced during time unit $j$ and consumed during time unit $k$ is shown to be alive from $'j + 1'$ to $'k'$.
  - The numbers in the bracket in the adjacent figure correspond to the # of live variables at each time partition.
**Forward-Backward Register Allocation Technique:**

Note: Hashing is done to avoid conflict during backward allocation.

---

**Steps of Register Allocation**

- Determine the minimum number of registers using lifetime analysis.
- Input each variable at the time step corresponding to the beginning of its lifetime. If multiple variables are input in a given cycle, these are allocated to multiple registers with preference given to the variable with the longest lifetime.
- Each variable is allocated in a forward manner until it is dead or it reaches the last register. In forward allocation, if the register $i$ holds the variable in the current cycle, then register $i + 1$ holds the same variable in the next cycle. If $(i + 1)$-th register is not free then use the first available forward register.
- Being periodic the allocation repeats in each iteration, so hash out the register $R_j$ for the cycle $i + N$ if it holds a variable during cycle $i$.
- For variables that reach the last register and are still alive, they are allocated in a backward manner on a first come first serve basis.
- Repeat previous two steps until the allocation is complete.
Functional-unit and interconnect-unit estimation

- Clique-partitioning can be applied
- For determining the number of FU’s required, construct a graph where
  - Each operation in behavior represented by a vertex
  - Edge connects two vertices if corresponding operations assigned different control steps there exists an FU that can implement both operations
- For determining the number of interconnect units, construct a graph where
  - Each connection between two units is represented by a vertex
  - Edge connects two vertices if corresponding connections are not used in same control step

Computing datapath area

- Bit-sliced datapath
  \[
  L_{bit} = \alpha \times tr(DP)
  \]
  \[
  H_{rt} = \frac{nets}{nets_{per\_track}} \times \beta
  \]
  \[
  area(bit) = L_{bit} \times (H_{cell} + H_{rt})
  \]
  \[
  area(DP) = \text{bitwidth}(DP) \times area(bit)
  \]
**Pin estimation**

- Number of wires at behavior's boundary depends on
  - Global data
  - Port accessed
  - Communication channels used
  - Procedure calls

**Software estimation model**

- Processor-specific estimation model
  - Exact value of a metric is computed by compiling each behavior into the instruction set of the targeted processor using a specific compiler.
  - Estimation can be made accurately from the timing and size information reported.
  - Bad side is hard to adapt an existing estimator for a new processor.

- Generic estimation model
  - Behavior will be mapped to some generic instructions first.
  - Processor-specific technology files will then be used to estimate the performance for the targeted processors.
Software estimation models

![Diagram of software estimation models]

Deriving processor technology files

<table>
<thead>
<tr>
<th>Generic instruction</th>
<th>8086 instructions</th>
<th>6820 instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem3=dmem1+dmem2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>8086 bytes</th>
<th>8086 clocks</th>
<th>6820 bytes</th>
<th>6820 clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ax,word ptr[bp+offset1]</td>
<td>(10)</td>
<td>3</td>
<td>(7)</td>
<td>2</td>
</tr>
<tr>
<td>add ax,word ptr[bp+offset2]</td>
<td>(9+EA1)</td>
<td>4</td>
<td>(2+EA2)</td>
<td>2</td>
</tr>
<tr>
<td>mov word ptr[bp+offset3],ax</td>
<td>(10)</td>
<td>3</td>
<td>(5)</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generic instruction</th>
<th>Execution time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem3=dmem1+dmem2</td>
<td>35 clocks</td>
<td>10 bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generic instruction</th>
<th>Execution time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem3=dmem1+dmem2</td>
<td>22 clocks</td>
<td>6 bytes</td>
</tr>
</tbody>
</table>
Software estimation

- **Program execution time**
  - Create basic blocks and compile into generic instructions
  - Estimate execution time of basic blocks
  - Perform probability-based flow analysis
  - Compute execution time of the entire behavior:
    \[ \text{exectime}(B) = \delta \times \left( \sum \text{exectime}(bi) \times \text{freq}(bi) \right) \]
    \( \delta \) accounts for compiler optimizations
  - accounts for compiler optimizations

- **Program memory size**
  \[ \text{progsz}(B) = \sum \text{instr\_size}(g) \]

- **Data memory size**
  \[ \text{datasz}(B) = \sum \text{datasize}(d) \]