Dimension of Synthesis Problem

• The task of searching a very large design space for the lowest cost structure to meet all the constraints imposed by technology and design specifications is a formidable one.
12.1 Benefits of Algorithmic Synthesis

- Shorter Design Cycle
- Lower Design Cost
- Lower Production Cost
- Fewer Design Errors
- Easier to Explore Design Space
- Easier to Document
- Easier to Change

12.2 Algorithmic Synthesis Tasks
12.2 Algorithmic Synthesis Tasks - Compilation

entity SYNEX1 is
  port (A, B, C, D, E: in INTEGER;
       X, Y: out INTEGER);
end SYNEX1;

architecture HIGH_LEVEL of SYNEX1 is
begin
  X <= E*(A+B+C);
  Y <= (A+C)*(C+D);
end HIGH_LEVEL;

12.2 Algorithmic Synthesis Tasks - Scheduling

• Place each operation in the appropriate control step (clock cycle)
• Consider a schedule using 3 adders and 1 multiplier
12.2 Algorithmic Synthesis Tasks - Allocation

• Specify the components and the interconnections between them
12.2 Algorithmic Synthesis Tasks - Allocation of Functional Units

- Bottom up allocation selects items from a library

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>3</td>
</tr>
<tr>
<td>Multiplier</td>
<td>1</td>
</tr>
<tr>
<td>Register</td>
<td>5</td>
</tr>
<tr>
<td>2 x 1 MUX</td>
<td>4</td>
</tr>
<tr>
<td>3 x 1 MUX</td>
<td>2</td>
</tr>
</tbody>
</table>

12.2 Algorithmic Synthesis Tasks - Allocation of Data Paths

[Diagram of data paths and components]
12.2 Algorithmic Synthesis Tasks - Scheduling and Allocation Interaction

- Schedules are dependent on numbers of functional units.
- Schedules are dependent on the number of clock cycles needed by a functional unit.
- Consider a constraint of having only one multiplier and one adder (The common term $A + C$ has been used here)

Component Quantity

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>1</td>
</tr>
<tr>
<td>Multiplier</td>
<td>1</td>
</tr>
<tr>
<td>Register</td>
<td>5</td>
</tr>
<tr>
<td>2 x 1 MUX</td>
<td>7</td>
</tr>
</tbody>
</table>


12.2 Algorithmic Synthesis Tasks - Allocation for Constrained Schedule

Component Quantity

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD1</td>
<td>--</td>
</tr>
<tr>
<td>MUX</td>
<td>--</td>
</tr>
</tbody>
</table>

Table:

<table>
<thead>
<tr>
<th>s0</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD1</td>
<td>--</td>
<td>b</td>
<td>b</td>
<td>--</td>
</tr>
<tr>
<td>MUX</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>b</td>
</tr>
</tbody>
</table>
12.2 Algorithmic Synthesis Tasks - Data Path for Constrained Schedule

architecture FSM of FSMEX1 is
  type STATE_TYPE is (S0, S1, S2, S3, S4);
  signal STATE: STATE_TYPE;
  signal R1, R2, R3, R4, R5: INTEGER;
begin
  -- Process to update state and perform register transfers.
  STATEP: process (CLK)
  begin
    if CLK'event and CLK='1' then
      case STATE is
        when S0 => R5 <= E; R4 <= B; R3 <= D; R2 <= C; R1 <= A;
          STATE <= S1;
        when S1 => R1 <= R1 + R2; STATE <= S2;
        when S2 => R2 <= R2 + R3; STATE <= S3;
        when S3 => R1 <= R4 + R1;
        when S4 => R4 <= R1 * R2; STATE <= S4;
        end case;
      end if;
    end process STATEP;
    X <= R3; Y <= R4;
  end FSM;
12.3 Scheduling Techniques

- Two approaches
  - Transformational Scheduling
  - Iterative/Constructive Scheduling
    - ASAP Scheduling
    - ALAP Scheduling
    - List Scheduling
    - Freedom-Directed Scheduling

- Transformational Scheduling
  - Take schedule and transform it, one example transformation is state splitting
  - It is prohibitively expensive to try all possible transformations.
12.3 Scheduling Techniques - ASAP Scheduling

For each control step (from inputs)
Schedule as many operations as possible for which data is available

\[ \text{Diagram showing ASAP scheduling steps} \]

12.3 Scheduling Techniques - ASAP Example

\[ \begin{align*}
S0 \\
S1 \\
S2 \\
S3 \\
S4 \\
S5 \\
S6
\end{align*} \]
12.3 Scheduling Techniques - ALAP Scheduling

While there is more work (from outputs)
Schedule as many operations as possible
Increment the number of control steps

12.3 Scheduling Techniques - ALAP Example

S0
S1
S2
S3
S4
S5
S6
12.3 Scheduling Techniques - List Scheduling

- The problem with ASAP and ALAP is that only local information is used
- List scheduling exploits global information
- First, define a critical path as a path in the data flow graph of maximum length.
- Also, assign a priority of the node which is the length of the longest path from the node to the bottom of the graph

**Algorithm**

Construct a list in descending priority
For each control step
    Schedule as many operations as possible from the head of the list

12.3 Scheduling Techniques - List Scheduling Example

S0

S1

S2

S3

S4

S5

S6
12.3 Scheduling Techniques - Freedom-Directed Scheduling

- This technique uses global information both in operation and control step selection.
- **Algorithm**
  Perform ASAP and ALAP schedules, obtaining range of control steps for each operation with an upper bound on the number of control steps
  For all operations
  Schedule the one with the least freedom
12.3 Scheduling Techniques - Freedom-Directed Example

S0
S1
S2
S3
S4
S5
S6

12.4 Allocation Techniques

- Two types
  - Global
    - Exhaustive Search - useful only on sub-problems
  - Iterative/Constructive Techniques
    - Greedy Allocation
    - Left Edge Algorithm
12.4 Allocation Techniques - Greedy Allocation

For each node from top to bottom
  Assign each operation to the next available functional unit, add a new resource if all are busy
  Assign each data value to the next available register, add a new resource if all are busy
  Assign each data path to the next available bus or multiplexer, add a new resource if all are busy

12.4 Allocation Techniques - Left Edge Algorithm

- One possibility is to assign registers first and then consider other allocations
- One algorithm commonly used for register allocation is the left edge algorithm
12.4 Allocation Techniques - The Left Edge Algorithm Continued

Uniquely label each storage device
Prepare data storage lifetime chart
Sort the lifetimes first by starting time (ascending), then by ending time (ascending)
Order the available registers
For each data value
   Assign data value to the highest priority available register

---

12.4 Allocation Techniques - Left-Edge Algorithm Example

[Diagram showing allocation]
12.4 Allocation Techniques - Making the Assignments

- Look again at the schedule given with the constraint of two adders and two multipliers

![Diagram showing allocation of adders and multipliers]

12.4.3 Allocation Techniques - Making the Assignments (continued)

<table>
<thead>
<tr>
<th>AD1</th>
<th>AD2</th>
<th>MU1</th>
<th>MU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>a</td>
<td>b</td>
<td>Op</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AD1</th>
<th>AD2</th>
<th>MU1</th>
<th>MU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>a</td>
<td>b</td>
<td>Op</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Table showing assignments]

![Table showing assignments]

Page 31 of 46
12.4 Allocation Techniques - Final Connection Diagram (Alternate 1)

12.4 Allocation Techniques - Final Connection Diagram (Alternate 2)
12.4 Allocation Techniques - Remaining Sections

- 12.4.5 Analysis of the Allocation Process - omit for now
- 12.4.6 Nearly Minimal Cluster Partitioning Algorithm - omit for now
- 12.4.7 Profit Directed Cluster Partitioning Algorithm (PDCPA) - omit for now

12.5 State of the Art in High-Level Synthesis

- Each step is hard, the complete problem, including interactions, is extremely hard.
- Two promising approaches
  - Expert knowledge guidance
  - Narrow the problem domain, i.e., DSP, microprocessors, pipelined designs
- Another problem is design verification.
12.6 Automated Synthesis of VHDL Constructs

- Constructs that Involve Selection
  - Mapping case Statements to Multiplexers
  - Mapping if..then..else Statements to Multiplexers
  - Mapping Indexed Vector References to Multiplexers
- Loop Constructs
- Functions and Procedures

```vhdl
package TYPES is
  attribute ENCODING: STRING;
  type ENUM is (A, B, C, D);
  attribute ENCODING of ENUM: type is "00 01 10 11";
end TYPES;

MUX1: process (CHOICE, X, Y)
begin
  case CHOICE is
    when A => Z1 <= X;
    when B => Z1 <= Y;
    when C => Z1 <= not X;
    when D => Z1 <= not Y;
  end case;
end process MUX1;
```
12.6 Automated Synthesis of VHDL - Mapping if..then..else to MUXes

MUX2: process (X, Y, VECT)
begin
    if X = '1' then
        Z2 <= VECT(3);
    elsif Y = '1' then
        Z2 <= VECT(2);
    else
        Z2 <= VECT(1) and VECT(0);
    end if;
end process MUX2;

12.6 Automated Synthesis of VHDL - Mapping Indexed Vectors to MUXes

MUX3: process (VECT, INDEX)
begin
    Z3 <= VECT(INDEX);
end process MUX3;
end MUX_CONSTRUCTS;
12.6 Automated Synthesis of VHDL - Loop Constructs (XOR)

entity XOR4 is
  port (A: in BIT_VECTOR (3 downto 0);
        X: out BIT);
end XOR4;
architecture XOR4_LOOP of XOR4 is
begin
  process (A)
  variable X_INT: BIT;
  begin
    X_INT := '0';
    for I in 0 to 3 loop
      X_INT := X_INT xor A(I);
    end loop;
    X <= X_INT;
  end process;
end XOR4_LOOP;

12.6 Automated Synthesis of VHDL - Loop Constructs (XOR 2)

entity XOR4 is
  port (A: in BIT_VECTOR (3 downto 0);
        X: out BIT);
end XOR4;
architecture XOR4_SPACE of XOR4 is
begin
  process (A)
  variable X_INT: BIT_VECTOR (4 downto 0);
  begin
    X_INT(0) := '0';
    for I in 0 to 3 loop
      X_INT(I+1) := X_INT(I) xor A(I);
    end loop;
    X <= X_INT(4);
  end process;
end XOR4_SPACE;
12.6 Automated Synthesis of VHDL - Loop Constructs (Adder)

**Entity ADD4**

```vhdl
entity ADD4 is
  port (A, B: in BIT_VECTOR (3 downto 0);
        CIN: in BIT;
        S: out BIT_VECTOR (3 downto 0);
        COUT: out BIT);
end ADD4;
```

**Architecture LOOP_ADDER**

```vhdl
architecture LOOP_ADDER of ADD4 is
begin
  process (A, B, CIN)
  variable CARRY: BIT := '0'; variable SUM: BIT_VECTOR (3 downto 0);
  begin
    CARRY := CIN;
    for I in 0 to 3 loop
      SUM(I) := A(I) xor B(I) xor CARRY;
      CARRY := (A(I) and B(I)) or (A(I) and CARRY) or
                (B(I) and CARRY);
    end loop;
    S <= SUM; COUT <= CARRY;
  end process;
end LOOP_ADDER;
```

12.6 Automated Synthesis of VHDL - Loop Constructs (Adder 2)

**Architecture SPACE_ADDER**

```vhdl
architecture SPACE_ADDER of ADD4 is
begin
  process (A, B, CIN)
  variable CARRY: BIT_VECTOR (4 downto 0) := "00000";
  variable SUM: BIT_VECTOR (3 downto 0);
  begin
    CARRY(0) := CIN;
    for I in 0 to 3 loop
      SUM(I) := A(I) xor B(I) xor CARRY(I);
      CARRY(I+1) := (A(I) and B(I)) or (A(I) and CARRY(I))
                   or (B(I) and CARRY(I));
    end loop;
    S <= SUM;
    COUT <= CARRY(4);
  end process;
end SPACE_ADDER;
```
12.6 Automated Synthesis of VHDL - Functions (Adder)

architecture FUNCTION_ADDER of ADD4 is
function FA_S (AIN, BIN, CIN: BIT) return BIT is
begin
return AIN xor BIN xor CIN;
end FA_S;

function FA_C (AIN, BIN, CIN: BIT) return BIT is
begin
return (AIN and BIN) or (AIN and CIN) or (BIN and CIN);
end FA_C;
begin
process (A, B, CIN)
variable CARRY: BIT_VECTOR (4 downto 0) := "00000";
variable SUM: BIT_VECTOR (3 downto 0) := "0000";
begin
CARRY(0) := CIN;
for I in 0 to 3 loop
SUM(I) := FA_S(A(I), B(I), CARRY(I));
CARRY(I+1) := FA_C(A(I), B(I), CARRY(I));
end loop;
S <= SUM; COUT <= CARRY(4); end process;
end FUNCTION_ADDER;

12.6.3 Automated Synthesis of VHDL - Procedure (Adder)

architecture PROCEDURE_ADDER of ADD4 is
procedure FA(AIN, BIN, CIN: in BIT;
SOUT, COUT: out BIT) is
begin
SOUT := AIN xor BIN xor CIN;
COUT := (AIN and BIN) or (AIN and CIN) or (BIN and CIN);
end FA;
begin
process (A, B, CIN)
variable CARRY: BIT_VECTOR (4 downto 0) := "00000";
variable SUM: BIT_VECTOR (3 downto 0) := "0000";
begin
CARRY(0) := CIN;
for I in 0 to 3 loop
FA(A(I), B(I), CARRY(I), SUM(I), CARRY(I+1));
end loop;
S <= SUM; COUT <= CARRY(4); end process;
end PROCEDURE_ADDER;