CPE 426/526
Spring 2003
Chapter 12 Half-Notes
Part 2
Dr. Rhonda Kay Gaede

UAH

Freedom Information

<table>
<thead>
<tr>
<th>Operation</th>
<th>Earliest ASAP</th>
<th>Latest ALAP</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
12.4 Allocation Techniques

- Two types
  - 12.4.1 Global
    - Exhaustive Search - useful only on sub-problems
  - 12.4.2 Iterative/Constructive Techniques
    - 12.4.2.1 Greedy Allocation
    - 12.4.2.2 Left Edge Algorithm
12.4.2.1 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

12.4.2.2 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
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

12.4.2.2  Left-Edge Algorithm Example

[Diagram showing the Left-Edge Algorithm example with nodes and edges labeled with numbers and letters, indicating the flow of the algorithm.]
12.4.3 Assigning Functional Units and Interconnection Paths

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

![Diagram showing functional units and connections](image-url)
12.4.3 Final Connection Diagram
(Alternate 1)

\[
\begin{array}{c}
\text{AD1} \\
\text{ME1} \\
R1 \\
\text{AD2} \\
\text{ME2} \\
R2 \\
\text{AD1} \\
\text{ME1} \\
R3 \\
\text{AD2} \\
\text{ME2} \\
R4 \\
\text{AD1} \\
\text{ME1} \\
R5 \\
\text{AD2} \\
\text{ME2} \\
R6 \\
\end{array}
\]

12.4.3 Final Connection Diagram
(Alternate 2)

\[
\begin{array}{c}
\text{AD1} \\
\text{ME1} \\
R1 \\
\text{AD2} \\
\text{ME2} \\
R2 \\
\text{AD1} \\
\text{ME1} \\
R3 \\
\text{AD2} \\
\text{ME2} \\
R4 \\
\text{AD1} \\
\text{ME1} \\
R5 \\
\text{AD2} \\
\text{ME2} \\
R6 \\
\end{array}
\]
12.4 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

- 12.6.1 Constructs that Involve Selection
  - 12.6.1.1 Mapping case Statements to Multiplexers
  - 12.6.1.2 Mapping if..then..else Statements to Multiplexers
  - 12.6.1.3 Mapping Indexed Vector References to Multiplexers

- 12.6.2 Loop Constructs

- 12.6.3 Functions and Procedures

12.6.1.1 Mapping case Statements to Multiplexers

```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;
```
12.6.1.2 Mapping \texttt{if..then..else} Statements to Multiplexers

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;

12.6.1.3 Mapping Indexed Vector Statements to Multiplexers

MUX3: process (VECT, INDEX)
begin
  Z3 <= VECT(INDEX);
end process MUX3;
end MUX_CONSTUCTS;
12.6.2 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;

12.6.2 Loop Constructs with Space-time Transformation (XOR)

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;
12.6.2 Loop Constructs (Adder)

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 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;

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;
12.6.3 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 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;