CprE / ComS 583 Reconfigurable Computing

Prof. Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University

Lecture #20 – Retiming

Quick Points

- HW #4 due today at 12:00pm
- Midterm, HW #3 graded by Wednesday
- Upcoming deadlines:
  - November 15 – project status updates
  - December 4,6 – project final presentations
  - December 14 – project write-ups due

Recap – Variables

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Numbits IS
   PORT ( X      : IN  STD_LOGIC_VECTOR(1 TO 3) ;
          Count  : OUT INTEGER RANGE 0 TO 3) ;
END Numbits ;
```


Variables – Example

```
ARCHITECTURE Behavior OF Numbits IS

BEGIN
   PROCESS(X) – count the number of bits in X equal to 1
        VARIABLE Tmp: INTEGER;
   BEGIN
      Tmp := 0;
      FOR i IN 1 TO 3 LOOP
           IF X(i) = '1' THEN
             Tmp := Tmp + 1;
           END IF;
      END LOOP;
      Count <= Tmp;
   END PROCESS;

END Behavior ;
```


Variables – Features

- Can only be declared within processes and subprograms (functions & procedures)
- Initial value can be explicitly specified in the declaration
- When assigned take an assigned value immediately
- Variable assignments represent the desired behavior, not the structure of the circuit
- Should be avoided, or at least used with caution in a synthesizable code

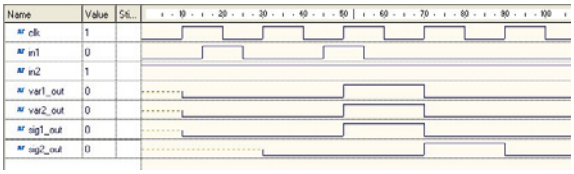
Variables vs. Signals

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY test_delay IS
   PORT(
        clk : IN STD_LOGIC;
        in1, in2 : IN STD_LOGIC;
        var1_out, var2_out : OUT STD_LOGIC;
        sig1_out : BUFFER STD_LOGIC;
        sig2_out : OUT STD_LOGIC
      );
END test_delay;
```

Variables vs. Signals (cont.)

```
ARCHITECTURE behavioral OF test_delay IS
BEGIN
    PROCESS(clk) IS
        VARIABLE var1, var2: STD_LOGIC;
    BEGIN
        if (rising_edge(clk)) THEN
            var1 := in1 AND in2;
            var2 := var1;

            sig1_out <= in1 AND in2;
            sig2_out <= sig1_out;
        END IF;

        var1_out <= var1;
        var2_out <= var2;

    END PROCESS;

END behavioral;
```


Simulation Result



Assert Statements

- Assert is a **non-synthesizable** statement whose purpose is to write out messages on the screen when problems are found during simulation
- Depending on the **severity of the problem**, the simulator is instructed to continue simulation or halt
- Syntax:
  - ASSERT condition [REPORT "message"] [SEVERITY severity_level];
  - The message is written when the condition is FALSE
  - Severity_level can be: Note, Warning, Error (default), or Failure

Array Attributes

A'left(N)	left bound of index range of dimension N of A
A'right(N)	right bound of index range of dimension N of A
A'low(N)	lower bound of index range of dimension N of A
A'high(N)	upper bound of index range of dimension N of A
A'range(N)	index range of dimension N of A
A'reverse_range(N)	index range of dimension N of A
A'length(N)	length of index range of dimension N of A
A'ascending(N)	true if index range of dimension N of A is an ascending range, false otherwise

Subprograms

- Include **functions** and **procedures**
- Commonly used pieces of code
- Can be placed in a library, and then reused and shared among various projects
- Use only sequential statements, the same as processes
- Example uses:
  - Abstract operations that are repeatedly performed
  - Type conversions

Functions – Basic Features

- Always return a single value as a result
- Are called using formal and actual parameters the same way as components
- Never modify parameters passed to them
- Parameters can only be constants (including generics) and signals (including ports);
- Variables are not allowed; the default is a CONSTANT
- When passing parameters, no range specification should be included (for example no RANGE for INTEGERS, or TO/DOWNTO for STD_LOGIC_VECTOR)
- Are always used in some expression, and not called on their own

Function Syntax and Example

```
FUNCTION function_name (<parameter_list>)
RETURN data_type IS
   [declarations]
BEGIN
  (sequential statements)
END function_name;
```

```
FUNCTION f1
 (a, b: INTEGER; SIGNAL c: STD_LOGIC_VECTOR)
RETURN BOOLEAN IS
BEGIN
  (sequential statements)
END f1;
```


Procedure Syntax and Example

```
PROCEDURE procedure_name (<parameter_list>) IS
   [declarations]
BEGIN
  (sequential statements)
END procedure_name;
```

```
PROCEDURE p1
 (a, b: in INTEGER; SIGNAL c: out STD_LOGIC)
   [declarations]
BEGIN
  (sequential statements)
END p1;
```


Problem

- **Given**: clocked circuit
- **Goal**: minimize clock period without changing (observable) behavior
- *I.e.* minimize maximum delay between any pair of registers
- **Freedom**: move placement of internal registers

Procedures – Basic Features

- Do not return a value
- Are called using formal and actual parameters the same way as components
- May modify parameters passed to them
- Each parameter must have a mode: IN, OUT, INOUT
- Parameters can be constants (including generics), signals (including ports), and variables
- The default for inputs (mode in) is a constant, the default for outputs (modes out and inout) is a variable
- When passing parameters, range specification should be included (for example RANGE for INTEGERS, and TO/DOWNTO for STD_LOGIC_VECTOR)
- Procedure calls are statements on their own

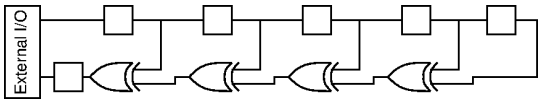
Outline

- Recap
- Retiming
  - Performance Analysis
  - Transformations
  - Optimizations
- Covering + Retiming

Other Goals

- Minimize number of registers in circuit
- Achieve target cycle time
- Minimize number of registers while achieving target cycle time

Simple Example

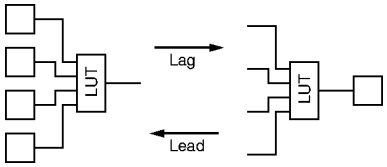


Path Length (L) = 4

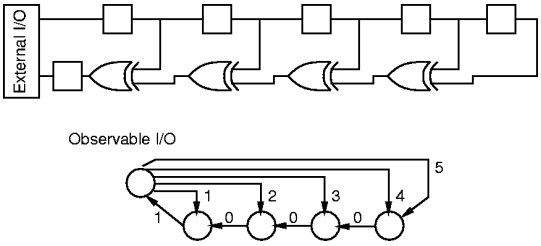
Can we do better?

Legal Register Moves

- Retiming Lag/Lead



Canonical Graph Representation

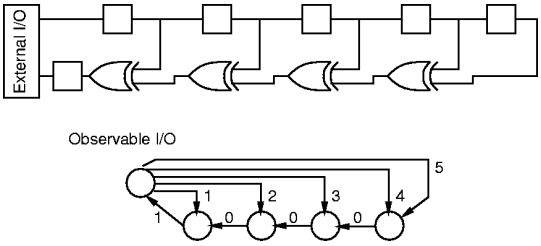


Separate arch for each path

Weight edges by number of registers

(weight nodes by delay through node)

Critical Path Length

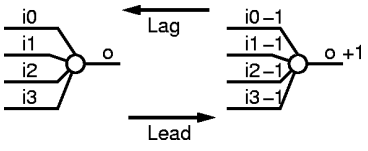


Critical Path: Length of longest path of zero weight nodes

Compute in O(|E|) time by levelizing network:

Topological sort, push path lengths forward until find register.

Retiming Lag/Lead



Retiming: Assign a lag to every vertex

weight(e′) = weight(e) + lag(head(e))-lag(tail(e))

Valid Retiming

- Retiming is valid as long as:
  - ∀e in graph
    - weight(e′) = weight(e) + lag(head(e))-lag(tail(e)) ≥ 0
- Assuming original circuit was a valid synchronous circuit, this guarantees:
  - Non-negative register weights on all edges
    - No traveling backward in time :-)
  - All cycles have strictly positive register counts
  - Propagation delay on each vertex is non-negative (assumed 1 for today)

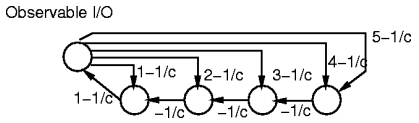
Retiming Task

- Move registers ≡ assign lags to nodes
  - Lags define all locally legal moves
- Preserving non-negative edge weights
  - (previous slide)
  - Guarantees collection of lags remains consistent globally

Optimal Retiming

- There is a retiming of
  - graph *G*
  - w/ clock cycle *c*
  - *iff* *G*-1/*c* has no cycles with negative edge weights
- *G*-α ≡ subtract α from each edge weight

G-1/*c*



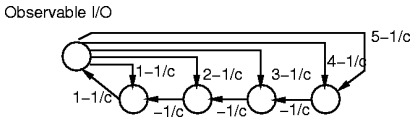
Intuition

- Must have at most *c* delay between every pair of registers
- So, count 1/c'th charge against register for every delay without out
  - (G provides credit of 1 register every time one passed)

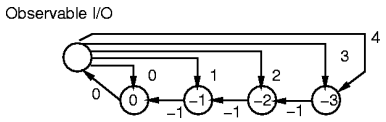
Compute Retiming

- Lag(v) = shortest path to I/O in *G*-1/*c*
- Compute shortest paths in O(|V||E|)
  - Bellman-Ford
  - also use to detect negative weight cycles when *c* too small

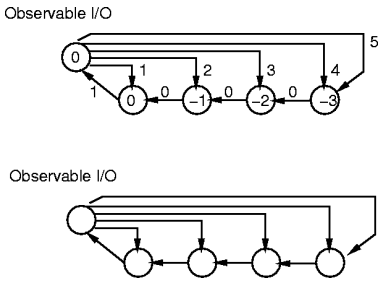
Apply to Example



Apply: Find Lags

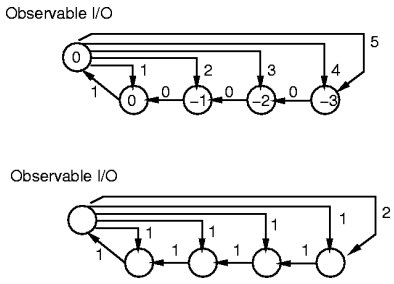


Apply: Move Registers

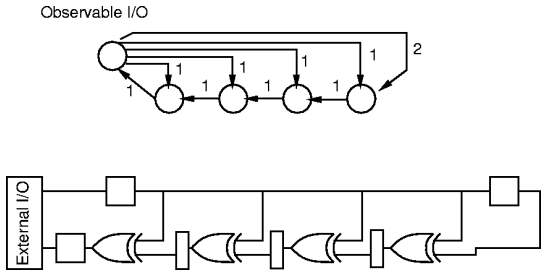


weight(e′) = weight(e) + lag(head(e))-lag(tail(e))

Apply: Retimed



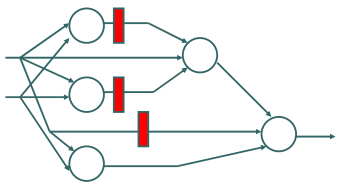
Apply: Retimed Design



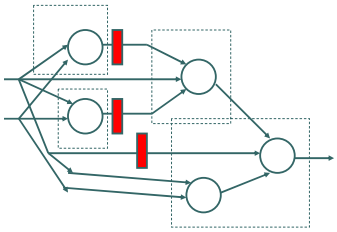
Pipelining

- Can use this retiming to pipeline
- Assume have enough (infinite supply) of registers at edge of circuit
- Retime them into circuit
- See [WeaMar03A] for details

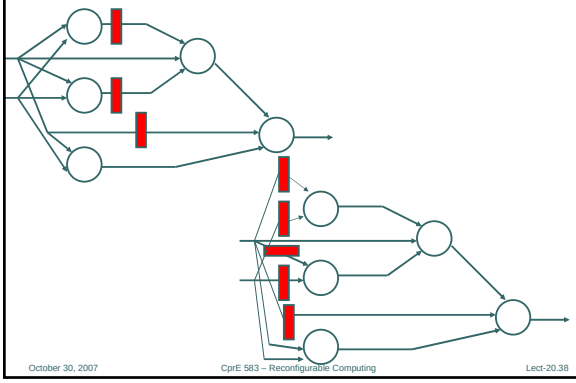
Cover + Retiming – Example



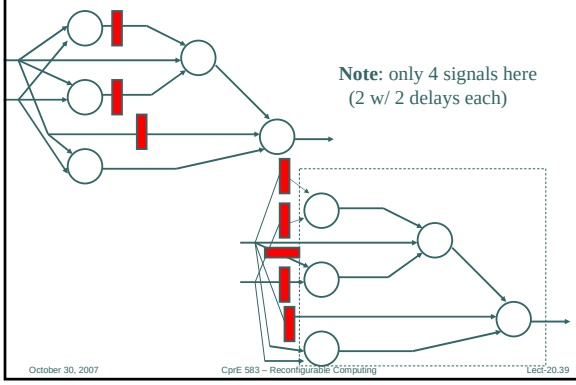
Cover + Retiming – Example (cont.)



Example: Retimed



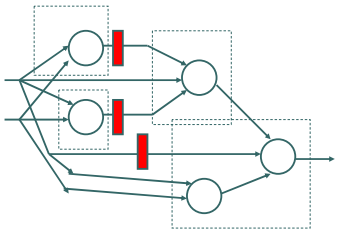
Example: Retimed (cont.)



Note: only 4 signals here
(2 w/ 2 delays each)

Basic Observation

- Registers break up circuit, limiting coverage
  - fragmentation
  - prevent grouping

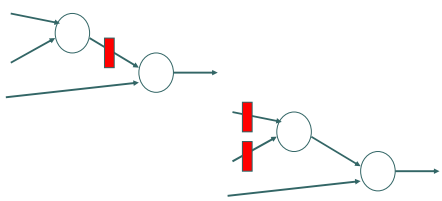


Phase Ordering Problem

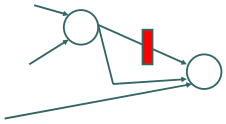
- General problem we've seen before
  - E.g. placement – don't know where connected neighbors will be if unplaced
  - Don't know effect/results of other mapping step
- In this case
  - Don't know delay (what can be packed into LUT) if retime first
  - If not retime first
    - fragmention: forced breaks at bad places

Observation #1

- Retiming flops to input of (fanout free) subgraph is trivial (and always doable)
  - Can cover *ignoring* flop placement
  - Then retime LUTs to input

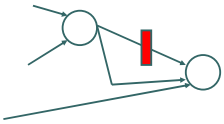


Fanout Problem?



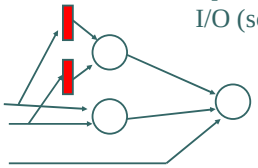
Can I use the same trick here?

Fanout Problem? (cont.)



Cannot retime without replicating

Replicating increase I/O (so cut size)



Summary

- Can move registers to minimize cycle time
- Formulate as a lag assignment to every node
- Optimally solve cycle time in O(|V||E|) time

- Can optimally solve
  - LUT map for delay
  - Retiming for minimum clock period
  - Solving separately does not give optimal solution to problem