ECE 327 – Lecture Notes

# VHDL Simulation

## Delta-Cycle Simulation

* **Delta delay**: infinitesimally small artificial unit of time
  + During each delta cycle:
    - Sample inputs – check if changed
    - Compute results if input changed
    - Drive results
  + When output stabilizes, increment time
* **Projected assignment**: new value remains invisible until next clock cycle

## RTL Simulation

* Label each process as **timed**, **clocked**, or **combinational**
* For each combinational process:
  + Divide process such that each only has single target value (output)
  + Sort in topological order (based on dependencies)
* During each clock cycle:
  + Run timed processes in **any order**, using old values of all signals
  + Run clocked processes in **any order**, using new values of timed signals and old values from clocked signals
  + Run combinational processes in **topological order**, using new values of all signals

# VHDL Synthesis

* Hardware is a concurrent, parallel description of an algorithm
  + VHDL **recursion** generates parallel hardware at compile-time
* Hardware logic and memory capacity are fixed
  + **No pointers** – must supply entire vectors directly
  + **Termination condition** – exists to ensure that resource use is bounded
* generate loop conditions must be **static** – known at compile-time
  + VHDL creates parallel copies of the function
* **Array sizes** cannot be dynamic– must be statically known at compile-time

* **Memory allocation:**
  + VHDL supports this with access types
  + Operations on memory become signals in VHDL
* **Conditional execution**:
  + Handled in hardware via multiplexers
    - if-then-else in sequential statements (e.g. in processes)
    - when-else in concurrent statements
  + If conditional statements are incomplete, will generate a **latch**

## Synthesizable vs. Unsynthesizable Code

* VHDL architecture contains **concurrent** statements that run in parallel
  + Concurrent statements:
    - Conditional: when-else
    - Select: with-select
    - Generate: for-generate, if-generate
    - Processes
    - Component instantiation
* **Processes** are concurrent statements that contain **sequential** statements in their bodies
  + Must have either a **sensitivity list** or at least one **wait** statement – cannot have both
  + Sequential statements:
    - Waits: wait until rising\_edge(clk)
    - Signal assignment: sig <= ‘1’
    - Conditional: if-then-else, case(sig) is
    - Loops: loop, while loop, for loop

## Combinational vs. Clocked Processes

**Combinational Processes**

* Must have sensitivity list
* Must not have any wait statements
* Must not have any rising\_edge/falling\_edge statements

**Clocked Processes**

* Target signals are outputs of flip-flops
* Must contain one or more wait or rising\_edge/falling\_edge statements

## Flip-Flops, Latches, and Combinational Circuitry

* If there are wait or rising\_edge/falling\_edge statements before the assignment, the signal is a **flip-flop**
* If the signal doesn’t appear in all branches of an if-then-else statement, it is a **latch**
* Otherwise, it is a **combinational gate**

# VHDL State Machines

# FPGAs

* Logic packed into **lookup tables** (LUTs)
  + Boolean table mapped to SRAM cells
  + A -input LUT has cells
* **Crossbars** allow the outputs of LUTs to connect to any LUT inputs
  + Each cross-point in the crossbar is one transistor + SRAM cell
  + For inputs and outputs, need SRAM cells
    - E.g. for an 8-input, 32-output switch, need 32 8:1 muxes; each mux can be implemented as one row in the crossbar

# Dataflow Pipelining

* **Throughput**: the rate at which the circuit can consume inputs
  + Highest throughput is 1, if it can accept new inputs during every clock cycle
  + **IPC**: instructions per cycle
  + **CPI**: clock cycles per instruction
* **Latency**: the number of clock cycles between the input and output
* **Clock period**: amount of time separation between clock edges

# Dataflow Scheduling

## Fully Spatial Design

* Use as many arithmetic operators as needed
* Use as many pipelining stages as required
* Cost doesn’t matter

## Fully Sequential Design

* Only have one of each – arithmetic operators, register, IO port
* Enable sharing of elements with multiplexers
* Low-cost design

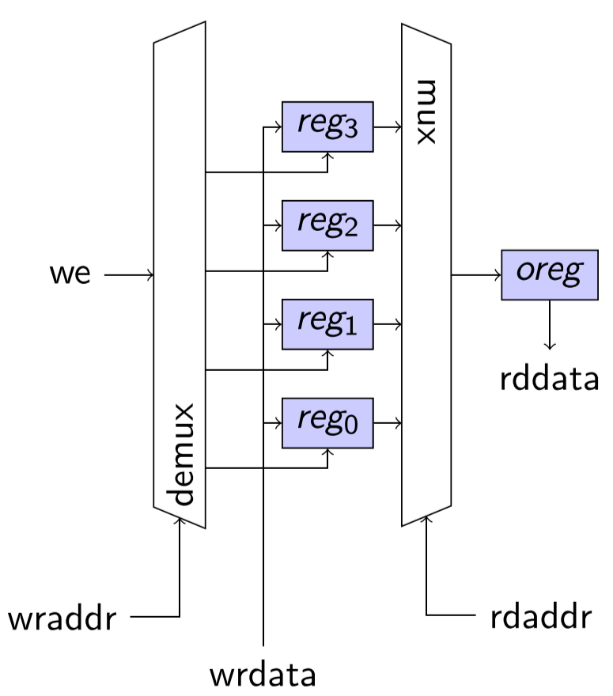
## Dataflow Analysis

* First, determine system constraints – how many of each resource do we have?
  + Number of each arithmetic operator
  + Number of IO ports
  + Number of registers
* Analyze how many of each resource the **fully spatial** design requires
* Put a multiplexer in front of every resource that is less than needed

Post-Midterm Material

# 8a: Memories

## Memory Operations



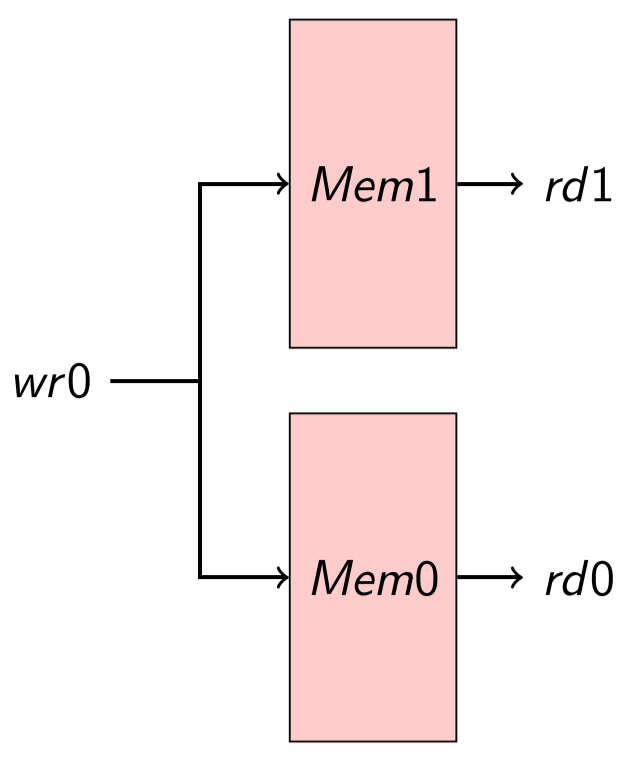
* **Write enable**: demux enabled by **write address**
* **Storage**: registers connected to **write data**, enabled by demux
* Output: mux enabled by **read address**
* Output register:outputs to **read data**

## Memory Behaviour

* Read and writes can happen in parallel
* Reads happen all the time at the **rdaddr** location (no read enable needed)
  + Read data is available one clock cycle after address is provided, or data changes in register
* Write need an explicit write enable
  + Write data is stored in register one clock cycle after write enable goes high
* End-to-end write to and read from the same memory address: 2 clock cycles
* What happens when you read and write to the same address in the same cycle?
  + Read data will be old data, since write data will be stored in register one clock cycle after

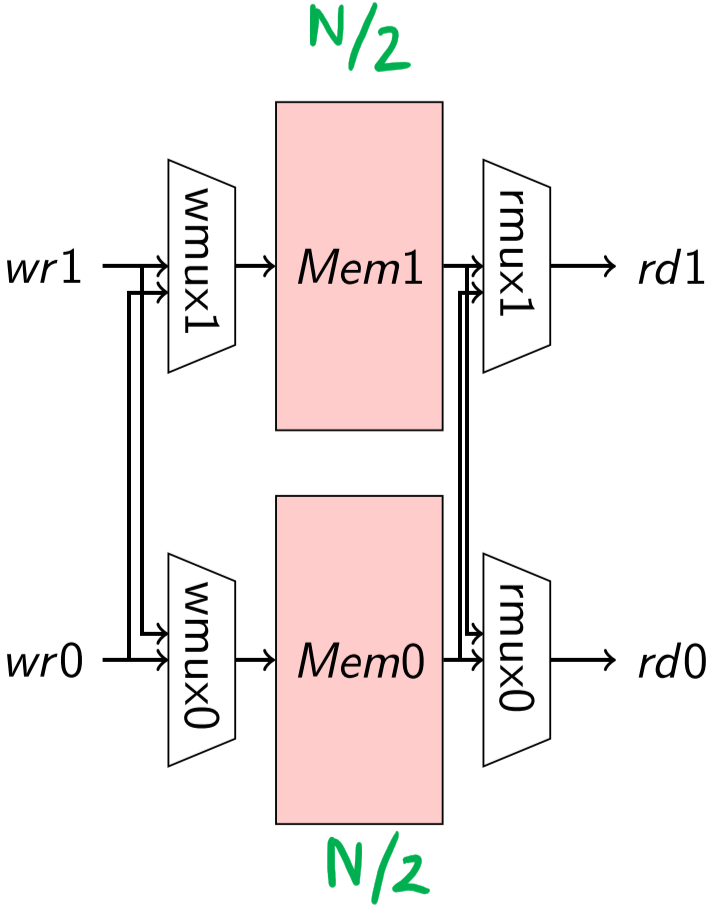
# 8a: Advanced Memories

## Multiple Read Ports



* Duplicate memory
* Only one write port, which performs the write operation twice

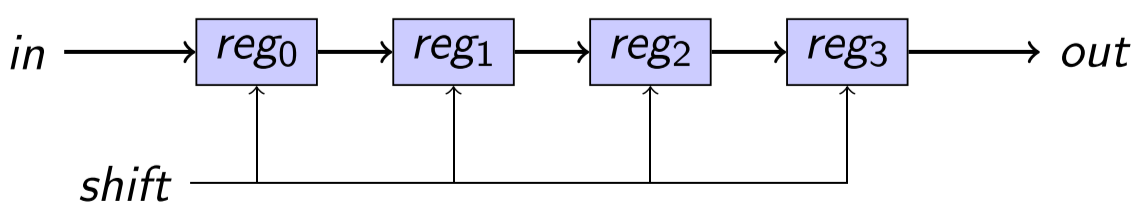
## Multiple Write Ports





* Use **banking** – partition memory of size into banks of size
  + Cannot do multiple writes or reads to the same bank
* All banks are connected to both write enable and read data muxes

## Shift Registers



* Fixed-size memory that adds storage and latency to a signal
* Constant (minimum) latency of data that passes through the shift register
* Useful when performing arithmetic over a window of inputs

## FIFO Memories

* Decouples production and consumption of data inside a chip
* Ideal for connecting hardware components with loose timing behaviour
* Implementation:
  + Memory + read/write address counters
    - Every write, increment write counter
    - Every read, increment read counter
    - Read address can never be greater than write address, write address can never be less than read address
  + Status indicators full and empty
    - When full, no writes possible
    - When empty, no reads possible
    - Computed by arithmetic on read and write counters