Topics

- Hardware modeling and simulation
  - Event-driven simulation
- Basics of register-transfer design:
  - data paths and controllers;
  - ASM charts.
- High-level synthesis

Verilog

- Initially a proprietary language, by *Gateway Design Automation* (a company later acquired by *Cadence*); introduced in 1984.
- Released to the public domain in 1990.
- New IEEE standard in 2001 with many enhancements (but backwards compatibility).

Verilog vs. VHDL (1)

- Verilog is weakly typed, VHDL uses strong typing (*wires* do not even to be declared in Verilog);
- No mechanism for introducing *new data types* and use them; no *packages* either;
- Only implicit rules for distinguishing between signed and unsigned arithmetic (solved in Verilog 2001);

Verilog vs. VHDL (2)

- No separation of entity and architecture and no configurations (partially solved in Verilog 2001);
- Very limited file I/O (can only read binary or hexadecimal files encoded in ASCII; solved in Verilog 2001);
- No *generate* loops (solved in Verilog 2001).
- Built-in support for gate-level modeling.
SystemVerilog

- New initiative to support system-level design with Verilog;
- Backwards compatible with Verilog;
- Adds strong typing (for new data types only) and object-oriented programming;
- Support for verification, with mechanisms such as assertions.

Sample C simulator

```c
while (TRUE) {
    switch (state) {
    case S0:
        x = a + b;
        state = S1;
        next;
        case S1:
            ...
    }
}
```

loop executed once per clock cycle

each case corresponds to a state; sets outputs, next state

Simulation coding

- Hardware description languages are typically supported by a simulation system: VHDL, Verilog, etc.
  - Simulation engine takes care of scheduling events during simulation.
- Can hand-code a simulation in a programming language like C.
  - Must be sure that register-transfer events happen in proper order.

C-based design (1)

- In recent years, hardware description based on C/C++ is gaining popularity:
  - it is the most popular programming language, avoids need to learn yet another language for hardware;
  - eases hardware-software co-design by keeping the entire system description in one language.
C-based design (2)

- Many variants of C/C++ for hardware, e.g.:
  - ART-C from Adelante (now ARM, to be used in course VLSI Signal Processing);
  - HANDEL-C from Celoxica;
  - SystemC, an open-source initiative supported by Synopsys, Adelante and many others (see www.systemc.org); it has a built-in event-driven simulator.

Simulation vs. programming

- Simulation tags computations with times.
  - Must know when signals change to properly simulate hardware.
- Simulation is parallel.
  - Many statements can execute at the same (simulation) time.
  - Just like hardware.

Types of simulation

- Compiled-code simulation.
  - Generate program that evaluates a hardware block.
  - Operational details within the hardware block are lost.
- Event-driven simulation.
  - Propagate events through simulation.
  - Don’t simulate a block until its inputs change.

Event-driven simulation

- An event is a change in a net’s value.
- An event has two components:
  - value;
  - time.
Event propagation

- Propagate events only when nets change value.
- If an input change doesn’t cause an output change, no event is propagated.

Timewheel

- The timewheel is a data structure in the simulator that efficiently determines the order of events processed.
- Events are placed on the timewheel in time order.
- Events are taken out of the head of the timewheel to process them in order.

Timewheel operation

- A register-transfer system is a sequential machine.
- Register-transfer design is structural—complex combinations of state machines may not be easily described solely by a large state transition graph.
- Register-transfer design concentrates on functionality, not details of logic design.
Register-transfer system example

A register-transfer machine has combinational logic connecting registers:

Data path-controller systems

- One good way to structure a system is as a data path and a controller:
  - data path executes regular operations (arithmetic, etc.), holds registers with data-oriented state;
  - controller evaluates irregular functions, sets control signals for data path.

Data and control are equivalent

- We can rewrite control into data and visa versa:
  - control: if i1 = ‘0’ then o1 <= a; else o1 <= b; end if;
  - data: o1 <= ((i1 = ‘0’) and a) or ((i1 = ‘1’) and b);
- Data/control distinction is useful but not fundamental.

Data operators

- Arithmetic operations are easy to spot in hardware description languages:
  - x <= a + b;
- Multiplexers are implied by conditionals. Must evaluate entire program to determine which sources of data for registers.
- Multiplexers also come from sharing adders, etc.
Conditionals and multiplexers

if \( x = '0' \) then
\[
\text{reg1} \leftarrow a;
\]
else
\[
\text{reg1} \leftarrow b;
\]
end if;

Alternate data path-controller systems

One controller, one data path

Two communicating data path-controller systems

ASM charts

- An ASM (augmented state machine) chart is a register-transfer description.
- ASM charts let us describe function without choosing a partitioning between control and data.
- Once we have specified the function, we can refine it into a block diagram which partitions data and control.

Sample ASM chart
ASM state

- An ASM state specifies a machine state and a set of actions in that state. All actions occur in parallel.

```
state s1:
x = a + b
y = c - d + e
o1 = 1
```

name of state (notation only)

Actions in state

- Actions in a state are unconditionally executed.
- A state can execute as many actions as you want, but you must eventually supply hardware for all those actions.
- A register may be assigned to only once in a state (single-assignment rule).

Implementing operations in an ASM state

- Sequences of states
  - States are linked by transitions.
  - States are executed sequentially. Each state may take independent actions (including assigning to a variable assigned to in a previous state).
Data paths from states

- Maximum amount of hardware in data path is determined by state which executes the most functionality.
- Function units implementing data operations may be reused across states, but multiplexers will be required to route values to the shared function units.

Function unit sharing example

mux allows + to compute a+b, a+c

Extracting data path and controller

- ASM chart notation helps identify data, control.
- Once you choose what values and operations go into the data path, you can determine by elimination what goes into the controller.
- Structure of the ASM chart gives structure of controller state transition graph.
High-level synthesis

- Sequential operation is not the most abstract description of behavior.
- We can describe behavior without assigning operations to particular clock cycles.
- High-level synthesis (behavioral synthesis) transforms an unscheduled behavior into a register-transfer behavior.

Tasks in high-level synthesis

- **Scheduling**: determines clock cycle on which each operation will occur.
- **Binding (allocation)**: chooses which function units will execute which operations.

Confusing terminology

- Mapping of an operation to a hardware *functional unit* is called *assignment*.
- The task of reserving functional units of the right type is called *allocation* or *module selection*.
- While *scheduling* strictly means mapping to a clock cycle, it is often used to include assignment as well.

Data dependencies

- Data dependencies describe relationships between operations:
  - \( x \leq a + b \); value of \( x \) depends on \( a, b \)
- High-level synthesis must preserve data dependencies.
Data flow graph

- Data flow graph (DFG) models data dependencies.
- Does not require that operations be performed in a particular order.
- Models operations in a basic block of a functional model—no conditionals.
- Requires single-assignment form.

Data flow graph construction

original code:

\[
\begin{align*}
x & \leq a + b; \\
y & \leq a \times c; \\
z & \leq x + d; \\
x & \leq y - d; \\
x & \leq x + c;
\end{align*}
\]

single-assignment form:

\[
\begin{align*}
x_1 & \leq a + b; \\
y & \leq a \times c; \\
z & \leq x_1 + d; \\
x_2 & \leq y - d; \\
x_3 & \leq x_2 + c;
\end{align*}
\]

Goals of scheduling and allocation

- Preserve behavior—at end of execution, should have received all outputs, be in proper state (ignoring exact times of events).
- Utilize hardware efficiently.
- Obtain acceptable performance.
Scheduling example

One feasible schedule for last DFG:

Cycle 1

Cycle 2

Cycle 3

Binding values to registers

registers fall on clock cycle boundaries

Choosing function units

muxes allow function units to be shared for several operations

See book, page 442.

Building the sequencer

sequencer requires three states, even with no conditionals
**Choices during high-level synthesis**

- Scheduling determines number of clock cycles required; binding determines area, cycle time.
- Area tradeoffs must consider shared function units vs. multiplexers, control.
- Delay tradeoffs must consider cycle time vs. number of cycles.

**Finding schedules**

- Two simple schedules:
  - As-soon-as-possible (ASAP) schedule puts every operation as early in time as possible.
  - As-late-as-possible (ALAP) schedule puts every operation as late in schedule as possible.
- Many schedules exist between ALAP and ASAP extremes.

**ASAP and ALAP schedules**

- **ASAP**
  - May execute several operations in sequence in one cycle—operator chaining.
- **ALAP**
  - Critical path

**Operator chaining**

- Delay through function units may not be additive, such as through several adders.
Control implementation

- Clock cycles are also known as control steps.
- Longer schedule means more states in controller.
- Cost of controller may be hard to judge from casual inspection of state transition graph.

Distributed control

- Product machine of FSM1 & FSM2
- One centralized controller
- Two distributed controllers

Synchronized communication between FSMs

To pass values between two machines, must schedule output of one machine to coincide with input expected by the other:

Hardwired vs. microcoded control

- Hardwired control has a state register and “random logic.”
- A microcoded machine has a state register which points into a microcode memory.
- Styles are equivalent; choice depends on implementation considerations.