ECE 587 – Hardware/Software Co-Design
Lecture 28 Final Exam Review

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

April 29, 2015
Final Exam

- Final exam: Wed. May 6, 8:00 AM – 10:00 AM
  - Closed book/notes, cheat sheet allowed
  - Take at this room or contact IIT online for proctor arrangement
Outline

System Synthesis

Software Synthesis

Hardware Synthesis

Verification
System Design Trends

- Traditional: board-based system design
  - Platform definition → HW design → BSP development → Application development → Prototype test
  - Sequential development of HW and SW causes delays and it's difficult to decide who should be accountable for bugs.

- Current: virtual platform based system design
  - Platform definition → Platform modeling → HW design/BSP development → Application development → Prototype test
  - SW and HW are developed in parallel while virtual platform serves as golden for both.
  - Platform definition still requires expertise and experiences.

- Future: model based system design
  - Platform generation → TLM generation → SW/HW synthesis → Prototype test
TLM Based Design Flow

FIGURE 4.4  TLM based design flow.  
(Gajski et al.)
System Synthesis

- Generate the platform if necessary
- Generate the mapping if necessary
- Generate TLM according to application and its mapping on the platform
System Synthesis Flow

FIGURE 4.24  System synthesis from application and constraints.  
(Gajski et al.)
Platform Generation Example

(a) Application graph

(b) Generated platform with mapping

FIGURE 4.26 Illustration of platform generation on a GSM Encoder example. (Gajski et al.)
Automatic Mapping Example

**FIGURE 4.18** Abstraction of profiled statistics into an application graph.

**FIGURE 4.21** Platform graph with communication costs.

(Gajski et al.)
FIGURE 4.13  Automatically Generated TLM from system specification.

(Gajski et al.)
What’s Next?

![Synthesis Overview Diagram](image)

*FIGURE 5.1 Synthesis overview* (Gajski et al.)
Outline

System Synthesis

Software Synthesis

Hardware Synthesis

Verification
Automated Software Synthesis

- Generate embedded software from system TLM
  - The final products are the binaries for processors.
- Benefits
  - No tedious and error-prone manual code writing
  - Demands less processor- and platform-specific knowledge from the designer
  - Each synthesis step can be individually verified.
- Increase productivity by reducing time for development and debugging
Challenges for Software Development/Synthesis

- Coupling to underlying hardware and external processes
  - Embedded software interacts with hardware accelerators.
  - Embedded software interacts with external physical process.
- Timeliness
  - Real-time constraints extend to software implementation
  - Correctness not only means correct functionality, but also the ability to meet deadlines.
  - Predictable execution time is more important than fast execution.
- Concurrency
  - Scheduling with real-time constraints is complicated.
- Resource constraints
  - Memory, computing power, energy consumption, power dissipation, etc.
FIGURE 5.2 Software synthesis flow

(Gajski et al.)
Code Generation

- Input: a single task specified in certain system design language, e.g. SystemC
- Output: a sequential program in a programming language that will be compiled to binaries later, e.g. C
- Translate module compositions and communications into available language constructs.
  - Use data abstractions to represent modules, ports, and their compositions
  - Use function call and global variables to resolve communications within the same task.
  - Other communications are delegated to lower levels.
Concurrency and RTOS

- Concurrency is not available at language level for many programming languages.
- Concurrency is usually provided through OS.
  - Task executions are interleaved on the processor via scheduling.
- For embedded systems, Real-Time Operating System (RTOS) is relevant.
  - General purpose OS typically schedules tasks for fairness, enabling all tasks to have a chance to execute.
  - RTOS emphasizes predictability, enabling execution within predictable bounds.
Characterization of Scheduling Algorithms

- Scheduling algorithm decides which task to gain access to a resource when many are ready.

- Preemptive vs. Non-preemptive
  - Preemptive: a task can be interrupted in the middle of its execution, and be suspended.
  - Non-preemptive: a task may not be interrupted unless it’s calling certain OS services.

- Static vs. Dynamic
  - Whether task scheduling parameters can be updated during runtime.

- Off-line vs. On-line
  - Off-line: the complete schedule for all tasks is determined before executing any task. Often used for hard real-time systems, minimum scheduling overhead but less flexible.
  - On-line: scheduling decisions are made at run-time.
Scheduling Policies

- **Round Robin (RR)**
  - Tasks take turn to execute based on time slices.
  - Guarantee fairness: every task has a chance to run

- **Priority-based scheduling**
  - Tasks are assigned a priority number depending on their importance.
  - The task with the highest priority among those are ready is selected to execute.
  - For on-line scheduling, typically preemptive and allow dynamic priority changes.

- **Earliest Deadline First (EDF)**
  - Require task deadlines to be available for decision making
  - Could be implemented on top of priority-based scheduling using deadlines as the priorities
Multi-Task Synthesis

- For dynamic on-line scheduling
- RTOS-based multi-tasking
  - User tasks are executed on top of an off-the-shelf RTOS and are scheduled by the RTOS scheduler.
  - Task management and synchronization are resolved by the RTOS Abstraction Layer (RAL) API that hides OS details.
  - Preferred when there is enough resource due to its flexibility and maturity
- Interrupt-based multi-tasking
  - Applicable when off-the-shelf RTOS’ are not suitable due to performance and resource constraints.
  - Also similar to how RTOS implements task management and task scheduling.
Interrupt-Based Multi-Tasking

- Specify tasks as FSMs explicitly.
  - Tasks are suspended/resumed after/before state transitions.
  - The states should encode everything including processor state and the stack.
- The task should be reorganized so the states can be introduced at strategic locations to save storage requirement.
  - e.g. it is not wise to introduce a state deep inside a function call hierarchy.
  - Memory constraints may demand to share a single stack among multiple tasks.
  - Executing times for state transitions are also constrained to achieve certain scheduling goals.
Outline

System Synthesis

Software Synthesis

Hardware Synthesis

Verification
Hardware Synthesis

- Synthesize HW components from specification to RTL
  - HW components as standard or custom processors or as special custom hardware units, usually known as intellectual property components (IPs).
  - Untimed specification, e.g. C
  - RTL in HDL for further synthesis

- Via high-level synthesis (HLS, C-to-RTL design)
  - Tasks: allocation/binding/scheduling
  - Also provide estimation of hardware performance metrics
  - A very difficult problem under stringent design constraints: any change in one aspect affects other aspects, trade-offs need to be explored but are not obvious

- Possible HLS flows
  - Complete the there tasks sequentially
  - Pre-allocation: define architecture for HW processor
  - Pre-binding: optimize register usage
  - Pre-scheduling: avoid structural dependency in inner loops
FIGURE 6.1 HW synthesis design flow

(Gajski et al.)
Although the output of HW synthesis is the RTL/FSM model of the component, we can divide it into pieces.

- Facilitate reasonings/communications
- Optimize with specialized algorithms

**Datapath**

- Perform complicate, but usually combinational, computations
- Produce status signals for decision making

**Controller**

- Provide control signals to the datapath, e.g. to collect input data and to distribute output data
- Interact with other components, e.g. to notify completion and to start computation once activated
Controller vs. Datapath: A Detailed Diagram

\textit{FIGURE 6.3} RTL diagram with FSM controller

(Gajski et al.)
Input Model: CDFG

```c
while(1) {
    while(Start == 0);
    Done = 0;
    Data = Input;
    Ocount = 0;
    Mask = 1;
    while(Data > 0) {
        Temp = Data & Mask;
        Ocount = Ocount + Temp;
        Data >>= 1;
    }
    Output = Ocount;
    Done = 1;
}

LISTING 6.2 RTL-based C code
```

(Gajski et al.)
General HLS Flow

- Synthesize CDFG into FSMD: schedule both the dataflow and the control flow w/ or w/o allocation
- Synthesize FSMD into RTL: HLS optimization is much easier when confined to a single clock cycle
Fundamental Scheduling Algorithms

- **ASAP (as-soon-as-possible)**
  - Assume each operation will take one clock cycle to finish
  - Assume an unlimited number of functional units are available
  - Schedule an operation as *soon* as all its operands are available
  - The execution is only constrained by data dependencies but not structural dependencies.
  - We will obtain a schedule with the shortest execution time.

- **ALAP (as-late-as-possible)**
  - Same assumption as ASAP
  - Schedule an operation as *late* as possible, but not too late so that the overall execution will take more time than a given bound.
ASAP and ALAP Scheduling Examples

![Diagram of ASAP and ALAP scheduling examples](image_url)

**FIGURE 6.28**

(a) ASAP  
(b) ALAP  

(Gajski et al.)
The criticality of an operation can be modeled by its *mobility*. 
- Mobility of an operation = Its starting time in ALAP - Its starting time in ASAP
- Critical operations to be those with 0 mobility
- Non-critical operations to be those with positive mobility

Mobility provides a measure to prioritize operations if we cannot schedule all operations that are ready.
- e.g. due to structural dependencies
- Other measures exist. But none is perfect for all cases.
Resource-constrained (RC) scheduling
- When the allocation is provided by the designer, we should follow it and schedule for the best performance.

Time-constrained (TC) scheduling
- When the desired execution time is provided by the designer, we should schedule all the operations within the bound using least amount of resource.
Assume one arithmetic unit and two shift units are available

(Gajski et al.)
FSMD as Output of Scheduling

\[ \text{Start} \]
\[ a = \text{In1} \]
\[ b = \text{In2} \]

\[ \begin{align*}
  t1 &= |a| \\
  t2 &= |b| \\
  x &= \max(t1, t2) \\
  y &= \min(t1, t2) \\
  t3 &= x >> 3 \\
  t4 &= y >> 1 \\
  t5 &= x - t3 \\
  t6 &= t4 + t5 \\
  t7 &= \max(t6, x) \\
\end{align*} \]

\[ \text{Done} = 1 \]
\[ \text{Out} = t7 \]

\[ (\text{Gajski et al.}) \]
Resource Optimization

- Register sharing: group variables with non-overlapping lifetimes into a single register
- Functional unit sharing: group similar operations into a single multifunction unit
- There may exist many ways to group registers/functional units into the minimum number of groups.
  - Break the tie by considering a second design metric
  - e.g. the connectivity cost measured as number of selector inputs
Connection Optimization

- Connection sharing: organize wires into buses to facilitate routing, group connections not used at the same time into buses
- Register merging: organize registers with non-overlapping access times into register files to save ports and connections
Pipelining

- Introduce additional flip-flops/latches into the functional units so the idling parts can be reused for other operations.
  - As if there are additional functional units
  - You may also pipeline a unit for a higher frequency

- Cost of pipelining
  - Additional internal sequential elements
  - A small overhead to clock period

- Complications
  - Pipelining changes cycle-to-cycle behavior of the design.
  - It is generally not possible to pipeline a RTL design so we prefer to apply it in HLS.
A unit can be pipelined into multiple stages

One set of new operands can be introduced per each clock cycle.

- As usual, they should persist throughout that clock cycle.
- Unlike multi-cycling, it is not necessary to maintain the inputs throughout the whole computation.

The output will be ready after designated number of cycles.

- Following the same order you send in the operands
- The same as multi-cycling
Interface Synthesis

- Integrate HW component into the system platform
  - Combine HW processes with communication channels for message transfers between system components.
- Extract code for processes and channels from the system TLM model and feed them to HW synthesis (HLS) flow
  - Special bus interfaces are necessary if HW and bus are on different clock periods.
- It is beneficial to use pre-implemented and tested bus interfaces to bridge HW and bus timing.
  - Allow HW synthesis flow to exploit the flexibility in deciding the timing for untimed code w/o being limited by bus timing
  - Use a special driver layer to interface the synthesized HW and the pre-implemented bus interface
Outline

System Synthesis

Software Synthesis

Hardware Synthesis

Verification
Functional Verification

- Analysis and reasoning on a computer model of the system
  - Before manufacturing
- Establish confidence in functional correctness before the product is shipped
  - Critical for systems where safety is the first concern
  - Prevent costly recall for non-critical systems as well
- Designers need to make sure the model at each step of design does reflect the original intent.
  - Catch bugs as early as they are introduced so one can locate them effectively
Both verification methods need a golden reference as part of the specification.

- Simulation based methods: stimuli (inputs) and monitors (expected outputs)
- Formal methods: mathematical model of desired properties

Simulation-based methods are effective to catch bugs at early design stages, though only as good as the stimuli.

Formal methods can provide a proof of correctness, but require more effort at design time.

Simulation is still predominant while formal methods are catching up.

- More formal verification tools become available.
- More designers are trained to use these tools.
- e.g. most gate-level verifications are based on formal methods nowadays
**FIGURE 7.1** A typical simulation environment

(Gajski et al.)
Performance Considerations

- Coverage may increase as you simulate with more test-cases.
  - However, simulation takes time.
  - Need to trade-off verification performance with quality.
- Stimulus optimization: simulate less cases
  - Use coverage feedback mechanism to improve test-cases that are otherwise generated randomly.
  - Not all test-cases are valid – only simulate with valid ones.
- Monitor optimization: discover bugs faster
  - White box testing – also monitor internal variables.
  - Use assertions instead of golden for internal variables.
  - Communicate more effectively with designers via visualization.
- Speed-up techniques: faster simulation
  - Faster algorithms.
  - Hardware assistance, e.g. on FPGA.
  - Simulate at higher abstraction levels, e.g. TLM, if certain details can be omitted.
Formal Verification Techniques

- Equivalence checking
  - Ensure the correctness of implementations and optimizations
  - Combinational equivalence checking: same output for same input
  - FSM equivalence checking: same output trace for same input trace from certain initial states
  - Cannot be applied if the functional equivalence goes beyond same inputs/outputs per cycle.

- Model checking
  - Safety: bad things never happen
  - Liveness: good things eventually happen

- Theorem proving via (interactive) deductive reasoning

- Bounded model checking

- Symbolic simulation
Thank You and Good Luck!