Reading Assignment

- This lecture: 5.1, 5.2, 5.3
- Next lecture: 5.4
Outline

Software Synthesis

Code Generation
Software synthesis: generate binary code for target platform
Software development dominates the design cost of modern complex multiprocessor systems.

- Complexity of embedded software increases.
  - Overall system complexity increases.
  - Designers prefer software-centric implementations because of productivity and flexibility.

- Automated software synthesis is preferred.
  - Especially for software that is tightly coupled to the underlying hardware.
Software development dominates the design cost of modern complex multiprocessor systems.

- Complexity of embedded software increases.
  - Overall system complexity increases.
  - Designers prefer software-centric implementations because of productivity and flexibility.

- Automated software synthesis is preferred.
  - Especially for software that is tightly coupled to the underlying hardware.
Software development dominates the design cost of modern complex multiprocessor systems.

- Complexity of embedded software increases.
  - Overall system complexity increases.
  - Designers prefer software-centric implementations because of productivity and flexibility.

- Automated software synthesis is preferred.
  - Especially for software that is tightly coupled to the underlying hardware.
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
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
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.
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.
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.
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.
Software Synthesis and Programming Languages

- Software synthesis leverages existing embedded software design tools.
- Embedded software can be generated in existing programming language(s)
  - Instead of binaries, which can be produced by existing tools
  - Allow designers to have better control over the final code
- What languages are available?
- Which one are we going to use?
Software Synthesis and Programming Languages

- Software synthesis leverages existing embedded software design tools.
- Embedded software can be generated in existing programming language(s)
  - Instead of binaries, which can be produced by existing tools
  - Allow designers to have better control over the final code
- What languages are available?
- Which one are we going to use?
Software Synthesis and Programming Languages

- Software synthesis leverages existing embedded software design tools.
- Embedded software can be generated in existing programming language(s)
  - Instead of binaries, which can be produced by existing tools
  - Allow designers to have better control over the final code
- What languages are available?
  - Which one are we going to use?
Software synthesis leverages existing embedded software design tools.

Embedded software can be generated in existing programming language(s)
  ▶ Instead of binaries, which can be produced by existing tools
  ▶ Allow designers to have better control over the final code

What languages are available?
Which one are we going to use?
Programming Languages

- **Assembly**: provide fine-grained control over processors
  - Processor dependent and overly verbose for large projects
- **C**: provide low-level features while being
  - Processor independent, require minimal run-time support
- **C++**: compatible with C, provide higher level abstractions
  - Complicated, large runtime overhead if not used properly
- **Java**: a simplified derivative of C++
  - Prevent unnecessary mistakes (both functional and performance-wise) by being less flexible
  - JVM provides supports of concurrency and communications at language level, though slow speed is a concern.
  - To meet deadlines is challenging due to garbage collection.
Programming Languages

- Assembly: provide fine-grained control over processors
  - Processor dependent and overly verbose for large projects
- C: provide low-level features while being
  - Processor independent, require minimal run-time support
- C++: compatible with C, provide higher level abstractions
  - Complicated, large runtime overhead if not used properly
- Java: a simplified derivative of C++
  - Prevent unnecessary mistakes (both functional and performance-wise) by being less flexible
  - JVM provides supports of concurrency and communications at language level, though slow speed is a concern.
  - To meet deadlines is challenging due to garbage collection.
Programming Languages

- **Assembly**: provide fine-grained control over processors
  - Processor dependent and overly verbose for large projects
- **C**: provide low-level features while being
  - Processor independent, require minimal run-time support
- **C++**: compatible with C, provide higher level abstractions
  - Complicated, large runtime overhead if not used properly
- **Java**: a simplified derivative of C++
  - Prevent unnecessary mistakes (both functional and performance-wise) by being less flexible
  - JVM provides supports of concurrency and communications at language level, though slow speed is a concern.
  - To meet deadlines is challenging due to garbage collection.
Programming Languages

- **Assembly**: provide fine-grained control over processors
  - Processor dependent and overly verbose for large projects
- **C**: provide low-level features while being
  - Processor independent, require minimal run-time support
- **C++**: compatible with C, provide higher level abstractions
  - Complicated, large runtime overhead if not used properly
- **Java**: a simplified derivative of C++
  - Prevent unnecessary mistakes (both functional and performance-wise) by being less flexible
  - JVM provides supports of concurrency and communications at language level, though slow speed is a concern.
  - To meet deadlines is challenging due to garbage collection.
Programming Languages

- **Assembly**: provide fine-grained control over processors
  - Processor dependent and overly verbose for large projects
- **C**: provide low-level features while being
  - Processor independent, require minimal run-time support
- **C++**: compatible with C, provide higher level abstractions
  - Complicated, large runtime overhead if not used properly
- **Java**: a simplified derivative of C++
  - Prevent unnecessary mistakes (both functional and performance-wise) by being less flexible
  - JVM provides supports of concurrency and communications at language level, though slow speed is a concern.
  - To meet deadlines is challenging due to garbage collection.
Software Synthesis Flow

FIGURE 5.2 Software synthesis flow (Gajski et al.)
How about to implement the TLM via running the necessary simulation kernels on target processors?

- Consume too much resource and lead to slow execution.

Figure 5.3: Input system TLM example (Gajski et al.)
Example Input TLM

How about to implement the TLM via running the necessary simulation kernels on target processors?

- Consume too much resource and lead to slow execution

FIGURE 5.3 Input system TLM example

(Gajski et al.)
Outline

Software Synthesis

Code Generation
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
- A task is inherently sequential.
  - The focus is therefore to translate module compositions and communications into available language constructs.
- Use data abstractions to represent modules, ports, and their compositions
- Communications
  - Use function call and global variables to resolve communications within the same task.
  - Use inter-process communication primitives to resolve communications among tasks on the same processor
  - Use drivers to communicate with tasks on other processors
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
- A task is inherently sequential.
  - The focus is therefore to translate module compositions and communications into available language constructs.
- Use data abstractions to represent modules, ports, and their compositions
- Communications
  - Use function call and global variables to resolve communications within the same task.
  - Use inter-process communication primitives to resolve communications among tasks on the same processor
  - Use drivers to communicate with tasks on other processors
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
- A task is inherently sequential.
  - The focus is therefore to translate module compositions and communications into available language constructs.
- Use data abstractions to represent modules, ports, and their compositions
- Communications
  - Use function call and global variables to resolve communications within the same task.
  - Use inter-process communication primitives to resolve communications among tasks on the same processor
  - Use drivers to communicate with tasks on other processors
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
- **A task is inherently sequential.**
  - The focus is therefore to translate module compositions and communications into available language constructs.
- **Use data abstractions to represent modules, ports, and their compositions**

**Communications**

- Use function call and global variables to resolve communications within the same task.
- Use inter-process communication primitives to resolve communications among tasks on the same processor
- Use drivers to communicate with tasks on other processors
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
- A task is inherently sequential.
  - The focus is therefore to translate module compositions and communications into available language constructs.
- Use data abstractions to represent modules, ports, and their compositions
- **Communications**
  - Use function call and global variables to resolve communications within the same task.
  - Use inter-process communication primitives to resolve communications among tasks on the same processor
  - Use drivers to communicate with tasks on other processors
Example Task Specification

```c
SC_MODULE(TaskB2) {
    CH1 ch11, ch12;
    B1 b11, b12;
    SCCTOR(TaskB2):
        ch11("ch11"), ch12("ch12"),
        b11("b11"), b12("b12") {
            b11.myCh(ch11); // connect ch11
            b12.myCh(ch12); // connect ch12
        }
    void main(void) {
        b11.main();
        b12.main();
    }
};
```

```c
SC_MODULE(B1) {
    int A;
    sc_port<iChannel> myCh;
    SCCTOR(B1) {
        void main(void) {
            A = 1;
            myCh->chCall(A*2);
        }
    };
```

![Diagram](image.png)

`LISTING 5.1  SystemC task specification`

`FIGURE 5.5  Task specification`

(Gajski et al.)
Rules for C Code Generation

**Rule 1:** module $\rightarrow$ struct

**Rule 2:** child modules $\rightarrow$ members

**Rule 3:** variables $\rightarrow$ members

**Rule 4:** ports $\rightarrow$ members

**Rule 5:** methods $\rightarrow$ function taking an extra parameter pointing to the struct representing the module

**Rule 6:** globally/statically instantiate the struct representing the top-most module for the task, bind channels to ports at initialization


C Task Code I

```c
SC_MODULE(B1){
    int A;
    sc_port<iChannel> myCh;
    SC_CTOR(B1){}
    void main(void) {
        A = 1;
        myCh->chCall(A*2);
    }
};

LISTING 5.1
```

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int a;
};

void B1_main(struct B1 *This) {
    (This->a) = 1;
    CH1_chCall(This->myCh, (This->a)*2);
}

LISTING 5.2 ANSI-C task code
```

(Gajski et al.)
SC_MODULE(TaskB2){
  CH1 ch11, ch12;
  B1 b11, b12;
  void main(void) {
    b11.main();
    b12.main();
  }
};

LISTING 5.1

struct TaskB2 {
  struct B1 b11, b12;
  struct CH1 ch11, ch12;
};

void TaskB2_main(struct TaskB2 *This) {
  B1.main(&(This->b11));
  B1.main(&(This->b12));
}

LISTING 5.2  ANSI-C task code

(Gajski et al.)
C Task Code III

```c
SC_MODULE(B1) {
    int A;
    sc_port<iChannel> myCh;
}

SC_MODULE(TaskB2) {
    CH1 ch11, ch12;
    B1 b11, b12;
    SCCTOR(TaskB2):
        ch11("ch11"), ch12("ch12"),
        b11("b11"), b12("b12") {
            b11.myCh(ch11); // connect ch11
            b12.myCh(ch12); // connect ch12
        }
```

**LISTING 5.1**

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int a;
};

struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};

struct TaskB2 taskB2 = {
    &(taskB2.ch11),0/*a*/}/*b11*/,
    & taskB2.ch12,0/*a*/}/*b12*/,
    } /*ch11*/, {} /*ch12*/
};

void TaskB2() {
    TaskB2.main( &task1);
}

**LISTING 5.2** ANSI-C task code
(Gajski et al.)
Summary

- Automated software synthesis increases productivity by reducing time for development and debugging.
- Code generation translates a task specified in a system design language into a sequential program with limited language constructs that can be compiled into binaries later.