Due: 5:00pm 4/15 Chicago time
Reading Assignment

- This lecture: 5.7 – 5.9
- Next lecture: this lecture note, 6
Outline

Additional Issues

Hardware Synthesis
Startup Code

/* processor startup code */

```c
void main(void) {
    PE_Struct_Init(&PE0);
    BSP_init();
    OSInit();

    c_os_handshake_init(&PE0->sem1);
    c_os_handshake_init(&PE0->sem2);

    BSP_UserIrqRegister(INT1, Int1Handler, /*..*/);
    BSP_UserIrqRegister(INT2, Int2Handler, /*..*/);

    taskCreate(task_b2b3, NULL,
               B2B3_main, &this->task_b2b3);

    OSStart();
}
```

LISTING 5.11  Startup code example

(Gajski et al.)

- Startup code is both platform-specific and application-specific
  - Initialize the underlying hardware
  - Registers interrupt handlers
  - Prepares multi-tasking
  - Start OS (the scheduler) to release multi-tasking
Binary Image Generation

**Figure 5.19** Binary image generation

(Gajski et al.)
Execution

- The produced binaries are downloaded onto the target platform and executed.
- The target platform may be implemented as an ASIC or using a FPGA prototyping platform.
  - Allow to validate functionality and timing in physical world.
- Alternatively, the binaries can be validated using an ISS-based virtual platform via simulation.
- Designers may utilize detailed feedback to further improve the system, and trigger the synthesis again for design iterations.
  - Automated synthesis speeds up the whole process so designers may explore more system implementations.
Execution

- The produced binaries are downloaded onto the target platform and executed.
- The target platform may be implemented as an ASIC or using a FPGA prototyping platform.
  - Allow to validate functionality and timing in physical world.
- Alternatively, the binaries can be validated using an ISS-based virtual platform via simulation.
- Designers may utilize detailed feedback to further improve the system, and trigger the synthesis again for design iterations.
  - Automated synthesis speeds up the whole process so designers may explore more system implementations.
Execution

- The produced binaries are downloaded onto the target platform and executed.
- The target platform may be implemented as an ASIC or using a FPGA prototyping platform.
  - Allow to validate functionality and timing in physical world.
- Alternatively, the binaries can be validated using an ISS-based virtual platform via simulation.
  - Designers may utilize detailed feedback to further improve the system, and trigger the synthesis again for design iterations.
    - Automated synthesis speeds up the whole process so designers may explore more system implementations.
Execution

- The produced binaries are downloaded onto the target platform and executed.
- The target platform may be implemented as an ASIC or using a FPGA prototyping platform.
  - Allow to validate functionality and timing in physical world.
- Alternatively, the binaries can be validated using an ISS-based virtual platform via simulation.
- Designers may utilize detailed feedback to further improve the system, and trigger the synthesis again for design iterations.
  - Automated synthesis speeds up the whole process so designers may explore more system implementations.
Virtual Platform

**FIGURE 5.20** ISS-based Virtual platform

(Gajski et al.)
Outline

Additional Issues

Hardware Synthesis
Hardware Synthesis via High-Level Synthesis (HLS)

- A method to generate hardware implementations from behavioral descriptions
  - Input
    - Behavioral descriptions, e.g. CDFG
    - RTL library: available hardware resources
    - Design constraints
  - Output: RTL netlist
    - A synchronous circuit consisting of functional units, registers, interconnects, and control logics.
  - From untimed behavior to cycle-accurate behavior
Hardware Synthesis via High-Level Synthesis (HLS)

- A method to generate hardware implementations from behavioral descriptions
- Input
  - Behavioral descriptions, e.g. CDFG
  - RTL library: available hardware resources
  - Design constraints
- Output: RTL netlist
  - A synchronous circuit consisting of functional units, registers, interconnects, and control logics.
- From untimed behavior to cycle-accurate behavior
Hardware Synthesis via High-Level Synthesis (HLS)

- A method to generate hardware implementations from behavioral descriptions

Input
- Behavioral descriptions, e.g. CDFG
- RTL library: available hardware resources
- Design constraints

Output: RTL netlist
- A synchronous circuit consisting of functional units, registers, interconnects, and control logics.

- From untimed behavior to cycle-accurate behavior
A method to generate hardware implementations from behavioral descriptions

Input
- Behavioral descriptions, e.g. CDFG
- RTL library: available hardware resources
- Design constraints

Output: RTL netlist
- A synchronous circuit consisting of functional units, registers, interconnects, and control logics.

From untimed behavior to cycle-accurate behavior
HLS Tasks

- **Allocation**: spatial aspects of the hardware system
  - Determine the type and quantity of hardware resources used
  - Directly affect chip area

- **Scheduling**: temporal aspects of the hardware system
  - Determine the clock period
  - Determine when (the clock cycle) the activities in CDFG should be executed
  - Directly affect total execution time

- **Binding**: connecting spatial and temporal aspects
  - Determine the hardware resources to execute each activities
  - Become important as metrics beyond chip area and execution time are of concern

- An optimal HLS algorithm should explore all possibilities in these tasks
  - A very difficult problem.
HLS Tasks

- **Allocation**: spatial aspects of the hardware system
  - Determine the type and quantity of hardware resources used
  - Directly affect chip area

- **Scheduling**: temporal aspects of the hardware system
  - Determine the clock period
  - Determine when (the clock cycle) the activities in CDFG should be executed
  - Directly affect total execution time

- **Binding**: connecting spatial and temporal aspects
  - Determine the hardware resources to execute each activity
  - Become important as metrics beyond chip area and execution time are of concern

- An optimal HLS algorithm should explore all possibilities in these tasks
  - A very difficult problem.
HLS Tasks

- **Allocation**: spatial aspects of the hardware system
  - Determine the type and quantity of hardware resources used
  - Directly affect chip area

- **Scheduling**: temporal aspects of the hardware system
  - Determine the clock period
  - Determine when (the clock cycle) the activities in CDFG should be executed
  - Directly affect total execution time

- **Binding**: connecting spatial and temporal aspects
  - Determine the hardware resources to execute each activities
  - Become important as metrics beyond chip area and execution time are of concern

- An optimal HLS algorithm should explore all possibilities in these tasks
  - A very difficult problem.
HLS Tasks

- **Allocation**: spatial aspects of the hardware system
  - Determine the type and quantity of hardware resources used
  - Directly affect chip area

- **Scheduling**: temporal aspects of the hardware system
  - Determine the clock period
  - Determine when (the clock cycle) the activities in CDFG should be executed
  - Directly affect total execution time

- **Binding**: connecting spatial and temporal aspects
  - Determine the hardware resources to execute each activities
  - Become important as metrics beyond chip area and execution time are of concern

- **An optimal HLS algorithm should explore all possibilities in these tasks**
  - A very difficult problem.
Simplified HLS Setting

- **Input**
  - Behavioral description based on DFG
  - Ignore design constraints
- **Fixed allocation for functional units**
  - Functional units are combinational: no pipelining, need storage units for input/output
- **Scheduling**
  - Fixed clock period
  - No *chaining*: no data-dependency among activities executed in one clock cycle
- **Objective**: generate a hardware implementation with reasonably good performance and cost
Simplified HLS Setting

▶ Input
  ▶ Behavioral description based on DFG
  ▶ Ignore design constraints

▶ Fixed allocation for functional units
  ▶ Functional units are combinational: no pipelining, need storage units for input/output

▶ Scheduling
  ▶ Fixed clock period
  ▶ No chaining: no data-dependency among activities executed in one clock cycle

▶ Objective: generate a hardware implementation with reasonably good performance and cost
Simplified HLS Setting

- **Input**
  - Behavioral description based on DFG
  - Ignore design constraints
- **Fixed allocation for functional units**
  - Functional units are combinational: no pipelining, need storage units for input/output
- **Scheduling**
  - Fixed clock period
  - No *chaining*: no data-dependency among activities executed in one clock cycle
- **Objective**: generate a hardware implementation with reasonably good performance and cost
Simplified HLS Setting

- **Input**
  - Behavioral description based on DFG
  - Ignore design constraints
- **Fixed allocation for functional units**
  - Functional units are combinational: no pipelining, need storage units for input/output
- **Scheduling**
  - Fixed clock period
  - No *chaining*: no data-dependency among activities executed in one clock cycle
- **Objective**: generate a hardware implementation with reasonably good performance and cost
Simplified Design Flow

1. DFG generation
2. Scheduling and functional units binding
3. Storage units allocation and binding
4. Control unit synthesis

- 2 and 3 are usually known as datapath synthesis
Simplified Design Flow

1. DFG generation
2. Scheduling and functional units binding
3. Storage units allocation and binding
4. Control unit synthesis
   ▶ 2 and 3 are usually known as datapath synthesis
Example Program

double u, w, y, dx;
int i, N;

for (i = 0; i < N; ++i) {
    double u1, u2, u3, u4, u5, u6, y1;

    u1 = u *dx;
    u2 = 5 *w;
    u3 = 3 *y;
    y1 = i *dx;
    w = w +dx;
    u4 = u1*u2;
    u5 = dx*u3;
    y = y +y1;
    u6 = u -u4;
    u = u6-u5;
}

▶ Assume we need to speed-up the loop body by hardware implementations
Step 1: DFG Generation

- Based on data dependency of loop body
- Vertices are operations (activities), edges are variables
Step 1: DFG Generation

- Based on data dependency of loop body
- Vertices are operations (activities), edges are variables
Step 1: DFG Generation

- Based on data dependency of loop body
- Vertices are operations (activities), edges are variables
Example Functional Units Allocation

- add: need 1 clock cycles to generate result
- sub: need 1 clock cycles to generate result
- mulA and mulB: need 4 clock cycles to generate result
Step 2: Scheduling and Functional Units Binding

- **Control step (cstep):** usually equivalent to a clock cycle
  - Correspond to the lifetime of a single state in the FSM representing the control unit

- Scheduling and functional units binding algorithm
  - Assign operations to functional units iteratively until all operations are assigned
  - Assume external variables to loop body (e.g. u, w, y, i, dx) are ready in cstep 0 and scheduling starts in cstep 1
  - Each iteration handles one cstep
  - Data dependency: only operations with no unfinished predecessor at the beginning of the cstep can start execution in the cstep
  - Structural dependency: subject to the availability of functional units
Step 2: Scheduling and Functional Units Binding

- **Control step** (cstep): usually equivalent to a clock cycle
  - Correspond to the lifetime of a single state in the FSM representing the control unit

- **Scheduling and functional units binding algorithm**
  - Assign operations to functional units iteratively until all operations are assigned
  - Assume external variables to loop body (e.g. u, w, y, i, dx) are ready in cstep 0 and scheduling starts in cstep 1
  - Each iteration handles one cstep
  - Data dependency: only operations with no unfinished predecessor at the beginning of the cstep can start execution in the cstep
  - Structural dependency: subject to the availability of functional units
Control Step 1: Beginning
Control Step 1: Ending
Control Step 2: Beginning
Control Step 5: Beginning
Control Step 9: Beginning
Control Step 13: Beginning
Step 3: Storage Units Allocation and Binding

- **Storage units:** registers (flip-flops)

  - The straight-forward approach: allocate a register to each variable
    - Drawbacks: may need more than necessary number of registers, increase chip area
  - Solution: share registers among variables
    - Variable *lifetime*: time interval between its definition to its last use
    - Two variables can share a register if their lifetimes don’t overlap

- Assume external variables to loop body (e.g. u, w, y, i, dx) won’t share registers with other variables
Step 3: Storage Units Allocation and Binding

- Storage units: registers (flip-flops)
- The straight-forward approach: allocate a register to each variable
  - Drawbacks: may need more than necessary number of registers, increase chip area
- Solution: share registers among variables
  - Variable lifetime: time interval between its definition to its last use
  - Two variables can share a register if their lifetimes don’t overlap
- Assume external variables to loop body (e.g. u, w, y, i, dx) won’t share registers with other variables
Step 3: Storage Units Allocation and Binding

- Storage units: registers (flip-flops)
- The straight-forward approach: allocate a register to each variable
  - Drawbacks: may need more than necessary number of registers, increase chip area
- Solution: share registers among variables
  - Variable *lifetime*: time interval between its definition to its last use
  - Two variables can share a register if their lifetimes don’t overlap
- Assume external variables to loop body (e.g. u, w, y, i, dx) won’t share registers with other variables
Step 3: Storage Units Allocation and Binding

- Storage units: registers (flip-flops)
- The straight-forward approach: allocate a register to each variable
  - Drawbacks: may need more than necessary number of registers, increase chip area
- Solution: share registers among variables
  - Variable lifetime: time interval between its definition to its last use
  - Two variables can share a register if their lifetimes don’t overlap
- Assume external variables to loop body (e.g. u, w, y, i, dx) won’t share registers with other variables
The Left Edge Algorithm for Register Allocation and Binding

1. Sort variables into a list by the starting points of their lifetimes in ascending order
2. Allocate a register R for the first variable and remove the variable from the list
3. Bind R to the first variable in the list such that there is no overlap of lifetimes and remove the variable from the list
4. Repeat 3 until no such variable exists
5. Repeat 2 to 4 until there is no variable in the list
Register Allocations

▶ Need $5+3=8$ instead of $5+7=12$ registers
Step 4: Control Unit Synthesis

- The Diagram in the previous slide is clearly not completed
  - An input port cannot be driven by multiple signals
  - A register should hold its data until being explicitly changed
- Use a mux at each input to choose the correct signal
- Control unit synthesis: design FSMs to generate the control signals for the mux’s
  - State transition depends on scheduling and binding
Step 4: Control Unit Synthesis

- The Diagram in the previous slide is clearly not completed
  - An input port cannot be driven by multiple signals
  - A register should hold its data until being explicitly changed

- Use a mux at each input to choose the correct signal
- Control unit synthesis: design FSMs to generate the control signals for the mux’s
  - State transition depends on scheduling and binding
Step 4: Control Unit Synthesis

- The Diagram in the previous slide is clearly not completed
  - An input port cannot driven by multiple signals
  - A register should hold its data until being explicitly changed
- Use a mux at each input to choose the correct signal
- Control unit synthesis: design FSMs to generate the control signals for the mux’s
  - State transition depends on scheduling and binding
<table>
<thead>
<tr>
<th>Port</th>
<th>csteps</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulA.x</td>
<td>Ru</td>
</tr>
<tr>
<td>mulA.y</td>
<td>Rdx</td>
</tr>
<tr>
<td>Ry</td>
<td>Ry</td>
</tr>
<tr>
<td>R1</td>
<td>*</td>
</tr>
<tr>
<td>R2</td>
<td>*</td>
</tr>
</tbody>
</table>
HLS Flow Summary

- Functional units allocation affects scheduling and function units binding
- Scheduling and function units binding affects storage units allocation and binding
- Scheduling and binding affects control unit design
HLS Flow Summary

- Functional units allocation affects scheduling and function units binding
- Scheduling and function units binding affects storage units allocation and binding
- Scheduling and binding affects control unit design
HLS Flow Summary

- Functional units allocation affects scheduling and function units binding
- Scheduling and function units binding affects storage units allocation and binding
- Scheduling and binding affects control unit design
Automated synthesis speeds up the whole process so designers may explore more system implementations.

Hardware synthesis is based on high-level synthesis that converts behavior specification into RTL.