Reading Assignment

- This lecture: 6.4, 6.5, 6.13
- Next lecture: 6.6, 6.7, 6.8
Outline

Register Sharing

Functional Unit Sharing

Interface Synthesis
Register Sharing

- We assume any variable is only written once.
  - Multiple writes to the same variable are resolved by renaming the variable for each write.
- Variable lifetime: set of states where the variable is alive
  - Write state: the state after it is assigned a new value
  - Read state: the states it is used on certain RHS’
  - All states between the write and the last read state
- Group variables with non-overlapping lifetimes and bind each group to a single register
  - Try to have as few registers as possible
- There may exist many ways to group variables 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 (mux’s to registers)
Register Sharing

- We assume any variable is only written once.
  - Multiple writes to the same variable are resolved by renaming the variable for each write.
- Variable lifetime: set of states where the variable is alive
  - Write state: the state after it is assigned a new value
  - Read state: the states it is used on certain RHS’
  - All states between the write and the last read state
- Group variables with non-overlapping lifetimes and bind each group to a single register
  - Try to have as few registers as possible
- There may exist many ways to group variables 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 (mux’s to registers)
Register Sharing

- We assume any variable is only written once.
  - Multiple writes to the same variable are resolved by renaming the variable for each write.

- Variable lifetime: set of states where the variable is alive
  - Write state: the state after it is assigned a new value
  - Read state: the states it is used on certain RHS’
  - All states between the write and the last read state

- Group variables with non-overlapping lifetimes and bind each group to a single register
  - Try to have as few registers as possible

- There may exist many ways to group variables 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 (mux’s to registers)
Register Sharing

- We assume any variable is only written once.
  - Multiple writes to the same variable are resolved by renaming the variable for each write.
- Variable lifetime: set of states where the variable is alive
  - Write state: the state after it is assigned a new value
  - Read state: the states it is used on certain RHS’
  - All states between the write and the last read state
- Group variables with non-overlapping lifetimes and bind each group to a single register
  - Try to have as few registers as possible
- There may exist many ways to group variables 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 (mux’s to registers)
The register can be shared at both input and output to reduce connectivity cost.
Connectivity Cost Optimization via Compatibility Graph

\[ a = \ln 1 \\
 b = \ln 2 \]

\[ t_1 = |a| \\
 t_2 = |b| \]

\[ x = \max(t_1, t_2) \\
 y = \min(t_1, t_2) \]

\[ t_3 = x \gg 3 \\
 t_4 = y \gg 1 \]

\[ t_5 = x - t_3 \]

\[ t_6 = t_4 + t_5 \]

\[ t_7 = \max(t_6, x) \]

\[ \text{Done} = 1 \\
 \text{Out} = t_7 \]

FIGURE 6.11 (a) Initial compatibility graph

(Gajski et al.)
Grouping Variables in Compatibility Graph

▶ Prefer to merge nodes with high gain

*(Gajski et al.)*
Final Variable Bindings

(e) Final compatibility graph

(f) Final register assignments

- $R_1 = \{ a, t1, x, t7 \}$
- $R_2 = \{ b, t2, y, t3, t5, t6 \}$
- $R_3 = \{ t4 \}$

**FIGURE 6.11** Variable merging for SRA example

- Prefer to merge nodes with high gains
Datapath after Register Sharing

Assume a function unit is available for each type of operation

Further savings on interconnects can be achieved via functional unit sharing.

FIGURE 6.12 SRA datapath with register sharing

(Gajski et al.)
Datapath after Register Sharing

▶ Assume a function unit is available for each type of operation
▶ Further savings on interconnects can be achieved via functional unit sharing.

*FIGURE 6.12* SRA datapath with register sharing

(Gajski et al.)
Outline

Register Sharing

Functional Unit Sharing

Interface Synthesis
Functional Unit Sharing

- Minimize number of functional units in datapath
  - Within any given state, a datapath will not perform every operation.
  - Similar operations can be grouped into a single multifunction unit if they are active at different states

- Increase unit utilizations

- Usually it’s not helpful to group dissimilar operations as they demand structurally different designs.
Functional Unit Sharing

- Minimize number of functional units in datapath
  - Within any given state, a datapath will not perform every operation.
  - Similar operations can be grouped into a single multifunction unit if they are active at different states

- Increase unit utilizations
  - Usually it’s not helpful to group dissimilar operations as they demand structurally different designs.
Functional Unit Sharing

- Minimize number of functional units in datapath
  - Within any given state, a datapath will not perform every operation.
  - Similar operations can be grouped into a single multifunction unit if they are active at different states
- Increase unit utilizations
- Usually it’s not helpful to group dissimilar operations as they demand structurally different designs.
Note the extra selectors required for functional unit sharing.

The sharing would be advantageous if the cost of an adder/subtractor and two selectors is less than the cost of a separate adder and subtractor.

*(Gajski et al.)*
Note the extra selectors required for functional unit sharing.

The sharing would be advantageous if the cost of an adder/subtractor and two selectors is less than the cost of a separate adder and subtractor.
Now the edge weights represent number of common sources and number of common destinations.

- Note that we count common registers instead of common variables.
Prefer to merge nodes with similar structures and high gains
Datapath after Register and Functional Unit Sharing

▶ Only 7 selector inputs are needed.

**FIGURE 6.15** SRA design after register and unit merging

(Gajski et al.)
Outline

Register Sharing

Functional Unit Sharing

Interface Synthesis
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
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
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
HW Synthesis Timing Constraints

(a) Freely schedulable code
(b) Schedule constraint code
(c) RTL bus interface
Timing Constraints Explained

- Freely schedulable code
  - Code for actual application/processes
  - Code for communication down to a set of MAC layer calls
- Schedule constraint code
  - Contain timing limitations that have to be observed during synthesis
  - e.g. MAC driver that interfaces between the freely schedulable code and the pre-implemented bus interface
  - Specific for the underlying bus interface
- Bus interface
  - Have to follow strict cycle-timing requirements of the bus protocol
  - Use pre-implemented component to guarantee correct bus timing (RTL using bus clock cycle)
Timing Constraints Explained

- Freely schedulable code
  - Code for actual application/processes
  - Code for communication down to a set of MAC layer calls
- Schedule constraint code
  - Contain timing limitations that have to be observed during synthesis
  - E.g. MAC driver that interfaces between the freely schedulable code and the pre-implemented bus interface
  - Specific for the underlying bus interface
- Bus interface
  - Have to follow strict cycle-timing requirements of the bus protocol
  - Use pre-implemented component to guarantee correct bus timing (RTL using bus clock cycle)
Timing Constraints Explained

- Freely schedulable code
  - Code for actual application/processes
  - Code for communication down to a set of MAC layer calls
- Schedule constraint code
  - Contain timing limitations that have to be observed during synthesis
  - e.g. MAC driver that interfaces between the freely schedulable code and the pre-implemented bus interface
  - Specific for the underlying bus interface
- Bus interface
  - Have to follow strict cycle-timing requirements of the bus protocol
  - Use pre-implemented component to guarantee correct bus timing (RTL using bus clock cycle)
Timing Constraints Explained

- Freely schedulable code
  - Code for actual application/processes
  - Code for communication down to a set of MAC layer calls

- Schedule constraint code
  - Contain timing limitations that have to be observed during synthesis
  - e.g. MAC driver that interfaces between the freely schedulable code and the pre-implemented bus interface
  - Specific for the underlying bus interface

- Bus interface
  - Have to follow strict cycle-timing requirements of the bus protocol
  - Use pre-implemented component to guarantee correct bus timing (RTL using bus clock cycle)
MAC Driver Example

FIGURE 6.34  FSMD for MAC driver

(Gajski et al.)
Synthesized HW Component w/ Bus Interface

Figure 6.35 Custom HW component with bus interface (Gajski et al.)
FIGURE 6.36  A typical bus protocol

(Gajski et al.)
Transducer Synthesis

Transducer is simply a PE translating between two bus protocols

(Gajski et al.)

FIGURE 6.37 Transducer structure
Resource usage can be optimized via sharing of registers and functional units.

Pre-implemented bus interfaces help to resolve differences among HW and bus timing. MAC drivers should be introduced as part of the applications to access such interfaces.