ECE 587 – Hardware/Software Co-Design
Lecture 16 Hardware Synthesis II

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

March 10, 2025

Reading Assignment

- This lecture: 6
- Next Lecture: 6

Outline

General HLS Flow

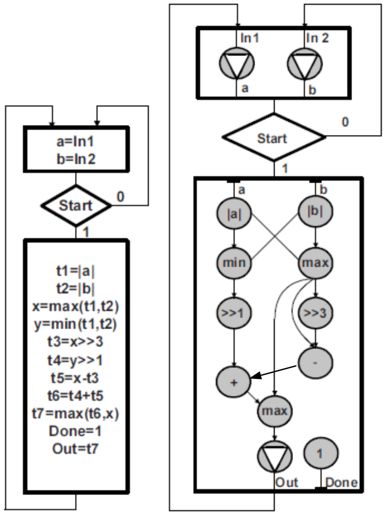
- What are missing from our simplified flow?
  - Input as CDFG instead of DFG
  - What if allocation is not available?
- A more general flow
  - Input as CDFG: don't worry about other input formats, they are at lower abstraction levels and will be handled as HLS goes.
  - Synthesize CDFG into FSMD: schedule 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

Example Application

- Approximate the square root of the sum of two squares.

  SRA: $\sqrt{a^2 + b^2} \approx \max(0.875 \max(a, b) + 0.5 \min(a, b), a, b)$

Untimed Specifications: C and CDFG



(a) Flowchart in C (b) CDFG model

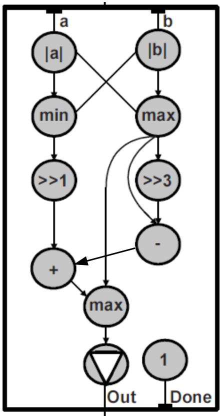
FIGURE 6.27 C and CDFG

(Gajski et al.)

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.
    - Assume the bound is the shortest execute time for now (as obtained in ASAP scheduling).

ASAP and ALAP Scheduling Examples



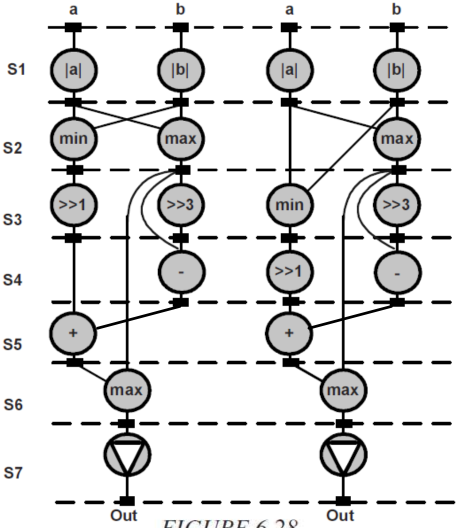


FIGURE 6.28
(a) ASAP (b) ALAP

(Gajski et al.)

Critical Path

- Without structural dependencies, the shortest execution time of a DFG is determined by its critical path.
  - Critical paths: paths with the maximum number of operations
  - Critical operations: operations on at least one critical path
  - Non-critical operations: operations not on any critical paths
- Critical operations are scheduled to the same state (cstep) in both ASAP and ALAP algorithms.
- Non-critical operations are scheduled to different states.
  - e.g. `min` and >>1 in our example
  - The earliest possible state to execute it is obtained from ASAP.
  - The latest possible state (deadline) to execute it is obtained from ALAP.

Mobility

- 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)
  - e.g. in our previous example, both `min` and >>1 have a mobility of 1, and others have a mobility of 0.
- We can extend mobility to the case where the time bound for ALAP is NOT the shortest execution time.
  - Redefine critical operations to be those with 0 mobility
  - Redefine 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.
  - Need to combine several for better results.

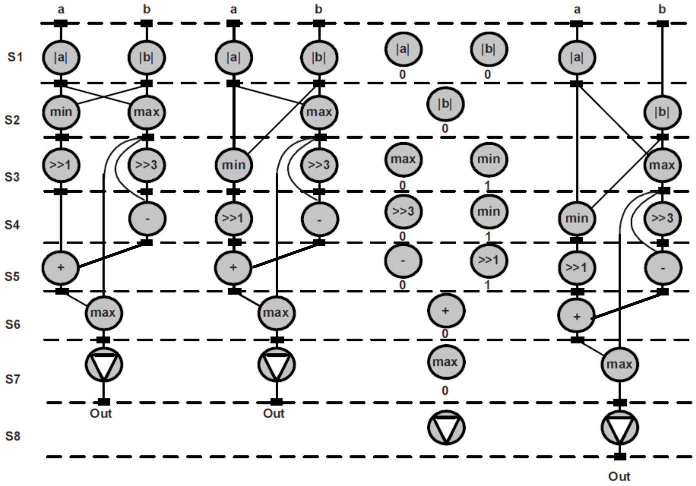
Advanced Scheduling Algorithms

- 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.

RC Scheduling

- Since we want to obtain a schedule with the best performance, it is reasonable to compute the shortest execution time ignoring the resource constraints first.
  - Perform ASAP scheduling
- Perform ALAP scheduling using the shortest execution time as bound and then compute mobility
  - Mobilities will be used to select operations that are ready but cannot be scheduled due to structural dependencies.
- Schedule operations from the first state, at each state:
  - Put all operations whose operands are ready to the *ready list*
  - Sort the ready list by the increasing order of mobilities, breaking ties using other measure like urgencies (distance to the deadline as computed in ALAP)
  - Scan the ready list from the beginning, bind operations to available functional units until no functional unit is available or the end of the ready list is reached.

RC Scheduling Example



(a) ASAP (b) ALAP (c) Ready list with nobil- ities (d) RC schedule

FIGURE 6.28 ASAP, ALAP, and RC schedules for SRA

(Gajski et al.)

- Assume one arithmetic unit and two shift units are available

TC Scheduling

- Perform ASAP scheduling to make sure the given bound on the execution time is feasible.
- Perform ALAP scheduling using the given bound and then compute mobility
- Utilize the mobility to reduce resource usages

TC Scheduling Example

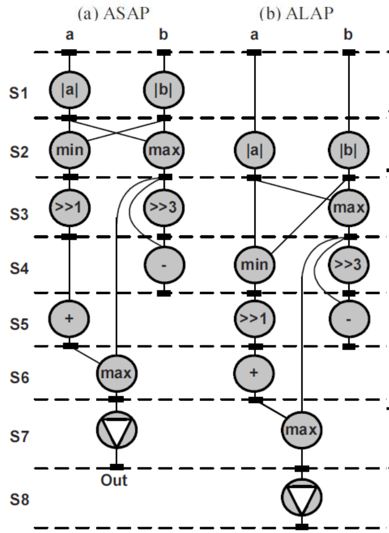
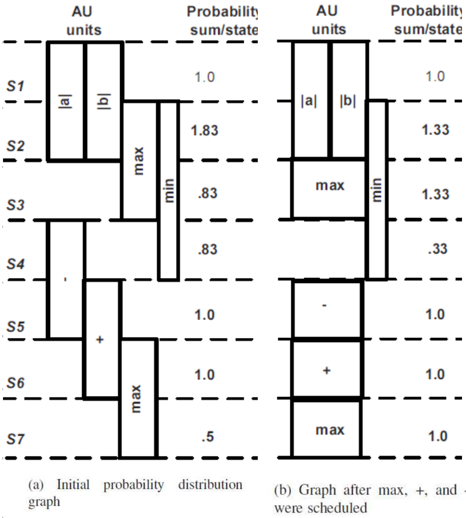


FIGURE 6.31



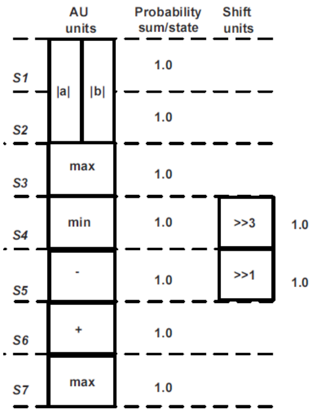
(a) Initial probability distribution graph

(b) Graph after max, +, and - were scheduled

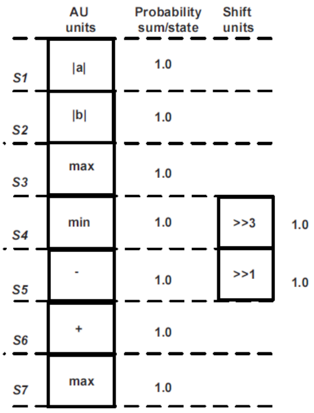
FIGURE 6.32

(Gajski et al.)

TC Scheduling Example (Cont.)



(c) Graph after max, +, -, min, »3, and »1 were scheduled



(d) Distribution graph for final schedule

FIGURE 6.32

(c) TC schedule

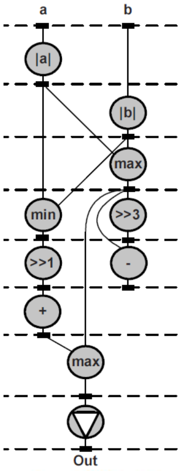
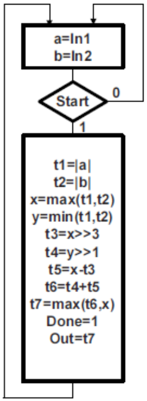
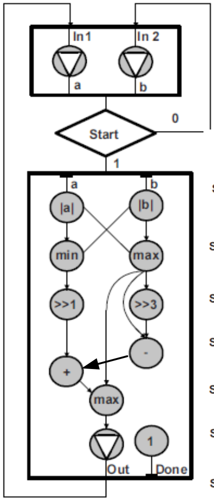


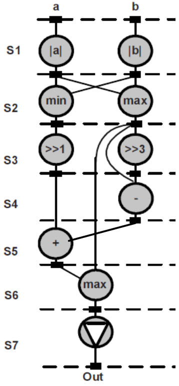
FIGURE 6.31
(Gajski et al.)

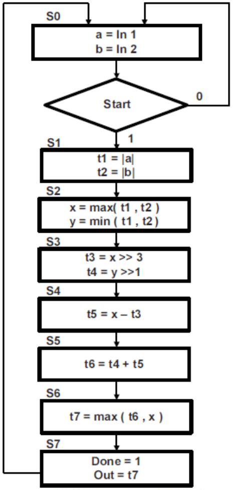
- Need one arithmetic unit and one shift unit.

FSMD as Output of Scheduling (Assume ASAP)









(b) FSMD model of SRA

FIGURE 6.8

(Gajski et al.)

After Scheduling: Optimization Ideas

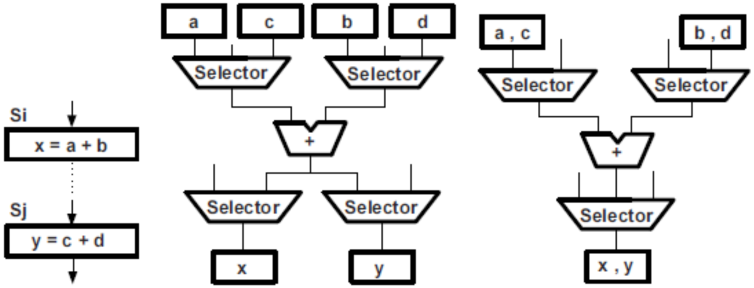
- Allow multiple variables to share the same register
  - If their lifetimes do not overlap
- Use register file/scratch-pad memory to save connections from/to registers
  - If they do not need to be used at the same time
- Share the same functional unit for multiple operations
  - Across different cycles
- Group connections into buses

Outline

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)

Variable Binding and Connectivity Cost



(a) Partial FSMD

(b) Datapath without register sharing

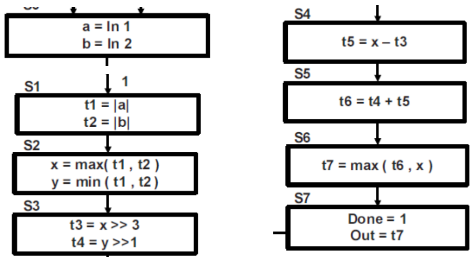
(c) Datapath with register sharing

FIGURE 6.9 Gain in register sharing

(Gajski et al.)

- The register can be shared at both input and output to reduce connectivity cost.

Connectivity Cost Optimization via Compatibility Graph



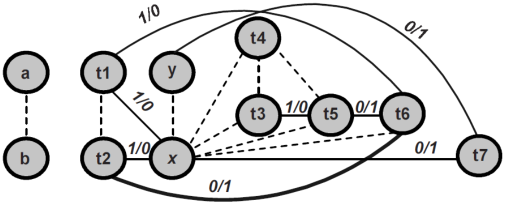
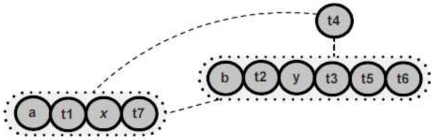


FIGURE 6.11 (a) Initial compability graph (Gajski et al.)

- Variables connected by dotted edge cannot share a register.
- Solid edges indicate gains if variables share a register.

Final Variable Bindings



(e) Final compatibility graph

- R1 = [a, t1, x, t_7]
- R2 = [b, t2, y, t3, t5, t6]
- R3 = [t4]

(f) Final register assignments

FIGURE 6.11 Variable merging for SRA example

(Gajski et al.)

- Prefer to merge nodes with high gains

Datapath after Register Sharing

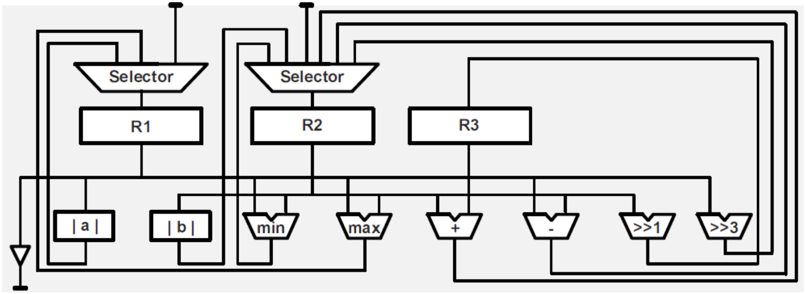


FIGURE 6.12 SRA datapath with register sharing

(Gajski et al.)

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

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 Gain

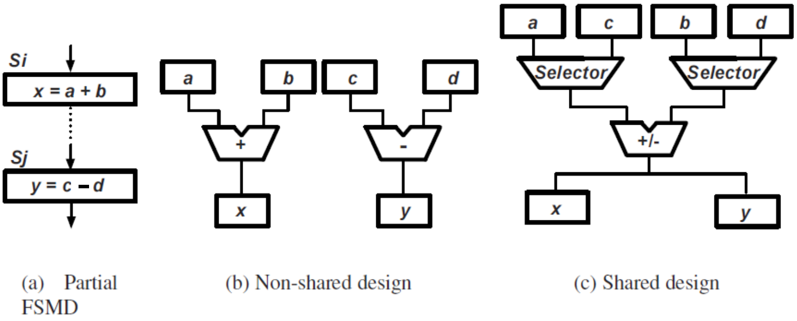
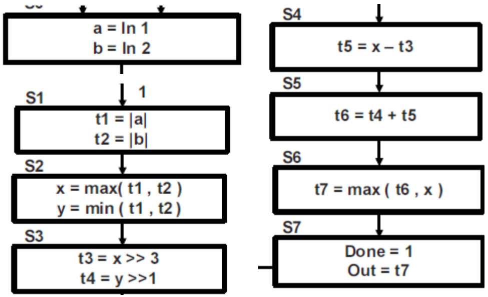


FIGURE 6.13 Gain in functional unit sharing

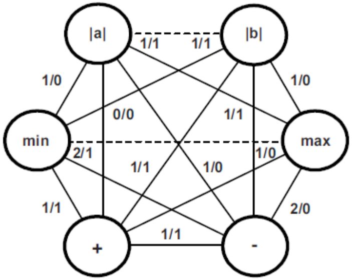
(Gajski et al.)

- 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.

Functional Unit Sharing via Compatibility Graph



- R1 = [a, t1, x, t_7]
- R2 = [b, t2, y, t3, t5, t6]
- R3 = [t4]



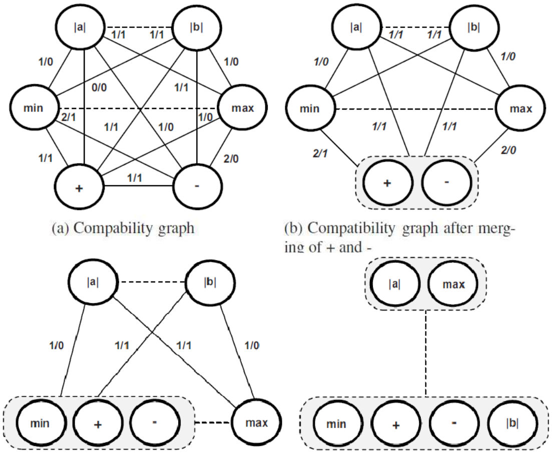
(a) Compability graph

FIGURE 6.14

(Gajski et al.)

- Dotted edges indicate units that cannot be merged.
- Weights on solid edges represent number of common sources and number of common destinations.
    - In common registers instead of common variables.

Functional Unit Merging



(a) Compability graph

(b) Compatibility graph after merging of + and -

(c) Compatibility graph after merging of min, +, and -

(d) Final graph partitions

FIGURE 6.14 Functional unit merging for SRA

(Gajski et al.)

- Prefer to merge nodes with similar structures and high gains

Datapath after Register and Functional Unit Sharing

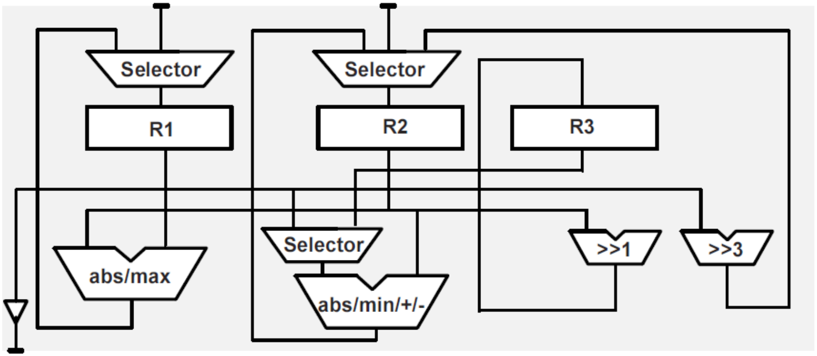


FIGURE 6.15 SRA design after register and unit merging

(Gajski et al.)

- Only 7 selector inputs are needed.

Connection Sharing

- Interconnects do consume considerable amount of resource in modern chip designs.
  - Consist of metal wires, vias, and buffers.
- Merge connections into buses to reduce resource usage
  - Group connections not used at the same time
  - Use tri-state buffer to connect connection sources to bus
  - We may implement the selectors the same way as how big mux's are implemented.

Connection Usage Table and Compatibility Graph

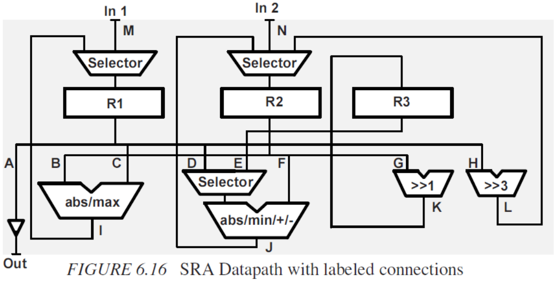
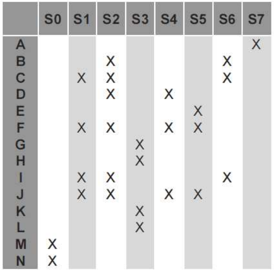
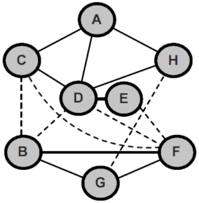


FIGURE 6.16 SRA Datapath with labeled connections



	S0	S1	S2	S3	S4	S5	S6	S7
A								X
B			X				X	
C		X	X				X	
D			X		X			
E						X		
F		X	X		X	X		
G				X				
H				X				
I		X	X				X	
J		X	X		X	X		
K				X				
L				X				
M	X							
N	X							

TABLE 6.6 Connection usage table

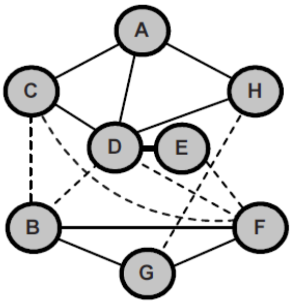


(a) Compatibility graph for input buses

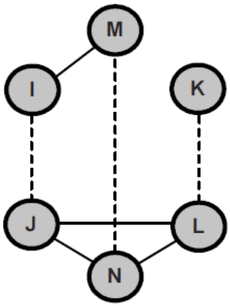
FIGURE 6.17

(Gajski et al.)

Bus Assignment



(a) Compatibility graph for input buses



(b) Compatibility graph for output buses

- Bus1 = [A, C, D, E, H]
- Bus2 = [B, F, G]
- Bus3 = [I, K, M]
- Bus4 = [J, L, N]

(c) Bus assignment

FIGURE 6.17 Connection merging for SRA

(Gajski et al.)

Updated Datapath with Buses

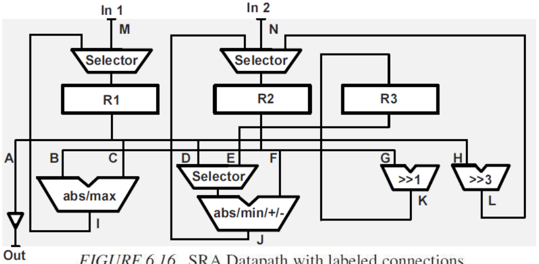


FIGURE 6.16 SRA Datapath with labeled connections

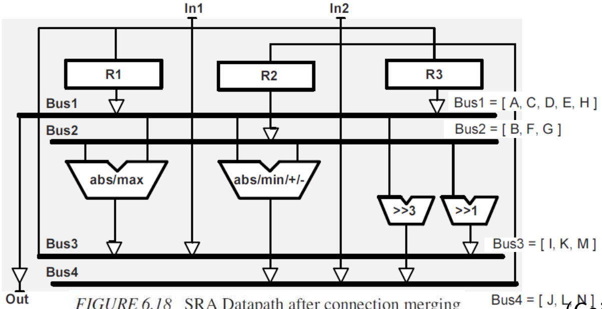


FIGURE 6.18 SRA Datapath after connection merging

(Gajski et al.)

Summary

- For a general HLS flow, the first step could be a scheduling that outputs the cycle-accurate behavior as FSMD.
- Allocations and bindings are applied to the FSMD model state-by-state, and further optimizations are also possible.
- Resource usage can be optimized via sharing of registers and functional units, as well as merging connections into buses.