



Allocation and binding

- **High-level synthesis tasks**, i.e., scheduling, resource allocation, and resource assignment neither need to be performed in a certain sequence nor to be considered as independent tasks
- Allocation is the assignment of operations to hardware possibly according to a given schedule, given constraints and minimizing a cost function
- Functional unit, storage and interconnection allocations
 - slightly different flavors:
 - module selection – selecting among several ones
 - binding – to particular hardware (a.k.a. assignment)
- Other HLS tasks...
 - *Memory management*: deals with the allocation of memories, with the assignment of data to memories, and with the generation of address calculation units
 - *High-level data path mapping*: partitions the data part into application specific units and defines their functionality
 - *Encoding* data types and control signals

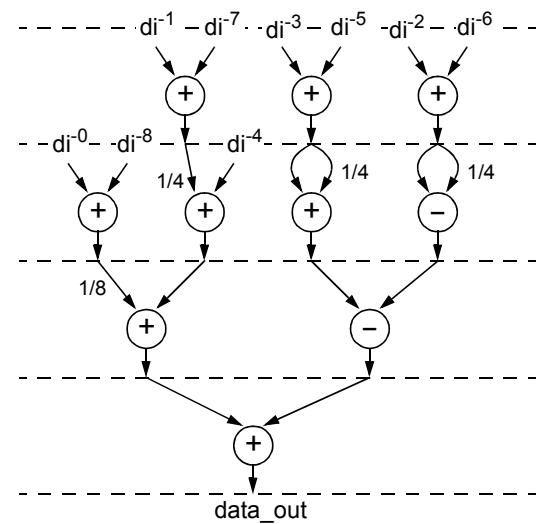


Completing the Data Path

- Subtasks after scheduling:
- Allocation
 - Allocation of FUs (if not allocated before scheduling)
 - Allocation of storage (if not allocated before scheduling)
 - Allocation of busses (if busses are required and not allocated in advance)
- Binding (assignment)
 - Assignment of operations to FU instances
(if not assignment before scheduling as in the partitioning approach)
 - Assignment of values to storage elements
 - Assignment of data to be transferred to buses (if busses are used)

Operation types and functional units

- An operation can be mapped onto different functional units
 - bit-width
 - 12-bit addition & 16-bit adder
 - supported operations
 - addition & adder/subtractor
 - cost trade-offs
 - universal modules are always more expensive
- Algebraic transformations
 - addition is commutative
 - $a+b == b+a$
 - double “inversion”
 - $(a+b)-(c+d) == (a-d)-(c-b) == (a-c)-(d-b)$

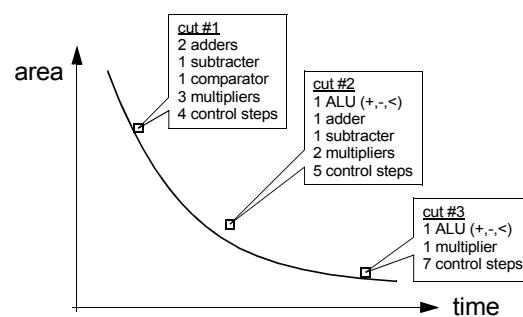
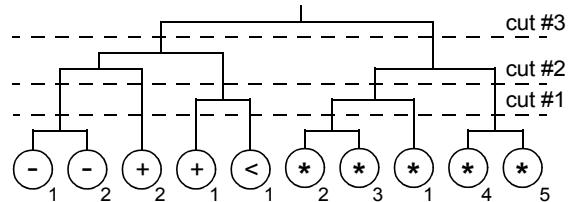


Allocation and binding approaches

- Rule based schemes (Cathedral II), used before scheduling
- Greedy (Adam)
- Iterative methods
- Branch and bound (interconnect levels)
- Integer linear programming (ILP)
- Graph theoretical (clicks, node coloring)
 - the most general approach

Partitioning approach

- **Partitioning approach** consists of bottom-up functional unit allocation and functional unit assignment before scheduling is introduced.
- Partitioning approach exploits topological information based on interconnections between clusters.
- A **functional unit cluster** is a bottom-up built multifunctional data path which either exists in the library or has to be built from several functional units available. *Distance* reflects how preferable it is to have two operations executed on the same cluster.



Distance measure in partitioning approach

- **Relative cost gain**

$$\Delta C(o_1, o_2) = (\text{cost}(o_1) + \text{cost}(o_2) - \text{cost}(o_1, o_2)) / \text{cost}(o_1, o_2)$$
- **Influence of potential parallelism**

$$P(o_1, o_2) = \begin{cases} 1 & \text{if } o_1 \text{ and } o_2 \text{ can be executed in parallel} \\ 0 & \text{otherwise} \end{cases}$$
- **Amount of interconnections**

$$\Delta X(o_1, o_2) = \frac{|\{(o_i, o_1), (o_1, o_i) \in E\} \cap \{(o_i, o_2), (o_2, o_i) \in E\}|}{\max(|\{(o_i, o_1), (o_1, o_i) \in E\}|, |\{(o_i, o_2), (o_2, o_i) \in E\}|)}$$
- **Distance measure D (between clusters, average between each pair):**

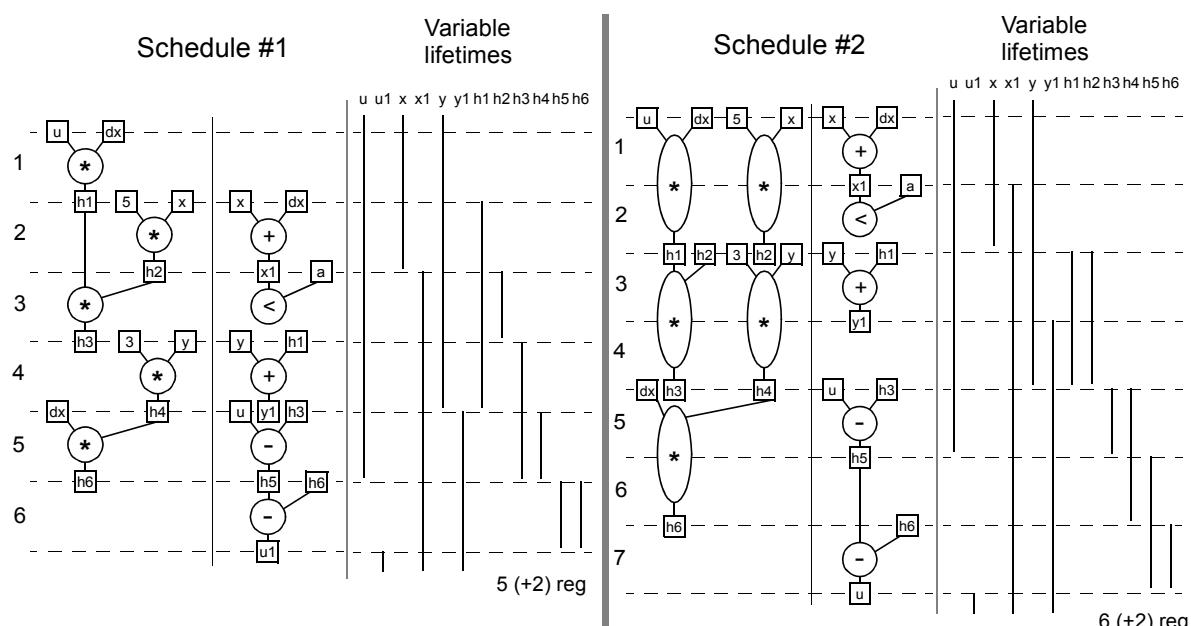
$$D(o_1, o_2; N, M) = (P(o_1, o_2) + 1)^N / (\Delta C(o_1, o_2) + M \Delta X(o_1, o_2))$$

Register allocation and binding

- Register allocation means the determination of the number of registers to be provided. Interconnect area depends on the grouping of values in registers.
 - Left-edge algorithm
 - Tseng's heuristics
 - Weight-directed clique partitioning
 - Graph coloring (conflict graph)
- The *lifetime* i_{val} or life interval (or life moments) of a value is defined as:

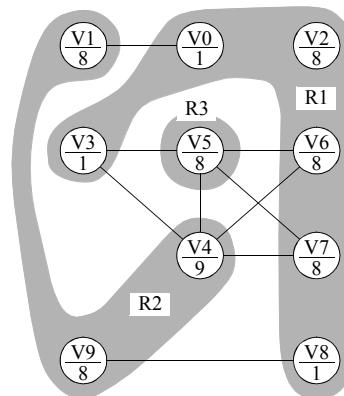
$$i_{val} = (T_{\sigma(o)} + \Delta(o), \max_{o' \in \text{Succ}(o, DFG)} (T_{\sigma(o')} + \Delta_{use}(o'))),$$
 where $\Delta_{use}(o')$ is the operation delay $\Delta(o')$ if operation o' is assigned to a combinational functional unit; σ is control step; and T corresponds to the control step boundary.

DFG and values lifetime table



Left-Edge algorithm and graph coloring

Variable	V0	V1	V2	V3	V4	V5	V6	V7	V8	V9	Register
	1	8	8	1	9	8	8	8	1	8	R1 R2 R3
S0	■■■										8 9 8
S1		■									
S2			■	■■■	■■■	■■■	■■■	■■■	■■■	■■■	
S3											
S4							■				
S5											
S6							■■■	■■■	■■■	■■■	
S7											
S8							■■	■■	■■	■■	
S9											



- a) lifetime moments of variables and allocated registers
- b) colored conflict graph

Left-Edge algorithm

- With: i_v - life interval of value v
 $\text{birth}(v)$ - birth time and $\text{death}(v)$ - death time
 assign v to registers R such that the minimum number of registers is allocated

```

BEGIN
    I_sorted := Sort V with increasing birth(v);
    number := 0;
    WHILE I_sorted ≠ NIL DO BEGIN
        number := number + 1;
        v_cur := HEAD(I_sorted);
        I_sorted := TAIL(I_sorted);
        R[number] := v_cur;
        I'_sorted := NIL;
        WHILE I_sorted ≠ NIL DO BEGIN
            v_next := HEAD(I_sorted);
            I_sorted := TAIL(I_sorted);
            IF death(v_cur) ≤ birth(v_next) THEN BEGIN
                R[number] := R[number] ∪ v_next;
                v_cur := v_next;
            END ELSE I'_sorted := APPEND(I'_sorted, v_next);
        END;
        I_sorted := I'_sorted;
    END;
END.
    
```

- Disadvantages of left-edge algorithm:
 - Not all lifetime tables may be interpreted as intersecting intervals on a line
 - The assignment produced is neither unique nor necessarily optimal



Tseng's heuristic (clique cover)

- Let's have a **compatibility graph** $G=(V,E)$. Find:
 - For all edges $e=(v,w)$ in graph G , the number of common neighbors v :
 $\nu(e) = |\{v' \in V : (v',v) \in E \text{ & } (v',w) \in E\}|$
 - the number of edges η to be deleted if the vertices v and w are merged into vertex v :
 $\eta(e) = |\{(v,v') \in E : (w,v') \notin E\} \cup \{(w,v') \in E\}|$
- Merging means assignment of values v and w to a register $\#v$.
- After merging, $\nu(e)$ and $\eta(e)$ are recalculated for affected nodes and the process is repeated for neighbors of v .
The list R_v stores the values assigned to register $\#v$.



Tseng's heuristic (clique cover)

- Remark 1: $E_{\text{sub}}(v)$ is the edge set of $G_{\text{sub}}(V_{\text{sub}}, E_{\text{sub}})$ with $v \in V_{\text{sub}}$ where G_{sub} is a disconnected subgraph of G .

```
BEGIN
    FOR all  $v \in V$  DO  $R_v := v$ ;
    FOR all  $e \in E$  DO calculate  $v$  and  $\eta$ ;
    REPEAT
         $e_{\text{best}} := (\text{ } e=(v,w) : \eta(e)=\min_{E.\max.v}(\eta)$ 
         $\text{where } E.\max.v=\{e' \in E : v(e')=\max_E(v)\} \text{ } )$ ;
         $R_v := \text{APPEND}(R_v, w)$ ;
         $E := E \setminus \{(v,v') \in E : (w,v') \notin E\} \setminus \{(w,v') \in E\}$ ;
         $V := V \setminus \{w\}$ ;
        FOR all  $e \in E_{\text{sub}}(v)$  DO calculate  $v$  and  $\eta$ ;
         $E_v := \{(v,v') \in E\}$ ;
        WHILE  $E_v \neq \emptyset$  DO BEGIN
             $e_{\text{best}} := (\text{ } e=(v,w) : \eta(e)=\min_{E.\max.v}(\eta)$ 
             $\text{where } E.\max.\eta=\{e' \in E : v(e')=\max_E(v)\} \text{ } )$ ;
             $R_v := \text{APPEND}(R_v, w)$ ;
             $E := E \setminus \{(v,v') \in E : (w,v') \notin E\} \setminus \{(w,v') \in E\}$ ;
             $V := V \setminus \{w\}$ ;
            FOR all  $e \in E_{\text{sub}}(v)$  DO calculate  $v$  and  $\eta$ ;
             $E_v := \{(v,v') \in E\}$ ;
        END;
    UNTIL  $E=\emptyset$ ;
END.
```

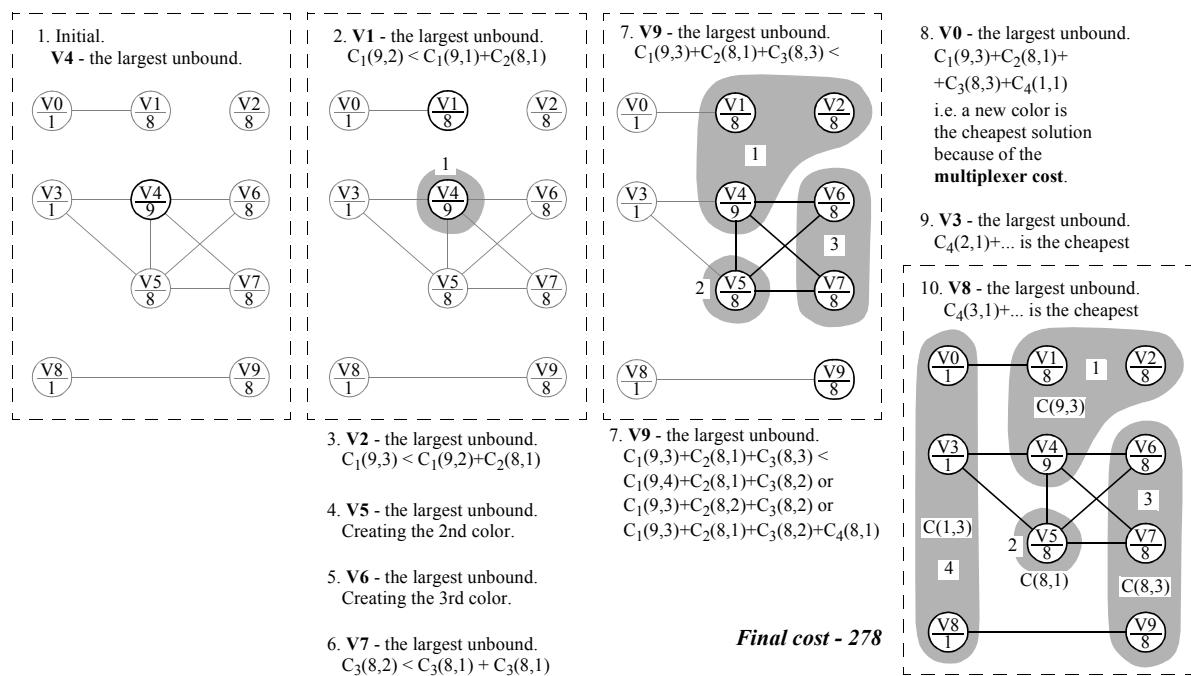
Other approaches

- **Weight directed clique partitioning** for optimization for interconnect area, the weights are assigned, expressing how preferable a merge of a compatible pair of values or intermediate registers is.
- **Graph coloring**. Based on conflict graph - a color corresponds to a register.
- **Two-step approach**. Treats separately the register allocation and register assignment, including interconnect optimization.

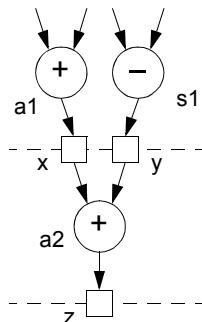
Algorithm	Registers	Mux inputs	2:1 Mux-s
Left-edge	12	50	35
Tseng	13	51	38
Weight-Directed	14	28	22
Two-step	12	28	20

Weighted graph coloring

- Uses conflict graph that is built from the lifetime intervals



Weighted graph coloring (cont)



conflict graphs

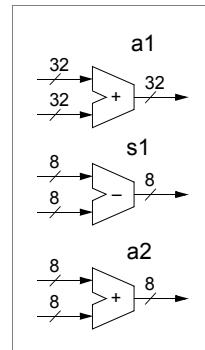


registers

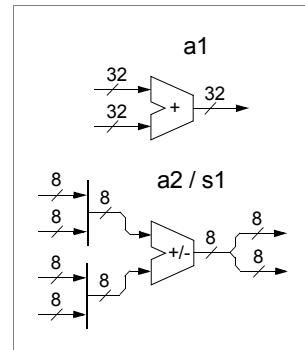
FU-s

CDFG

possible
bindings



VS.



```

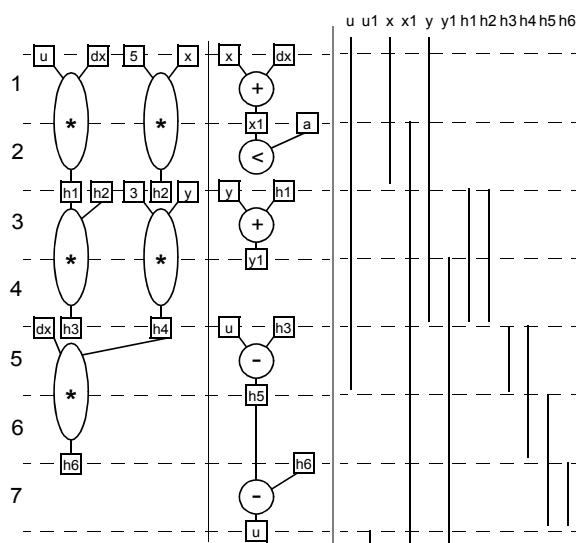
procedure StaticGreedy ( G, selection ) begin
    while UncoloredNodes(G) ≠ Ø loop begin
        node := NextUncoloredNode(G,selection);
        <best_color,best_type,best_cost> := BestBinding(G,node);
        G := Color( G, node, <best_color,best_type> );
    end loop;
    return G;
end procedure;

```

Example synthesis approach

- **Differential Equation example, multiplexed data part architecture**
- **Functional unit allocation**
- **Resource constrained scheduling**
- **Functional unit assignment**
- **Register allocation**
- **Register assignment**
- **Multiplexer extraction**

Schedule and lifetime table



- Functional unit (FU) assignment

FU	M1	M2	ALU
result	h1, h3, h6	h2, h4	x1, cc, y1, h5, u1

- Register assignment

Reg.	R1	R2	R3	R4	R5	R6
var.	u (u1) h5	x y1	y	x1	h1 h3 h6	h2 h4

Multiplexer optimization

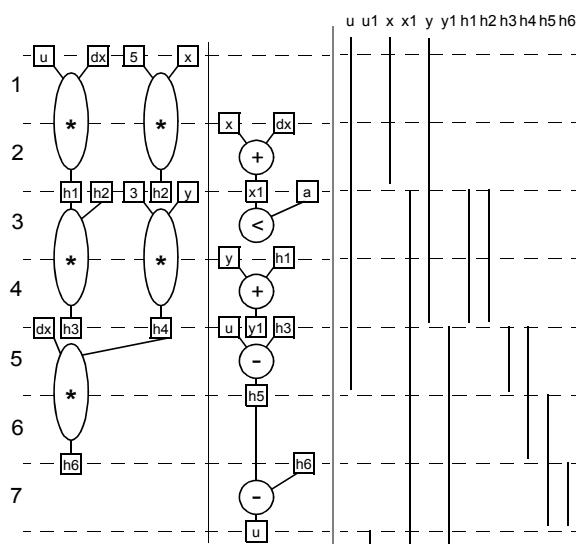
- Functional units & registers
 - M1: h1:4[u,dx], h3:3[h1,h2], h6:2[dx,h4]; M2: h2:4[5,x], h4:3[3,y]
 - ALU: h5:2[u,h3], x1:2[x,dx], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]
 - Ra; Rdx; R1 (u,u1,h5); R2 (x,y1); Ry; Rx1; R5 (h1,h3,h6); R6 (h2,h4)
- Multiplexers

step	1	2	3	4	5	6	7
L	M1	R1	R1	R5	R5	Rdx	Rdx
		Rdx	Rdx	R6	R6	R6	-
R	M2	5	5	3	3	-	-
		R2	R2	Ry	Ry	-	-

	1	2	3	4	5	6	7
ALU	R2	Rx1	Ry	-	R1	-	R1
	Rdx	Ra	R5	-	R5	-	R5

- M1.L - 3, M1.R - 2, M2.L - 2, M2.R - 2, ALU.L - 4, ALU.R - 3
- M1 has the same source (Rdx) on both multiplexers - swap inputs at the first step
- Result – 22 multiplexer inputs:
 - M1.L - 2 (Rdx, R5), M1.R - 2 (R1, R6), M2.L - 2 (5, 3), M2.R - 2 (R2, Ry), ALU.L - 4 (R2, Rx1, Ry, R1), ALU.R - 3 (Rdx, Ra, R5), Ra - 0 (inp), Rdx - 0 (inp), R1 - 2 (inp, ALU), R2 - 3 (inp, ALU, R4), Ry - 2 (inp, R2), Rx1 - 0 (ALU), R5 - 0 (M1), R6 - 0 (M2)

Schedule and lifetime table – example #2



- Functional unit (FU) assignment

FU	M1	M2	ALU
result	h1, h3, h6	h2, h4	x1, cc, y1, h5, u1

- Register assignment

Reg.	R1	R2	R3	R4	R5
var.	u (u1) h5	x x1	y y1	h1 h3 h6	h2 h4

Multiplexer optimization – example #2

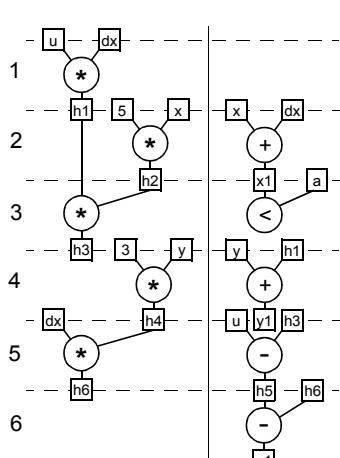
- Functional units & registers
 - M1: h1:4[u,dx], h3:3[h1,h2], h6:2[dx,h4]; M2: h2:4[5,x], h4:3[3,y]
 - ALU: h5:2[u,h3], x1:2[x,dx], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]
 - Ra; Rdx; R1 (u,u1,h5); Rx (x,x1); Ry (y,y1); R4 (h1,h3,h6); R5 (h2,h4)
- Multiplexers

step	1	2	3	4	5	6	7
L	M1	R1	R1	R4	R4	Rdx	Rdx
		Rdx	Rdx	R5	R5	R5	-
R	M2	5	5	3	3	-	-
		Rx	Rx	Ry	Ry	-	-

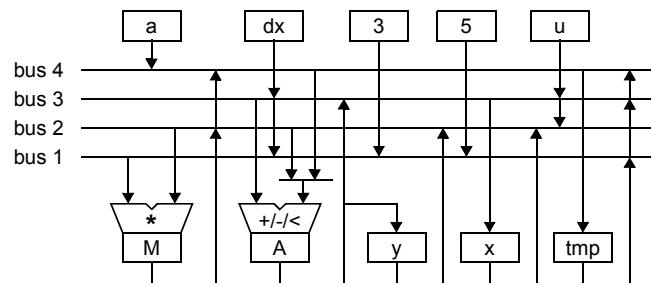
	1	2	3	4	5	6	7
ALU	-	Rx	Rx	Ry	R1	-	R1
	-	Rdx	Ra	R4	R4	-	R4

- M1.L - 3, M1.R - 2, M2.L - 2, M2.R - 2, ALU.L - 3, ALU.R - 3
- M1 has the same source (Rdx) on both multiplexers - swap inputs at the first step
- Result – 20 multiplexer inputs:
 - M1.L - 2 (Rdx, R4), M1.R - 2 (R1, R5), M2.L - 2 (5, 3), M2.R - 2 (Rx, Ry), ALU.L - 3 (Rx, Ry, R1), ALU.R - 3 (Rdx, Ra, R4), Ra - 0 (inp), Rdx - 0 (inp), R1 - 2 (inp, ALU), Rx - 2 (inp, ALU), Ry - 2 (inp, ALU), R4 - 0 (M1), R5 - 0 (M2)

Bidirectional bus architecture



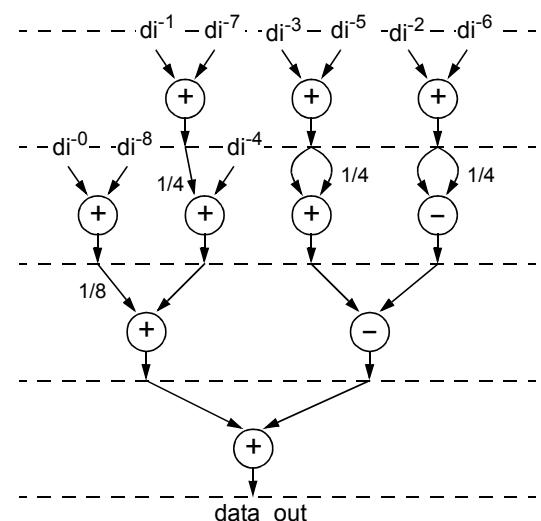
	bus 1	bus 2	bus 3	bus 4
1	$dx \rightarrow M.I$	$u \rightarrow M.r$	-	-
2	$5 \rightarrow M.I$	$x \rightarrow M.r, A.r$	$dx \rightarrow A.I$	$M \rightarrow tmp$
3	$tmp \rightarrow M.I$	$M \rightarrow M.r$	$A \rightarrow A.I, x$	$a \rightarrow A.r$
4	$3 \rightarrow M.I$	$y \rightarrow M.r, A.r$	$tmp \rightarrow A.I$	$M \rightarrow tmp$
5	$dx \rightarrow M.I$	$M \rightarrow M.r$	$u \rightarrow A.I$	$tmp \rightarrow A.r$
6	-	$M \rightarrow A.r$	$A \rightarrow A.I$	-



Binding example #1

- **FIR filter**
 - $\text{data_out} = 0.125*di^{-0} + 0.25*di^{-1} - 0.75*di^{-2} + 1.25*di^{-3} + 1.0*di^{-4} + 1.25*di^{-5} - 0.75*di^{-6} + 0.25*di^{-7} + 0.125*di^{-8}$
- **transformations**
 - shift-add trees & input swapping
- **10 operations & 9 variables**

	additions	subtractions
1	$v1=di^{-1}+di^{-7}; v2=di^{-2}+di^{-6}; v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}; v5=di^{-4}+v1/4; v6=v3+v3/4$	$v7=v2-v2/4$
3	$v8=v4/8+v5$	$v9=v6-v7$
4	$\text{data_out}=v8+v9$	



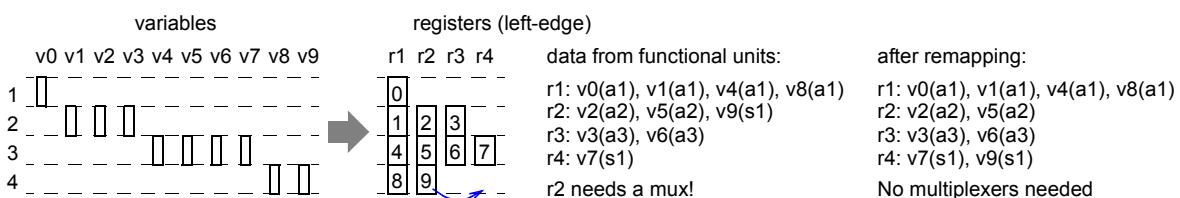


Binding example #1 (cont.)

- 4 steps, 10 operations, 9 variables
 - Assumptions – sample in (di^{-0}) & result out ($v0$) at step 1; di^{-n} are shifted at step 4

	additions	subtraction
1	$v1 = di^{-1} + di^{-7}$; $v2 = di^{-2} + di^{-6}$; $v3 = di^{-3} + di^{-5}$	
2	$v4 = di^{-0} + di^{-8}$; $v5 = di^{-4} + v1/4$; $v6 = v3 + v3/4$	$v7 = v2 - v2/4$
3	$v8 = v4/8 + v5$	$v9 = v6 - v7$
4	$data_out = v8 + v9$	

	add #1	add #2	add #3	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	$v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}$	$v5=di^{-4}+v1/4$	$v6=v3+v3/4$	$v7=v2-v2/4$
3	$v8=v4/8+v5$			$v9=v6-v7$
4	$v0=v8+v9$			



© Peeter Ellervee / Kalle Tammemäe

vlsi - allocation & binding - 23



Binding example #1 (cont.)

- Storing data from functional units into registers

data from functional units:

r1: v0(a1), v1(a1), v4(a1), v8(a1)	r1	r2	r3	r4
r2: v2(a2), v5(a2)	0	1	2	3
r3: v3(a3), v6(a3)	4	5	6	7
r4: v7(s1), v9(s1)	8			9



	r1	r2	r3	r4
1	a1	a2	a3	--
2	a1	a2	a3	s1
3	a1	--	--	s1
4	a1	--	--	--

`a1` - writes new value
`[]` - keeps previous value
`--` - don't care

- Multiplexers at functional units' inputs (plus shifters '>')

	add #1	add #2	add #3	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	$v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}$	$v5=di^{-4}+v1/4$	$v6=v3+v3/4$	$v7=v2-v2/4$
3	$v8=v4/8+v5$			$v9=v6-v7$
4	$v0=v8+v9$			

	a1 L	a1 R	a2 L	a2 R	a3 L	a3 R	s1 L	s1 R
1	di ⁻¹	di ⁻⁷	di ⁻²	di ⁻⁶	di ⁻³	di ⁻⁵	--	--
2	di ⁻⁰	di ⁻⁸	di ⁻⁴	r1>	r3	r3>	r2	r2>
3	r1>	r2	--	--	--	--	r3	r4
4	r1	r4	--	--	--	--	--	--

- Components: 3 add, 1 sub, 4 reg, 2 4-mux, 6 2-mux
 - $3 \times 125 + 139 + 4 \times 112 + (2 \times 3 + 6) \times 48 = 1538$ (+controller)

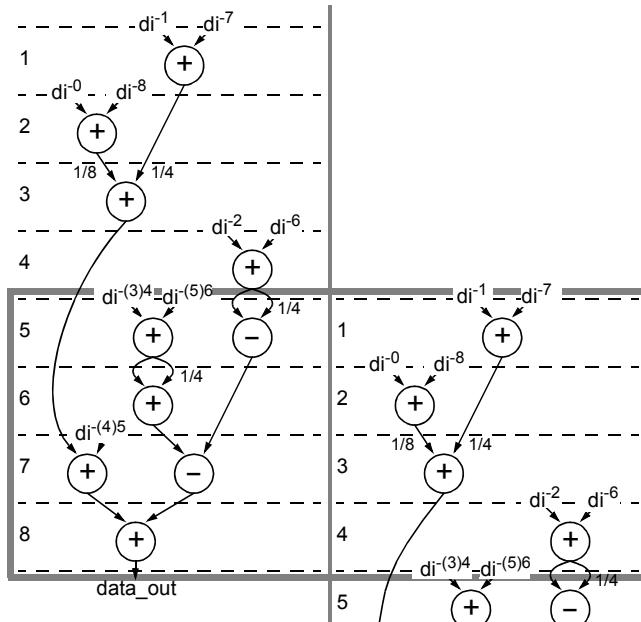
© Peeter Ellervee / Kalle Tammemäe

vlsi - allocation & binding - 24

Binding example #2

- Introducing pipelining – additional delay at the output
- Distribution of operations must be analyzed at both stages
- 10 operations & 9 variables

	additions	subtractions
1	$v_1 = di^{-1} + di^{-7}$	
(5)	$v_5 = di^{-4} + di^{-6}$	$v_6 = v_4 - (v_4/4)$
2	$v_2 = di^{-0} + di^{-8}$	
(6)	$v_7 = v_5 + (v_5/4)$	
3	$v_3 = (v_2/8) + (v_1/4)$	
(7)	$v_8 = v_3 + di^{-5}$	$v_9 = v_7 - v_6$
4	$v_4 = di^{-2} + di^{-6}$	
(8)	$v_4 = di^{-2} + di^{-6}$; data_out = $v_8 + v_9$	

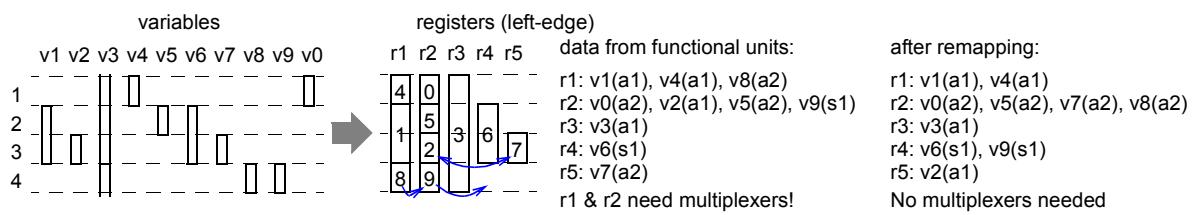


Binding example #2 (cont.)

- 4+4 steps, 10 operations, 9 variables
- Assumptions – sample in (di^{-0}) & result out (v_0) at step 1; di^{-n} are shifted at step 4

	additions	subtraction
1 (5)	$v_1 = di^{-1} + di^{-7}; v_5 = di^{-4} + di^{-6}$	$v_6 = v_4 - (v_4/4)$
2 (6)	$v_2 = di^{-0} + di^{-8}; v_7 = v_5 + (v_5/4)$	
3 (7)	$v_3 = (v_2/8) + (v_1/4); v_8 = v_3 + di^{-5}$	$v_9 = v_7 - v_6$
4 (8)	$v_4 = di^{-2} + di^{-6}$; data_out = $v_8 + v_9$	

	add #1	add #2	sub #1
1	$v_1 = di^{-1} + di^{-7}$	$v_5 = di^{-4} + di^{-6}$	$v_6 = v_4 - (v_4/4)$
2	$v_2 = di^{-0} + di^{-8}$	$v_7 = v_5 + (v_5/4)$	
3	$v_3 = (v_2/8) + (v_1/4)$	$v_8 = v_3 + di^{-5}$	$v_9 = v_7 - v_6$
4	$v_4 = di^{-2} + di^{-6}$	$v_0 = v_8 + v_9$	

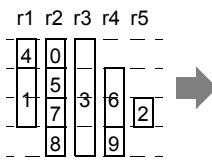


Binding example #2 (cont.)

- Storing data from functional units into registers

data from functional units:

r1: v1(a1), v4(a1)
 r2: v0(a2), v5(a2), v7(a2), v8(a2)
 r3: v3(a1)
 r4: v6(s1), v9(s1)
 r5: v2(a1)



	r1	r2	r3	r4	r5
1	a1	a2	[]	s1	--
2	[]	a2	[]	[]	a1
3	--	a2	a1	s1	--
4	a1	a2	[]	--	--

a1 - writes new value
 [] - keeps previous value
 -- - don't care

- Multiplexers at functional units' inputs (plus shifters '>')

	add #1	add #2	sub #1
1	$v1 = di^{-1} + di^{-7}$	$v5 = di^{-4} + di^{-6}$	$v6 = v4 - (v4/4)$
2	$v2 = di^{-0} + di^{-8}$	$v7 = v5 + (v5/4)$	
3	$v3 = (v2/8) + (v1/4)$	$v8 = v3 + di^{-5}$	$v9 = v7 - v6$
4	$v4 = di^{-2} + di^{-6}$	$v0 = v8 + v9$	

	a1 L	a1 R	a2 L	a2 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-4}	di^{-6}	r1	$r1 >$
2	di^{-0}	di^{-8}	$r2 >$	r2	--	--
3	$r5 >$	$r1 >$	r3	di^{-5}	r2	r4
4	di^{-2}	di^{-6}	r5	r4	--	--

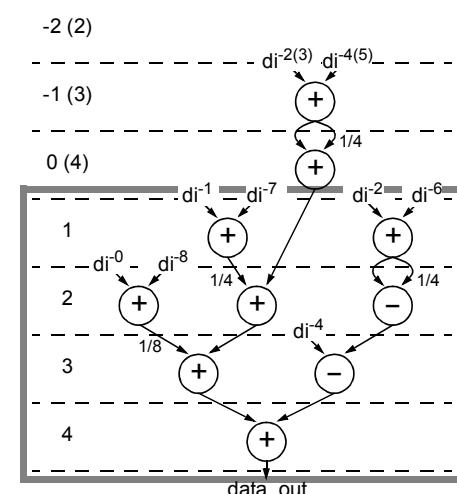
- Components: 2 add, 1 sub, 5 reg, 4 4-mux, 2 2-mux

- $2*125+139+5*112+(4*3+2)*48 = 1621$ (+controller) – less FU-s (-1) but more reg-s (+1) & mux-s (+3)

Binding example #3

- Out-of-order execution (functional pipelining)
- Earlier samples are available!
- 4 steps, 10 operations, 9 variables

	additions	subtractions
1	$v1 = di^{-1} + di^{-7}; v2 = di^{-2} + di^{-6}$	
2	$v3 = di^{-0} + di^{-8}; v4 = (v1/4) + v9$	$v5 = v2 - (v2/4)$
3	$v6 = (v3/8) + v4; [v8 = di^{-2} + di^{-4}]$	$v7 = di^{-4} - v5$
4	$v9 = v8 + (v8/4)$	

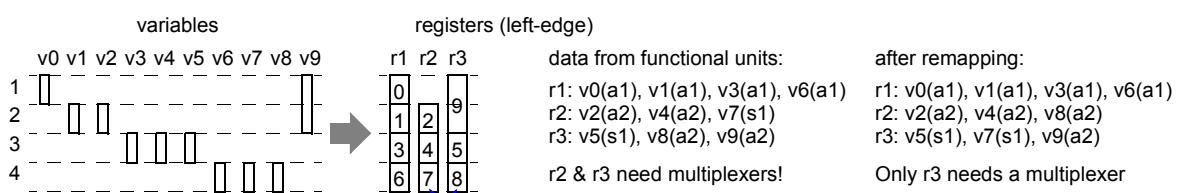


Binding example #3 (cont.)

- 4 steps, 10 operations, 9 variables, out-of-order execution
- Assumptions – sample in (di^{-0}) & result out ($v0$) at step 1; di^{-n} are shifted at step 4

	additions	subtraction
1	$v1=di^{-1}+di^{-7}; v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}; v4=(v1/4)+v9$	$v5=v2-(v2/4)$
3 (-1)	$v6=(v3/8)+v4; [v8=di^{-2}+di^{-4}]$	$v7=di^{-4}-v5$
4 (0)	$data_out=v6+v7; [v9=v8+(v8/4)]$	

	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$	$v4=(v1/4)+v9$	$v5=v2-(v2/4)$
3	$v6=(v3/8)+v4$	$v8=di^{-2}+di^{-4}$	$v7=di^{-4}-v5$
4	$v0=v6+v7$	$v9=v8+(v8/4)$	



Binding example #3 (cont.)

- Storing data from functional units into registers

data from functional units:
 r1: $v0(a1), v1(a1), v3(a1), v6(a1)$
 r2: $v2(a2), v4(a2), v8(a2)$
 r3: $v5(s1), v7(s1), v9(a2)$

	r1	r2	r3
0			
1	a1	a2	[]
2	a1	a2	s1
3	a1	a2	s1
4	a1	--	a2

	r1	r2	r3
1	a1	a2	[]
2	a1	a2	s1
3	a1	a2	s1
4	a1	--	a2

a1 - writes new value
 [] - keeps previous value
 -- - don't care

- Multiplexers at functional units' inputs (plus shifters ' $>$ ')

	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$	$v4=(v1/4)+v9$	$v5=v2-(v2/4)$
3	$v6=(v3/8)+v4$	$v8=di^{-2}+di^{-4}$	$v7=di^{-4}-v5$
4	$v0=v6+v7$	$v9=v8+(v8/4)$	

	a1 L	a1 R	a2 L	a2 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-2}	di^{-6}	--	--
2	di^{-0}	di^{-8}	$r1>$	$r3$	$r2$	$r2>$
3	$r1>$	$r2$	di^{-2}	di^{-4}	di^{-4}	$r3$
4	$r1$	$r3$	$r2$	$r2>$	--	--

- Components: 2 add, 1 sub, 3 reg, 3 4-mux, 1 3-mux, 3 2-mux
 - $2*125+139+3*112+(3*3+5)*48 = 1397$ (+controller) – less FU-s (-1) & reg-s (-1) but more mux-s (+2)



Creating synthesizable code

- Behavioral level code is not synthesizable
- Register-transfer level code is synthesizable
- What about “Behavioral RTL”?
 - ... or other intermediate levels
- Step-by-step code refinement
 - from idea to model
 - validating model's behavior by simulation
 - from model to structure
 - transforming behavioral level code into RT level code
 - pure RTL gives the best results (FSM & data-path == no ambiguities)
 - from structure to schematics (==synthesis)



Creating synthesizable code

- Use bit-vector data types
 - corresponds to actual implementation, e.g. no overflow detection
- Simplify behavioral hierarchy
 - avoid timing control in subroutines
- Introduce structural hierarchy
 - only few processes per design unit
 - one process would be ideal
- No tricks with clock signal(s)
- Follow coding rules to avoid
 - latches in combinational processes
 - duplication of registers
- Behavioral level construct

```
wait until sign_1 = val_2 for 25 sec;
```
- Behavioral RT level (not synthesizable)
 - timer & counter introduced

```
for counter in 0 to 49 loop -- 25 sec
    exit when sign_1 = val_2;
    wait on timer until timer='1';
end loop;
```
- Behavioral RT level (synthesizable)
 - synthesizable counter

```
counter := 0; -- 25 sec
while counter < 50 and
    sign_1 /= val_2 loop
    counter := counter + 1;
    wait on timer until timer='1';
end loop;
```
- Pure RTL == FSM + data-path
 - one ‘wait’ statement ~~ one state in FSM