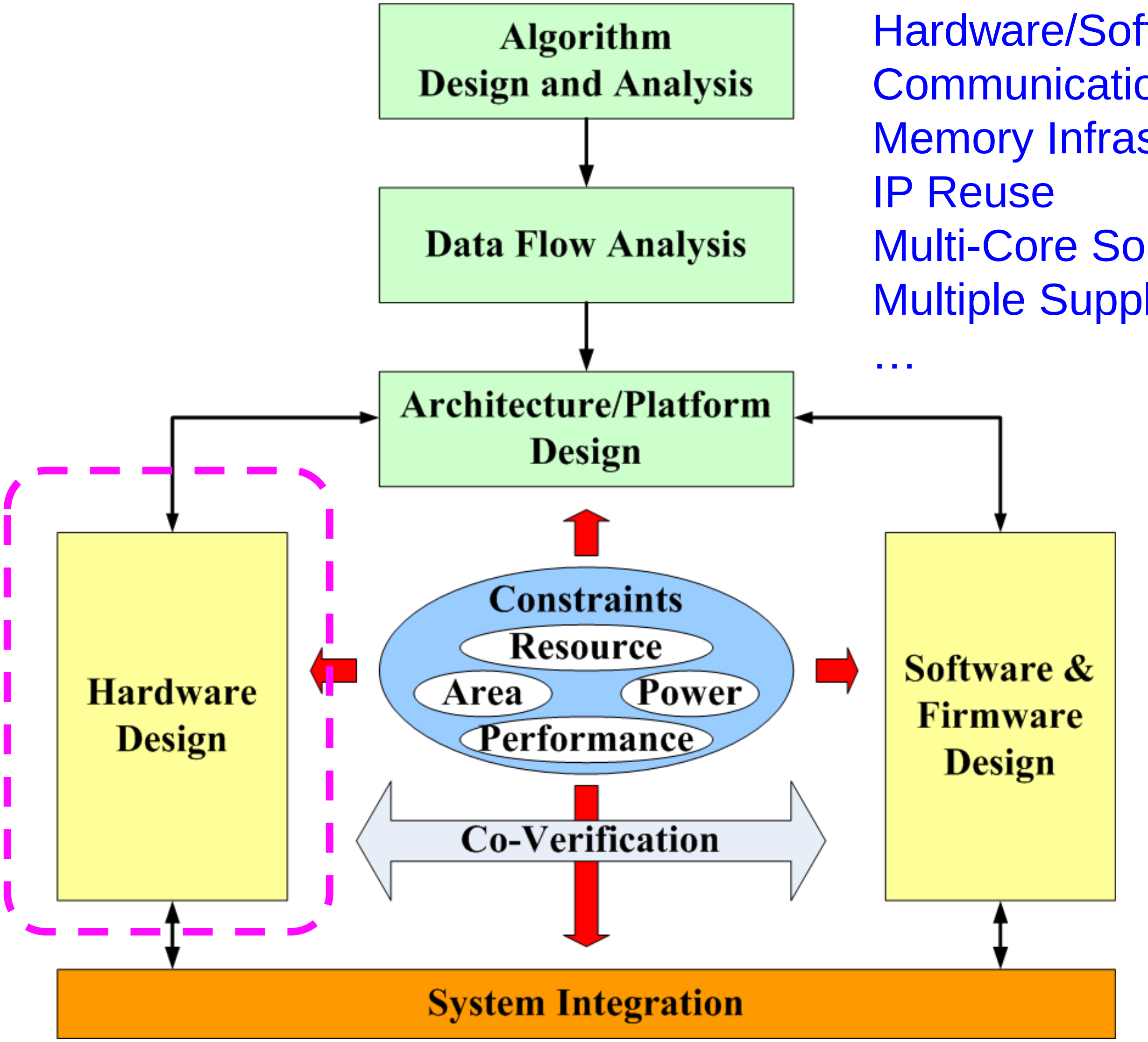
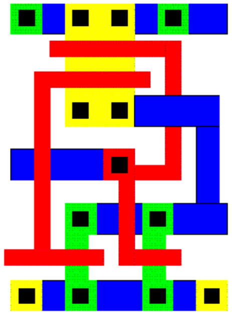
Introduction to High-level Synthesis

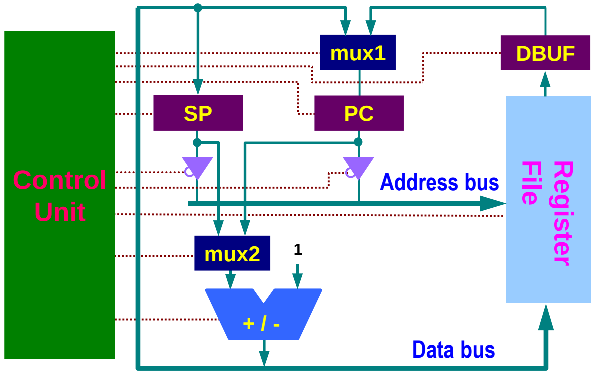
Design Flow for a Typical SoC

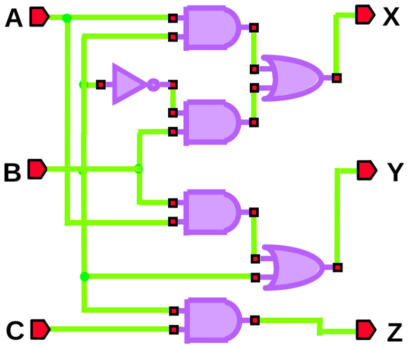


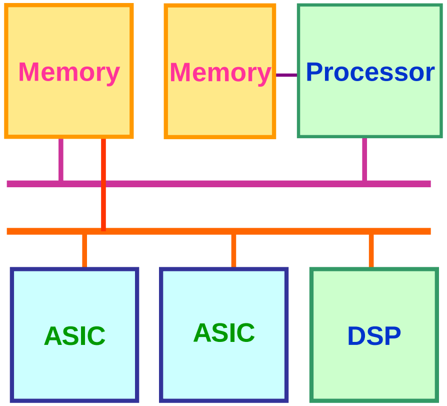
Hardware/Software Partitioning
Communication Synthesis
Memory Infrastructure
IP Reuse
Multi-Core SoC
Multiple Supply Voltages
…

Evolution of IC Design



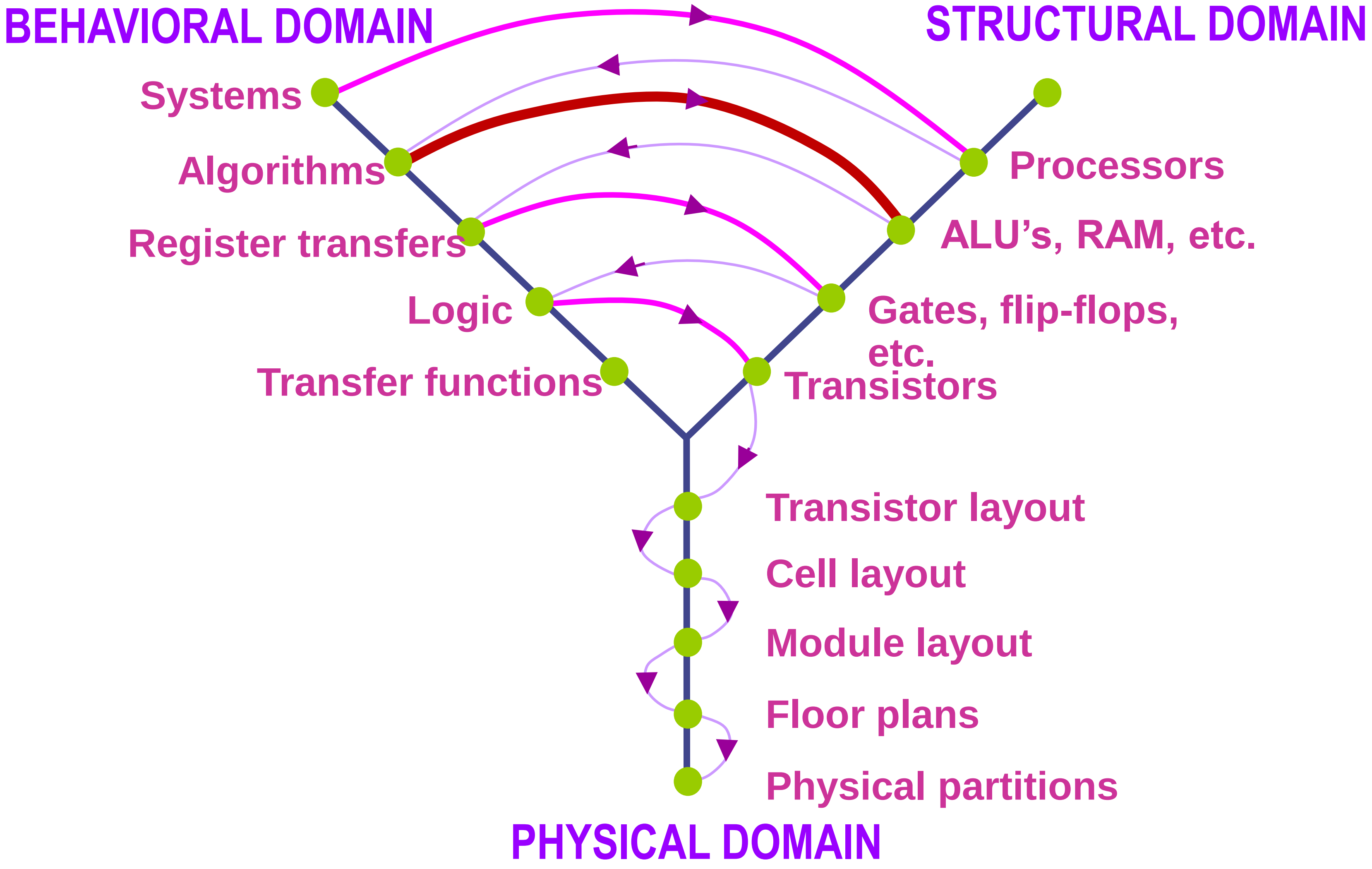




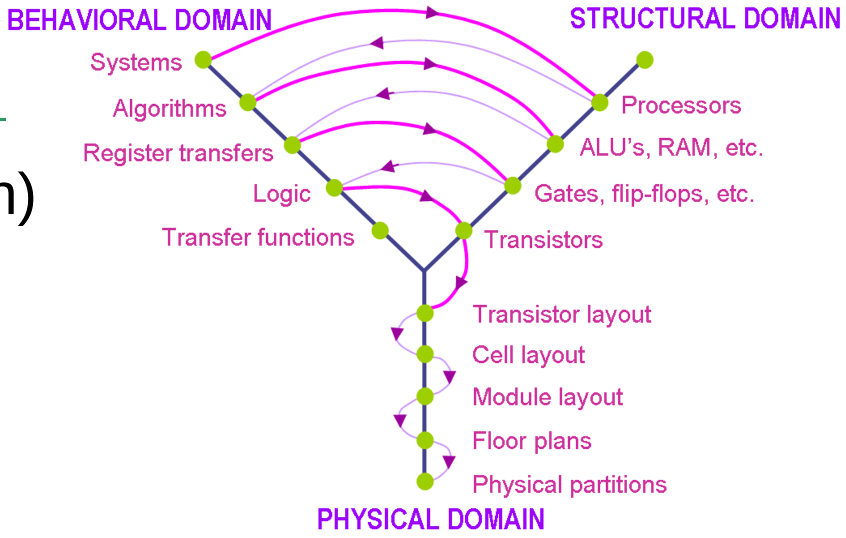


microcells → macro-cells → IP cores → SoC

Design Flow

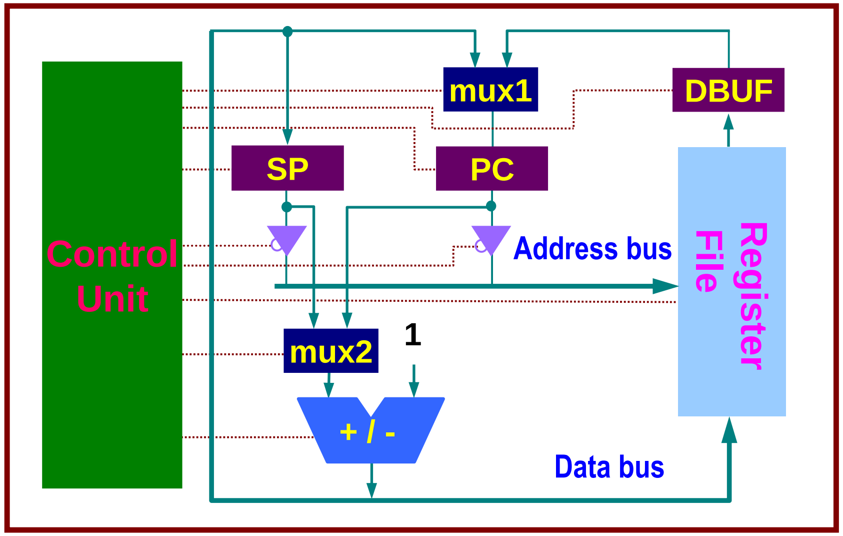

BEHAVIORAL DOMAIN
STRUCTURAL DOMAIN
Systems
Algorithms
Register transfers
Logic
Transfer functions
Processors
ALU’s, RAM, etc.
Gates, flip-flops, etc.
Transistors
Transistor layout
Cell layout
Module layout
Floor plans
Physical partitions
PHYSICAL DOMAIN

Algorithm-Level Design



- Specification (behavioral description)
  - Programming languages
    - C/C++ or Pascal
  - Hardware description language
    - VHDL or Verilog
- High-level (or Behavioral) synthesis
  - algorithmic behavioral level ⇒ RTL structural descriptions

```
if IR(3) = '0' then
      PC          := PC + 1;
else
      DBUF        := MEM(PC);
      MEM(SP)     := PC + 1;
      SP          := SP – 1;
      PC          := DBUF;
end if;
```



Examples

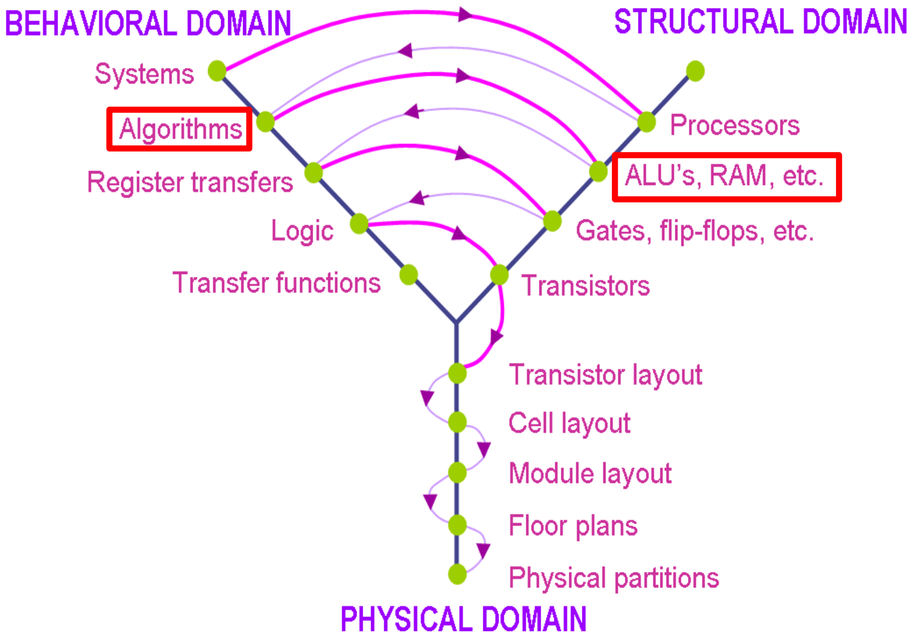
```
while (x<a) loop
    x1 :=  x + dx;
    u1 :=  u – (3*x*u*dx) – (3*y*dx);
    y1 :=  y + (u*dx);
    x :=  x1;
    u :=  u1;
    y :=  u1;
end loop;
```

$$
\begin{aligned}
Y &= 0.299\times R+0.587\times G+0.114\times B \\
U &= -0.169\times R-0.331\times G+0.5\times B+128 \\
V &= 0.5\times R-0.419\times G-0.081\times B+128
\end{aligned}
$$

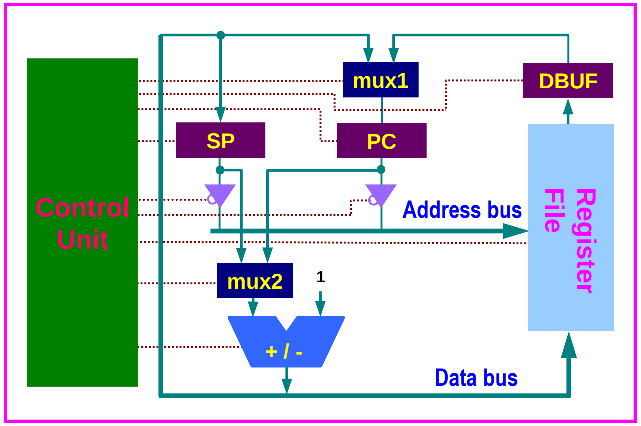
```
           A = Inport;
           B = Inport;
           done = 0;
Repeat:    while ($A \geq B$)
               A = A – B;
           R = A;
           if (A != 0) {
               A = B;
               B = R;
               goto Repeat;
           }
           else  goto End;
End:       Outport = B;
           done = 1;
```

High-level Synthesis (1/4)

- High-level Synthesis (Behavioral Synthesis)
  - The process of mapping a behavioral description at the algorithmic level to a structural description (RTL) in terms of functional units, memory elements and interconnections (e.g. multiplexers and buses)



```
if IR(3) = '0' then
    PC          := PC + 1;
else
    DBUF        := MEM(PC);
    MEM(SP)     := PC + 1;
    SP          := SP – 1;
    PC          := DBUF;
end if:
```



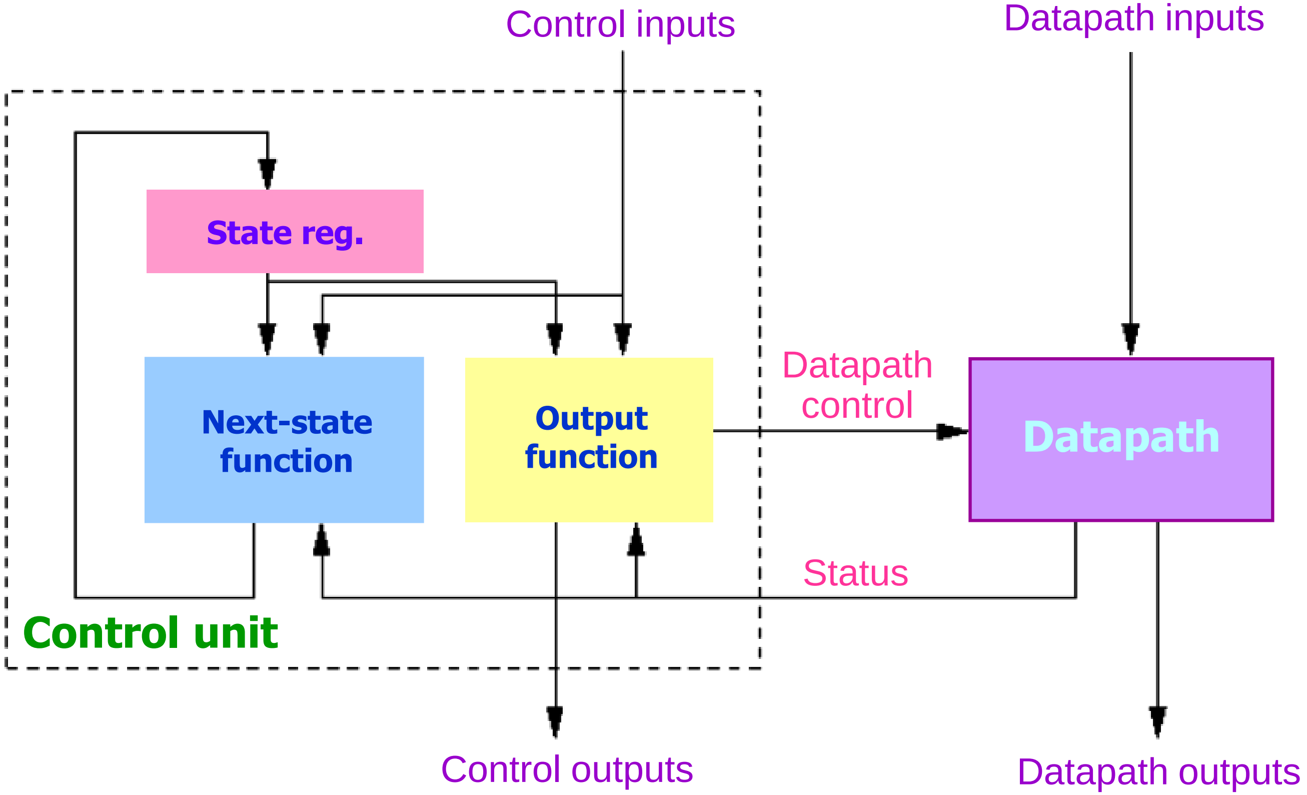
High-level Synthesis (2/4)

- Goals and Terminology
  - Mapping of the internal description to a hardware configuration that obeys the hardware model
    - each operation in DFG: time step, functional unit
    - remaining part of the hardware configuration: memory elements, interconnections
  - Data-dominated and control-dominated

- Hardware is normally partitioned into two parts:
  - Datapath
    - a network of functional units, registers, multiplexers and buses
    - The actual ‘‘computation’’ takes place in the data path
  - Control unit
    - the part of the hardware that takes care of having the data present at the right place at a specific time, of presenting the right instructions to a programmable unit, etc.

High-level Synthesis (3/4)

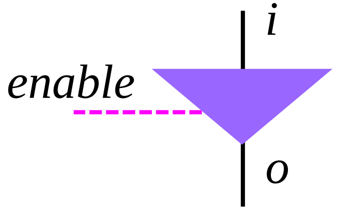
- Finite State Machines with a Datapath (FSMD)



FSMD = Control unit + Data Path

High-level Synthesis (4/4)

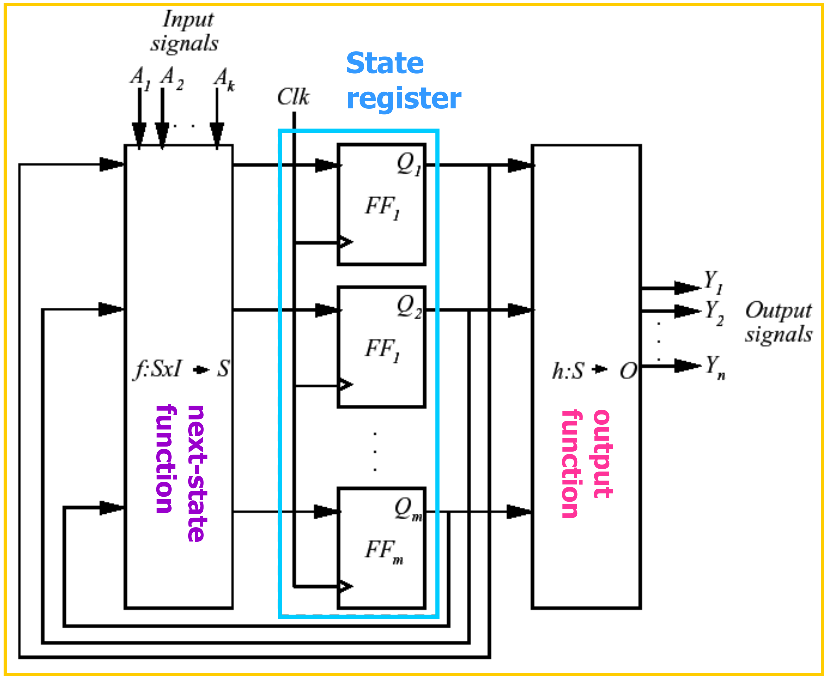
- The actual mapping consists of scheduling and allocation
- Most systems generate synchronous hardware and build it with the following parts:
  - Functional units: they can perform one or more computations, e.g. addition, multiplication, comparison, ALU
  - Registers: they store inputs, intermediate results and outputs; sometimes several registers are taken together to form a register file
  - Multiplexers: from several inputs, one is passed to the output
  - Busses: a connection shared between several hardware elements, such that only one element can write data at a specific time
    - Three-state (tri-state) drivers: control the exclusive writing on the bus

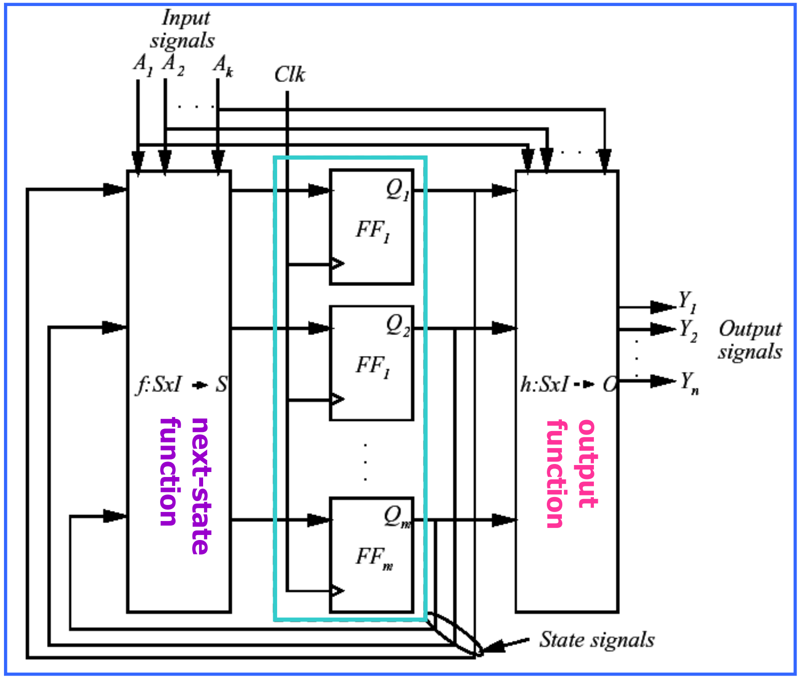


Three-state (tri-state) drivers

Control Unit

- $<S, I, O, f, h>$
  - S, I, O: a set of states, a set of inputs, and a set of output
  - f, h: next-state and output function
- State-based (Moore) FSM: $h: S \rightarrow O$
- Input-based (Mealy) FSM: $h: S \times I \rightarrow O$



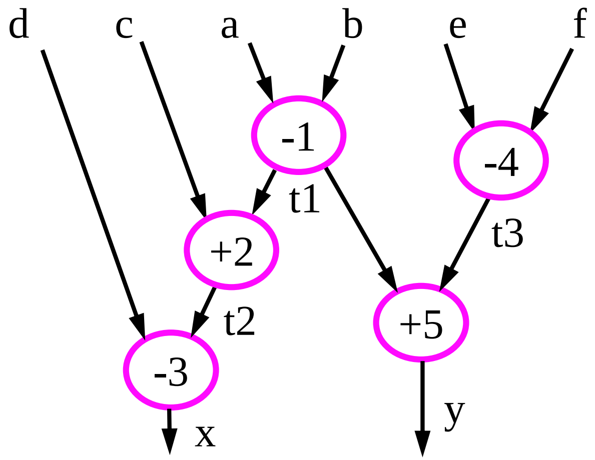


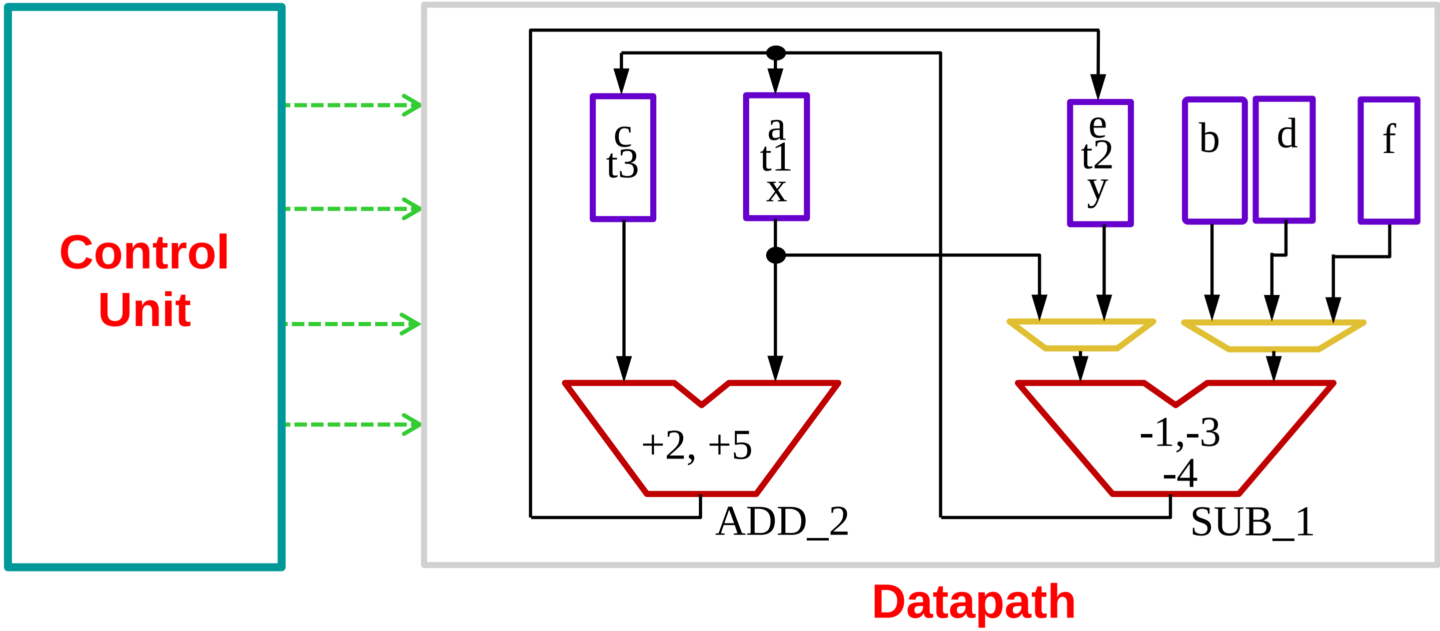
Subtasks in Behavioral Synthesis

- Scheduling
  - determine for each operation the time at which it should be performed such that no precedence constraint is violated
- Allocation (Datapath synthesis, Datapath allocation)
  - map each operation to a specific functional unit, each variable to a register, and data transfers to interconnections
  - module selection, functional unit allocation, storage allocation, interconnection allocation
- Control unit synthesis

Simple Example

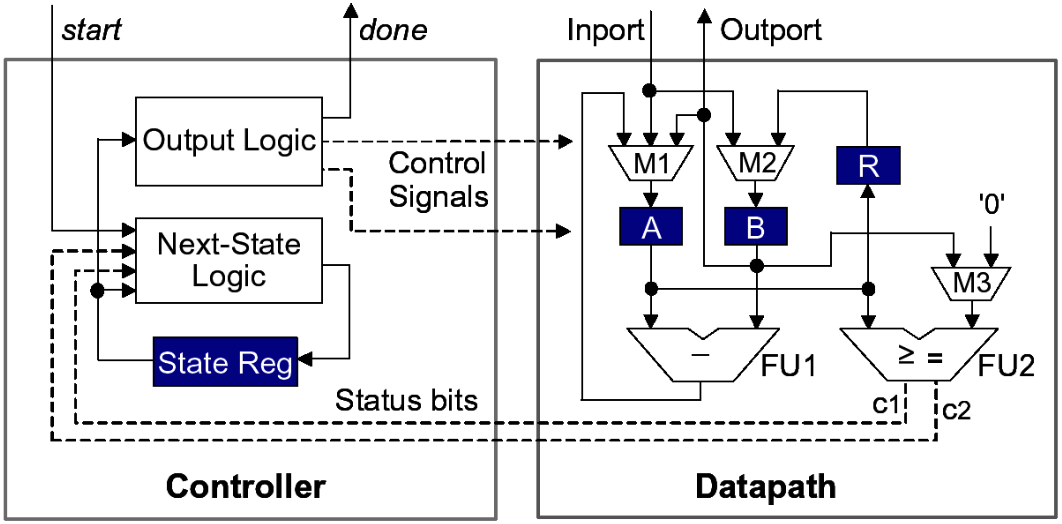
$$t1 = a - b;$$
$$x = t1 + c - d;$$
$$y = e - f + t1;$$

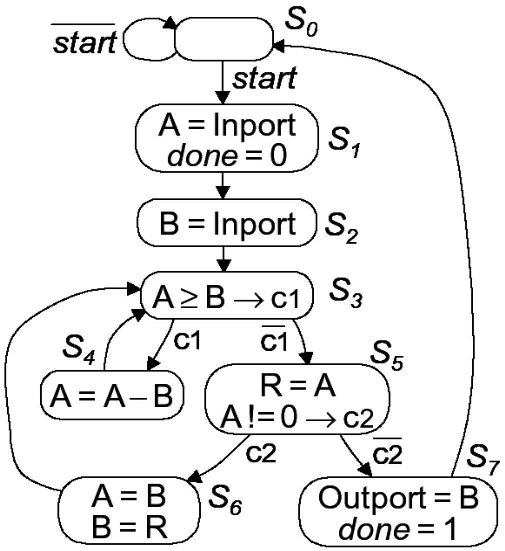




GCD Example

```
         A = Inport;
         B = Inport;
         done = 0;
Repeat:  while (A ≥ B)
             A = A – B;
         R = A;
         if (A != 0) {
             A = B;
             B = R;
             goto Repeat;
         }
         else  goto End;
End:      Outport = B;
         done = 1;
```





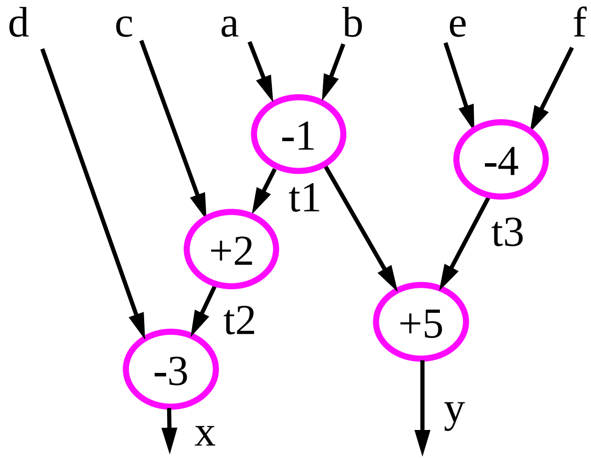
PS	Input	NS	Control signals						
			A	B	R	M1	M2	M3	*done*
S_0	*start* = 0	S_0	-	-	-	-	-	-	0
	start = 1	S_1							
S_1	-	S_2	1	-	-	Inport	-	-	0
S_2	-	S_3	0	1	-	-	Inport	-	0
S_3	c1 = 1	S_4	0	0	0	-	-	B	0
	c1 = 0	S_5							
S_4	-	S_3	1	0	0	FU1	-	-	0
S_5	c2 = 1	S_6	0	0	1	-	-	'0'	0
	c2 = 0	S_7							
S_6	-	S_3	1	1	0	B	R	-	0
S_7	-	S_0	0	0	0	-	-	-	1

Optimization Issues

- The objective function to be optimized is similar to the one for the design of VLSI circuits in general
  - It is a combination of speed, area and power consumption
- Often optimization is constrained
  - **Time-constrained** synthesis: optimize area when the minimum speed is given
  - **Resource-constrained** synthesis: optimize speed when a maximum for each resource type is given
    - another version of resource-constrained synthesis: limit the overall cost

A Simple Example (1/3)

$$t1 = a - b;$$
$$x = t1 + c - d;$$
$$y = e - f + t1;$$



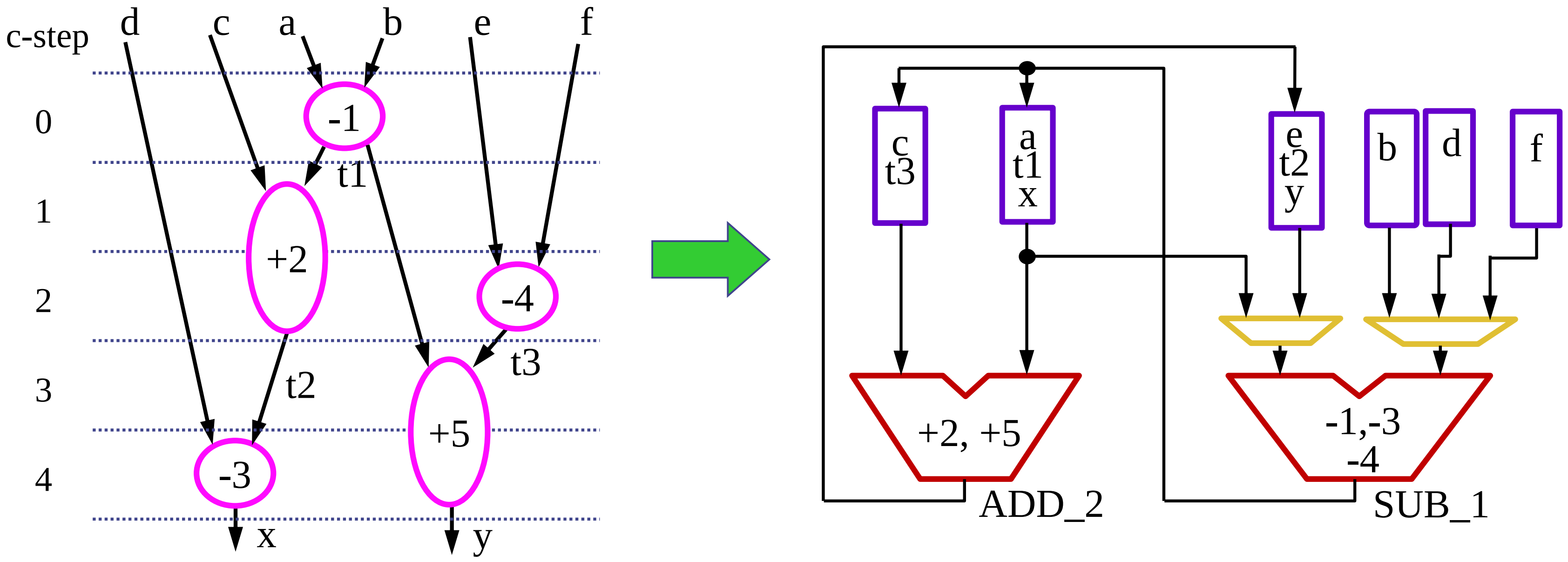
Clock cycle = 10ns, Latency = 5, and time constraint = 50ns

m-type	ADD_1	ADD_2	SUB_1	SUB_2	SUB_3	Reg	Mux_2	Mux_3	wire
index	1	2	3	4	5				
cost (area)	20	10	24	12	6	15	8	12	10
delay (ns)	7	16	7	16	38	1/1	1.5	2	3
power (nW)	0.62	0.36	0.68	0.38	0.25				

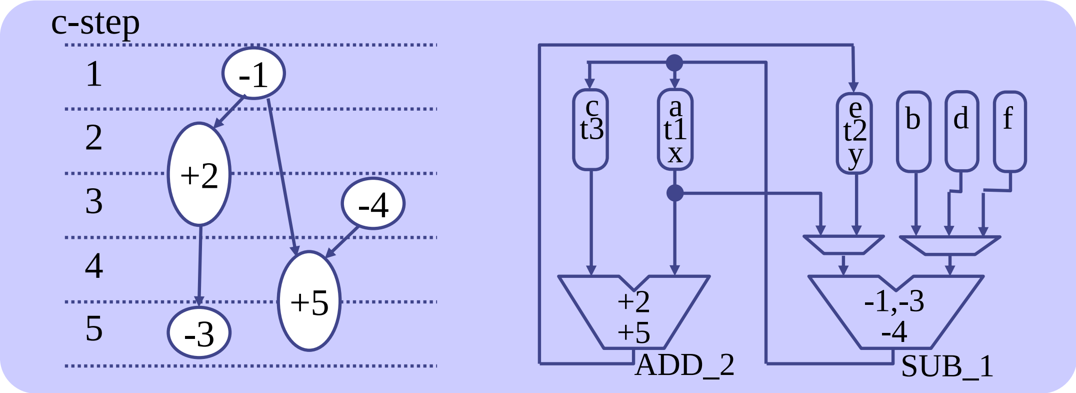
A Simple Example (2/3)

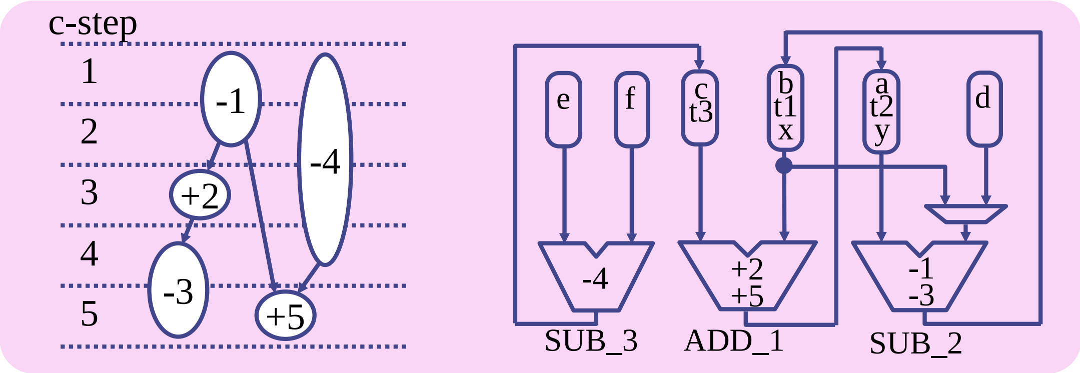
Clock cycle = 10ns, Latency = 5, and time constraint = 50ns

m-type	ADD_1	ADD_2	SUB_1	SUB_2	SUB_3	Reg	Mux_2	Mux_3	wire
index	1	2	3	4	5				
cost (area)	20	10	24	12	6	15	8	12	10
delay (ns)	7	16	7	16	38	1/1	1.5	2	3
power (nW)	0.62	0.36	0.68	0.38	0.25				



A Simple Example (3/3)



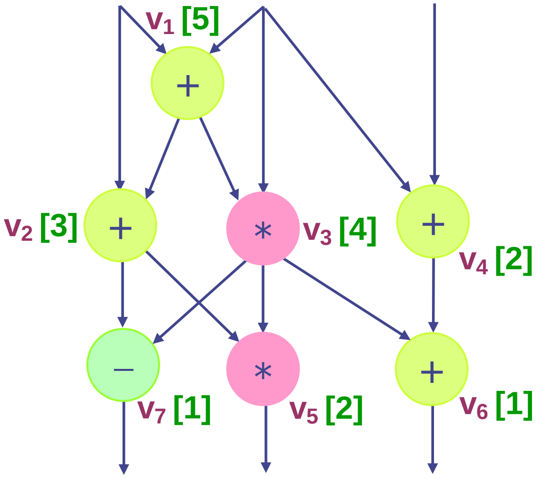


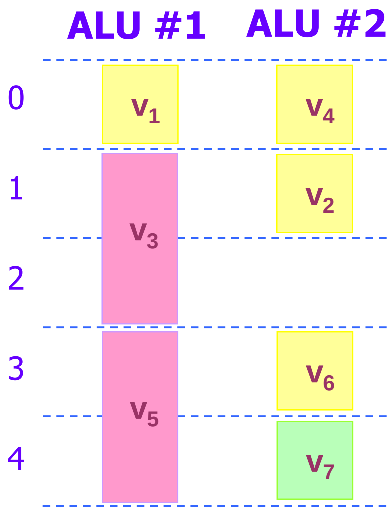
Circuit	**1**	**2**
module cost	34	38
MUX cost	20	8
wire cost	120	110
Register cost	90	90
Total cost	**264**	**246**

List Scheduling (1/2)

- List Scheduling
  - A resource-constrained scheduling method
  - Start at time zero and increase time until all operations have been scheduled
  - The ready list contains all operations that can start their execution at the current time step or later
  - If more operations are ready than there are resources available, use some priority function to choose, e.g. the longest-path to the output node ⇒ *critical-path list scheduling*

List Scheduling (2/2)





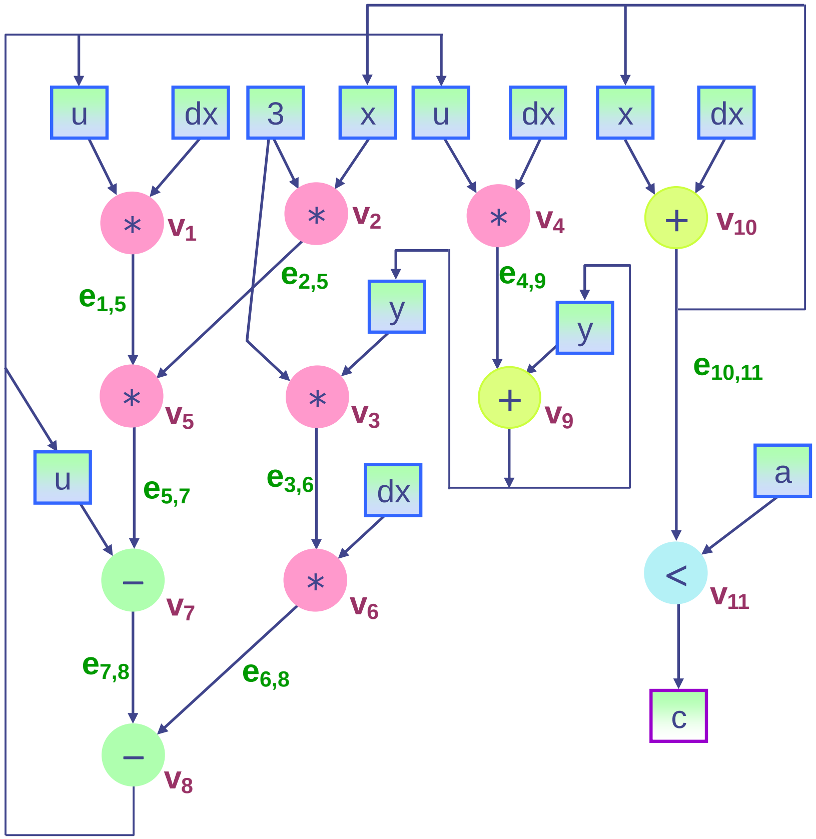
t	Ready List
0	{ v_1 [5], v_4 [2] }
1	{ v_3 [4], v_2 [3] }
2	ϕ
3	{ v_5 [2], v_6 [1], v_7 [1] }
4	{ v_7 [1] }

Time-constrained Scheduling

- Scheduling example

```
while (x<a) loop
      x1 :=  x + dx;
      u1 :=  u – (3*x*u*dx) – (3*y*dx);
      y1 :=  y + (u*dx);
      x :=  x1;
      u :=  u1;
      y :=  u1;
end loop;
```

VHDL Behavior

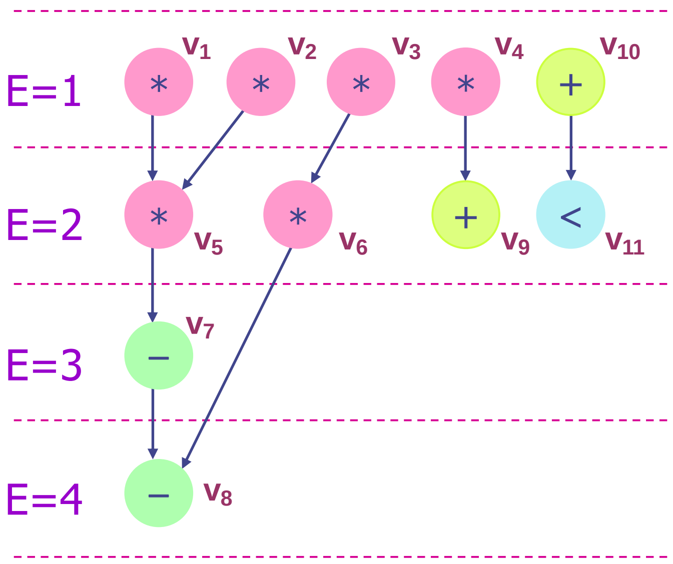


DFG Representation

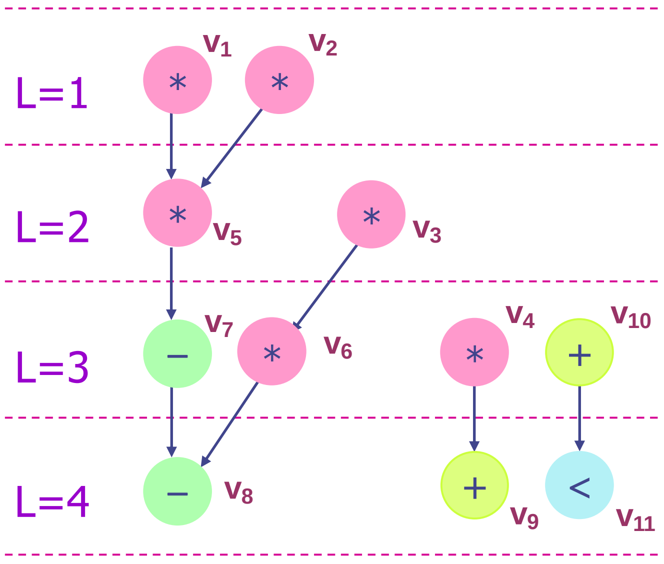
ASAP and ALAP Scheduling (1/2)

- ASAP Scheduling
  - As soon as possible (ASAP) scheduling maps an operation to the earliest possible starting time not violating the precedence constraints
  - It is easy to compute by finding the longest paths in a directed acyclic graph
  - It does not make any attempt to optimize the resource cost
- ALAP Scheduling
  - As late as possible (ALAP) scheduling

ASAP and ALAP Scheduling (2/2)



ASAP Scheduling



ALAP Scheduling

Integer Linear Programming Method

- Integer Linear Programming Method

$$\boxed{\textit{Minimize} \quad \sum_{k=1}^{m} (C_{t_k} \times N_{t_k})}$$

N_{t_k}: the number of units performing operation t_k
C_{t_k}: the cost of unit

(1) $\forall i, 1 \le i \le n, \quad \sum_{E_i \le j \le L_i} x_{i,j} = 1$ Time

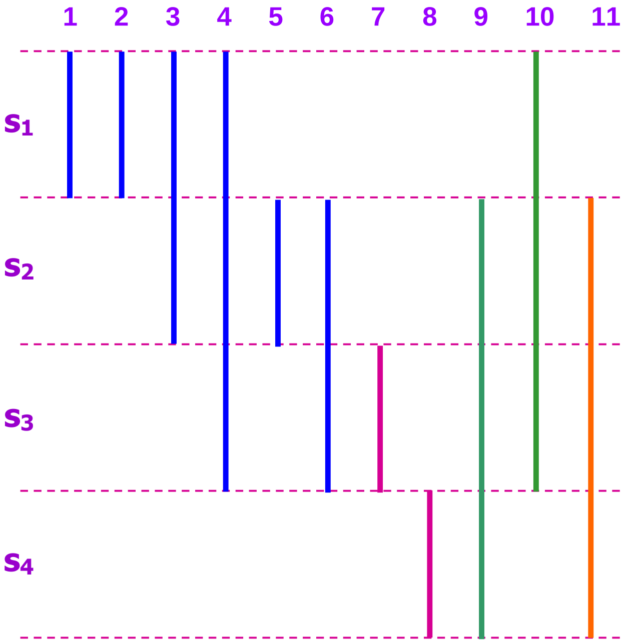
(2) $\forall j, 1 \le j \le r,$
$\forall k, 1 \le k \le m, \quad \sum_{i \in INDEX_{t_k}} x_{i,j} \le N_{t_k}$ Resource

(3) $\forall i, j, o_i \in Pred_{o_j} \quad \sum_{E_i \le k \le L_i} (k \times x_{i,k}) - \sum_{E_j \le l \le L_j} (l \times x_{j,l}) \le -1$

Precedence $k < l$

Minimize

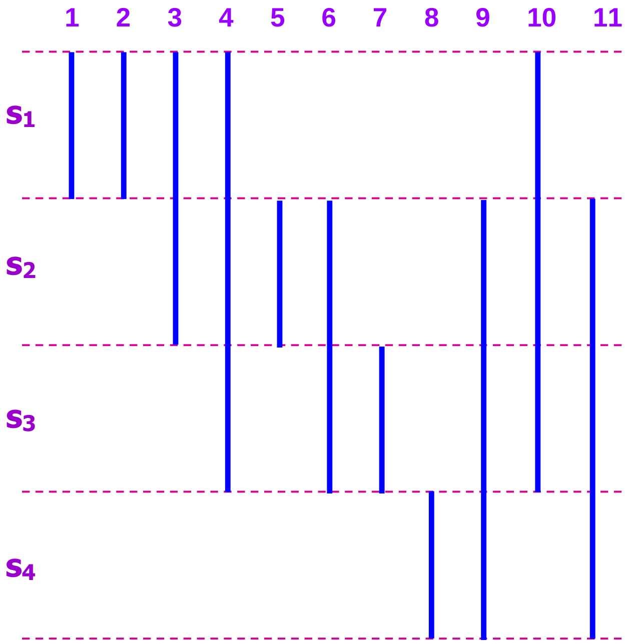
$$C_m \times N_m + C_a \times N_a + C_s \times N_s + C_c \times N_c$$

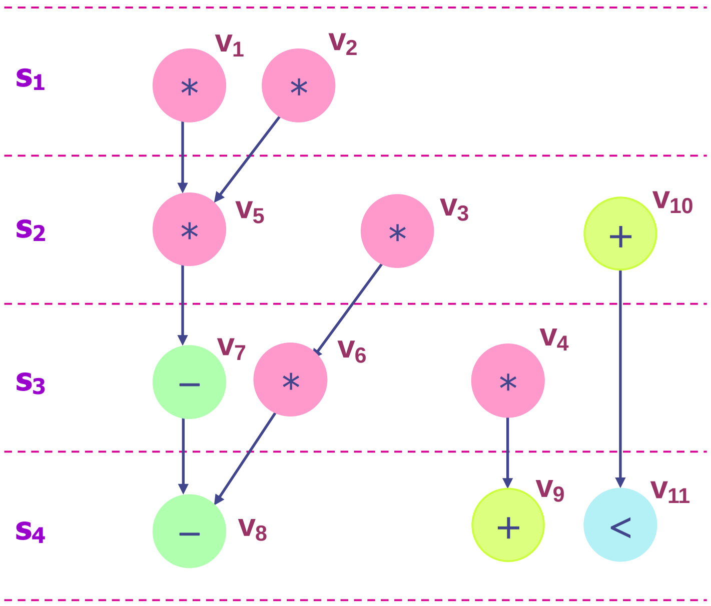


$$\left.\begin{aligned}
x_{1,1} &= 1\\
x_{2,1} &= 1\\
x_{3,1} + x_{3,2} &= 1\\
x_{4,1} + x_{4,2} + x_{4,3} &= 1\\
x_{5,2} &= 1\\
x_{6,2} + x_{6,3} &= 1\\
x_{7,1} &= 1\\
x_{8,4} &= 1\\
x_{9,2} + x_{9,3} + x_{9,4} &= 1\\
x_{10,1} + x_{10,2} + x_{10,3} &= 1\\
x_{11,2} + x_{11,3} + x_{11,4} &= 1
\end{aligned}\right\} \quad (1)$$

$$\left.\begin{aligned}
x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} &\le N_m\\
x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} &\le N_m\\
x_{4,3} + x_{6,3} &\le N_m\\
x_{7,3} &\le N_s\\
x_{8,4} &\le N_s\\
x_{10,1} &\le N_a\\
x_{9,2} + x_{10,2} &\le N_a\\
x_{9,3} + x_{10,3} &\le N_a\\
x_{9,4} &\le N_a\\
x_{11,2} &\le N_c\\
x_{11,3} &\le N_c\\
x_{11,4} &\le N_c
\end{aligned}\right\} \quad (2)$$

$$
\begin{aligned}
1x_{3,1} + 2x_{3,2} - 2x_{6,2} - 3x_{6,3} &\le -1 \\
1x_{4,1} + 2x_{4,2} + 3x_{4,3} - 2x_{9,2} - 3x_{9,3} - 4x_{9,4} &\le -1 \qquad (3) \\
1x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} &\le -1
\end{aligned}
$$





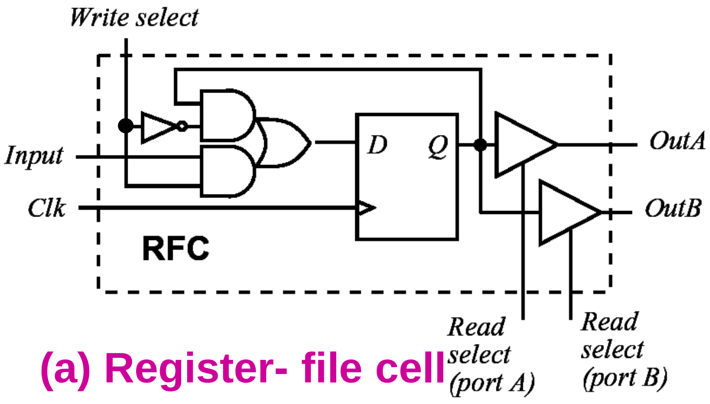
Subtasks in Allocation (1/2)

- Unit selection
  - selects the number and types of different functional and storage units form the RT component library
  - RT component library contains multiple types of functional units with different characteristics (e.g., functionality, size, delay, power, …)
- Operation-to-FU assignment (Functional-unit binding)
  - map a computation to an FU of an appropriate type
- Value grouping (Storage binding)
  - A subset does not contain values that are read or written simultaneously ⇒ is realized as a *register bank*
  - *Multiport memories*: the conditions for grouping should be adapted accordingly
- Value-to-register assignment (Storage binding)
  - Assign a memory location to storage values in the same group
  - Values with nonoverlapping *lift times* can share the same location

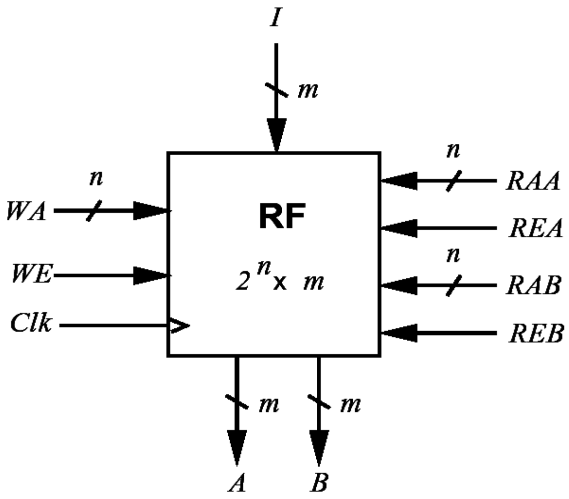
Subtasks in Allocation (2/2)

- Transfer-to-wire assignment (Interconnection binding)
  - Transfer: the actual transport of data from one hardware unit to another
  - Bus-based architecture: choose which bus to write
    - Three-state drivers: connect to the unit from which the transfer originates
  - Mux-oriented architecture
    - Multiplexers: connect to the unit receiving the transfer

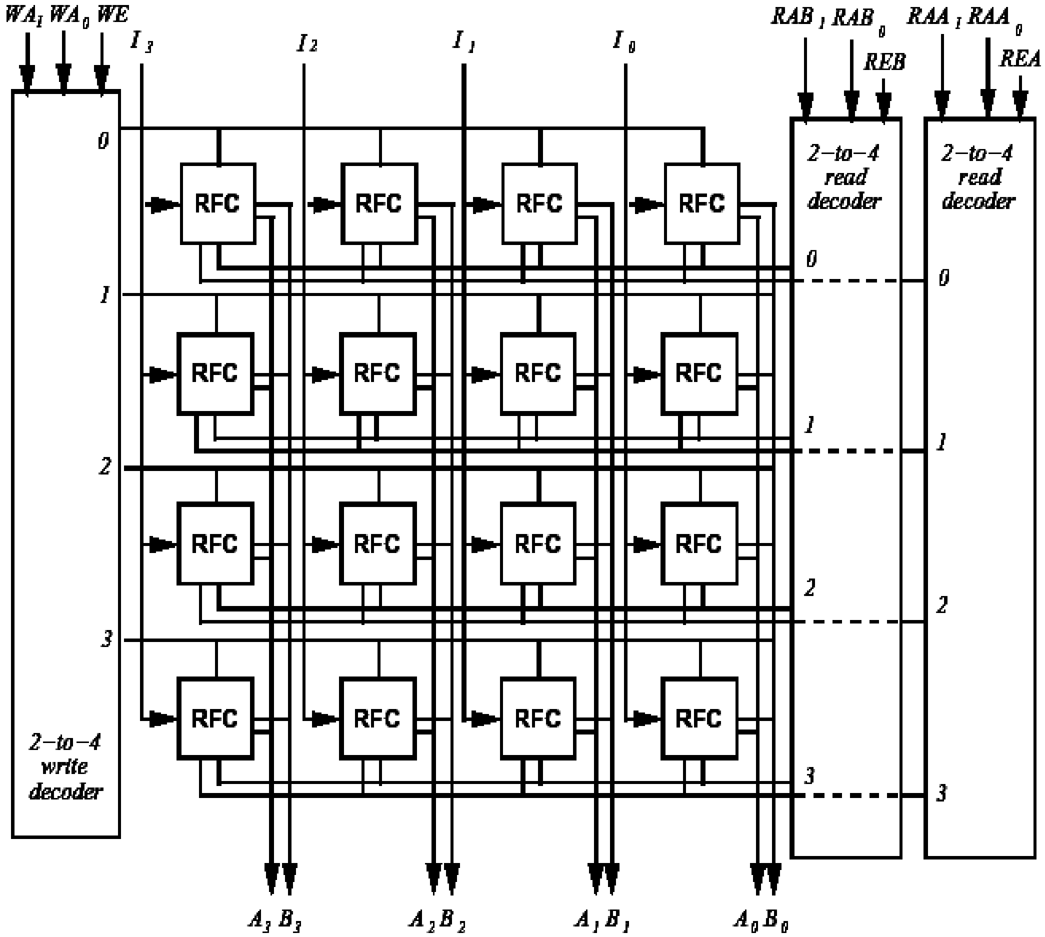
Register-file with 1 write port and 2 read ports



(a) Register- file cell

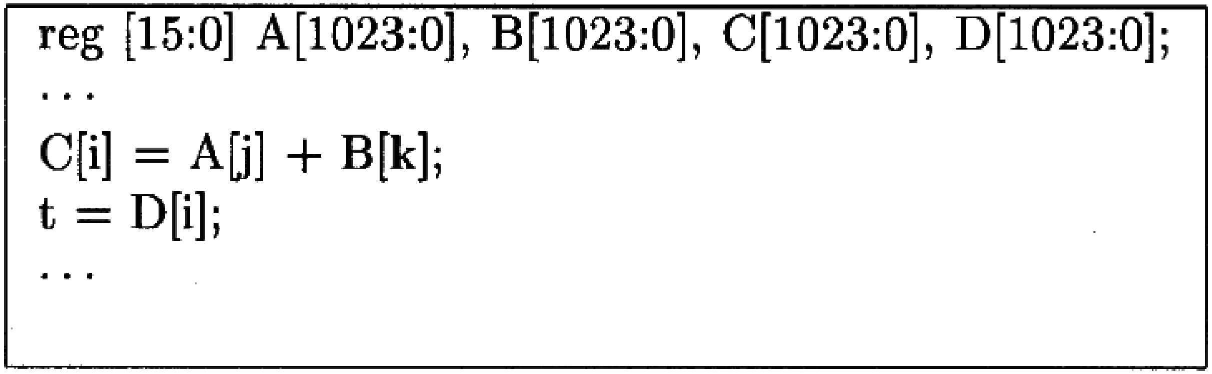


(b) Graphic symbol

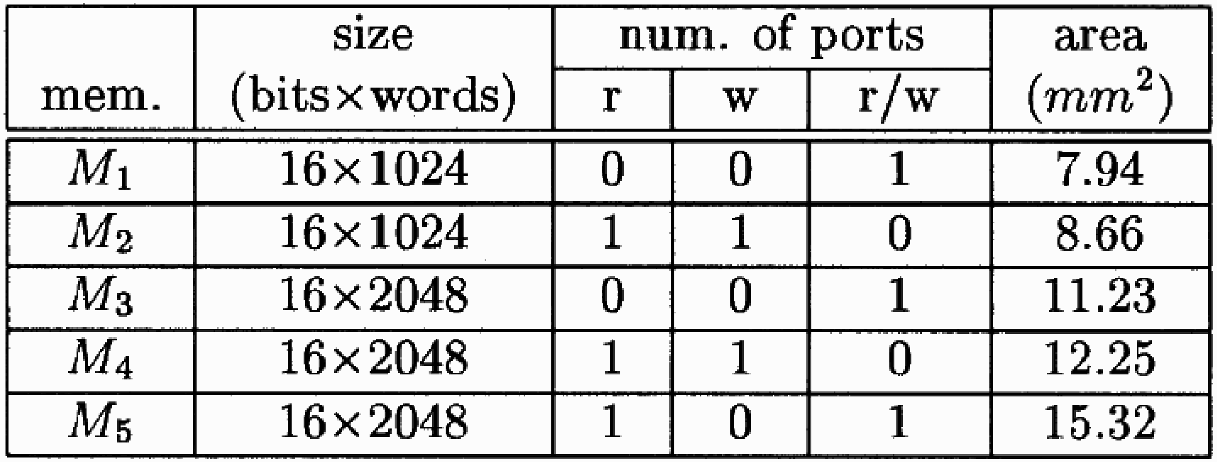


(c) Logic Schematic

Memory Allocation (1/2)



```
reg [15:0] A[1023:0], B[1023:0], C[1023:0], D[1023:0];
...
C[i] = A[j] + B[k];
t = D[i];
...
```

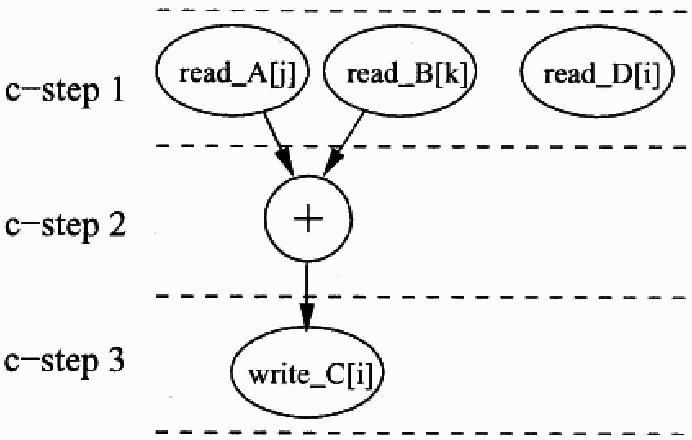


	size	num. of ports			area
mem.	(bits×words)	r	w	r/w	(mm^2)
M_1	16×1024	0	0	1	7.94
M_2	16×1024	1	1	0	8.66
M_3	16×2048	0	0	1	11.23
M_4	16×2048	1	1	0	12.25
M_5	16×2048	1	0	1	15.32

Memory Allocation (2/2)

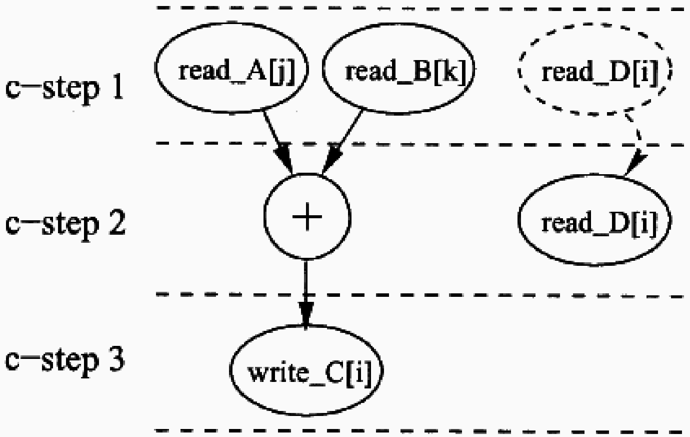
mem.	size (bits×words)	num. of ports			area (mm^2)
		r	w	r/w	
M_1	16×1024	0	0	1	7.94
M_2	16×1024	1	1	0	8.66
M_3	16×2048	0	0	1	11.23
M_4	16×2048	1	1	0	12.25
M_5	16×2048	1	0	1	15.32

schedule–1



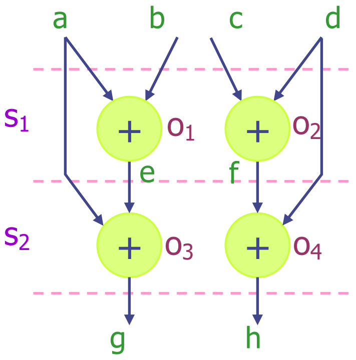
mem. configuration (array group : memory)	area (mm^2)
A:M_1, B:M_1, C:M_1, D:M_1	31.76
A:M_1, B:M_1, CD:M_3	27.11
A:M_1, C:M_1, BD:M_5	31.20
...	...
AB:M_5, CD:M_3	**26.55**
...	...

schedule–2

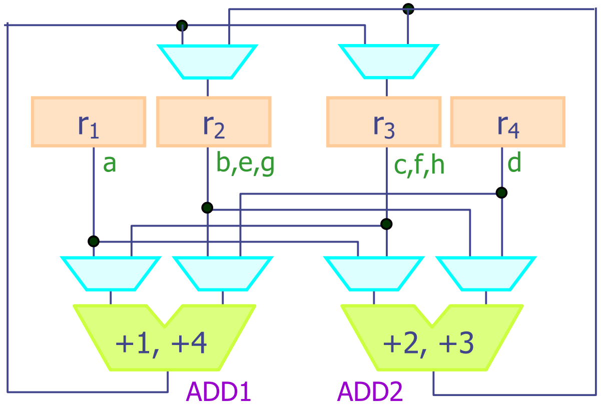


mem. configuration (array group : memory)	area (mm^2)
A:M_1, B:M_1, C:M_1, D:M_1	31.76
A:M_1, B:M_1, CD:M_3	27.11
A:M_1, C:M_1, BD:M_5	31.20
C:M_1, D:M_1, AB:M_3	27.11
...	...
AC:M_3, BD:M_3	**22.46**
AB:M_5, CD:M_3	26.55
...	...

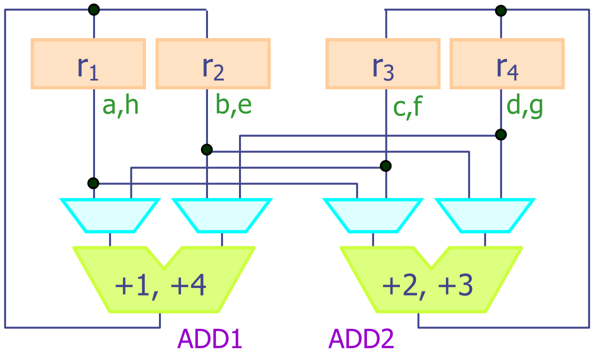
An Example of Allocation



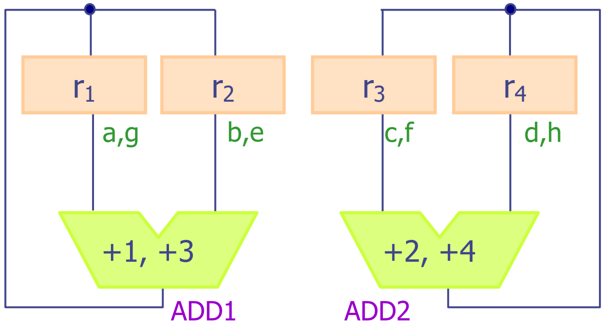
Scheduled DFG



FU Binding (6 Muxes)



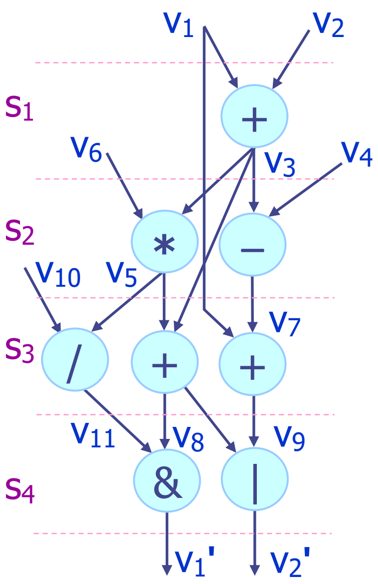
Register Reallocation (4 Muxes)



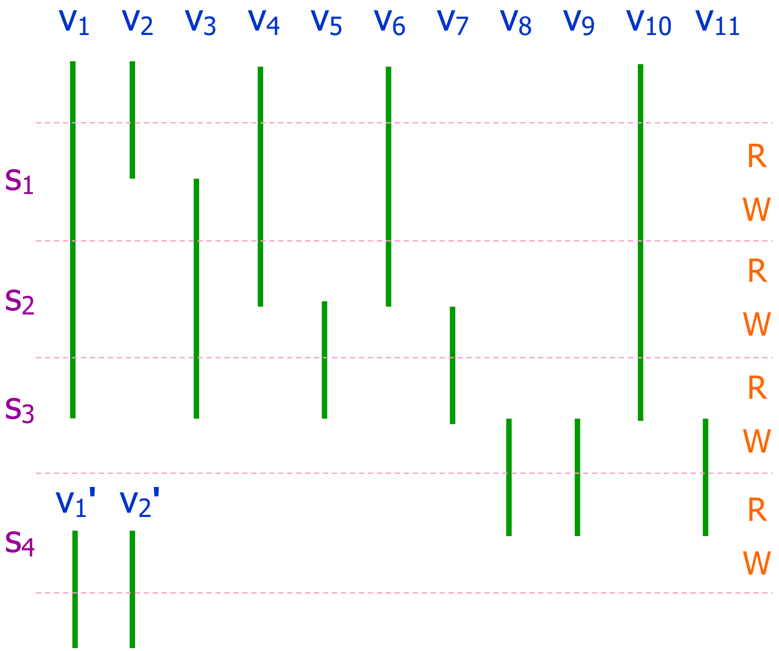
FU Rebinding

Left-Edge Algorithm (1/3)

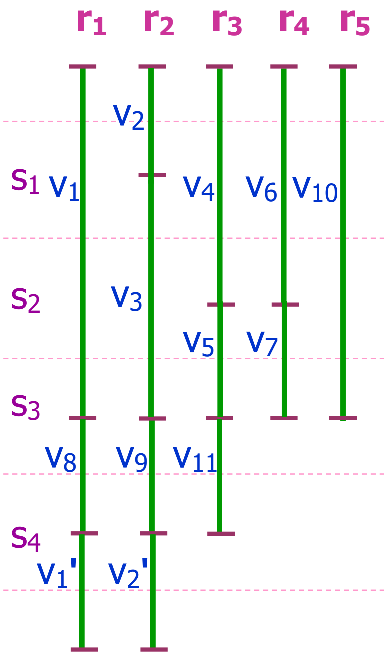
- Left-Edge Algorithm for Register Binding



DFG



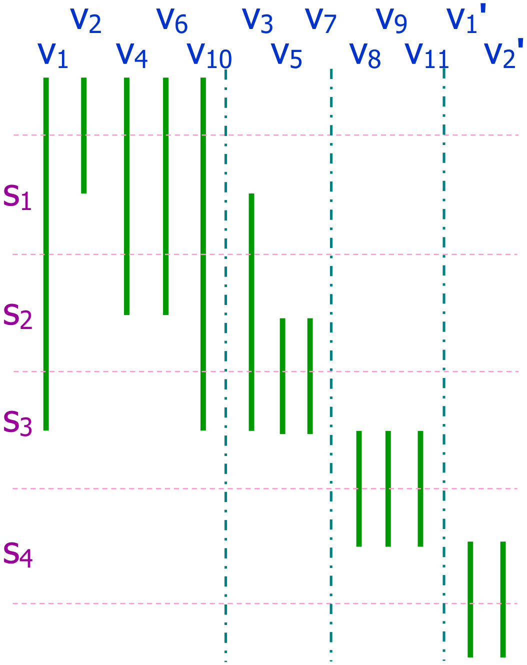
Lifetime intervals

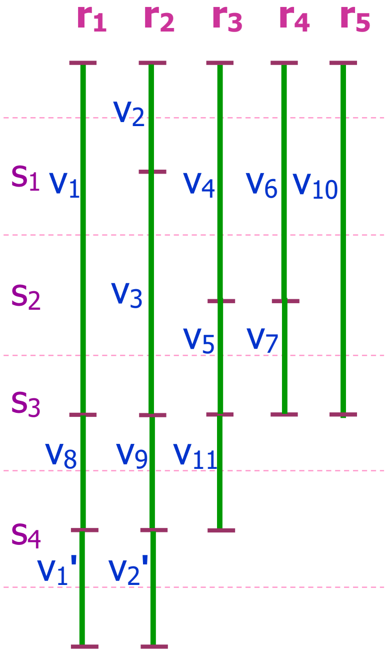


Registers

Left-Edge Algorithm (2/3)

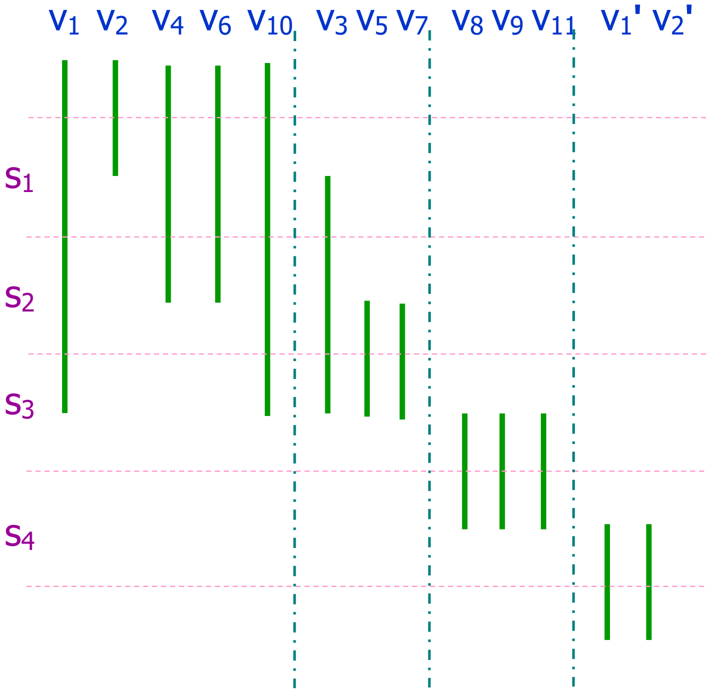
```
for all $v \in L$ do MAP[$v$]=0;  endfor
SORT($L$);  /* sort the variables in L in acending order with their start times*/
reg_index = 0;
while $L \neq \varnothing$ do
    reg_index = reg_index + 1;
    curr_var = FIRST($L$);
    last = 0;
    while curr_var ≠ null do
        if Start(curr_var) ≥ last then
            /* share the register */
            MAP[curr_var] = reg_index;
            last = End(curr_var);
            temp_var = curr_var;
            curr_var = NEXT(L, curr_var);
            DELETE(L, temp_var);
        else
            curr_var = NEXT(L, curr_var);
        endif
    endwhile
endwhile
```

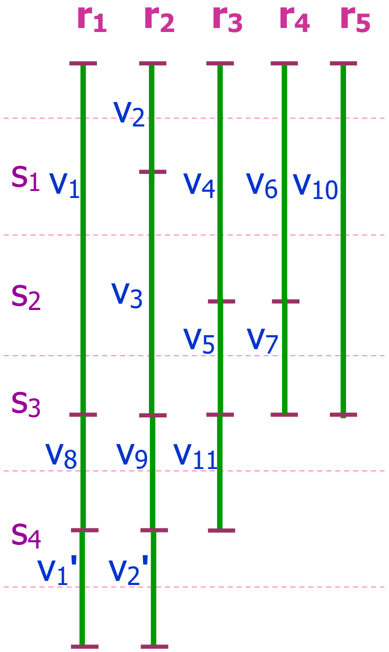




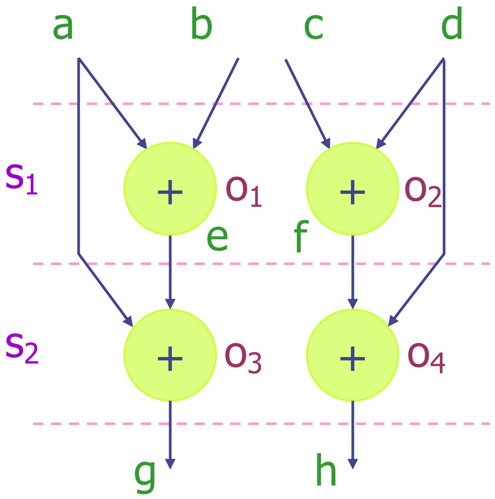
Left-Edge Algorithm (3/3)

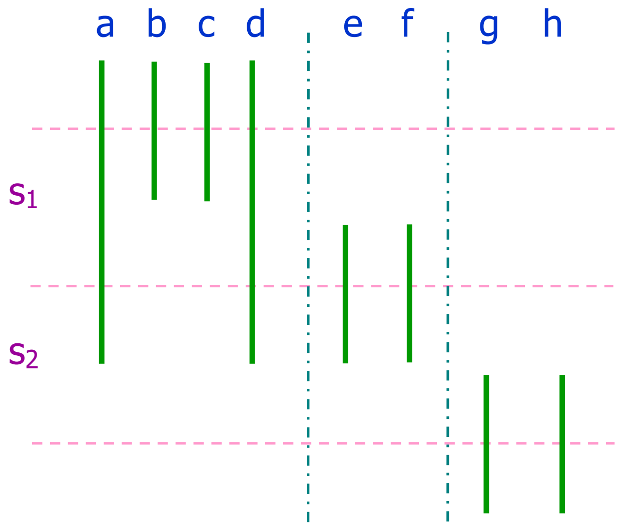
- Left-Edge Algorithm for Register Binding (Cont.)

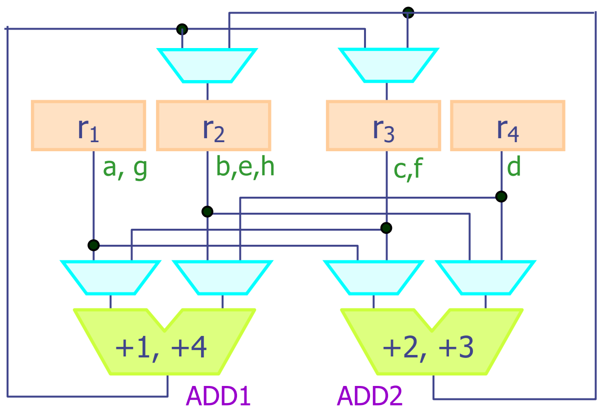

v_1 v_2 v_4 v_6 v_{10} v_3 v_5 v_7 v_8 v_9 v_{11} v_1' v_2'
s_1
s_2
s_3
s_4

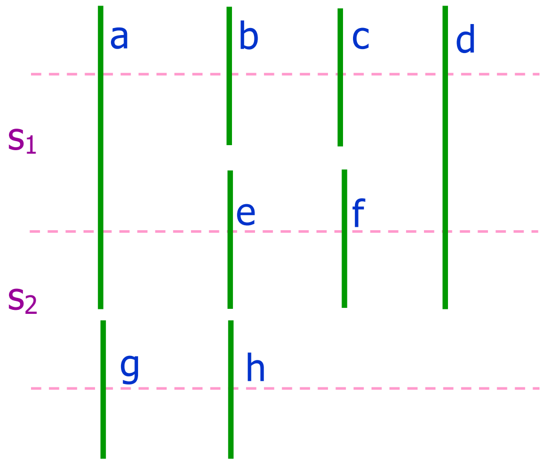


An Example


a
b
c
d
s1
+
o1
+
o2
e
f
s2
+
o3
+
o4
g
h

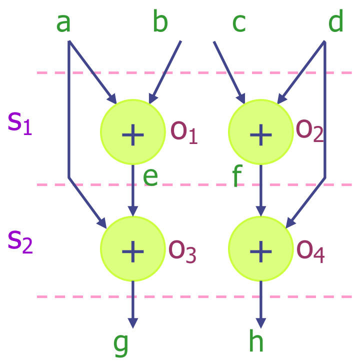

a
b
c
d
e
f
g
h
s1
s2

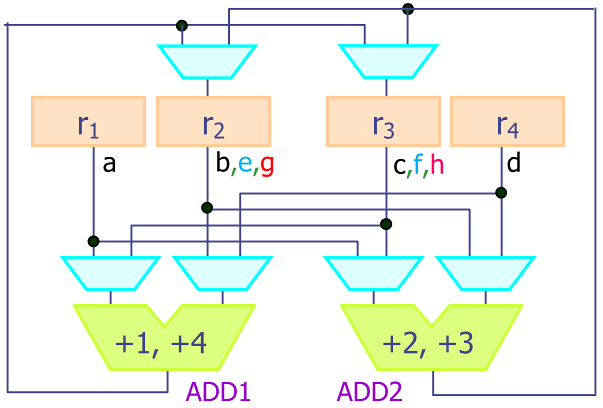

r1
r2
r3
r4
a, g
b,e,h
c,f
d
+1, +4
+2, +3
ADD1
ADD2

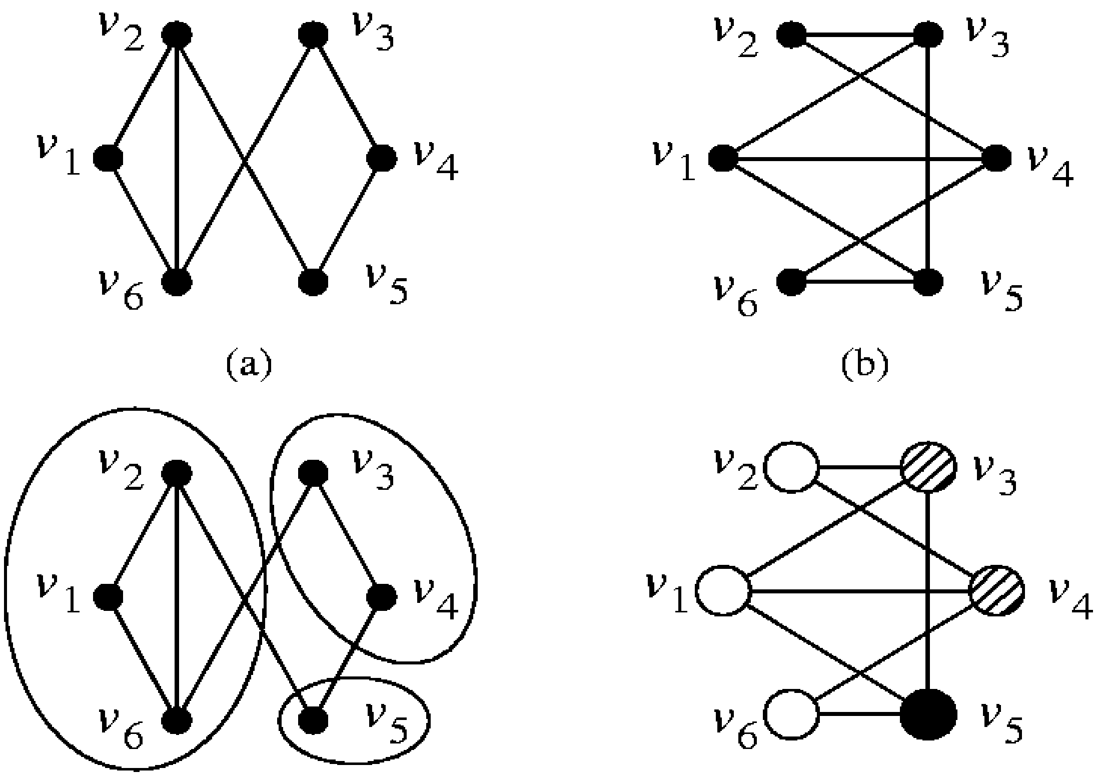

a
b
c
d
s1
e
f
s2
g
h

Clique Partitioning (1/3)

- Compatibility and Conflict Graphs
  - *Clique partitioning* gives an assignment in a compatibility graph
  - *Graph coloring* gives an assignment in the complementary conflict graph







(a)

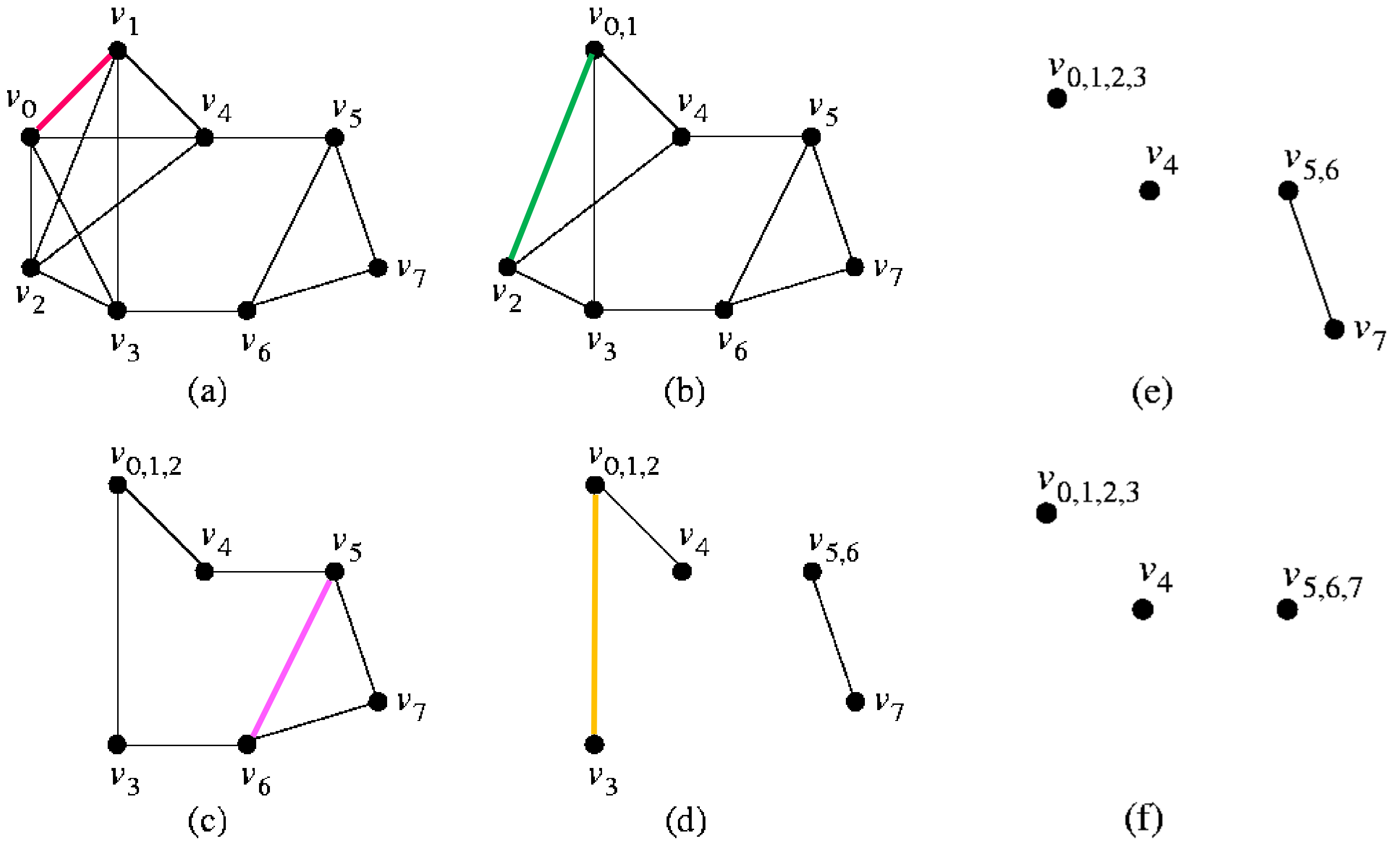
(b)

(c)

(d)

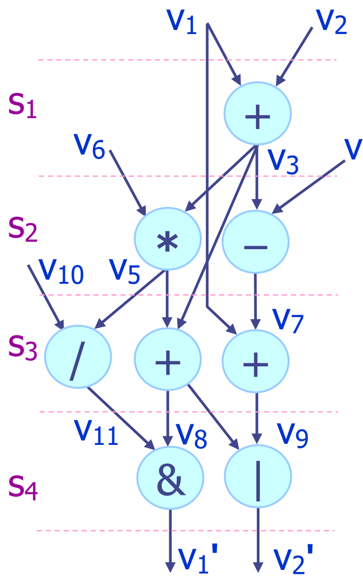
Clique Partitioning (2/3)

- Assignment by clique partitioning

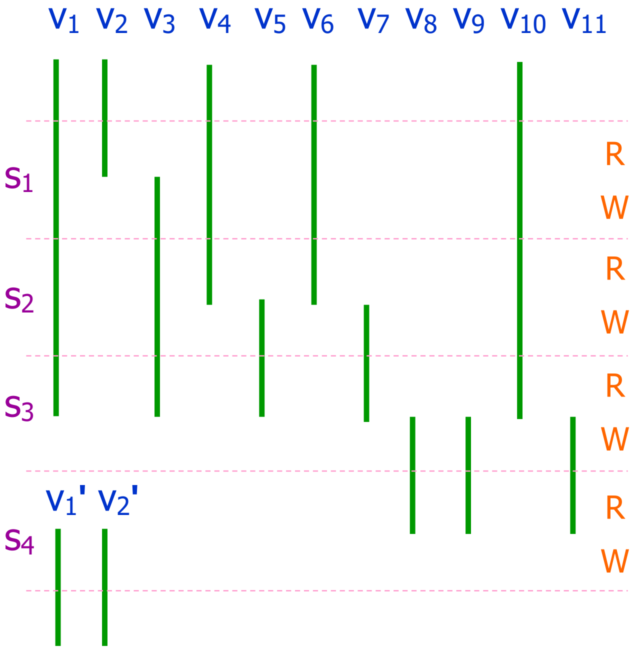


Clique Partitioning (3/3)

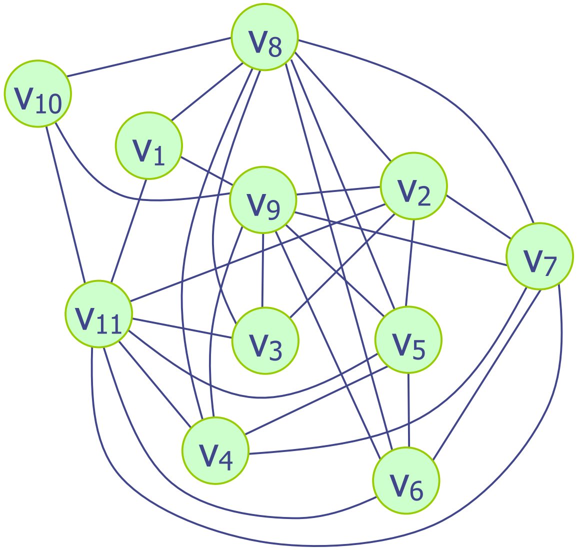
- Clique Partitioning for Register Binding



DFG



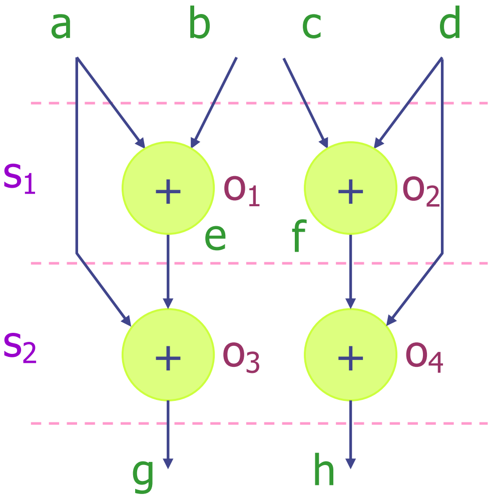
Lifetime intervals

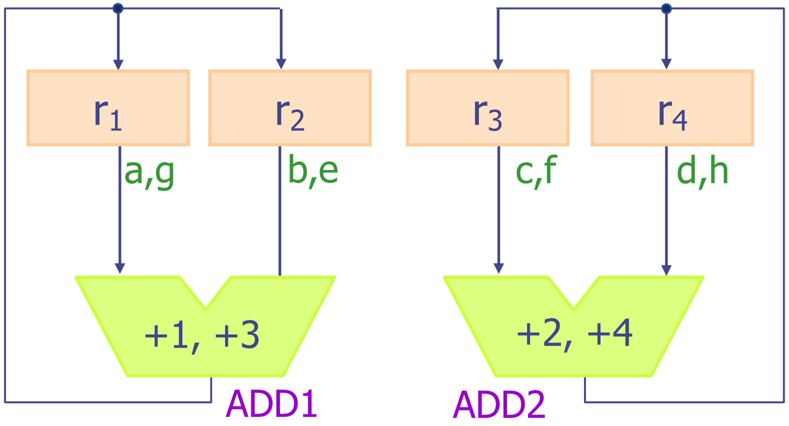


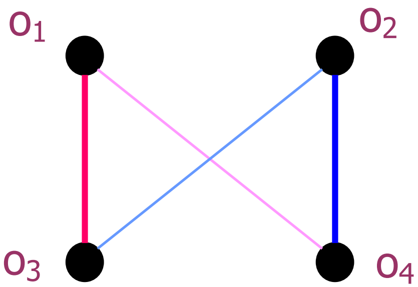
Graph model

Cliques: $r_1=\{v_1, v_8\}$, $r_2= \{v_2, v_3 , v_9\}$, $r_3= \{v_4, v_5 , v_{11}\}$,
$r_4=\{v_6, v_7\}$, $r_5= \{v_{10}\}$

An Example







(1, 3): 2+1+1 = 4

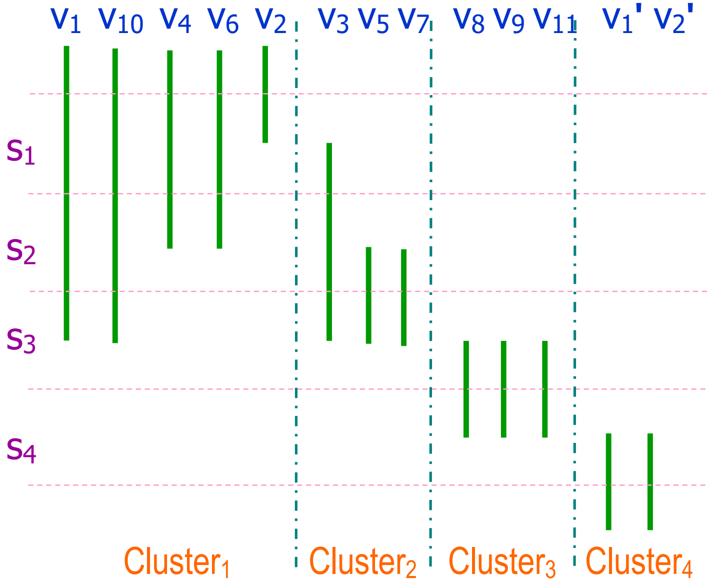
(1, 4): 0+1+1 = 2

(2, 4): 2+1+1 = 4

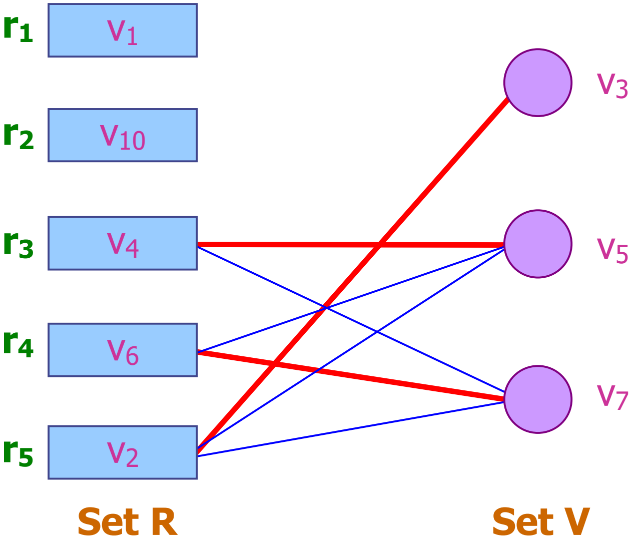
(2, 3): 0+1+1 = 2

Bipartite Matching (1/2)

- Bipartite Matching for Register Binding


v_1 v_{10} v_4 v_6 v_2 v_3 v_5 v_7 v_8 v_9 v_{11} v_1' v_2'
s_1
s_2
s_3
s_4
Cluster$_1$
Cluster$_2$
Cluster$_3$
Cluster$_4$

Sorted Lifetime Intervals with Clusters



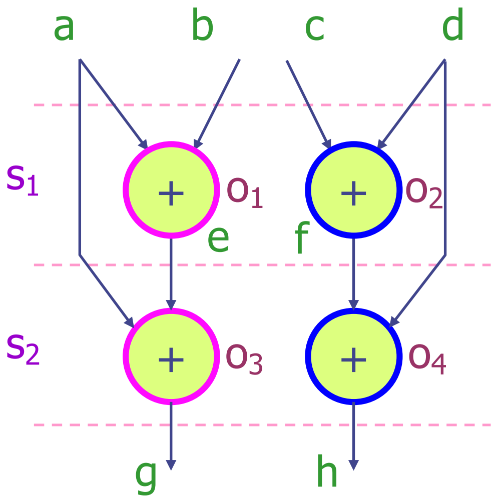
Bipartite Graph for Binding
Vars in Cluster2 after Cluster1

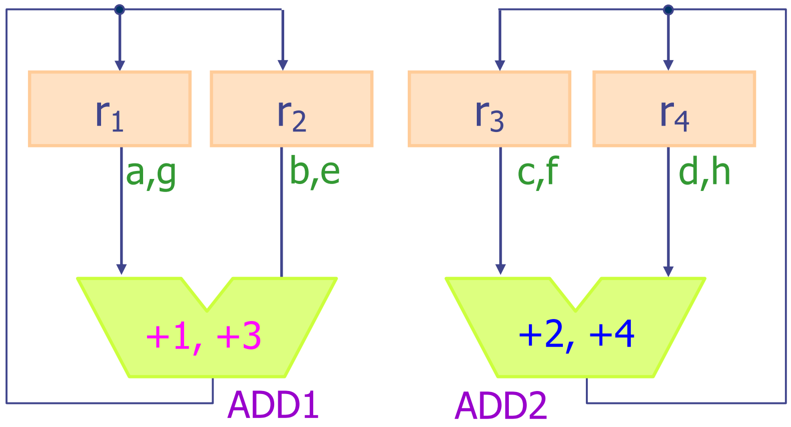
$r_1=\{v_1, v_8, v_1'\}$, $r_2=\{v_9, v_{10}\}$, $r_3=\{v_4, v_5, v_{11}\}$,
$r_4=\{v_6, v_7\}$, $r_5=\{v_2, v_3, v_2'\}$

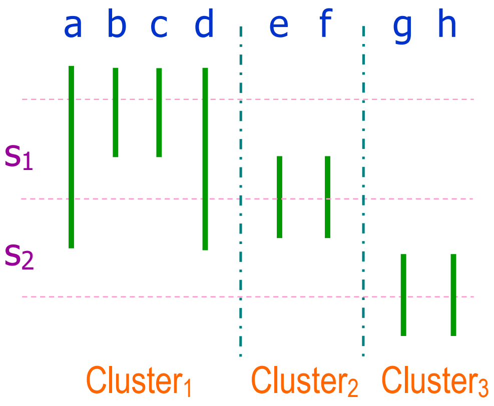
Bipartite Matching (2/2)

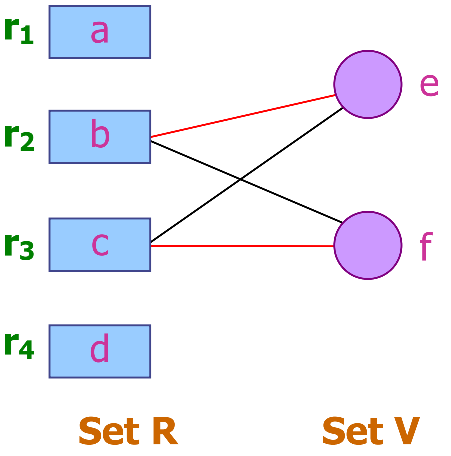
```
for all $v \in L$ do $MAP[v]=0$;  endfor
SORT($L$);
$clus\_num = 0$;
while $L \neq \emptyset$ do
    $clus\_num = clus\_num + 1$;
    $Cluster_{clus\_num} = \emptyset$;
    $last = 0$;
    while ($L \neq \emptyset$) and OVERLAP($Cluster_{clus\_num}$, FIRST($L$)) do
        $Cluster_{clus\_num} = Cluster_{clus\_num} \cup \{FIRST(L)\}$;
        $L$ = DELETE($L$, FIRST($L$));
    endwhile
endwhile

for $k$=1 to $clus\_num$ do
    $V = Cluster_k$;
    $E$ = BUILD_GRAPH($R$, $V$);
    $E'$ = MATCHING(G(R∪V, E));
    for each $e \in E'$, where $v_j \in V$ and $r_i \in R$ do
        $MAP[v_j] = r_i$;
    endfor
endfor
```

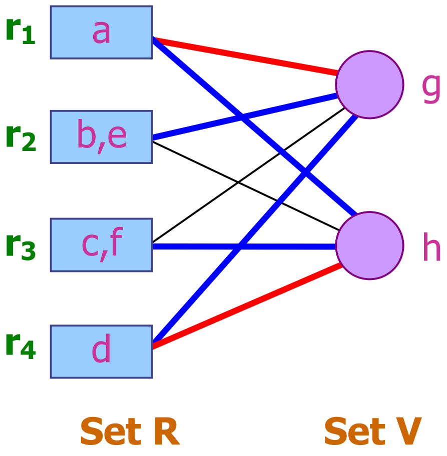
An Example (1/4)



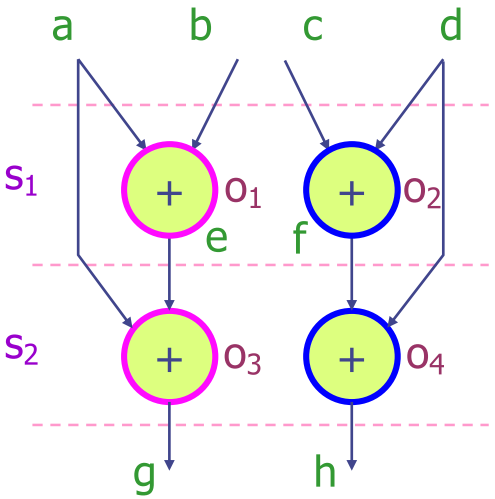


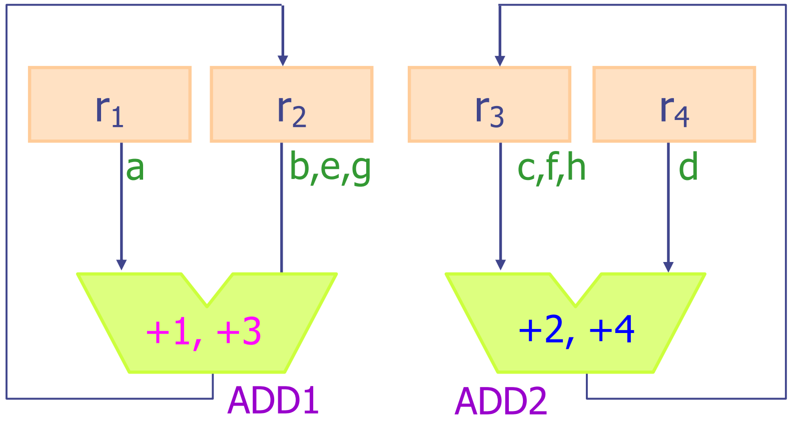


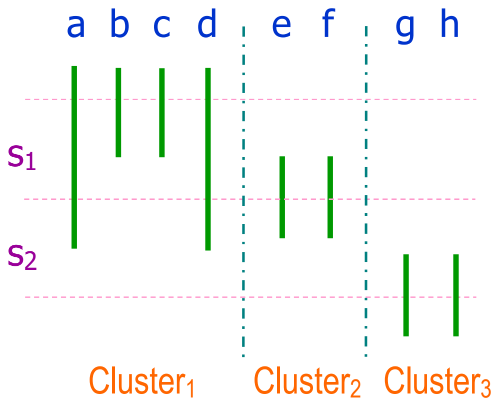


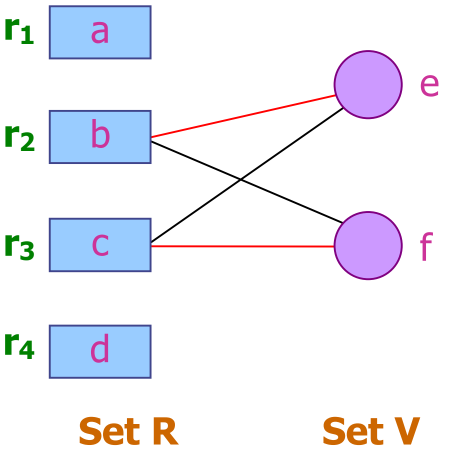


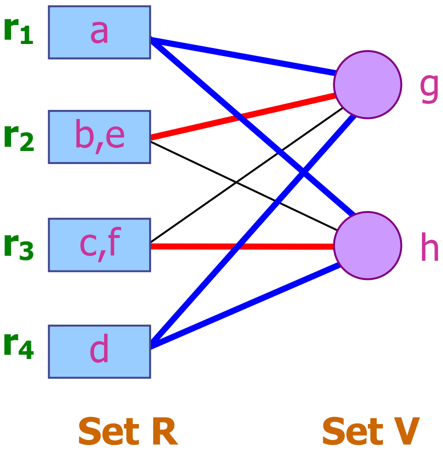
An Example (2/4)



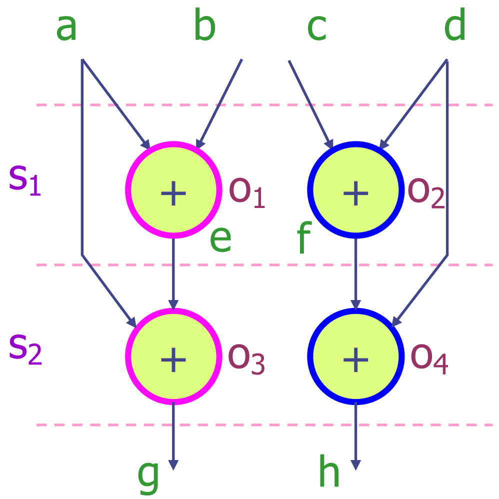


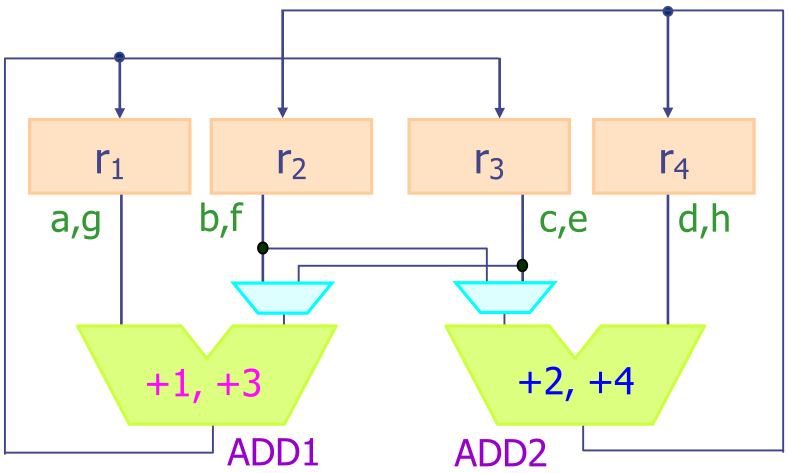


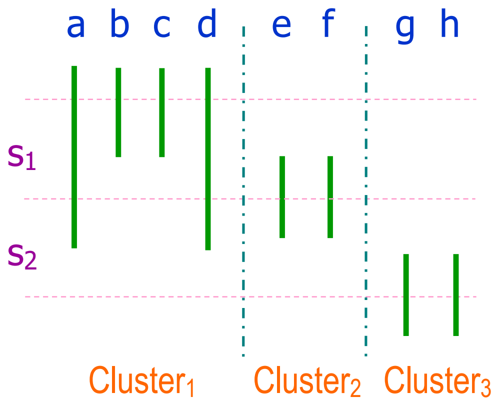


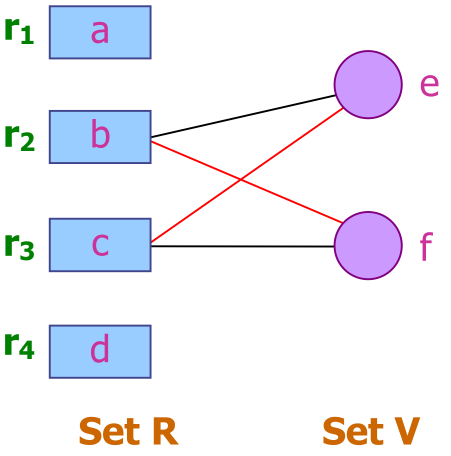


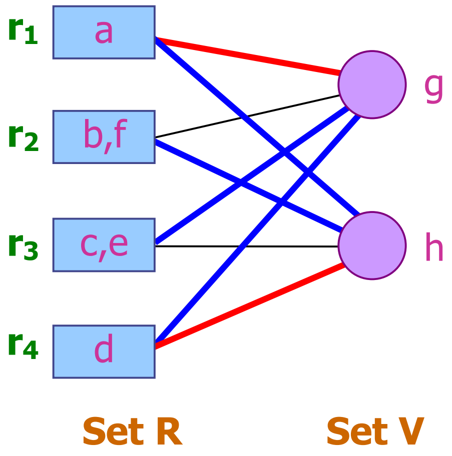
An Example (3/4)



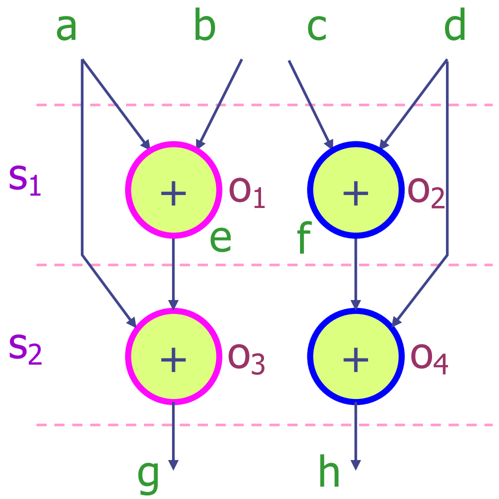


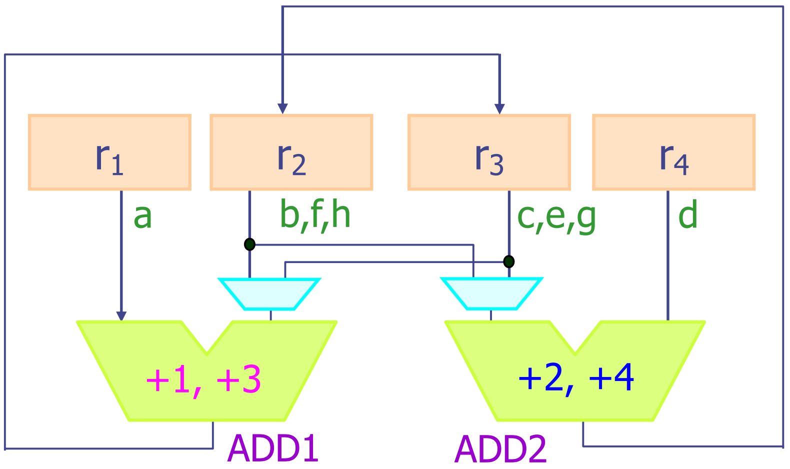


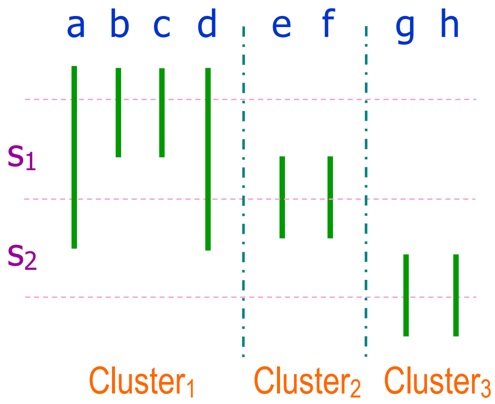


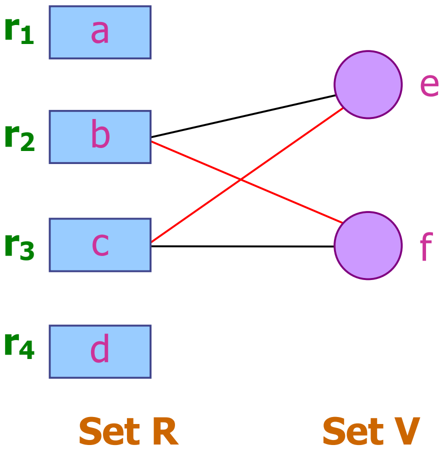


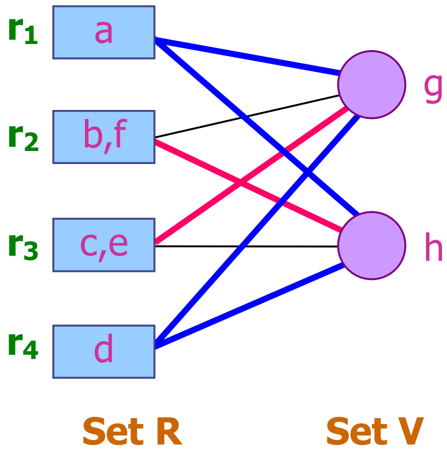
An Example (4/4)



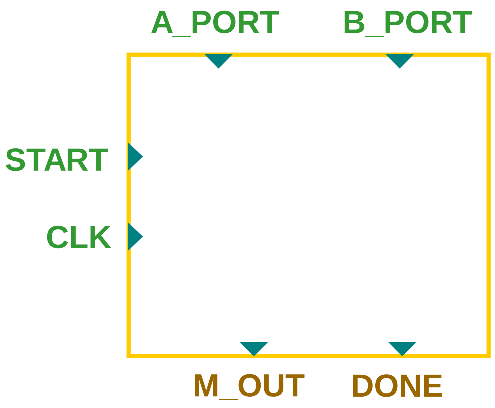








Shift-multiplier Example (Control-dominated) (1/2)

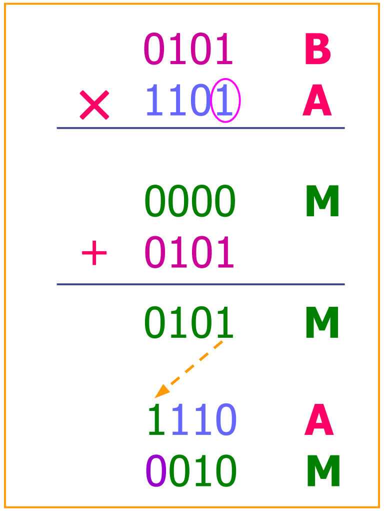


```
entity MULT is
port ( A_PORT,
       B_PORT: in bit_vector(3 downto 0);
       M_OUT: out bit_vector(7 downto 0);
       CLK: in CLOCK;
       START: in BIT;
       DONE: out BIT;
     );
End MULT;
```

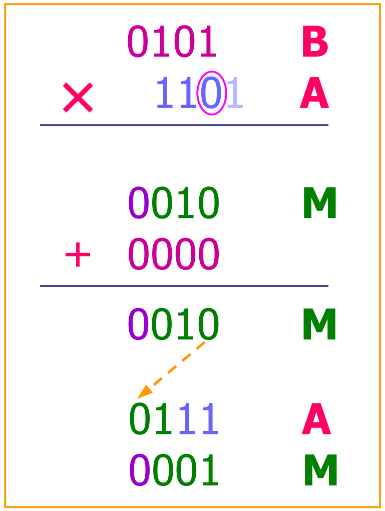
```
Architecture SHIFT_MULT of MULT is
begin
    process
        variable A, B, M:  BIT_VECTOR;
        variable COUNT: INTEGER;
    begin
        wait until (START = 1);
        A := A_PORT;     COUNT := 0;
        B := B_PORT;     DONE <= ‘0’;
        M := B”0000”;
        while (COUNT < 4) loop
            if (A(0) = ‘1’) then
                  M := M + B;
            end if;
            A := SHR(A, M(0));
            M := SHR(M, ’0’);
            COUNT := COUNT + 1;
        end loop;
        M_OUT <= M & A;
         DONE <= ‘1’;
     end processor;
```

Shift-multiplier Example (Control-dominated) (2/2)

```
A := A_PORT;   B := B_PORT;
M := B"0000";    COUNT := 0;
while (COUNT < 4) loop
        if (A(0) = '1') then
                M := M + B;
        end if;
        A := SHR(A, M(0));
        M := SHR(M, '0');
        COUNT := COUNT + 1;
end loop;
M_OUT <= M & A;
```

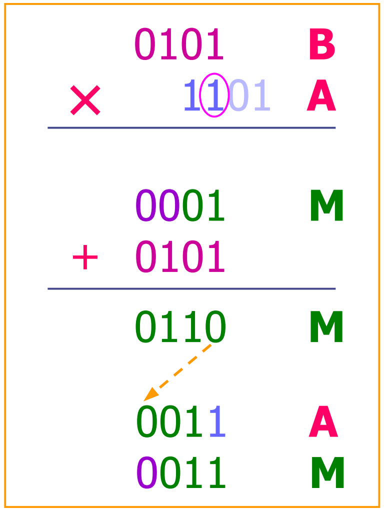


```
      0101   B
  ×   1101   A
  -------------
      0000   M
  +   0101
  -------------
      0101   M

      1110   A
      0010   M
```

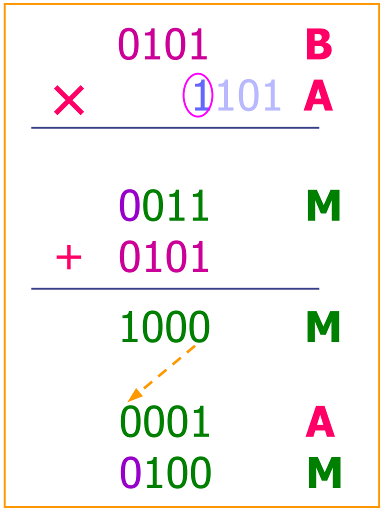


```
      0101   B
  ×   1101   A
  -------------
      0010   M
  +   0000
  -------------
      0010   M

      0111   A
      0001   M
```

```
          0101    B
      ×   1101    A
   -----------------
          0101
         0000
        0101
   +   0101
   -----------------
       01000001
```



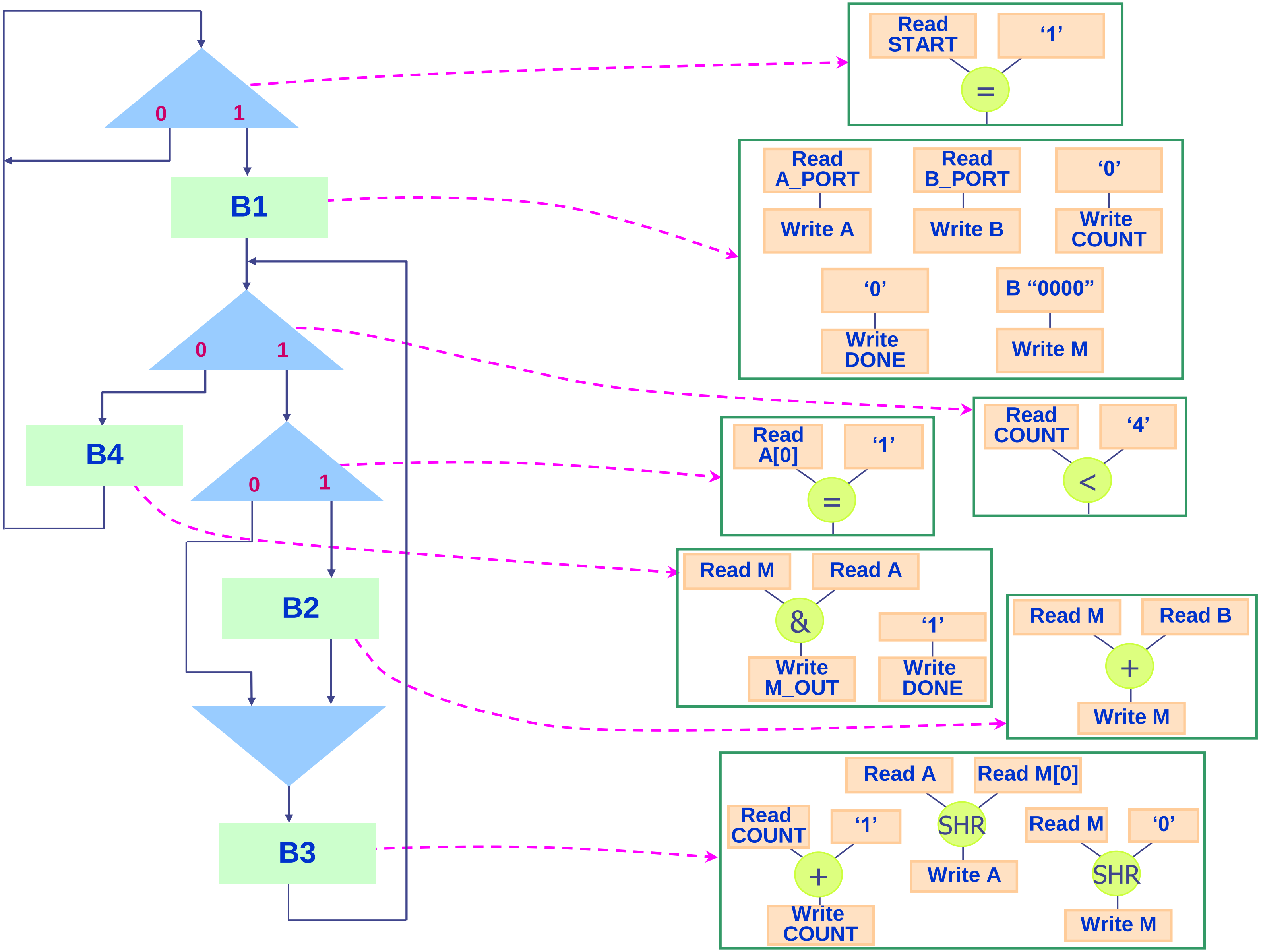
```
      0101   B
  ×   1101   A
  -------------
      0001   M
  +   0101
  -------------
      0110   M

      0011   A
      0011   M
```



```
      0101   B
  ×   1101   A
  -------------
      0011   M
  +   0101
  -------------
      1000   M

      0001   A
      0100   M
```

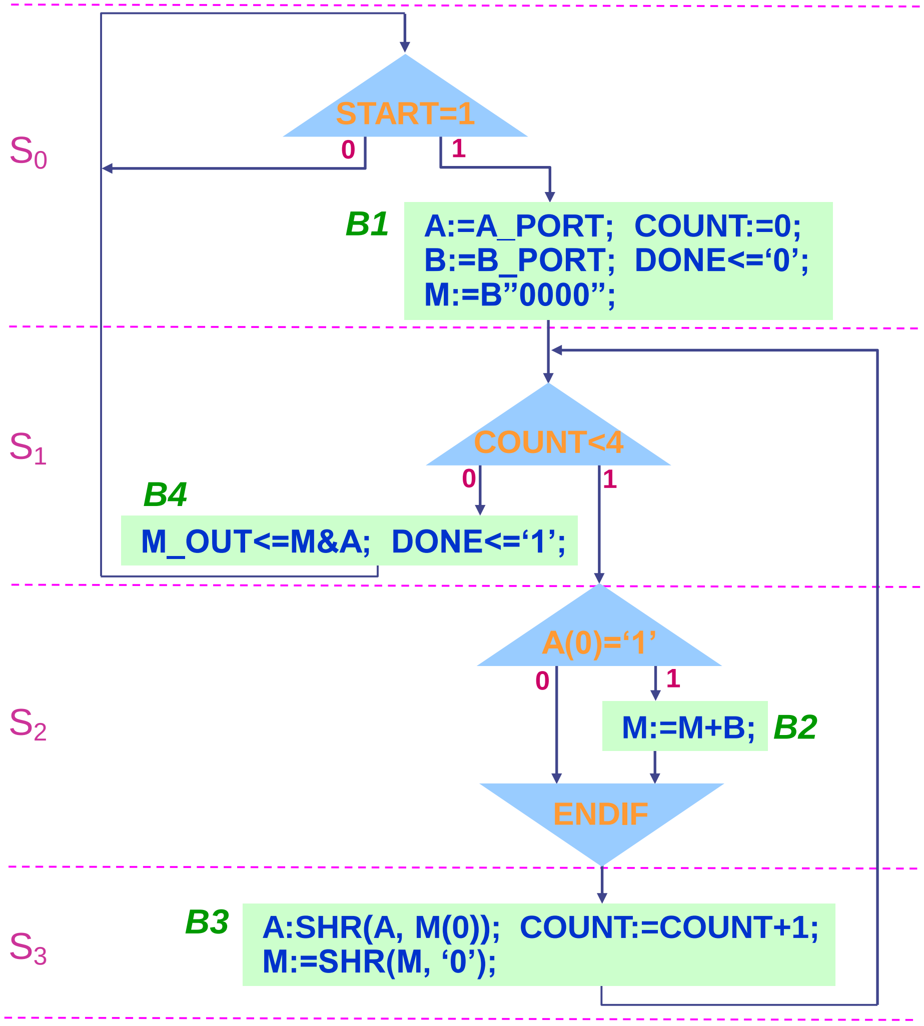
Control-flow graph

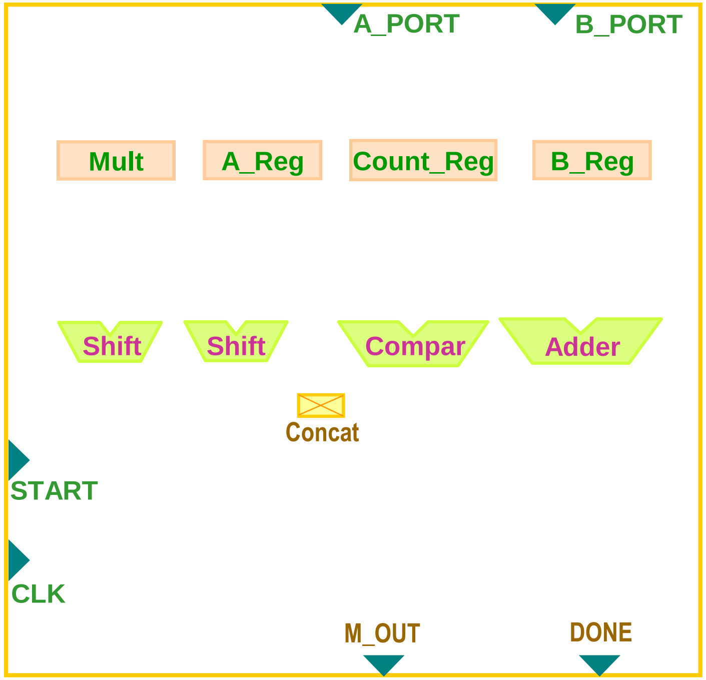
Data-flow graphs



Scheduling

- Shift-multiplier example: Scheduled CDFG





Initial Allocation

State Table

- Shift-multiplier example: State Table

Present State	Condition	Value	Actions	Next State

S0	START=1	T	A:=A_PORT;
			B:=B_PORT;
			COUNT:=0;
			DONE:=‘0’;
			M:=“0000”;
				S1

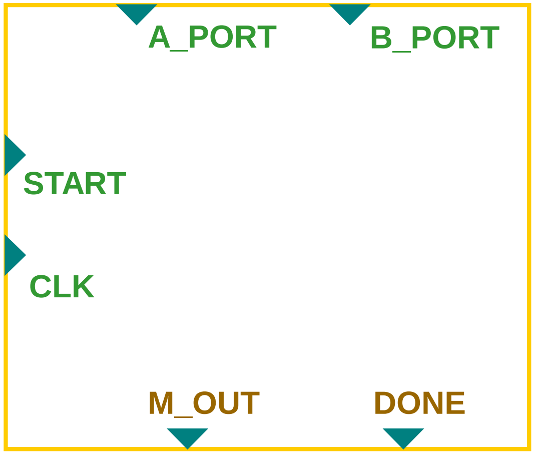
		F		S0
S1	COUNT<4	T		S2

		F	M_OUT:=M@A;
			DONE:=‘1’;
				S0

S2	A[0]=1	T	M:=M+B;	S3
		F		S3

S3			A:=SHR(A,M[0]);
			M:=SHR(M, ‘0’);
			COUNT:=COUNT+1;
				S1

FSMD state table



I/O ports

Allocation

- Shift-multiplier example: After Allocation

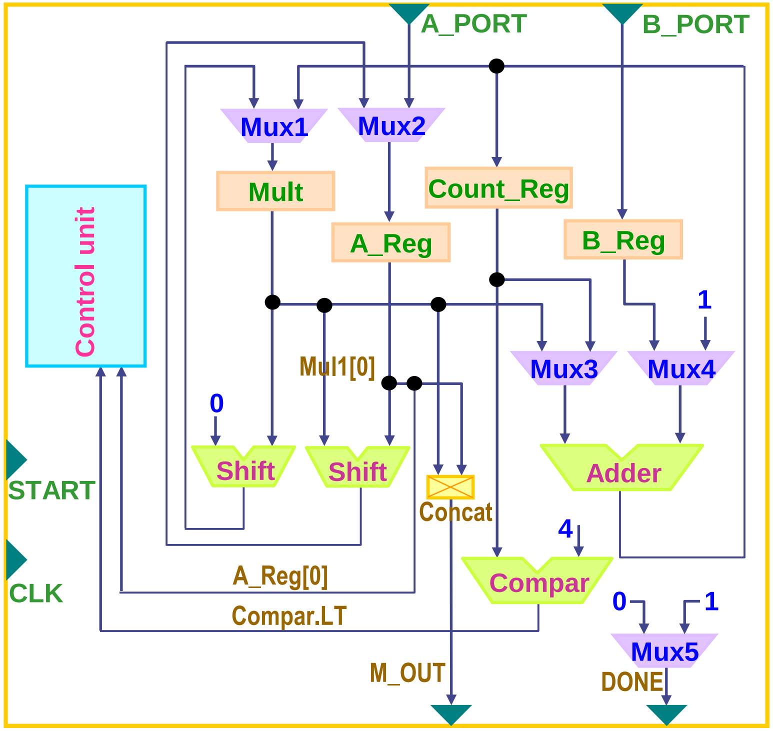
Present State	Condition	Value	Actions	Next State
S1	Compar.LT	1		S2

		0	Concat(OP:concat, INPS:Mult, A_Reg);
			M_OUT(OP: load, INPS:Concat);
			Mux5(OP:c1, INPS:'0', '1');
			DONE(OP:load, INPS:Mux5);
				S0

S2	A_Reg[0]	1	Mux3(OP:c0, INPS:Mult, Count_Reg);
			Mux4(OP:c0, INPS: B_Reg, "0001");
			Adder(OP:add, INPS:Mux3, Mux4);
			Mux1(OP:c1, INPS:Shift1, Adder);
			Mult(OP: load, INPS:Mux1);
				S3

		0		S3

Component-based state table



Partial design

Control Generation (1/2)

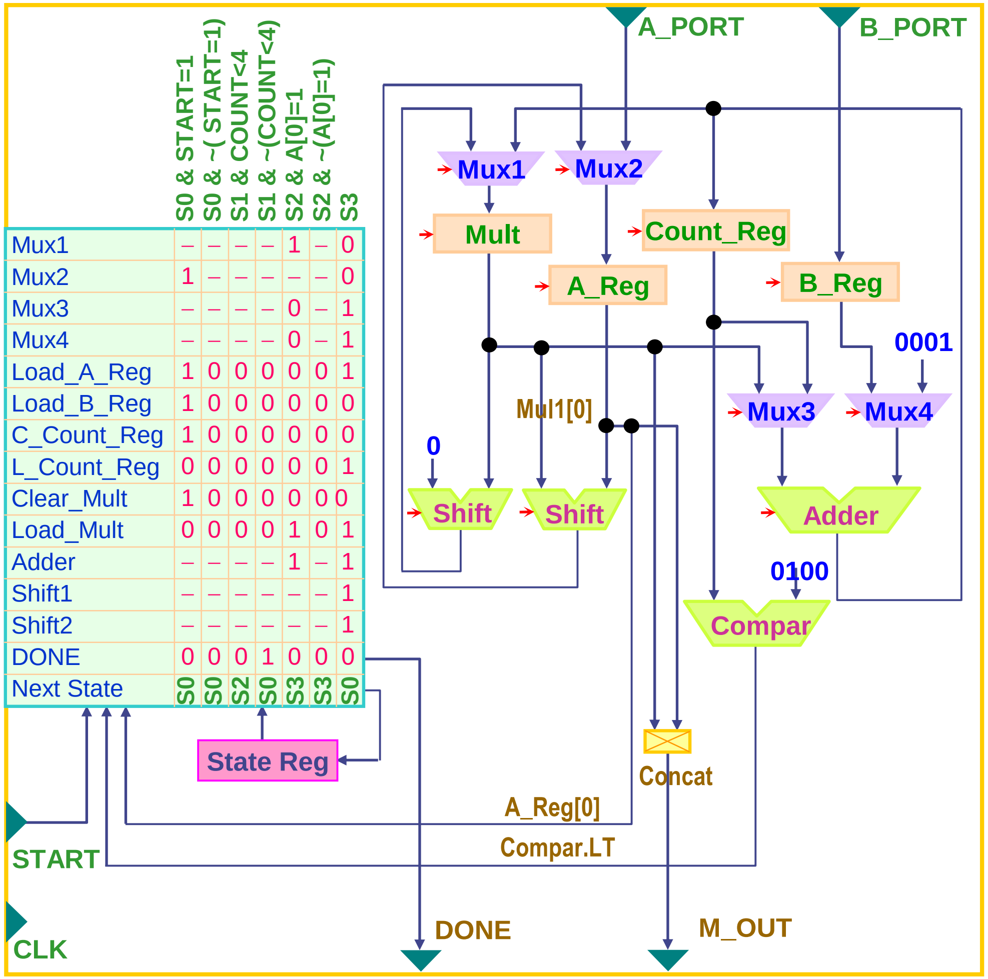
- Shift-multiplier example: After Control Generation

Present State	Condition	Value	Actions	Next State
S1	Compar.LT	1		S2
		0	DONE:=1;	S0

S2	A_Reg[0]	1	Mux3.sel:=0;
			Mux4.sel:=0;
			Adder.add:=1;
			Mux1.sel:=1;
			Mult.load:=1;
				S3

		0		S3

Symbolic Control Table

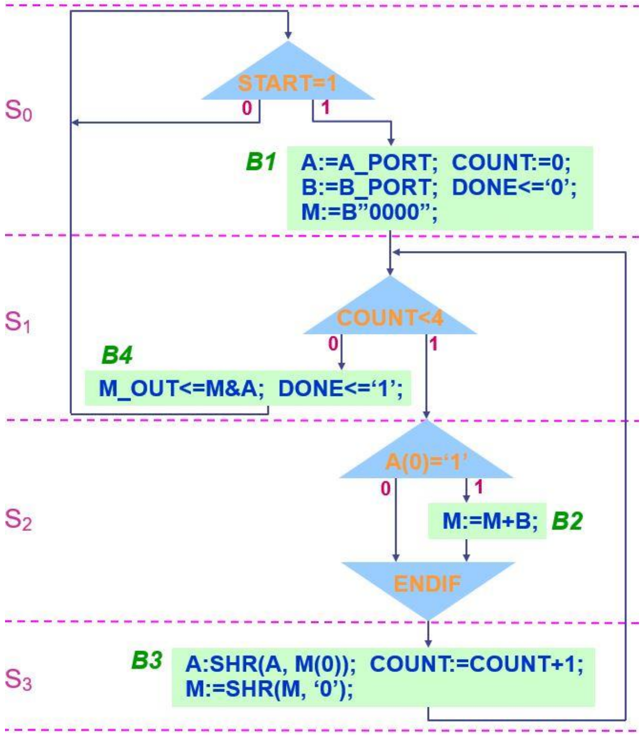


	S0 & START=1	S0 & ~(START=1)	S1 & COUNT<4	S1 & ~(COUNT<4)	S2 & A[0]=1	S2 & ~(A[0]=1)	S3
Mux1	–	–	–	–	1	–	0
Mux2	1	–	–	–	–	–	0
Mux3	–	–	–	–	0	–	1
Mux4	–	–	–	–	0	–	1
Load_A_Reg	1	0	0	0	0	0	1
Load_B_Reg	1	0	0	0	0	0	0
C_Count_Reg	1	0	0	0	0	0	0
L_Count_Reg	0	0	0	0	0	0	1
Clear_Mult	1	0	0	0	0	0	0
Load_Mult	0	0	0	0	1	0	1
Adder	–	–	–	–	1	–	1
Shift1	–	–	–	–	–	–	1
Shift2	–	–	–	–	–	–	1
DONE	0	0	0	1	0	0	0
Next State	S0	S0	S2	S0	S3	S3	S0

Complete Design

Control Generation (2/2)

Present State	Input	Next State	Control Signals										
			M1	M2	M3	M4	L_A	L_B	C_C	L_C	Clear_M	L_M	done
S0	Start = 0	S0	-	-	-	-	0	0	0	0	0	0	0
	Start = 1	S1	-	1	-	-	1	1	1	0	1	0	0
S1	Count ≥ 4	S0	-	-	-	-	0	0	0	0	0	0	1
	Count < 4	S2	-	-	-	-	0	0	0	0	0	0	0
S2	A[0] = 0	S3	1	-	-	-	0	0	0	0	0	0	0
	A[0] = 1	S3	-	-	0	0	0	0	0	0	0	1	0
S3		S1	0	0	1	1	1	0	0	1	0	1	0



S_0

START=1

0 1

B1 A:=A_PORT; COUNT:=0;
B:=B_PORT; DONE<='0';
M:=B"0000";

S_1

COUNT<4

0 1

B4 M_OUT<=M&A; DONE<='1';

S_2

A(0)='1'

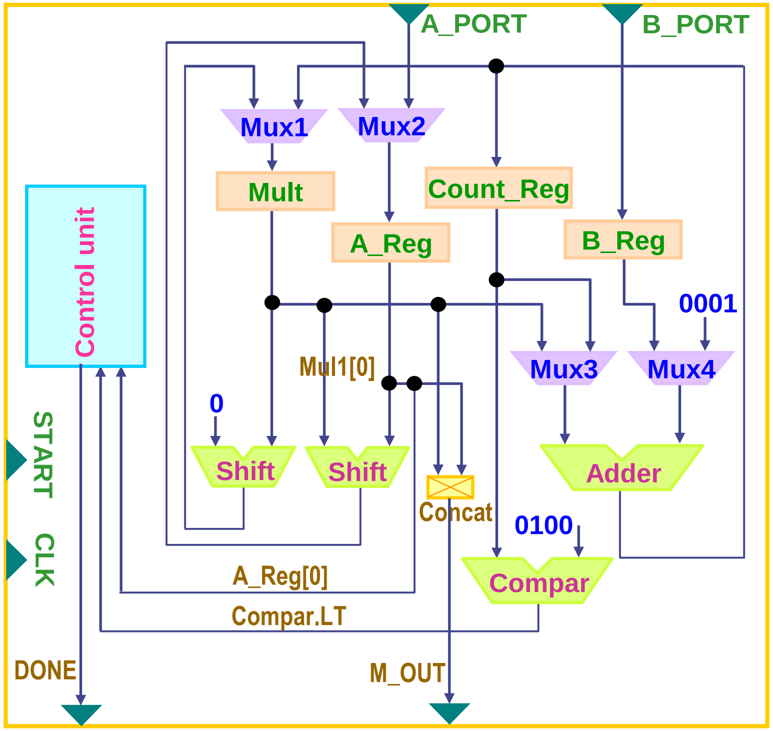
0 1

M:=M+B; B2

ENDIF

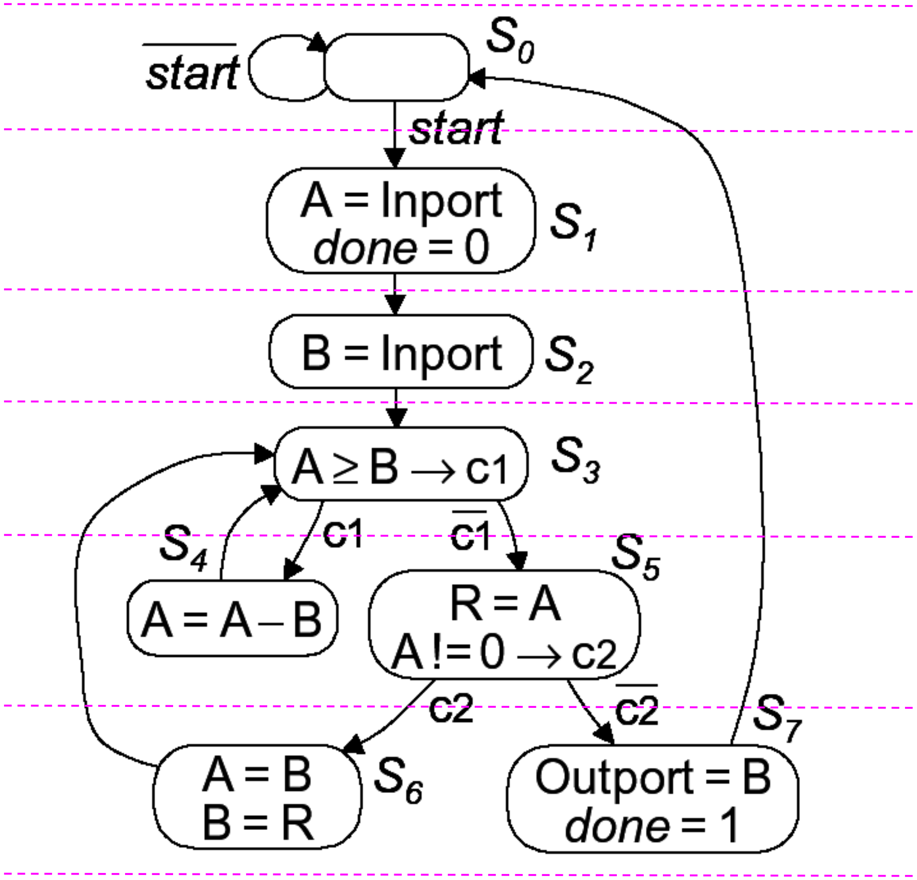
S_3

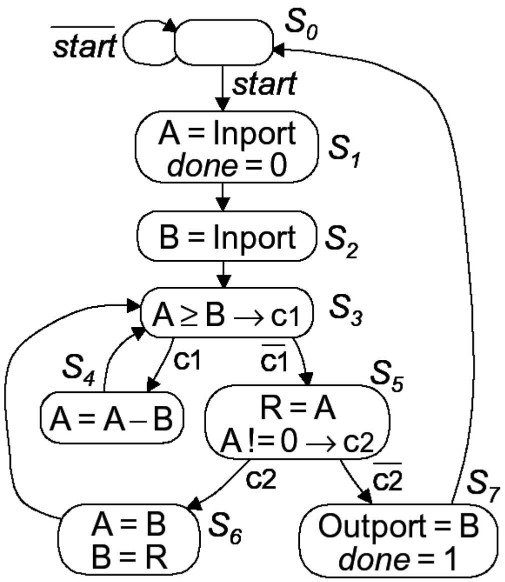
B3 A:SHR(A, M(0)); COUNT:=COUNT+1;
M:=SHR(M, '0');



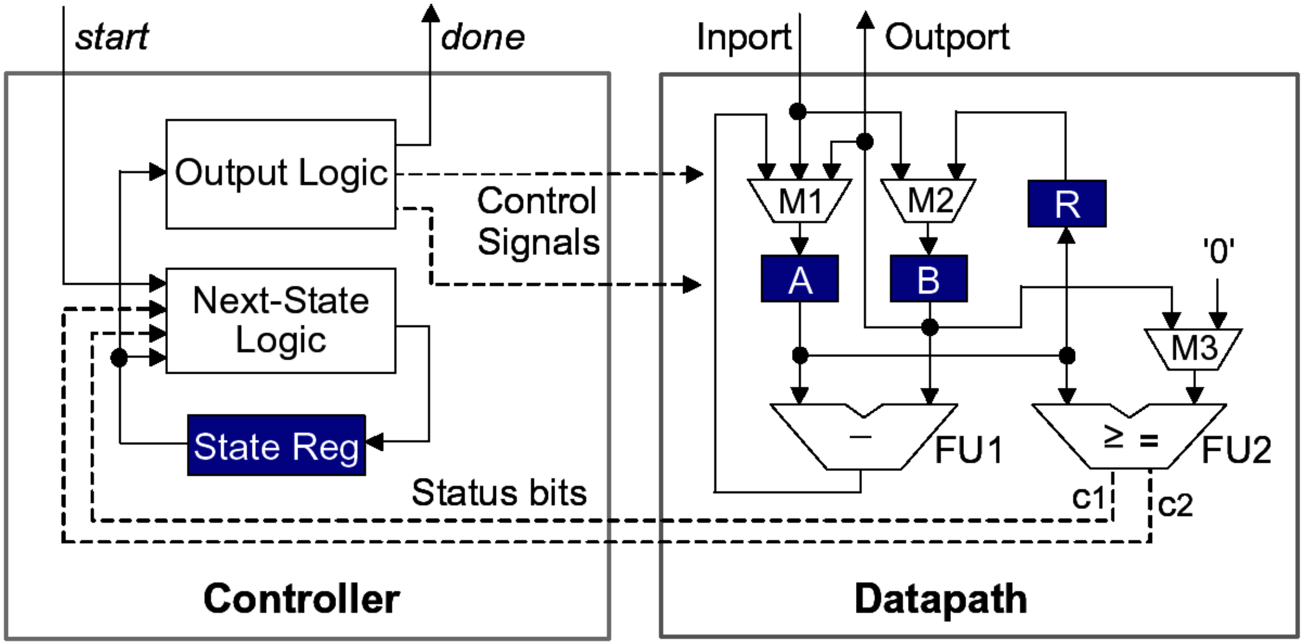
GCD Example (Control-dominated)

```
         A = Inport;
         B = Inport;
         done = 0;
Repeat:  while (A ≥ B)
             A = A – B;
         R = A;
         if (A != 0) {
             A = B;
             B = R;
             goto Repeat;
         }
         else  goto End;
End:     Outport = B;
         done = 1;
```



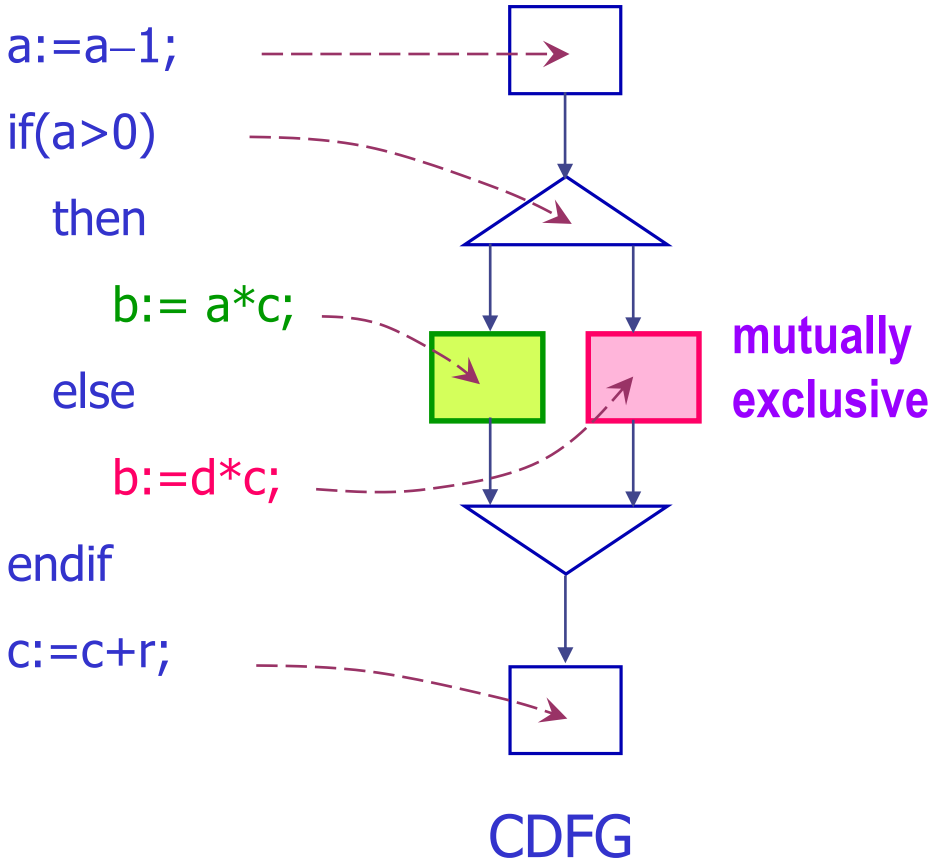


PS	Input	NS	Control signals						
			A	B	R	M1	M2	M3	*done*
S_0	*start* = 0	S_0	-	0	-	-	-	-	0
	start = 1	S_1							
S_1	-	S_2	1	-	-	Inport	-	-	0
S_2	-	S_3	0	1	-	-	Inport	-	0
S_3	c1 = 1	S_4	0	0	0	-	-	B	0
	c1 = 0	S_5							
S_4	-	S_3	1	0	0	FU1	-	-	0
S_5	c2 = 1	S_6	0	0	1	-	-	'0'	0
	c2 = 0	S_7							
S_6	-	S_3	1	1	0	B	R	-	0
S_7	-	S_0	0	0	0	-	-	-	1



Other Issues (1/6)

- Conditional Constructs
- Clocking of functional units
  - chaining
  - multicycling
- Pipelining
  - units pipelining
  - datapath pipelining
  - control pipelining



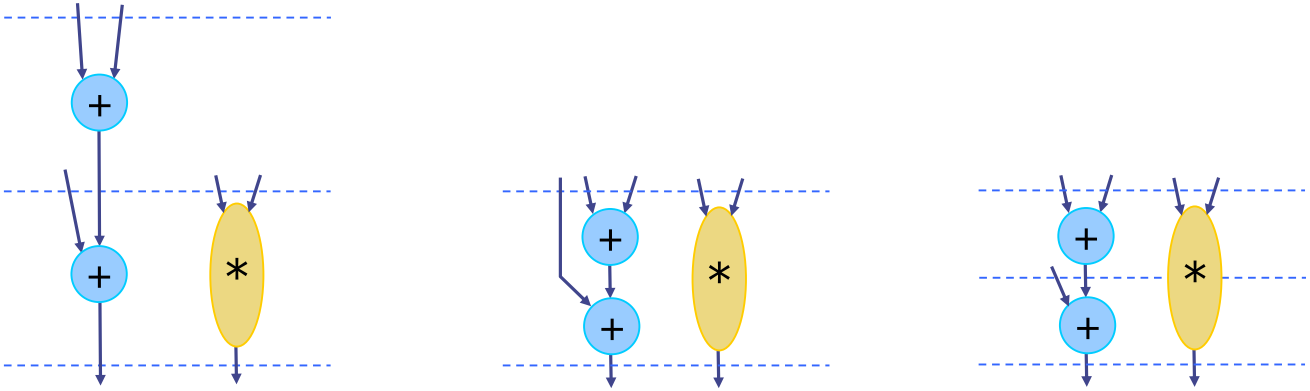
```
a:=a–1;
if(a>0)
   then
      b:= a*c;
   else
      b:=d*c;
endif
c:=c+r;
```

mutually exclusive

CDFG

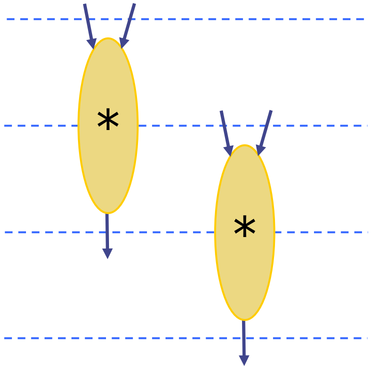
Other Issues (2/6)

- Chaining or Multicycling
  - are used on noncritical paths to improve resource utilization and performance
  - Chaining
    - allows serial execution of two or more operations in each state
    - reduces number of states and increases performance
  - Multicycling
    - allows one operation to be executed over two or more clock cycles
    - reduces size of functional units

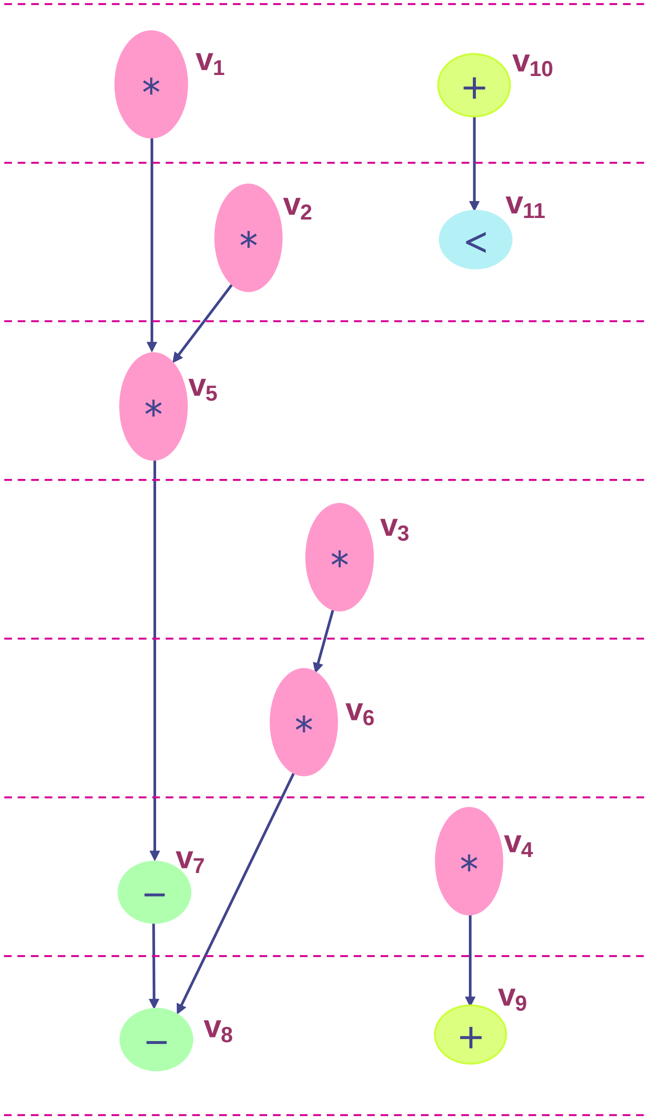


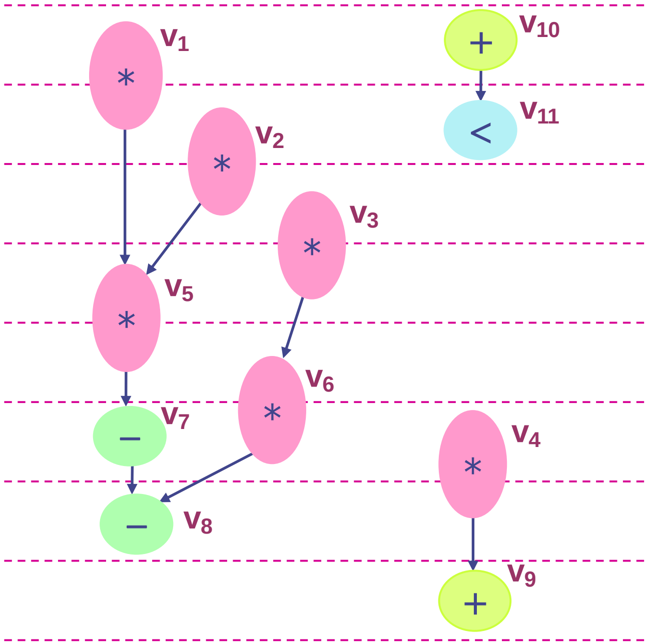
Other Issues (3/6)

- Pipelining
  - improves performance at a very small additional cost
  - divides resources into stages and uses all stage concurrently for different data (assembly line principle)
  - works on several levels:
    - units pipelining
    - datapath pipelining
    - control pipelining



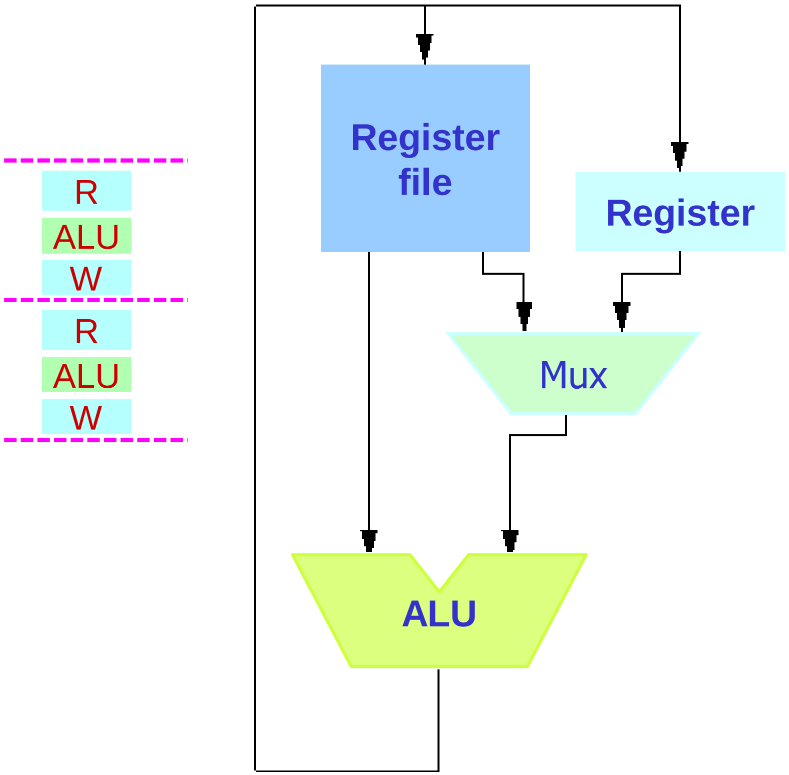
An Example



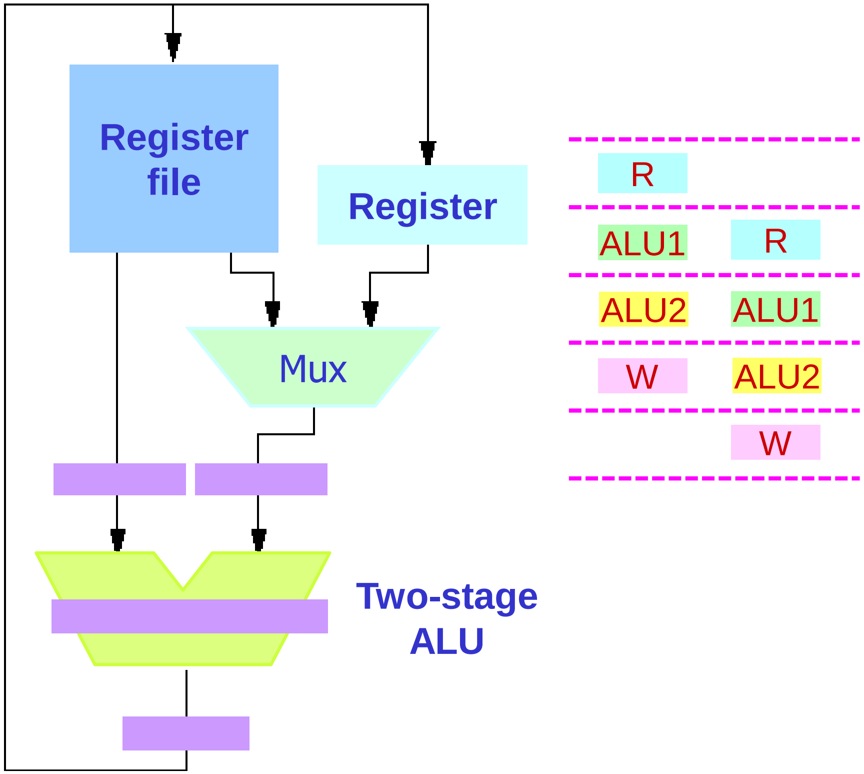


Other Issues (4/6)

- Datapath Pipelining

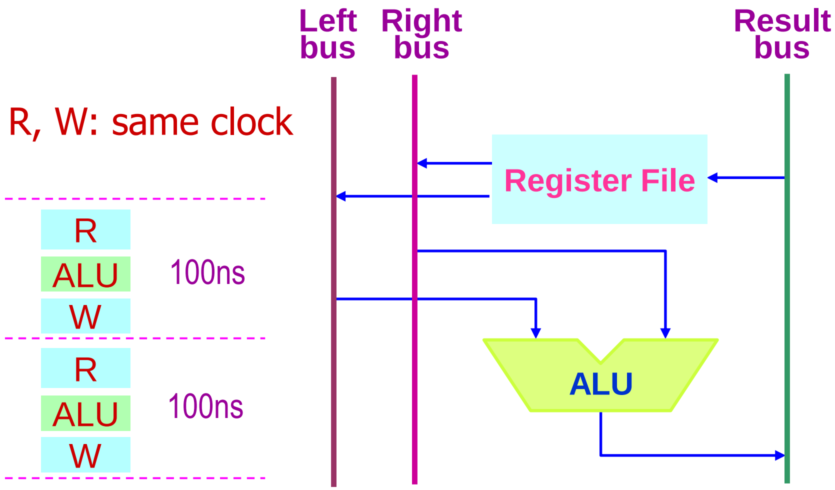


Non-pipelined datapath

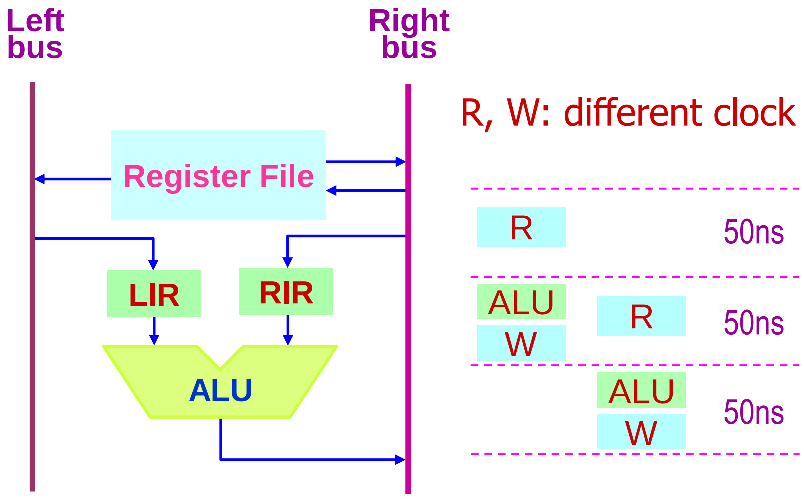


Pipelined datapath With 2-stage adder

Other Issues (5/6)



3-bus nonpipelined design



2-bus pipelined design

Program 1: x <= a + b; (100ns)
y <= c – x; (100ns)

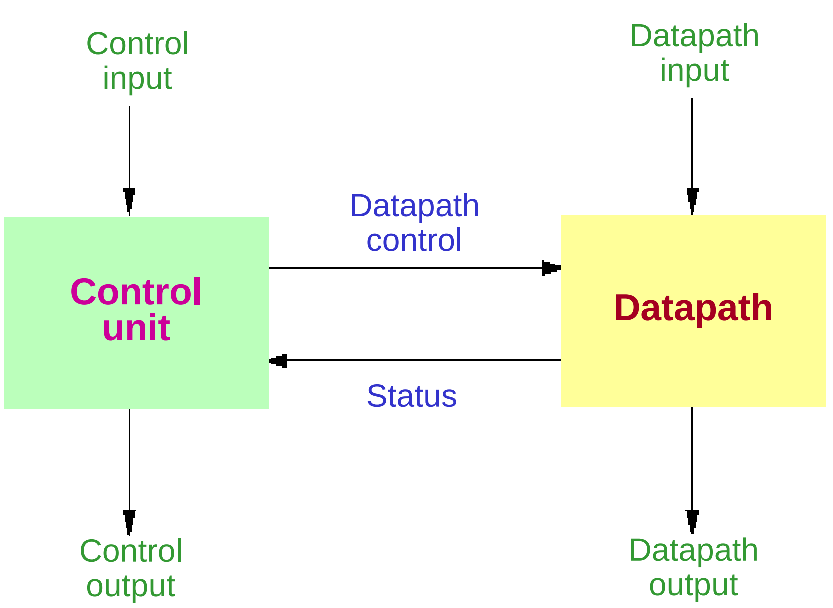
LIR <= a; RIR <= b; (50ns)
x, RIR <= LIR + RIR; LIR <= c; (50ns)
y <= LIR – RIR; (50ns)

Program 2: x <= a + b; (100ns)
y <= c – d; (100ns)

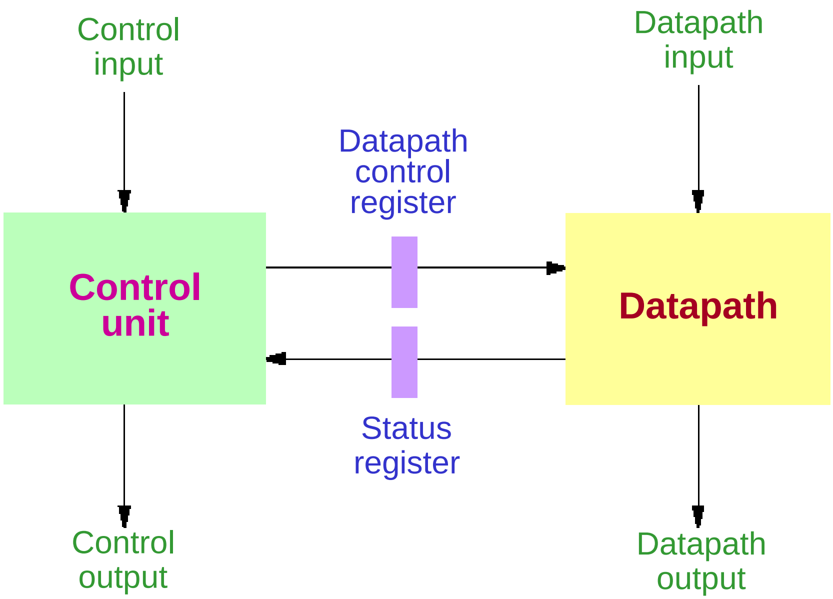
LIR <= a; RIR <= b; (50ns)
x <= LIR + RIR; (50ns)
LIR <= c;   RIR <= d; (50ns)
y <= LIR – RIR; (50ns)

Other Issues (6/6)

- Control Pipelining



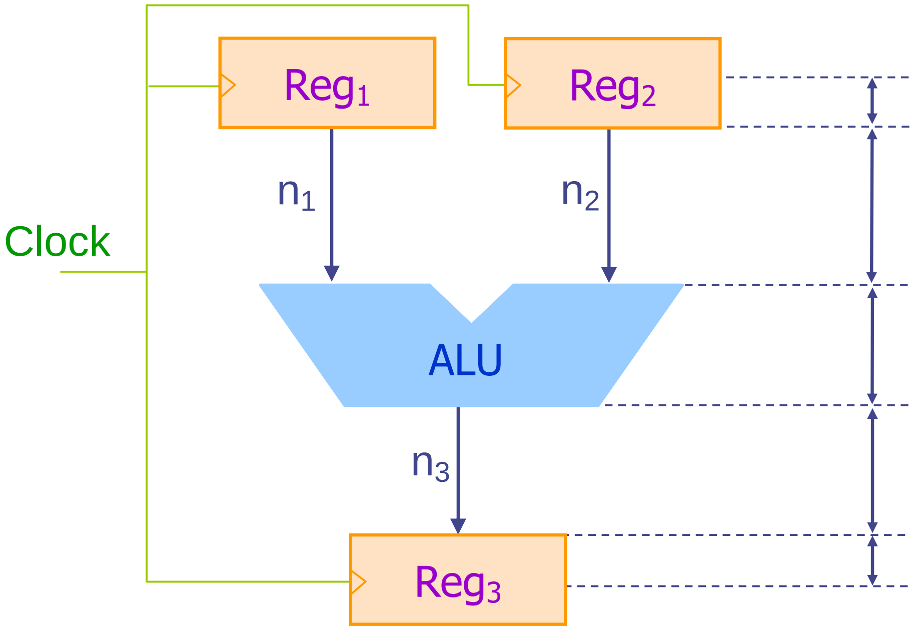
Non-pipelined control unit



Pipelined control unit

System Clock (1/2)

- Register-transfer path



$MAX\ (t_p(\mathrm{Reg}_1),\ t_p(\mathrm{Reg}_2))$

$MAX\ (t_p(\mathrm{n}_1),\ t_p(\mathrm{n}_2))$

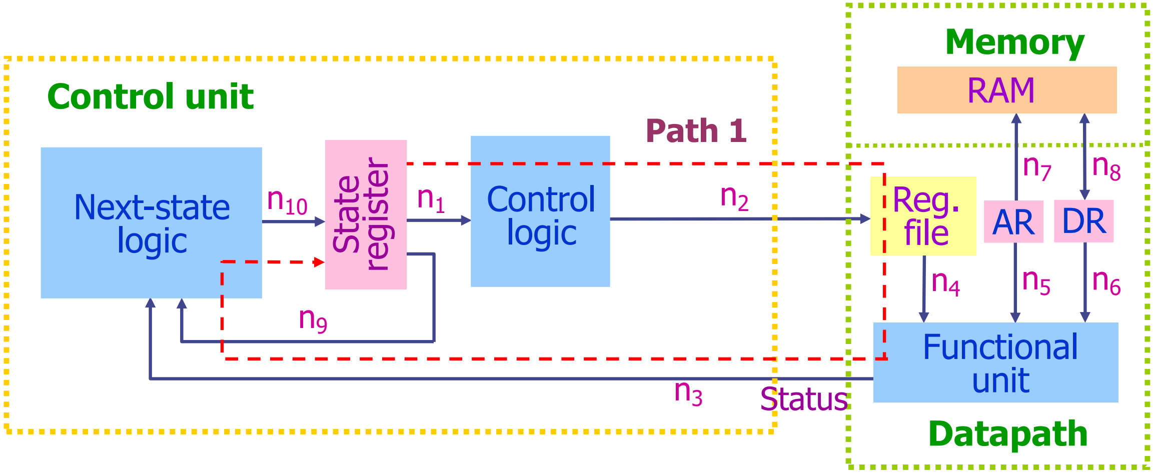
$t_p(\mathrm{ALU})$

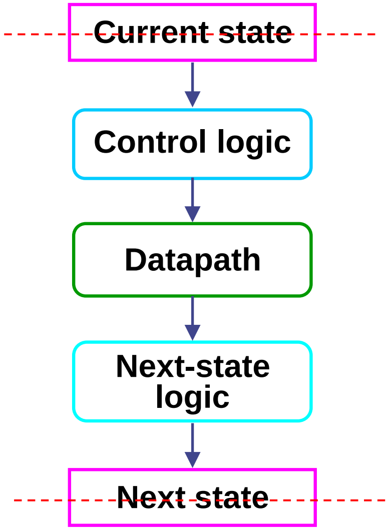
$t_p(\mathrm{n}_3)$

$t_{setup}(\mathrm{Reg}_3)$

System Clock (2/2)

- System Clock

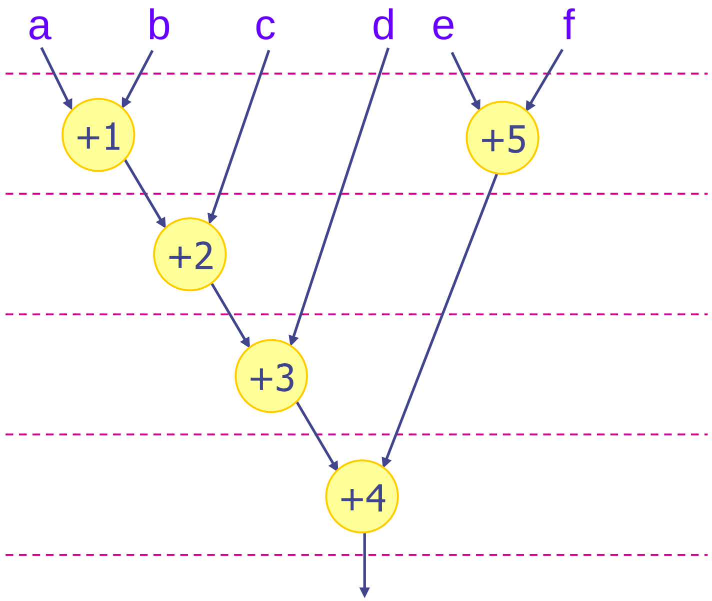




$$
\begin{aligned}
\mathbf{t}_{clock} = \; & \mathbf{t}_p\,(State\ register) + \mathbf{t}_p\,(Control\ logic) + \mathbf{t}_p\,(Reg.\ file) \\
& + \mathbf{t}_p\,(FU) + \mathbf{t}_p\,(Next\text{-}state\ logic) + \mathbf{t}_{setup}\,(State\ register) \\
& + \sum_{i=1,2,3,4,10} \mathbf{t}_p\,(n_i)
\end{aligned}
$$

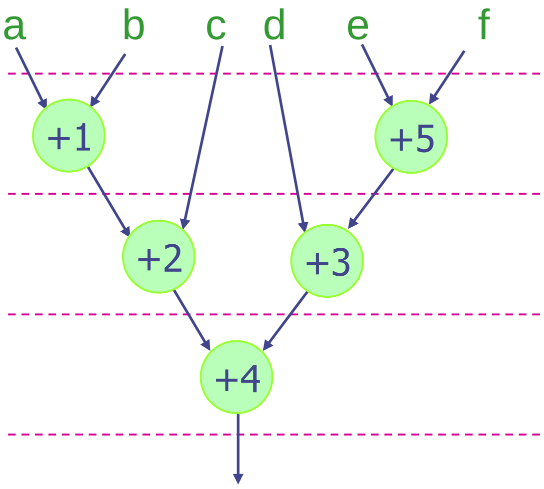
High-level Transformations (1/2)

- Tree-Height Reduction: *a+b+c+d+e+f*



(((a+b)+c)+d) + (e+f)

before

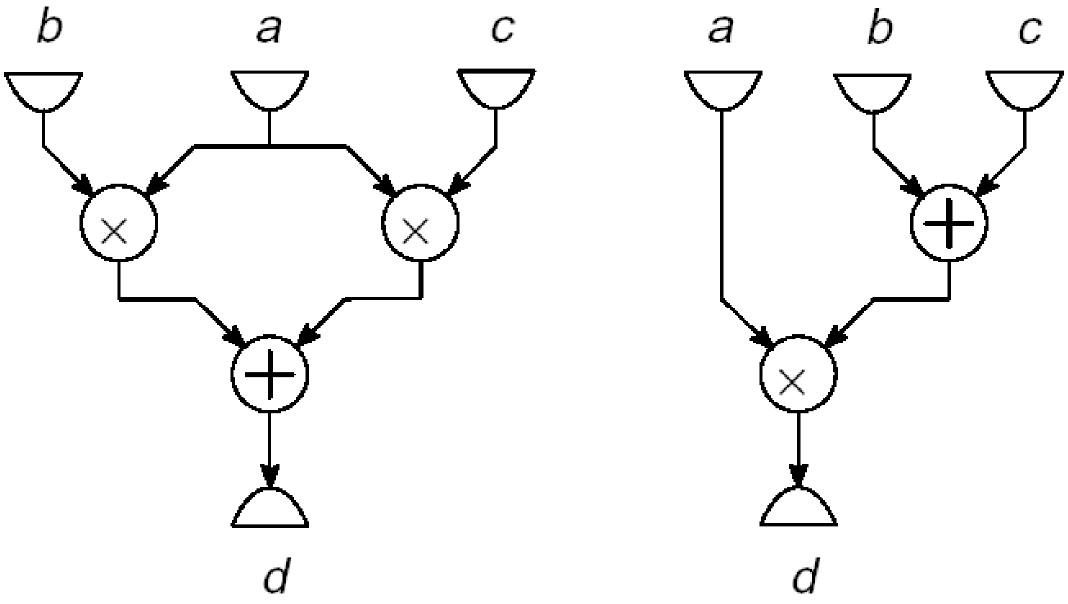


((a+b)+c) +(d +(e+f))

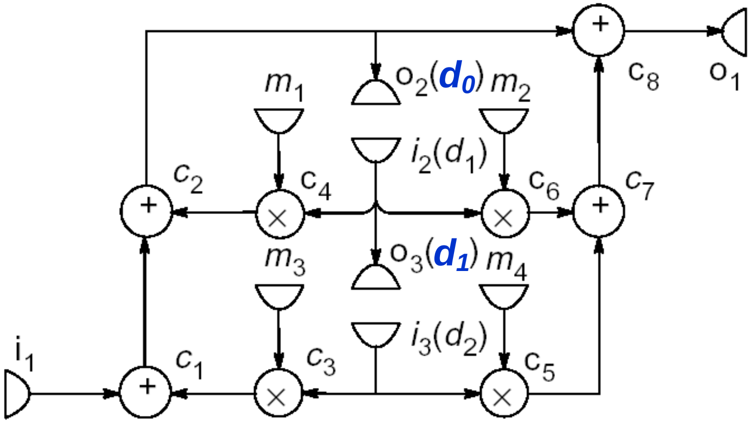
after

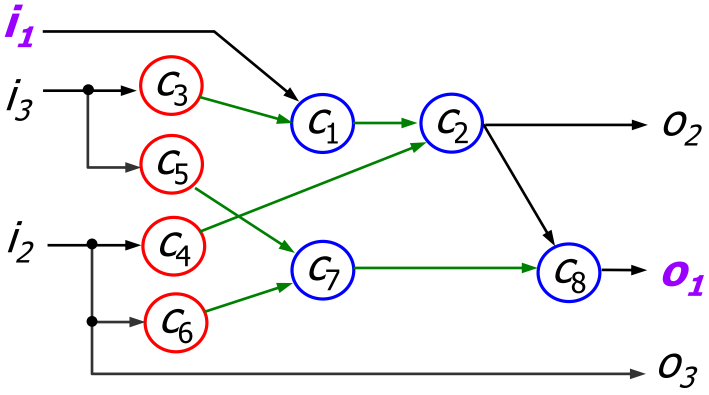
High-level Transformations (2/2)

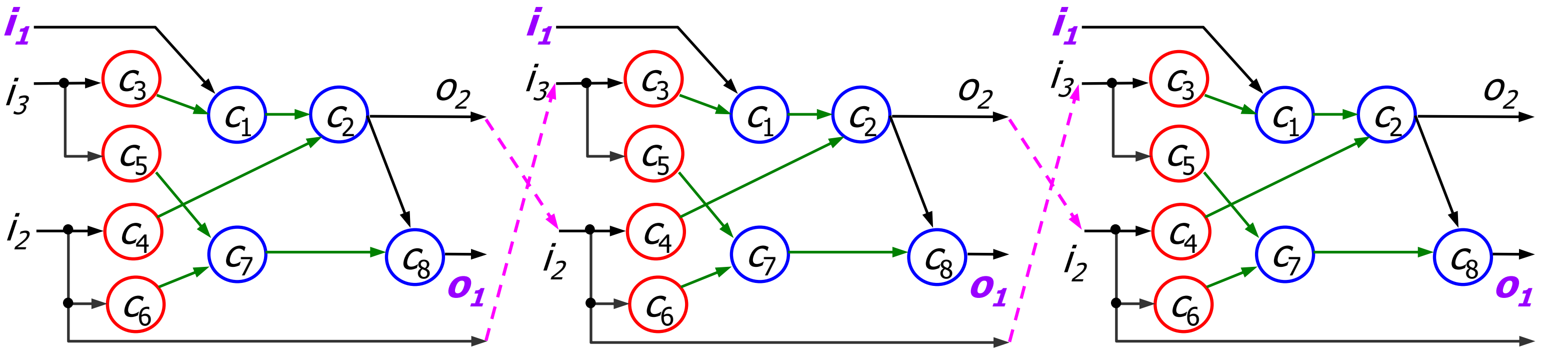
- Distributivity: $a \times b + a \times c = a \times (b + c)$
  - Is perfectly valid form a mathematical point of view
  - Expressions are performed on hardware
    - does not necessarily hold due to “finite word-length effects”



Second-order Filter (1/3)

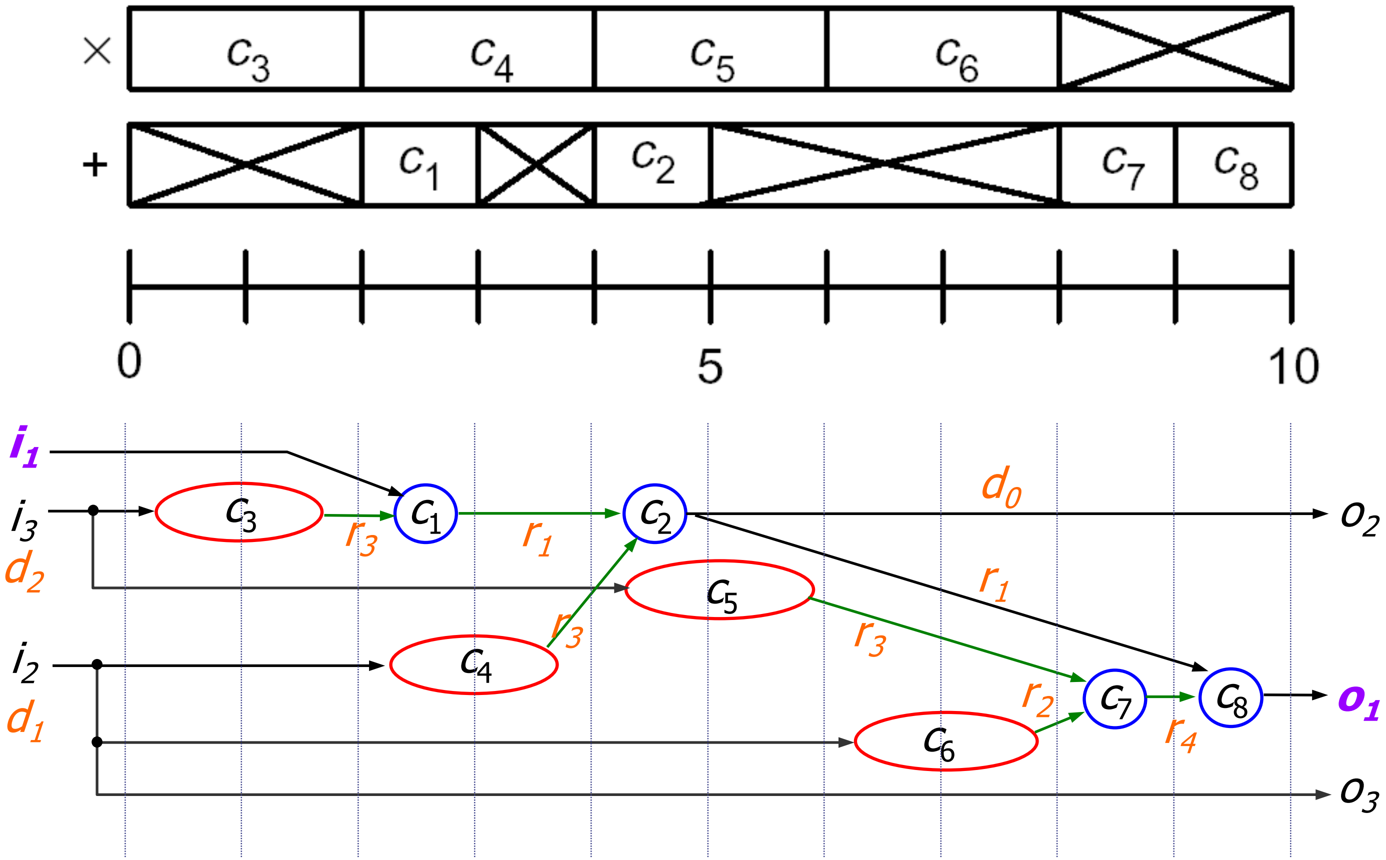






Second-order Filter (2/3)

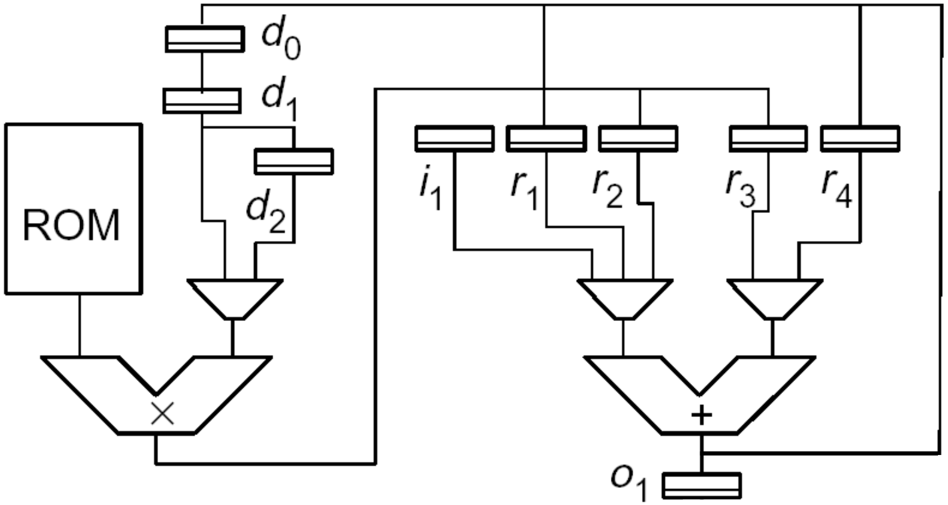
- Another Detailed Example
  - The schedule (precedence relations) and operation assignment with an allocation of one adder and one multiplier



Second-order Filter (3/3)

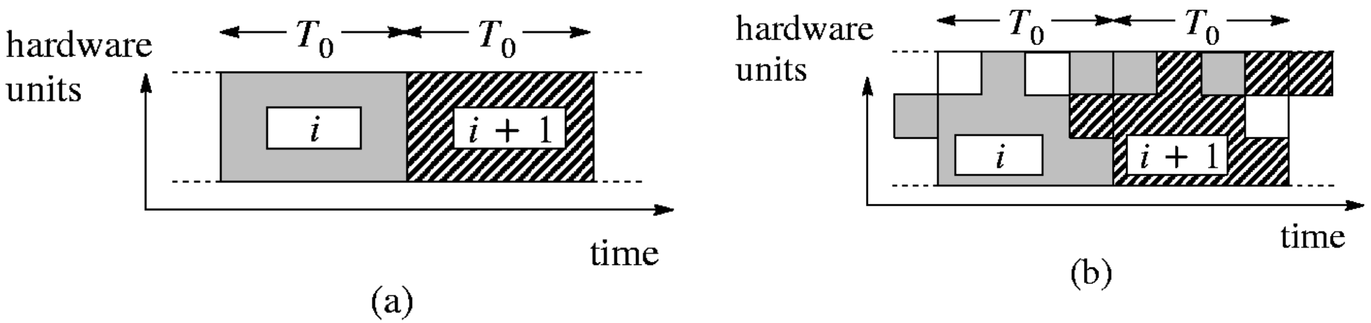
- Resulting data path

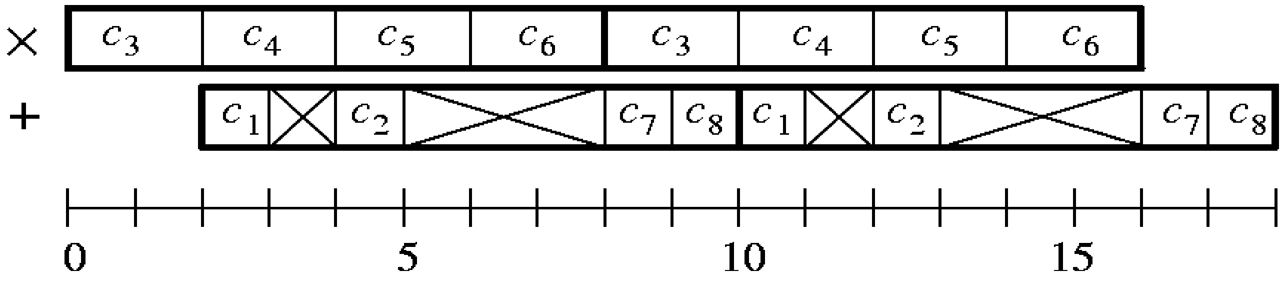
operation	first input	second input	output
c_1	i_1	r_3	r_1
c_2	r_1	r_3	r_1, d_0
c_3	ROM	d_2	r_3
c_4	ROM	d_1	r_3
c_5	ROM	d_2	r_3
c_6	ROM	d_1	r_2
c_7	r_2	r_3	r_4
c_8	r_1	r_4	o_1

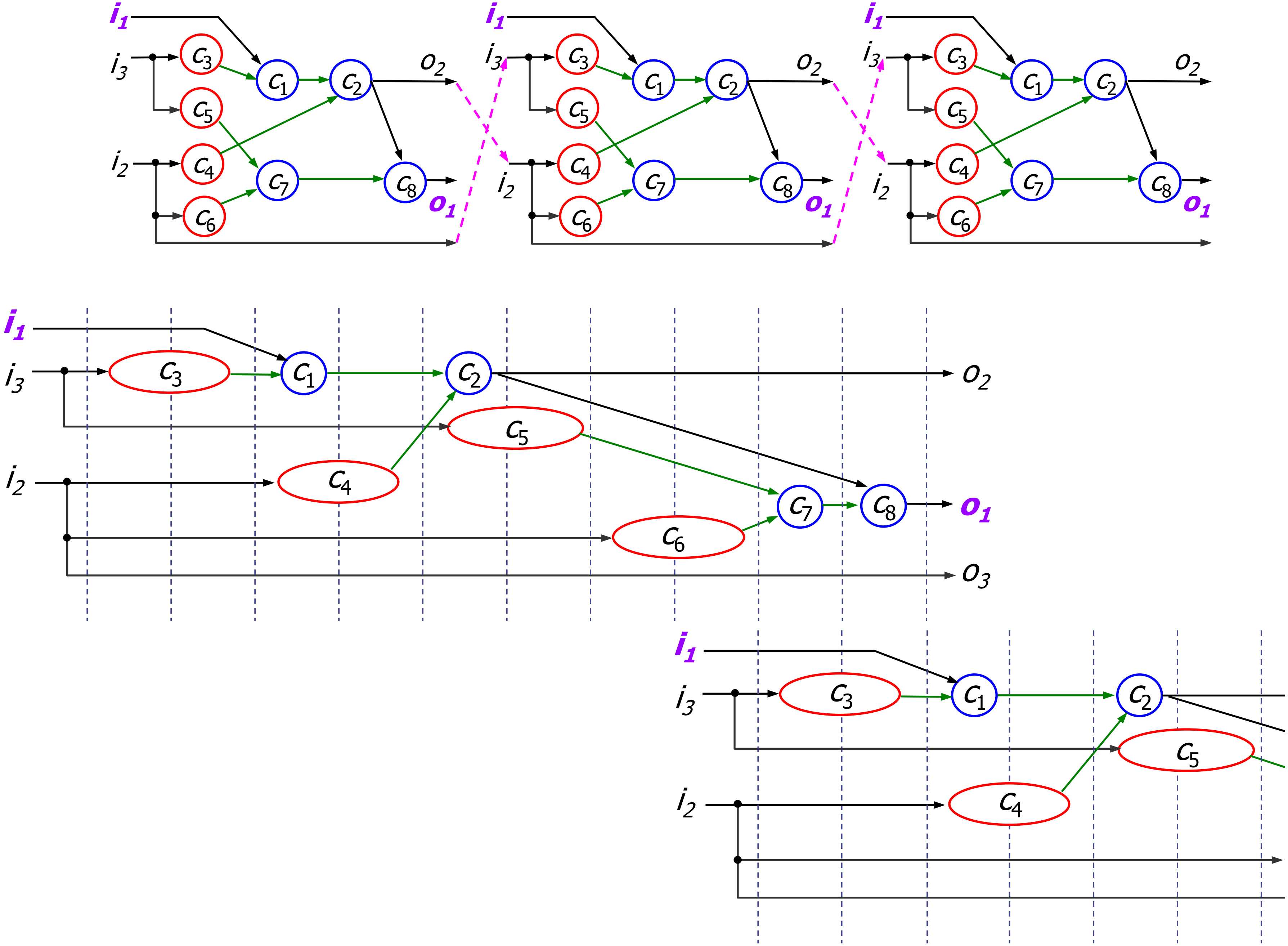


Loop Scheduling (1/3)

- The parallelism between operations belongs to
  - The same iteration: intra-iteration parallelism
  - The different iterations: inter-iteration parallelism
    - The search space is larger
    - Overlapped schedules can be generated (loop folding, software pipelining)



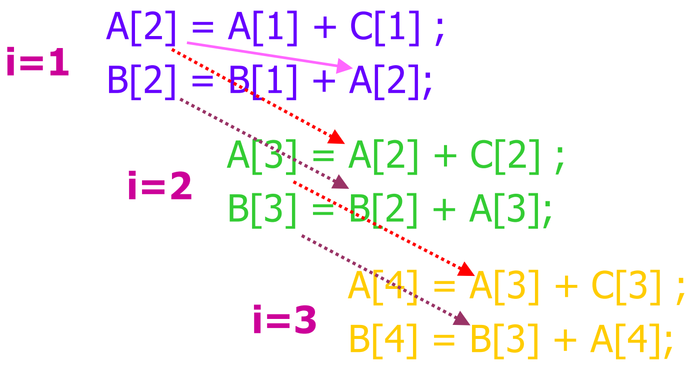



i_1
i_3
i_2
c_1
c_2
c_3
c_4
c_5
c_6
c_7
c_8
o_2
o_1
o_3

Loop Scheduling (2/3)

- loop-carried dependence
  - computations in later iteration are dependent on data values produced in earlier iterations

```
for (i=1 ; i<=100; i=i+1) {
      A[i+1] = A[i] + C[i] ;
      B[i+1] = B[i] + A[i+1];
}
```



i=1
A[2] = A[1] + C[1] ;
B[2] = B[1] + A[2];

i=2
A[3] = A[2] + C[2] ;
B[3] = B[2] + A[3];

i=3
A[4] = A[3] + C[3] ;
B[4] = B[3] + A[4];

Loop Scheduling (3/3)

Sequential execution

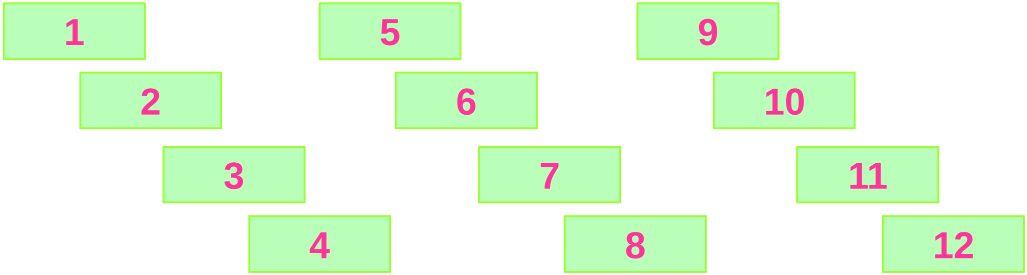
1	2	3	4	5	6	7	8	9	10	11	12

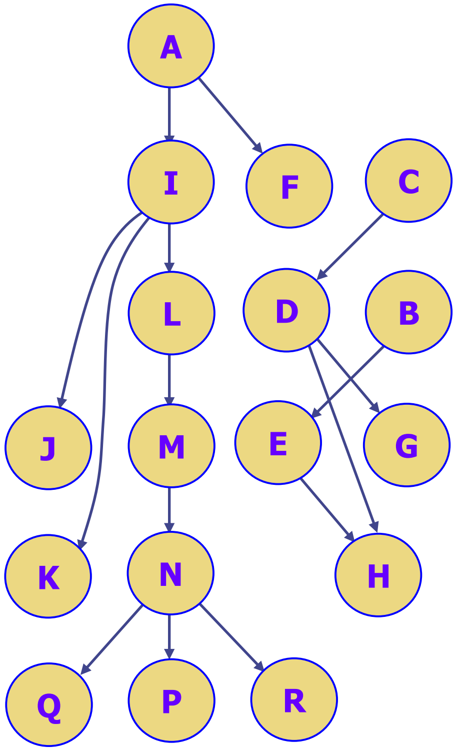
time

Partial loop unrolling

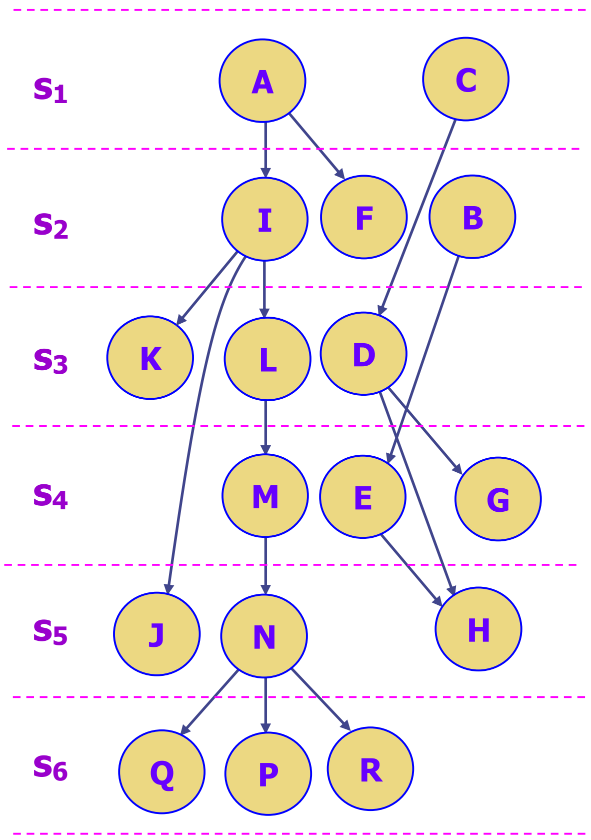
1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12

Loop folding

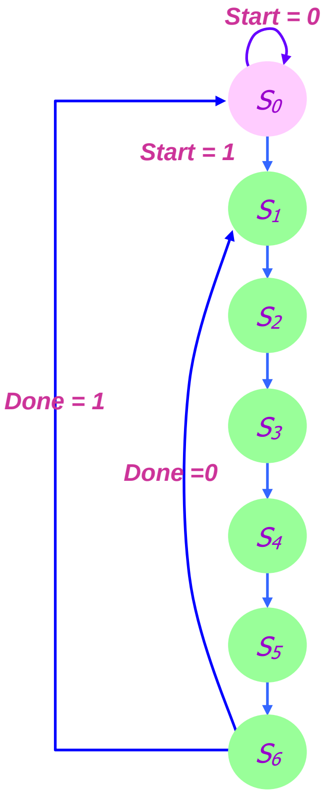


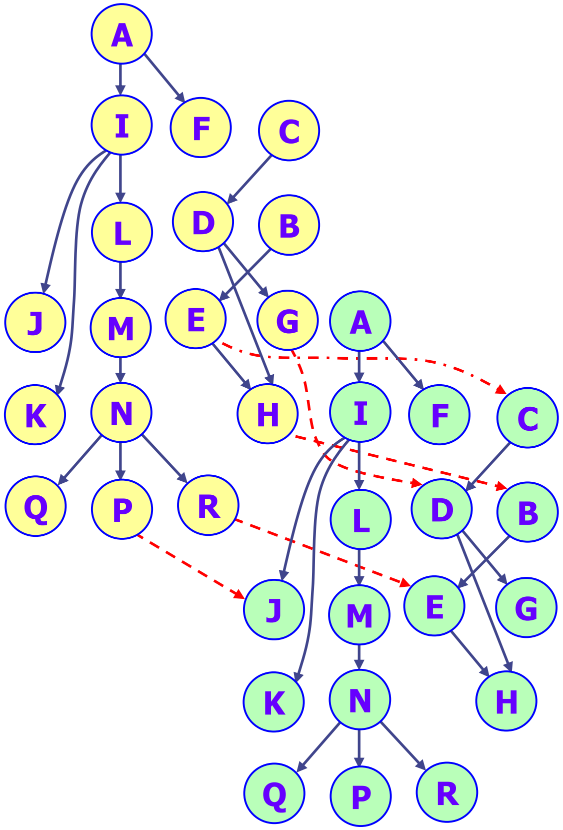


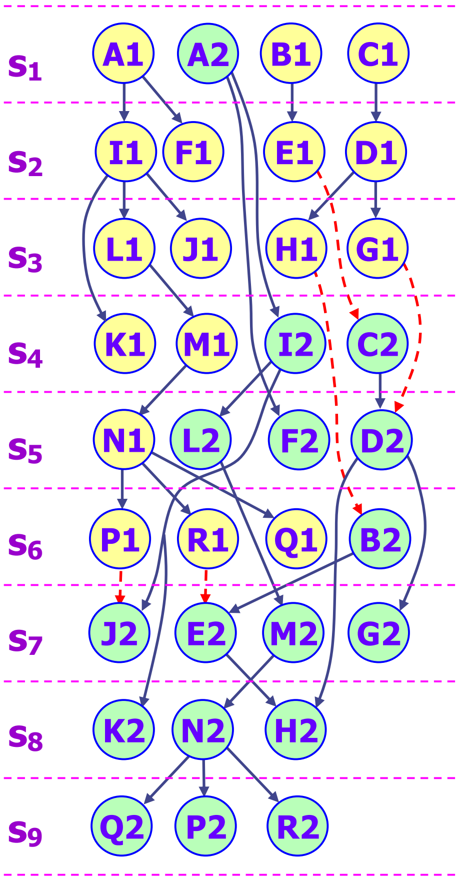
DFG



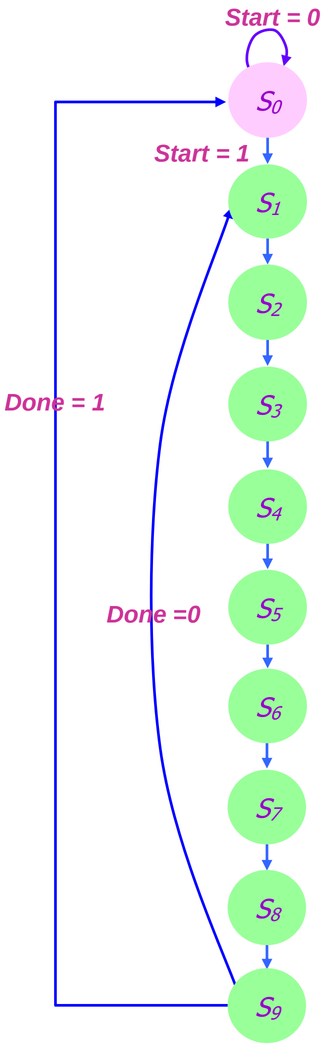
Seq. Schedule w/3 FU’s

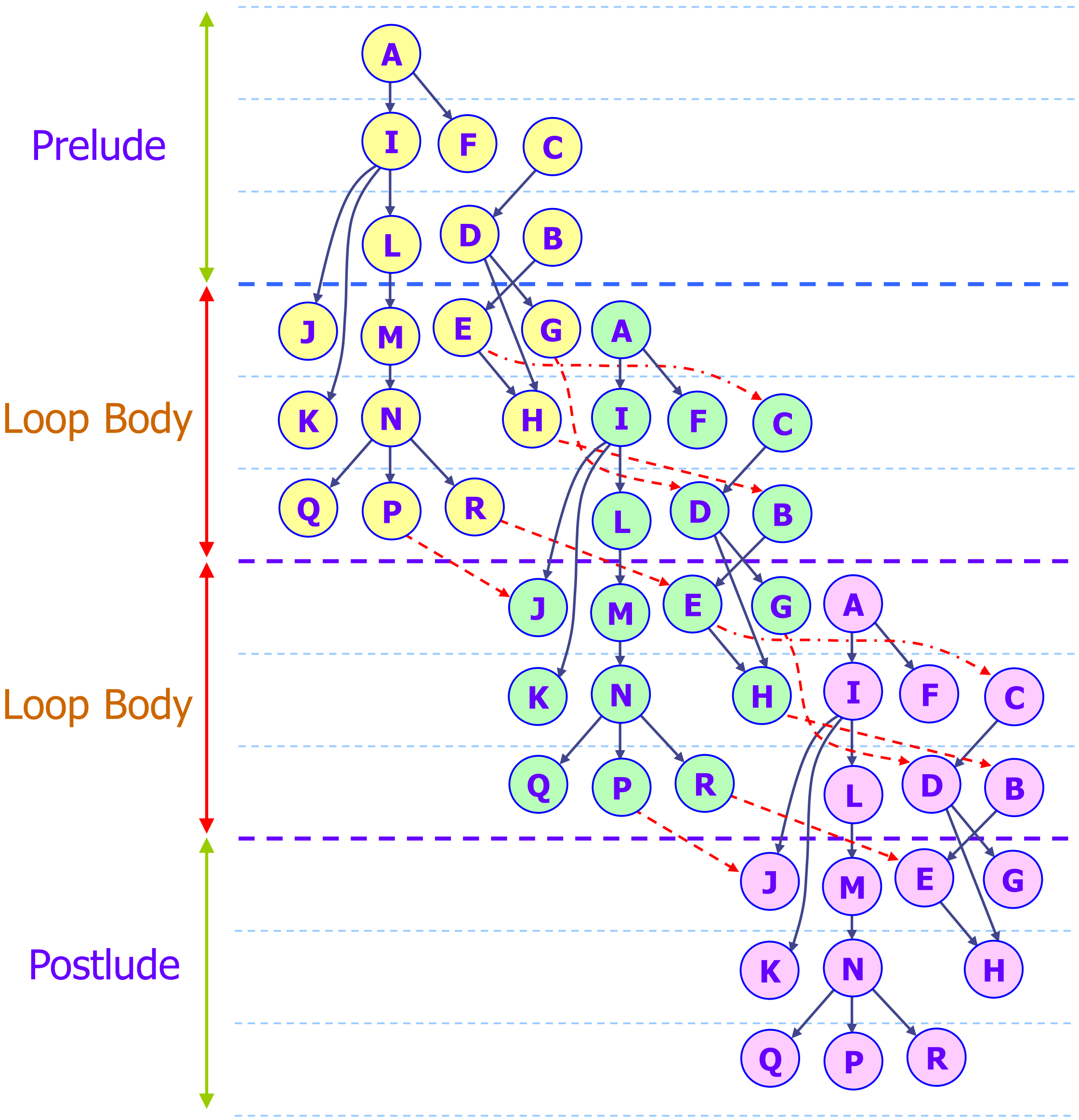


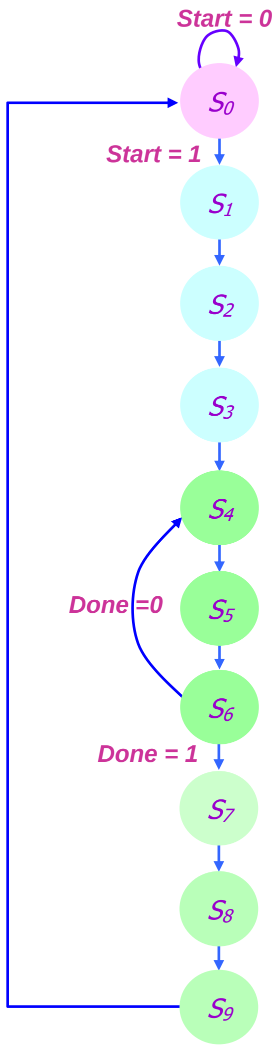




Loop Unrolling

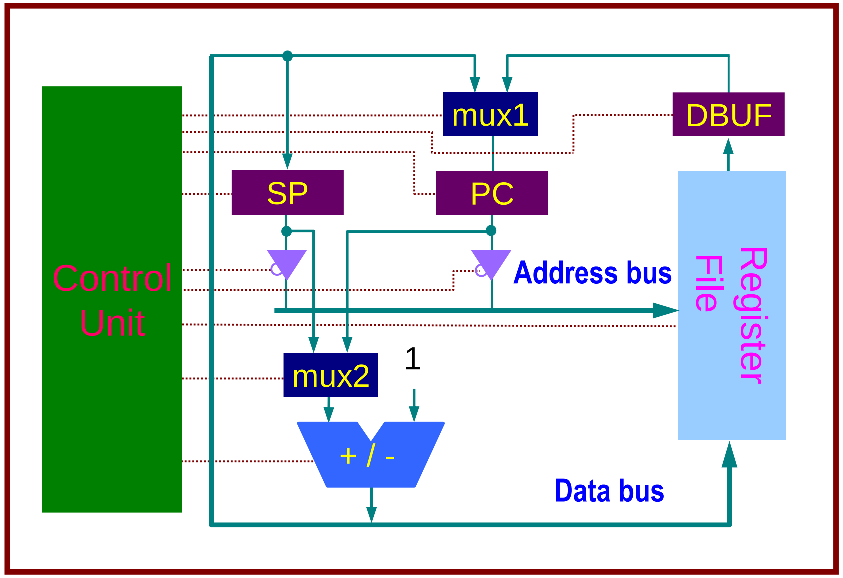


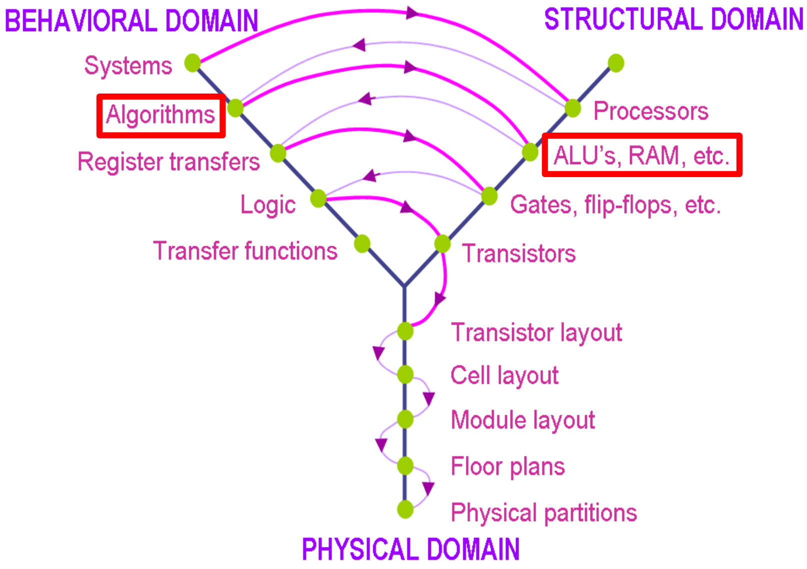


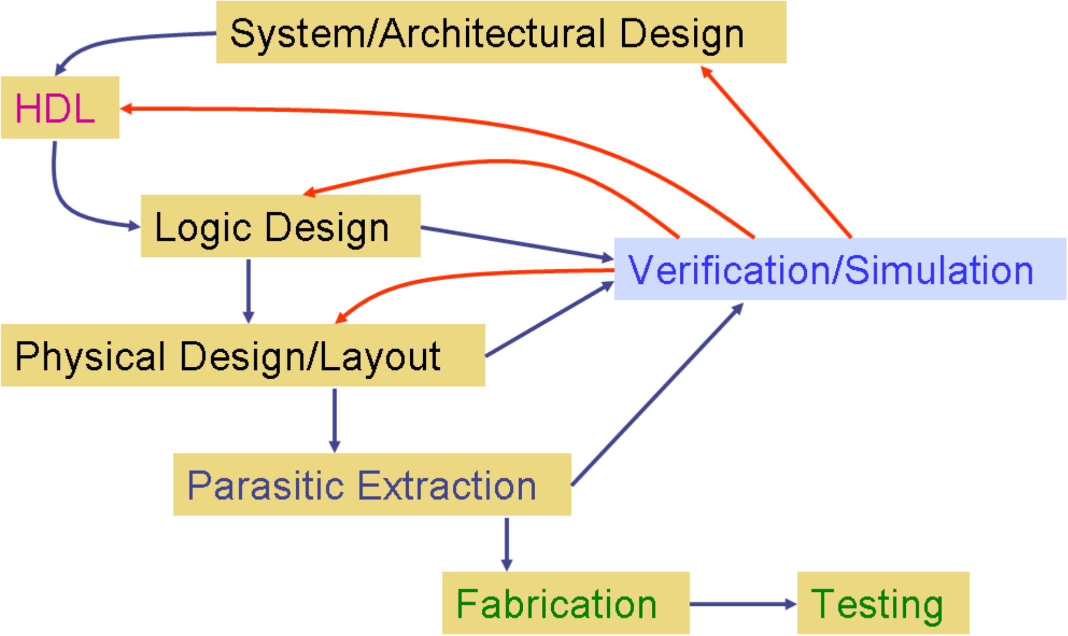


Loop Folding

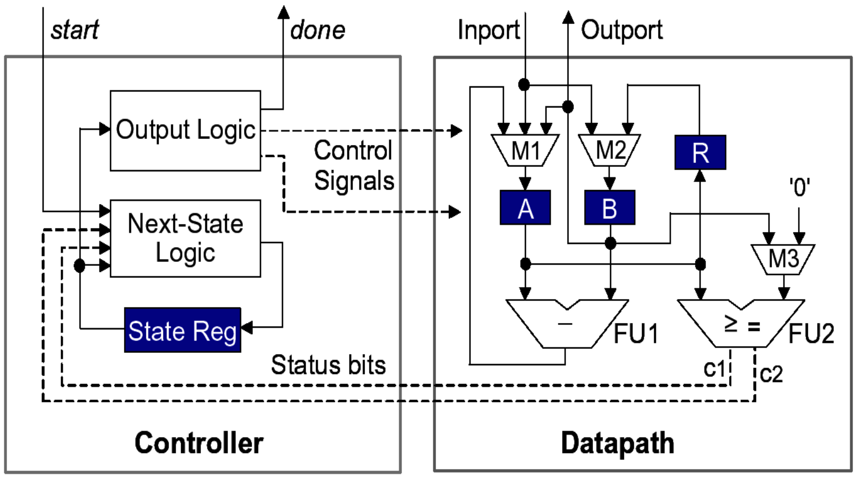
```
if IR(3) = '0' then
      PC          := PC + 1;
else
      DBUF        := MEM(PC);
      MEM(SP)     := PC + 1;
      SP          := SP – 1;
      PC          := DBUF;
end if;
```







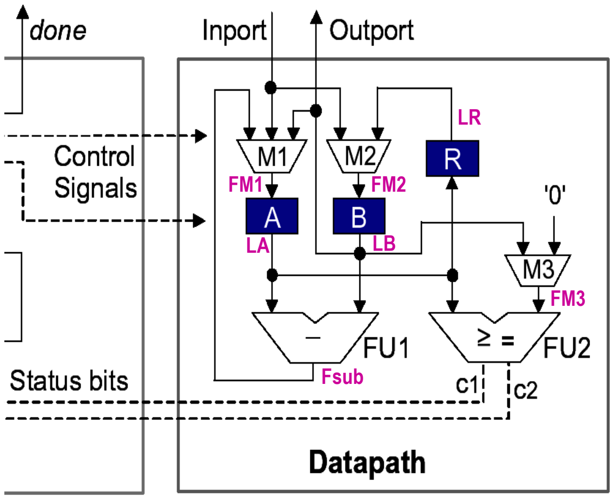
GCD Example (1/4)



```
`define bits 8

module GCD(start, inport, clk, rst, outport, done);
input       start, clk, rst;
input       [`bits-1:0] inport;
output      done;
output      [`bits-1:0] outport;

wire        C_1, C_2;
wire        [6:0] control;

Controller Controller( .start(start), .C_1(C_1), .C_2(C_2), .rst(rst),
                                .clk(clk), .control(control), .done(done));
Datapath Datapath( .inport(inport), .control(control),
                                .clk(clk), .rst(rst), .outport(outport), .C_1(C_1), .C_2(C_2));

endmodule
```

GCD Example (2/4)

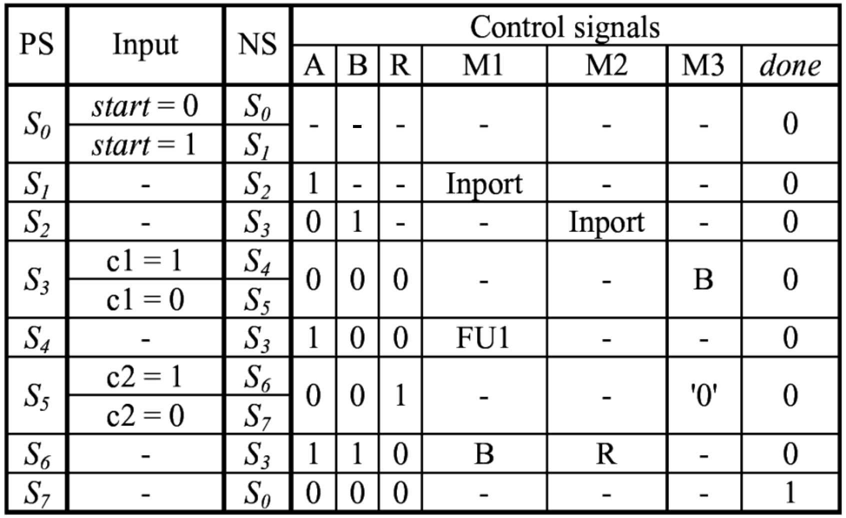
```
module Datapath(inport, control, clk, rst, outport, C_1, C_2);
input       [`bits-1:0] inport;
input       [6:0] control;
input       clk, rst;
output      [`bits-1:0] outport;
output      C_1, C_2;

wire        [`bits-1:0] FM1, FM2, FM3, Fsub, LA, LB, LR;

Register A( .D(FM1), .Q(LA), .load(control[6]), .reset(rst), .clk(clk));
Register B( .D(FM2), .Q(LB), .load(control[5]), .reset(rst), .clk(clk));
Register R( .D(LA), .Q(LR), .load(control[4]), .reset(rst), .clk(clk));

MUX3        M1( .A(Fsub), .B(inport), .C(LB), .S(control[3:2]), .Y(FM1) );
MUX2        M2( .A(inport), .B(LR), .S(control[1]), .Y(FM2) );
MUX2        M3( .A(LB), .B(8'd0), .S(control[0]), .Y(FM3) );

Sub         FU1( .A(LA), .B(LB), .Sub(Fsub) );

Compare     FU2( .A(LA), .B(FM3), .C_1(C_1), .C_2(C_2) );

assign      outport = LB;

endmodule
```



```
`define bits 8

module Sub(A, B, Sub);
input       [`bits-1:0] A, B;
output      [`bits-1:0] Sub;

assign Sub = A - B;

endmodule
```

GCD Example (3/4)

```
module Controller(start, C_1, C_2 , rst, clk, control, done);
input           start, C_1, C_2, clk, rst;
output          done;
output          [6:0] control;

reg             done;
reg             [6:0] control;
reg             [2:0] Current_State, Next_State;

always @(posedge clk or negedge rst)
begin
    if(~rst)   Current_State <= `S0;
    else   Current_State <= Next_State;
end

always @(Current_State or start or C_1 or C_2)
begin
     case (Current_State)
         `S0:
          begin
                control = 7'b0_0_0_00_0_0;   //7 bit:A_B_R_M1_M2_M3
                done = 1'b0;
                if(~start) Next_State = `S0;
                else  Next_State = `S1;
          end

          `S1:
           begin
                control = 7'b1_0_0_01_0_0;
                done = 1'b0;
                Next_State = `S2;
           end
```



PS	Input	NS	Control signals						
			A	B	R	M1	M2	M3	*done*
S_0	*start* = 0	S_0	-	-	-	-	-	-	0
	start = 1	S_1							
S_1	-	S_2	1	-	-	Inport	-	-	0
S_2	-	S_3	0	1	-	-	Inport	-	0
S_3	c1 = 1	S_4	0	0	0	-	-	B	0
	c1 = 0	S_5							
S_4	-	S_3	1	0	0	FU1	-	-	0
S_5	c2 = 1	S_6	0	0	1	-	-	'0'	0
	c2 = 0	S_7							
S_6	-	S_3	1	1	0	B	R	-	0
S_7	-	S_0	0	0	0	-	-	-	1

```
`define    bits  8
`define    S0    3'b000
`define    S1    3'b001
`define    S2    3'b010
`define    S3    3'b011
`define    S4    3'b100
`define    S5    3'b101
`define    S6    3'b110
`define    S7    3'b111
```

GCD Example (4/4)

```
`S2:
begin
        control = 7'b0_1_0_00_0_0;
        done = 1'b0;
        Next_State = `S3;
end

`S3:
begin
         control = 7'b0_0_0_00_0_0;
         done = 1'b0;
         if(~C_1)  Next_State = `S5;
         else  Next_State = `S4;
end

`S4:
begin
         control = 7'b1_0_0_00_0_0;
         done = 1'b0;
         Next_State = `S3;
end

`S5:
begin
         control = 7'b0_0_1_00_0_1;
         done = 1'b0;
         if(~C_2)  Next_State = `S7;
         else  Next_State = `S6;
end
```

```
        `S6:
        begin
                control = 7'b1_1_0_10_1_0;
                done = 1'b0;
                Next_State = `S3;
        end

        `S7:
        begin
                control = 7'b0_0_0_00_0_0;
                done = 1'b1;
                Next_State = `S0;
         end
    endcase
end

endmodule
```