Principles Of
Digital Design

C to RTL

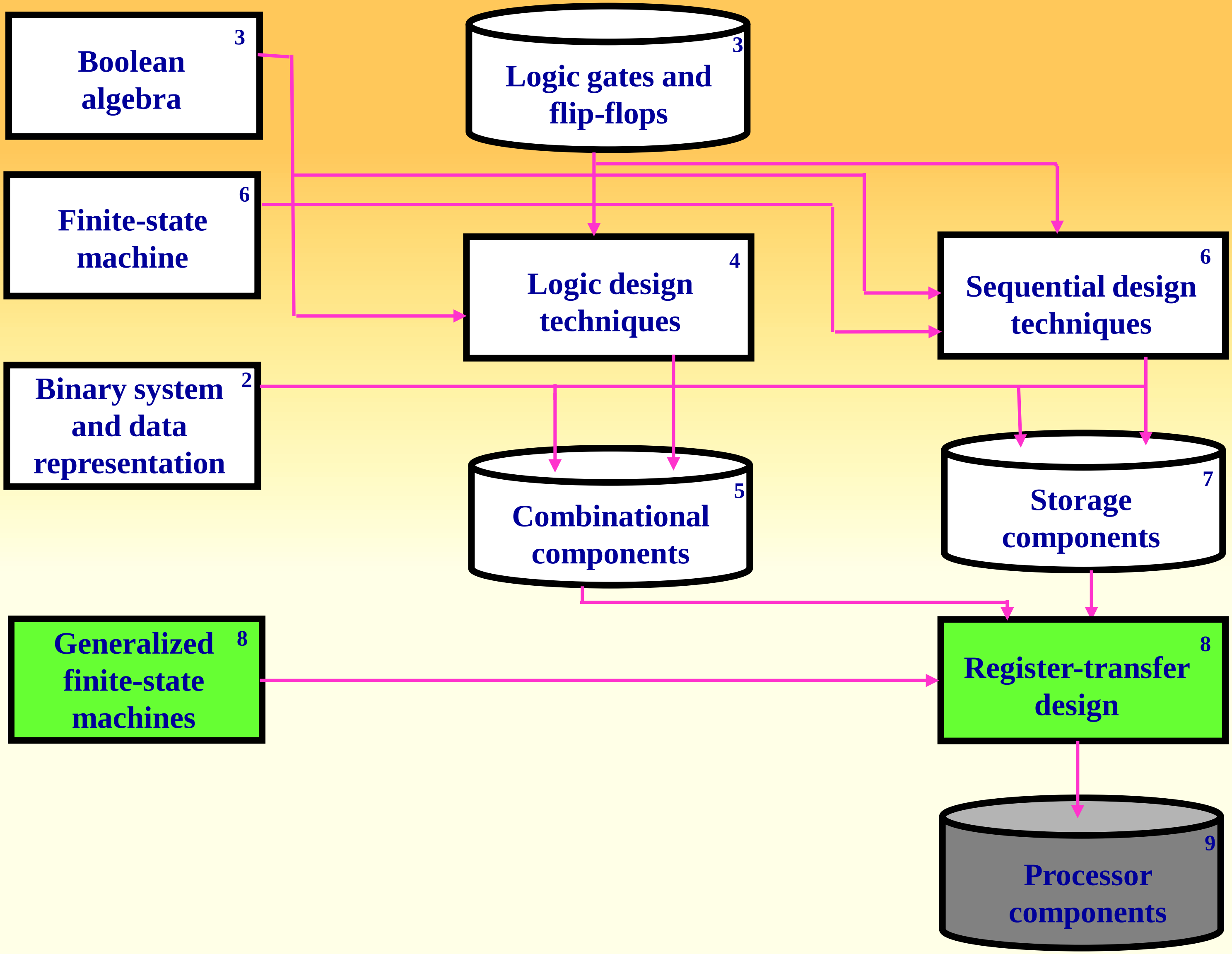
Control/Data flow graphs

Finite-state-machine with data

IP design

- Component selection
- Connection selection
- Operator and variable mapping
- Scheduling and pipelining

Topic preview



Boolean algebra 3

Logic gates and flip-flops 3

Finite-state machine 6

Logic design techniques 4

Sequential design techniques 6

Binary system and data representation 2

Combinational components 5

Storage components 7

Generalized finite-state machines 8

Register-transfer design 8

Processor components 9

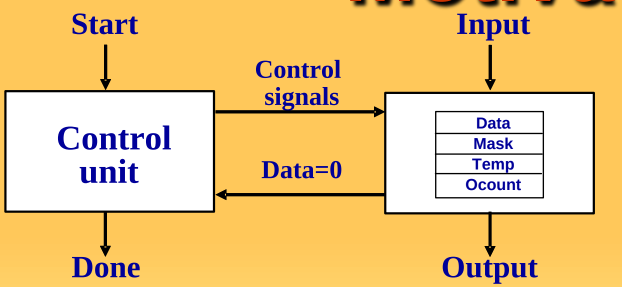
Register-transfer-level design

- Each standard or custom IP components consists of one or more datapaths and control units.

- To synthesize such IP we use the models of a CDFG and FSMD.

- We demonstrate IP synthesis (RTL Design) including
  - component and connectivity selection,
  - expression mapping
  - scheduling and pipelining

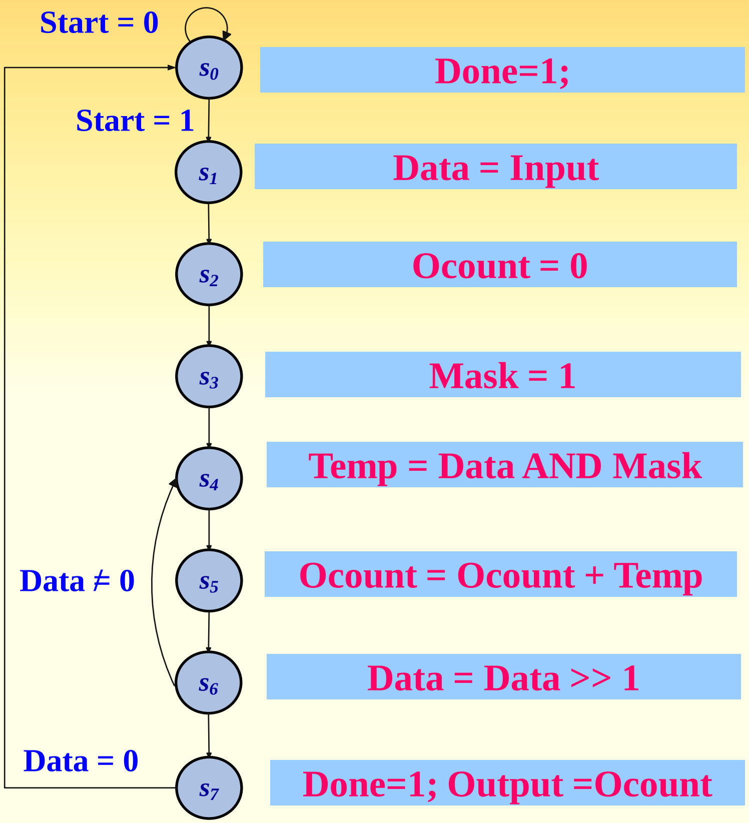
Motivation: Ones-counter



Problem:

Generate controller & control words for given FSMD & Datapath

20-bit control words



Done=1;

Data = Input

Ocount = 0

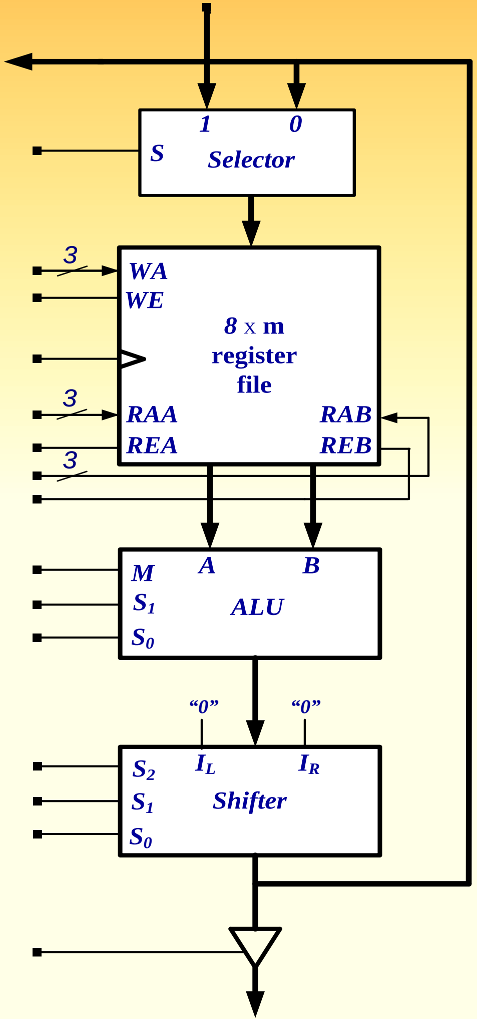
Mask = 1

Temp = Data AND Mask

Ocount = Ocount + Temp

Data = Data >> 1

Done=1; Output =Ocount



Ones Counter from C Code

- **Programming language semantics**
  - **Sequential execution,**
  - **Coding style to minimize coding**
- **HW design**
  - **Parallel execution,**
  - **Communication through signals**

```
int OnesCounter(int Data){
 int Ocount = 0;
 int Temp, Mask = 1;
 while (Data > 0) {
  Temp = Data & Mask;
  Ocount = Data + Temp;
  Data >>= 1;
 }
 return Ocount;
}
```

Function-based C code

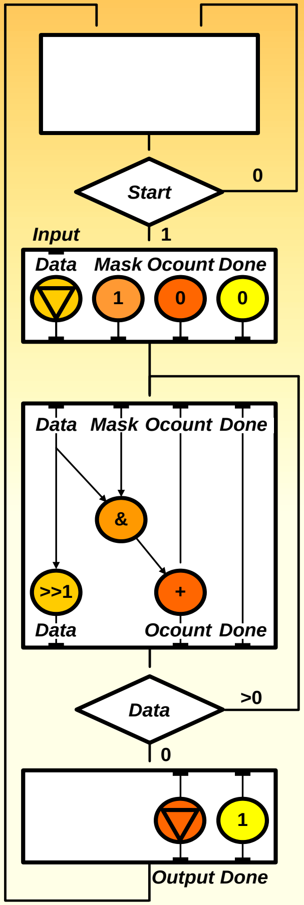
```
while(1) {
  while (Start == 0);
  Done = 0;
  Data = Input;
  Ocount = 0;
  Mask = 1;
  while (Data>0) {
    Temp = Data & Mask;
    Ocount = Ocount + Temp;
    Data >>= 1;
  }
  Output = Ocount;
  Done = 1;
}
```

RTL-based C code

CDFG for Ones Counter

```
while(1) {
  while (Start == 0);
  Done = 0;
  Data = Input;
  Ocount = 0;
  Mask = 1;
  while (Data>0) {
    Temp = Data & Mask;
    Ocount = Ocount + Temp;
    Data >>= 1;
  }
  Output = Ocount;
  Done = 1;
}
```

RTL-based C code

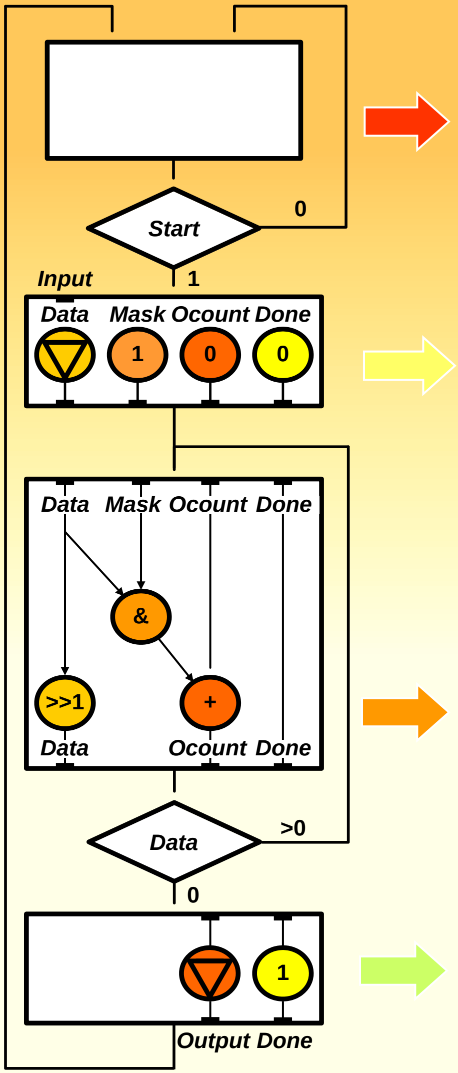


CDFG

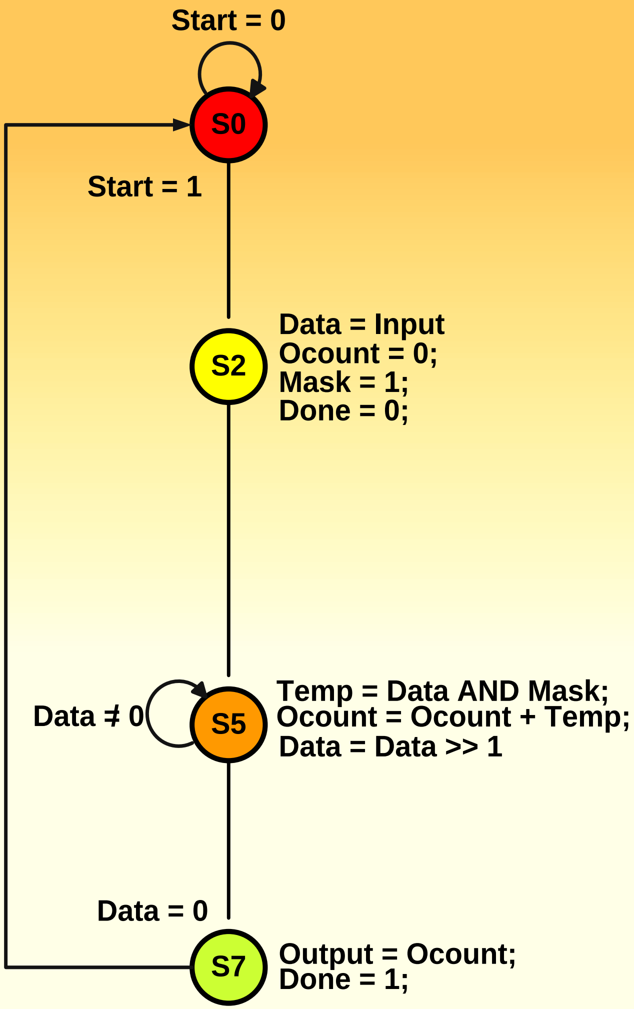
Control/Data flow graph

- •Resembles programming language
  - •Loops, ifs, basic blocks (BBs)
- •Explicit dependencies
  - •Control dependences between BBs
  - •Data dependences inside BBs
- •Missing dependencies between BBs

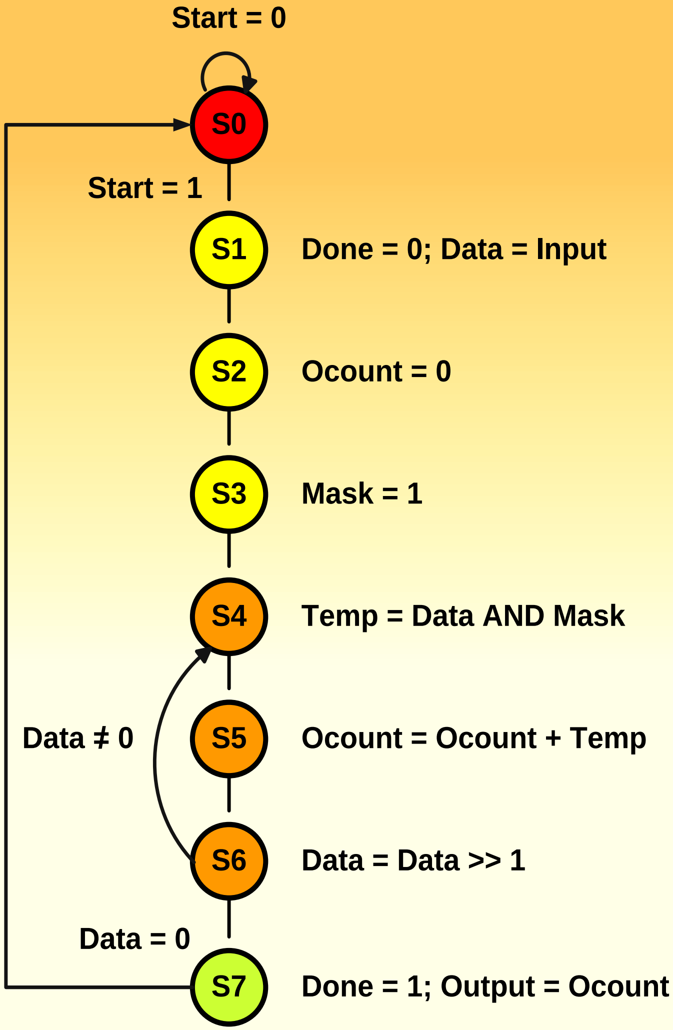
CDFG to FSMD for Ones Counter



CDFG



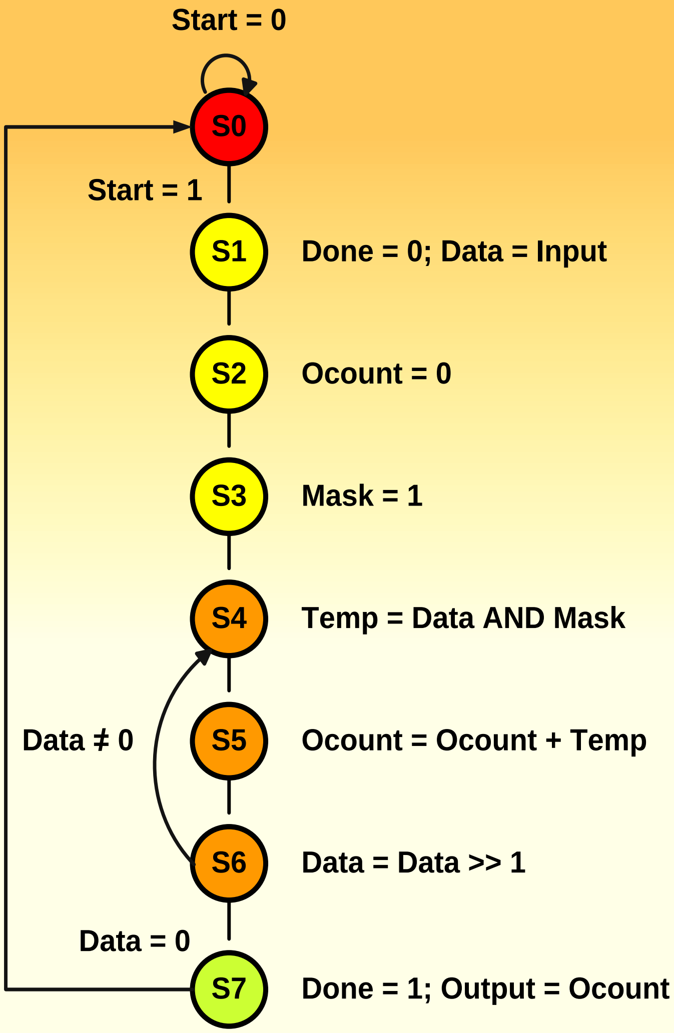
Super-state FSMD



Cycle-accurate FSMD

FSMD for Ones Counter

- FSMD more detailed then CDFG
    - States may represent clock cycles
    - Conditionals and statements executed concurrently
    - All statement in each state executed concurrently
    - Control signal and variable assignments executed concurrently
- FSMD includes scheduling
- FSMD doesn't specify binding or connectivity



FSMD Definition

We defined an FSM as a quintuple $< S, I, O, f, h >$ where S is a set of states, I and O are the sets of input and output symbols:

$$f : S \times I \rightarrow S \quad , \text{ and } \quad h : S \rightarrow O$$

More precisely,

$$I = A_1 \times A_2 \times \dots A_k$$

$$S = Q_1 \times Q_2 \times \dots Q_m$$

$$O = Y_1 \times Y_2 \times \dots Y_n$$

Where A_i, is an input signal, Q_i, is the state register output and Y_i, is an output signal.

To define a FSMD, we define a set of variables, $V = V_1 \times V_2 \times \dots V_q$, which defines the state of the datapath by defining the values of all variables in each state with the set of expressions Expr(V):

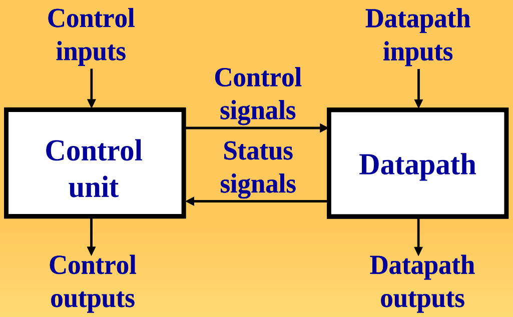
$$\text{Expr}(V) = \text{Const} \cup V \cup \{e_i \# e_j \mid e_i, e_j \text{ el of Expr}(V), \# \text{ is an operation}\}$$

Notes: 1. Status signal is a signal in I;

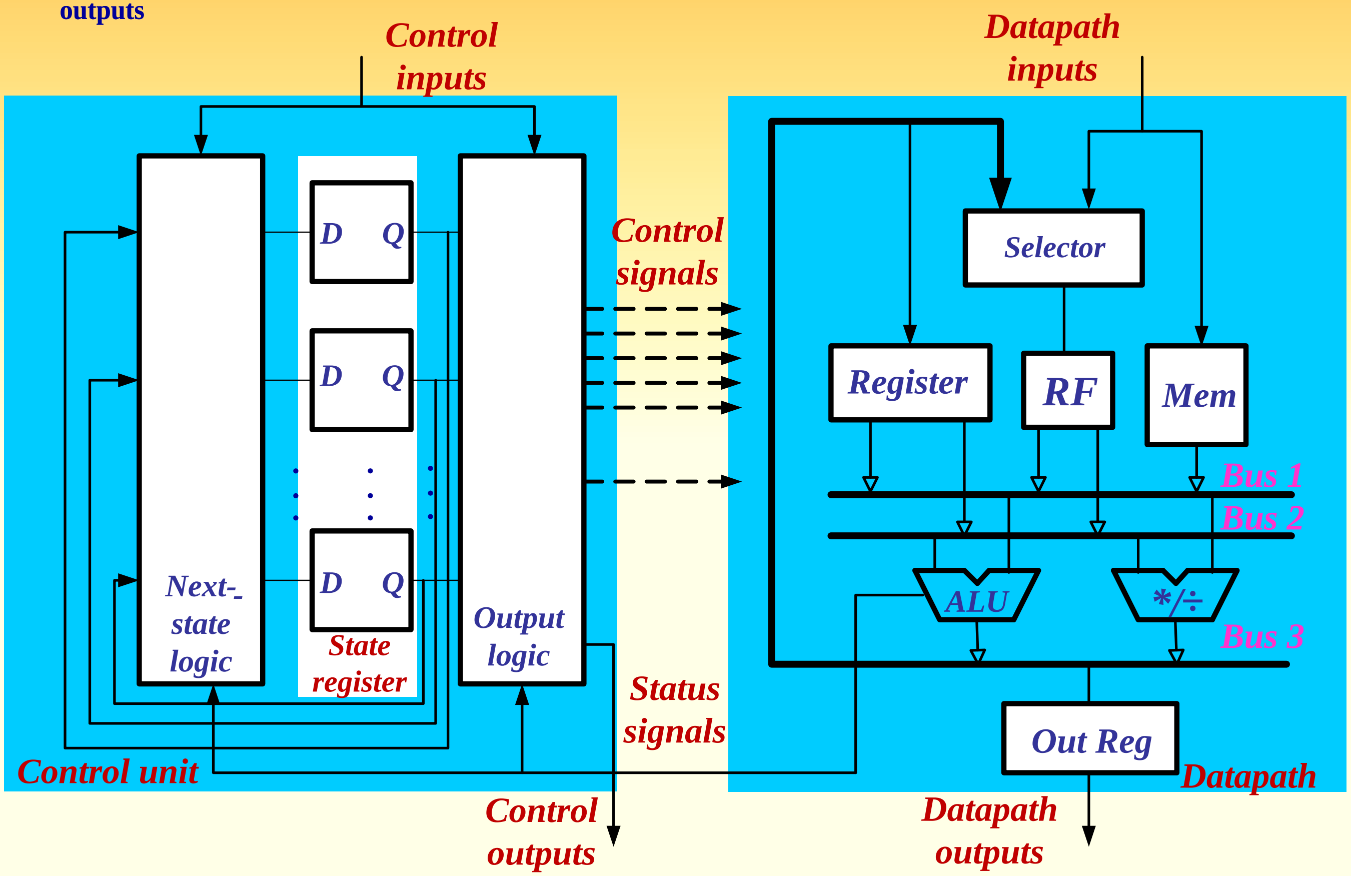
2. Control signals are signals in O;

3. Datapath inputs and outputs are variables in V

RTL Design Model

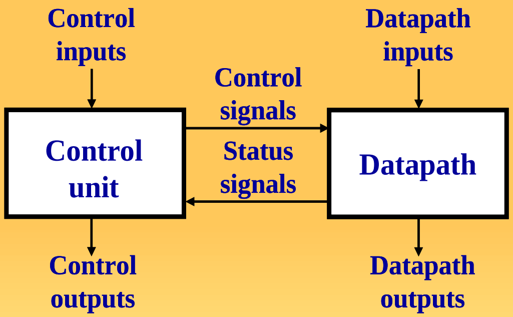


High-level block diagram

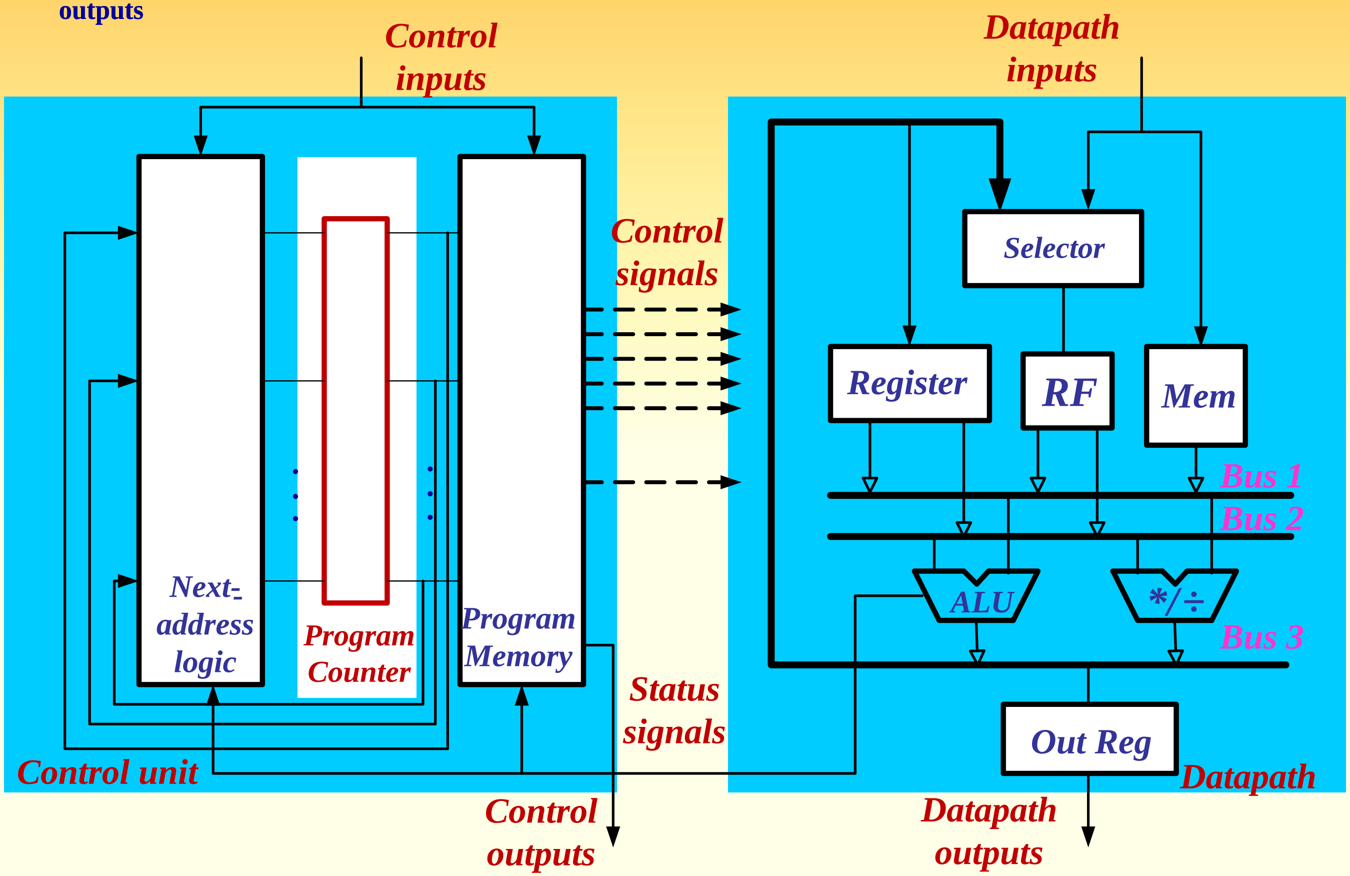


Register-transfer-level block diagram

RTL Design Model



High-level block diagram



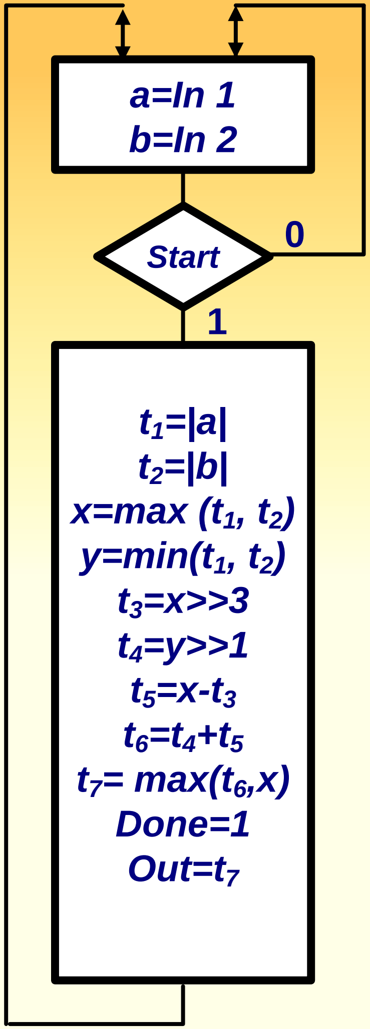
Register-transfer-level block diagram

C-to-RTL design

- RTL generation requires definition of
  - controller
  - datapath
- RTL generation of a controller requires choice of
  - state register (program counter)
  - output logic (program memory)
  - next-state logic (next-address generator)
- RTL generation of a datapath
  - RTL component and connectivity selection,
  - expression mapping (variable and operation mapping)
  - scheduling and pipelining

Square Root Approximation: C to CDFG

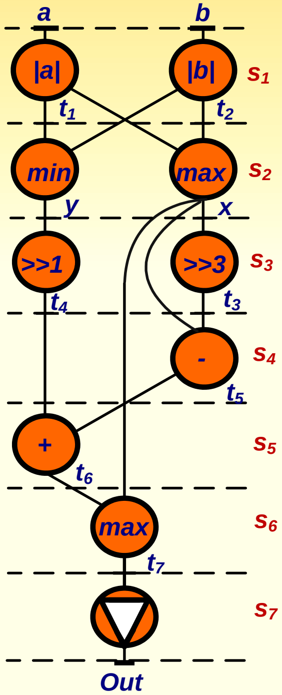
Example: Sq root (a + b) = max(0.875 x + 0.5 y), where x = max(|a|, |b|), y = min (|a|, |b|)



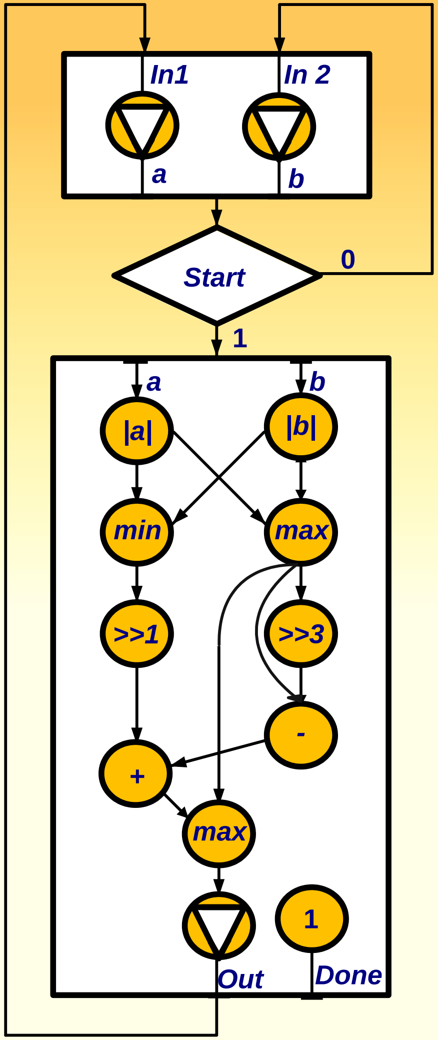
a=In 1
b=In 2

t_1=|a|
t_2=|b|
x=max (t_1, t_2)
y=min(t_1, t_2)
t_3=x>>3
t_4=y>>1
t_5=x-t_3
t_6=t_4+t_5
t_7= max(t_6,x)
Done=1
Out=t_7

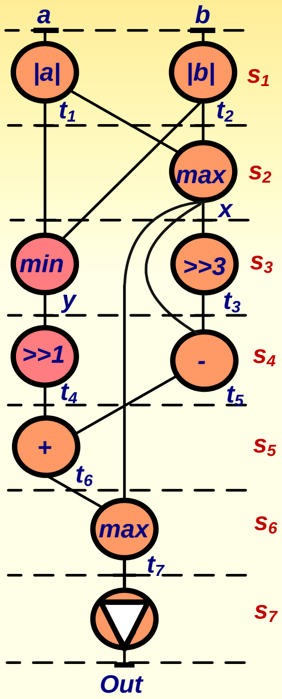
Flowchart



ASAP



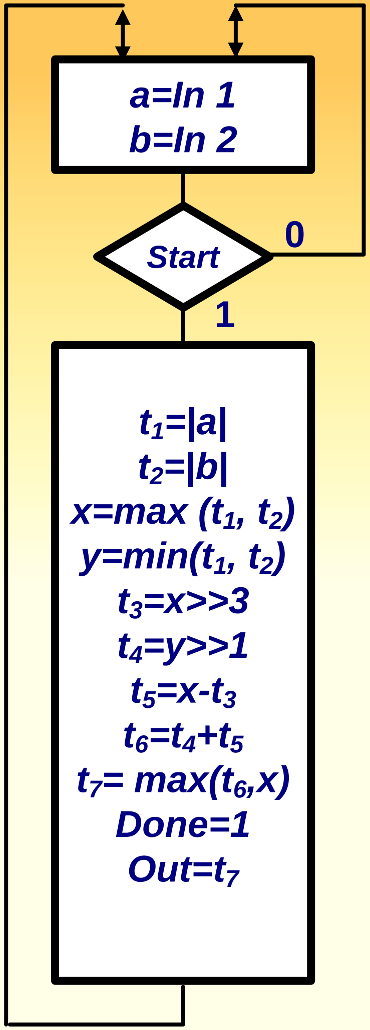
Control/Data flow graph



ALAP

Square Root Approximation: Scheduling

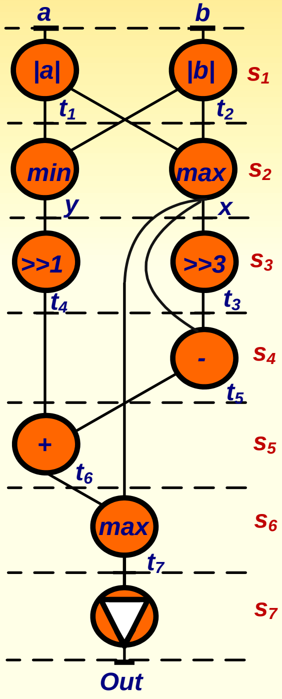
Example: Sq root (a + b) = max(0.875 x + 0.5 y), where x = max(|a|, |b|), y = min (|a|, |b|)



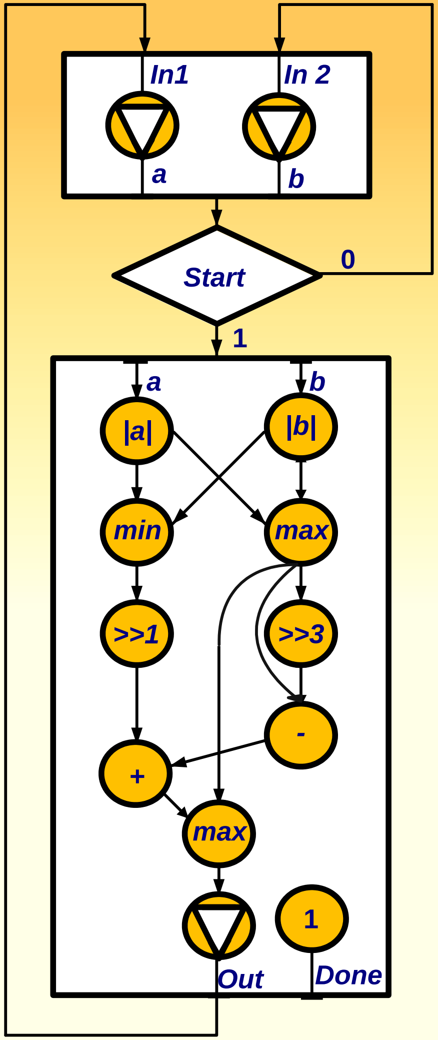
$a=In\ 1$
$b=In\ 2$

$t_1=|a|$
$t_2=|b|$
$x=max\ (t_1,\ t_2)$
$y=min(t_1,\ t_2)$
$t_3=x>>3$
$t_4=y>>1$
$t_5=x-t_3$
$t_6=t_4+t_5$
$t_7=\ max(t_6,x)$
$Done=1$
$Out=t_7$

Flowchart

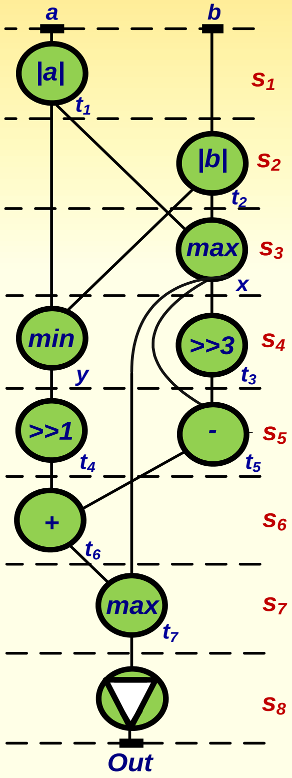


ASAP



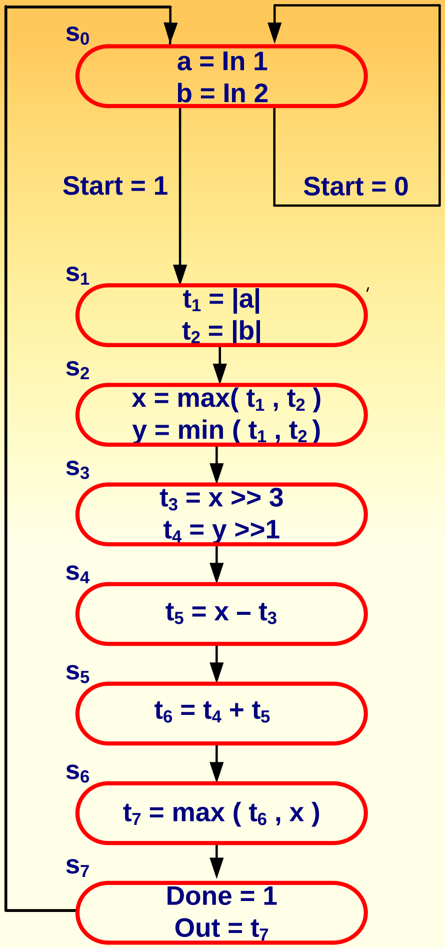
Control/Data flow graph

Resource-constrained



Square Root Approximation: CDFG to FSMD

Example: Sq root (a + b) = max(0.875 x + 0.5 y), where x = max(|a|, |b|), y = min (|a|, |b|)



s_0: a = In 1, b = In 2

Start = 1

Start = 0

s_1: t_1 = |a|, t_2 = |b|

s_2: x = max(t_1 , t_2), y = min (t_1 , t_2)

s_3: t_3 = x >> 3, t_4 = y >>1

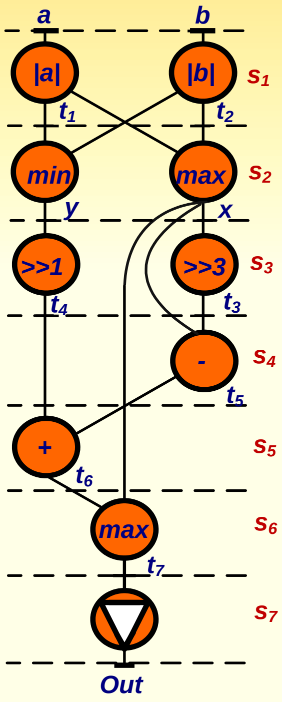
s_4: t_5 = x – t_3

s_5: t_6 = t_4 + t_5

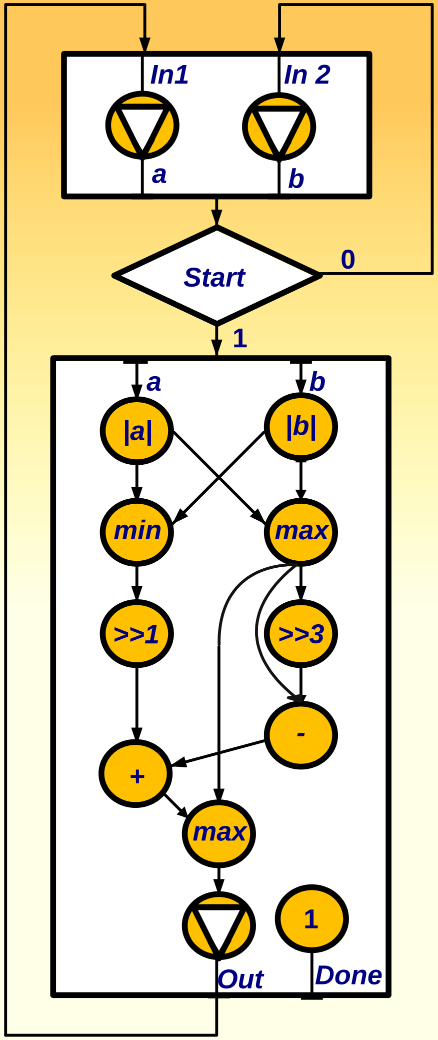
s_6: t_7 = max (t_6 , x)

s_7: Done = 1, Out = t_7

FSMD



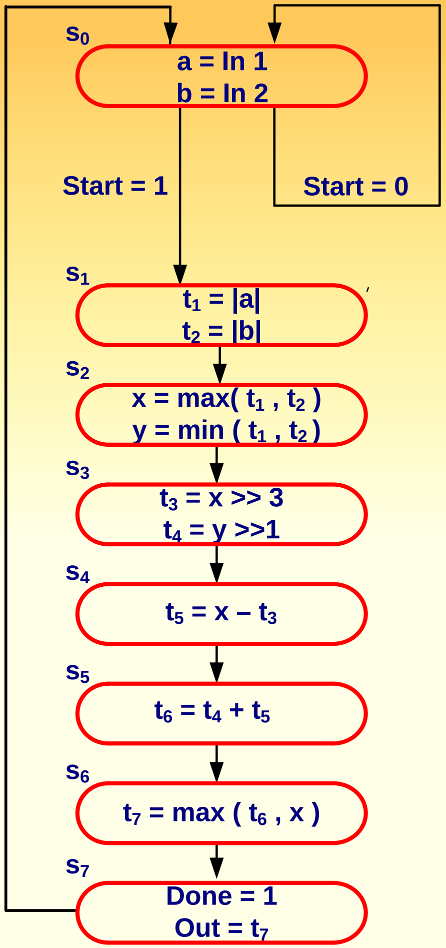
ASAP

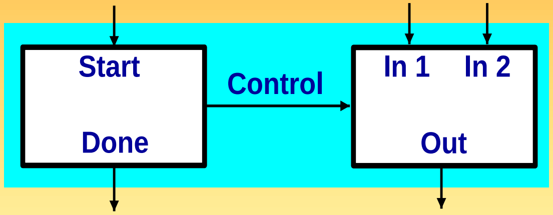


Control/Data flow graph

Square Root Approximation: FSMD Design

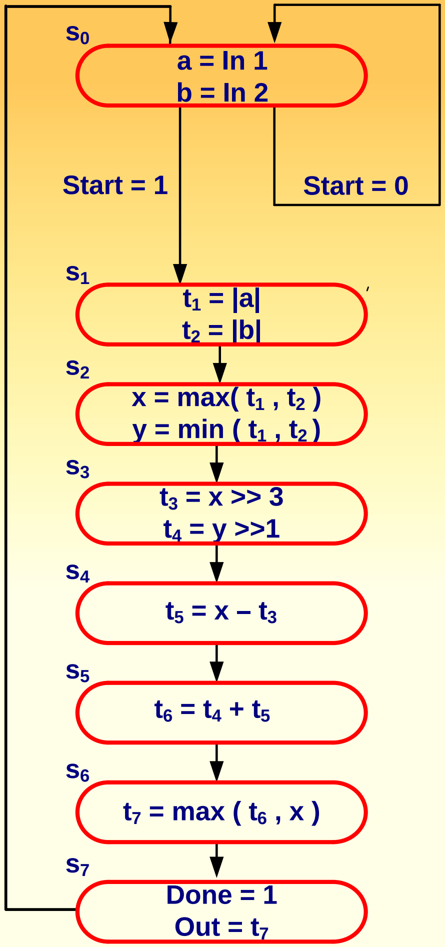
Example: Sq root (a + b) = max(0.875 x + 0.5 y), where x = max(|a|, |b|), y = min (|a|, |b|)





- Storage allocation and sharing
- Functional unit allocation and sharing
- Bus allocation and sharing

Resource usage in SRA



Square-root approximation

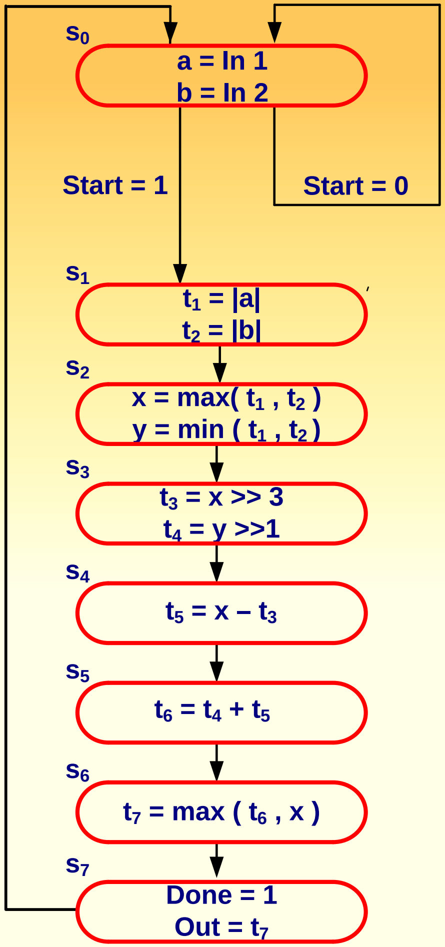
Variable usage

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
a	X						
b	X						
t_1		X					
t_2		X					
x			X	X	X	X	
y			X				
t_3				X			
t_4				X	X		
t_5					X		
t_6						X	
t_7							X
No. of live variables	2	2	2	3	3	2	1

Operation usage

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	Max. no. of units
abs	2							2
min		1						1
max		1				1		1
>>			2					2
-				1				1
+					1			1
No. of operations	2	2	2	1	1	1		

Resource usage in SRA



s_0: a = In 1, b = In 2

Start = 1

Start = 0

s_1: t_1 = |a|, t_2 = |b|

s_2: x = max(t_1 , t_2), y = min (t_1 , t_2)

s_3: t_3 = x >> 3, t_4 = y >>1

s_4: t_5 = x – t_3

s_5: t_6 = t_4 + t_5

s_6: t_7 = max (t_6 , x)

s_7: Done = 1, Out = t_7

Square-root approximation

Connectivity usage

	a	b	t_1	t_2	x	y	t_3	t_4	t_5	t_6	t_7
abs1	i		o								
abs2		i		o							
min			i	i	o						
max			i	i	i	o				i	o
>>3					i		o				
>>1						i		o			
-					i		i		o		
+								i	i	o	

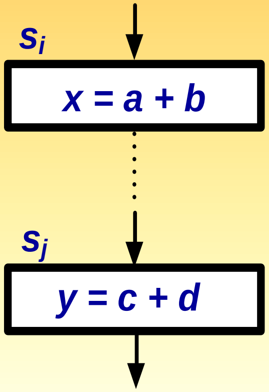
Operation usage

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	Max. no. of units
abs	2							2
min		1						1
max		1				1		1
>>			2					2
-				1				1
+					1			1
No. of operations	2	2	2	1	1	1		

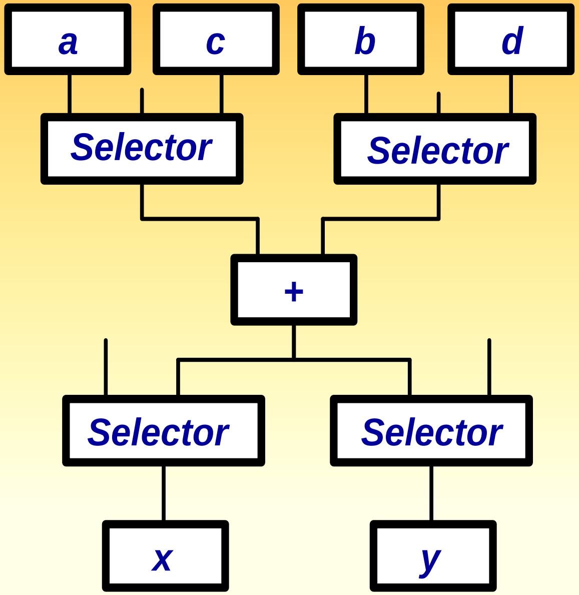
Register sharing (Variable merging)

- Group variables with non-overlaping lifetimes
- Each group shares one register
- Grouping reduces number of registers needed in the design
- There are many partitioning algorithms

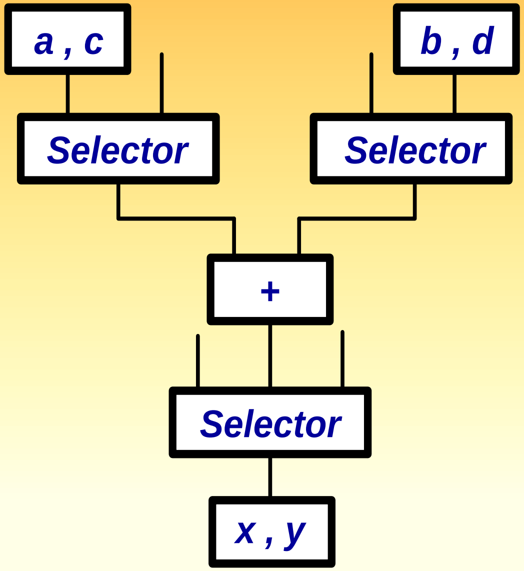
Merging variables with common sources and destination



FSMD

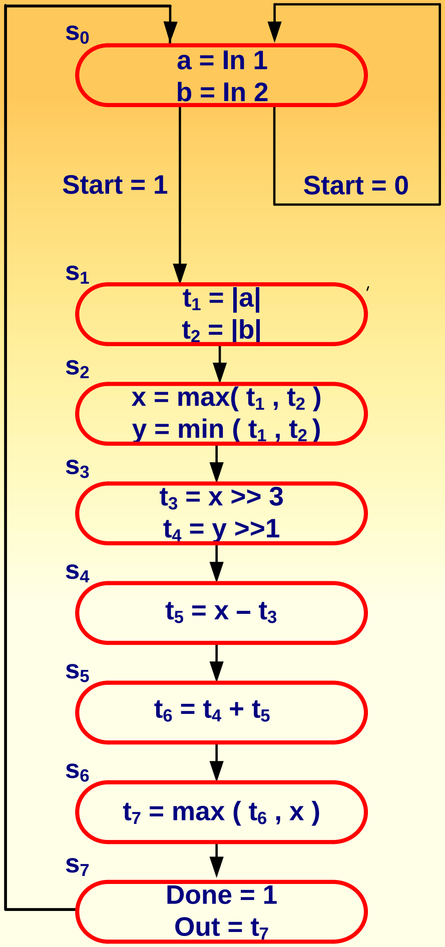


Datapath without register sharing



Datapath with register sharing

Register sharing (Variable merging)



s_0

a = In 1

b = In 2

Start = 1

Start = 0

s_1

$t_1 = |a|$

$t_2 = |b|$

s_2

$x = \max(t_1, t_2)$

$y = \min(t_1, t_2)$

s_3

$t_3 = x >> 3$

$t_4 = y >> 1$

s_4

$t_5 = x - t_3$

s_5

$t_6 = t_4 + t_5$

s_6

$t_7 = \max(t_6, x)$

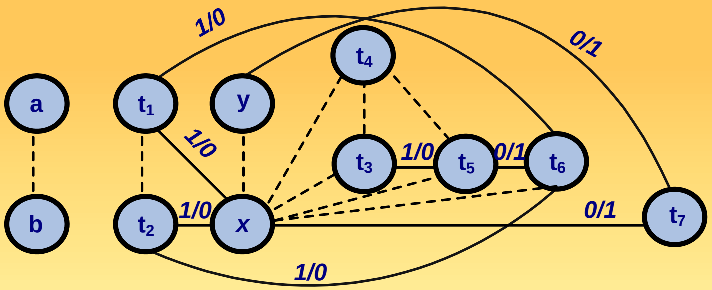
s_7

Done = 1

Out = t_7

Square-root approximation

Compatibility graph

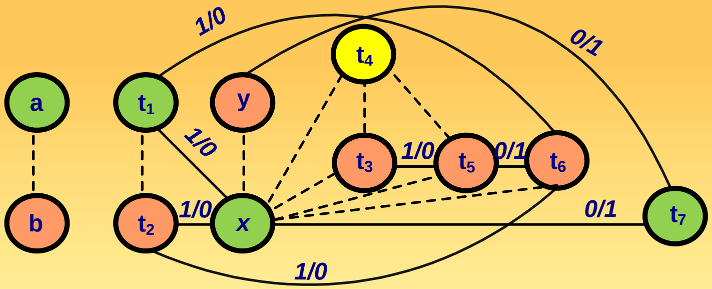


	s_1	s_2	s_3	s_4	s_5	s_6	s_7
a	X						
b	X						
t_1		X					
t_2		X					
x			X	X	X	X	
y			X				
t_3				X			
t_4				X	X		
t_5					X		
t_6						X	
t_7							X
No. of live variables	2	2	2	3	3	2	1

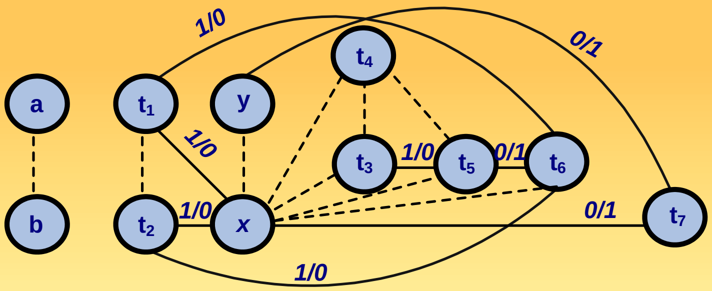
Variable usage

Register sharing (Variable merging)

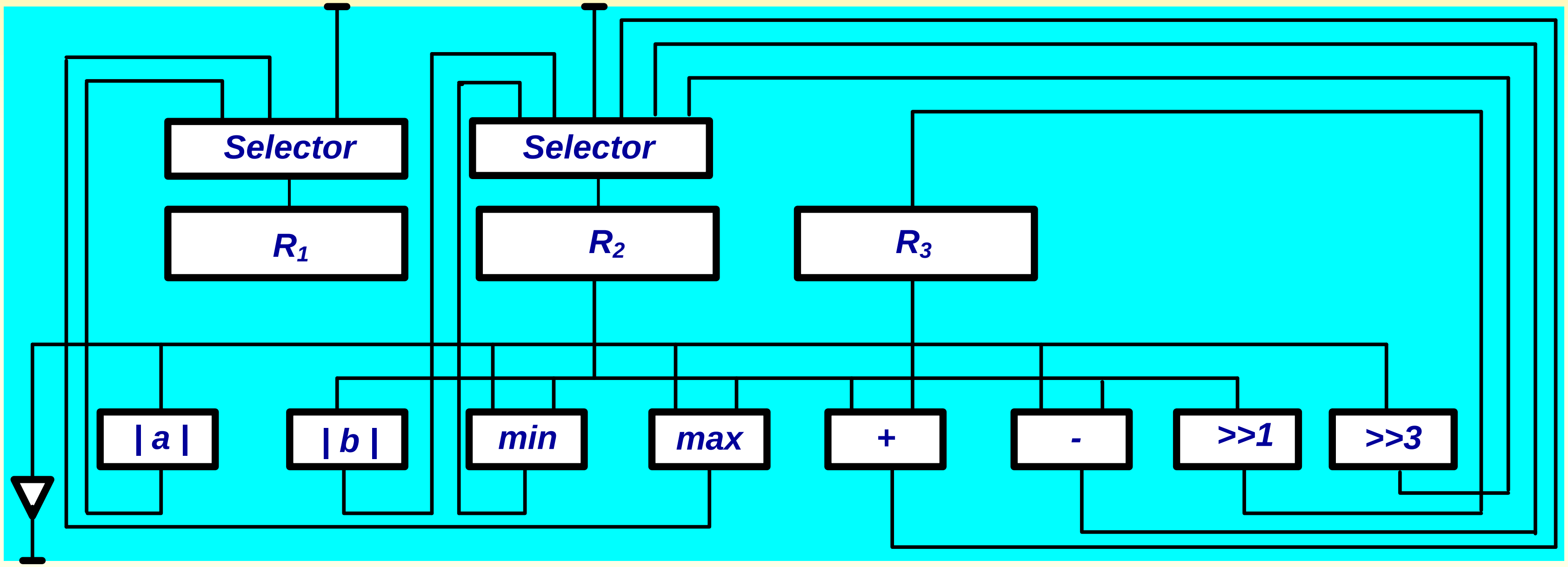
Partitioned compatibility graph



Compatibility graph



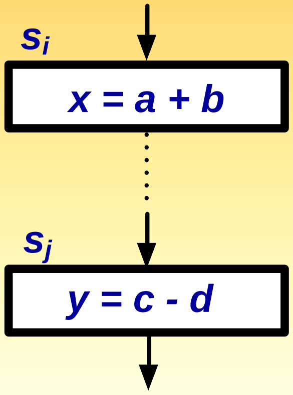
R1 = [a , t1 , x , t7]   R2 = [b , t2 , y , t3 , t5 , t6]   R3 = [t4]



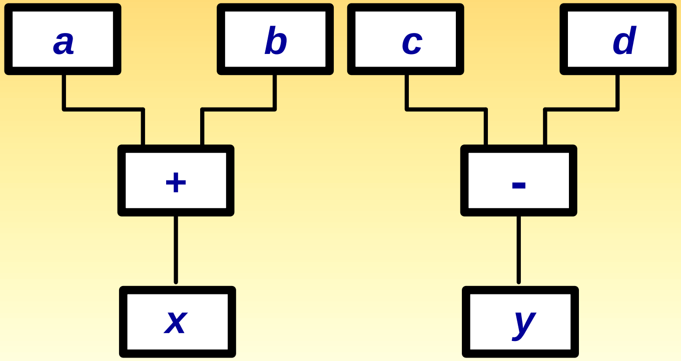
FU sharing (Operator merging)

- Group non-concurrent operations
- Each group shares one functional unit
- Sharing reduces number of functional units
- Grouping also reduces connectivity
- Clustering algorithms are used for grouping

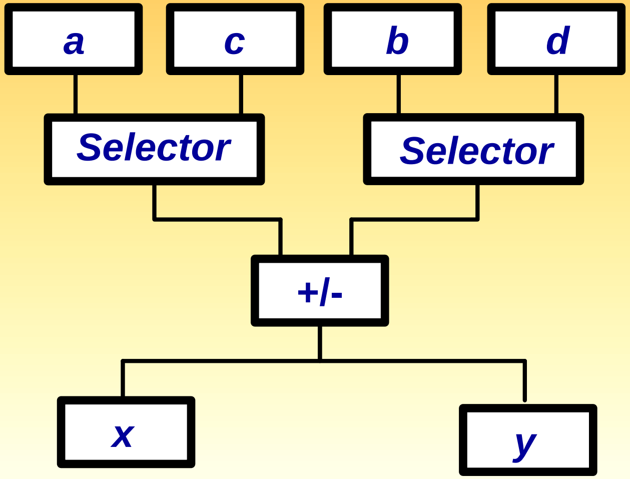
FU-sharing motivation



Partial FSMD

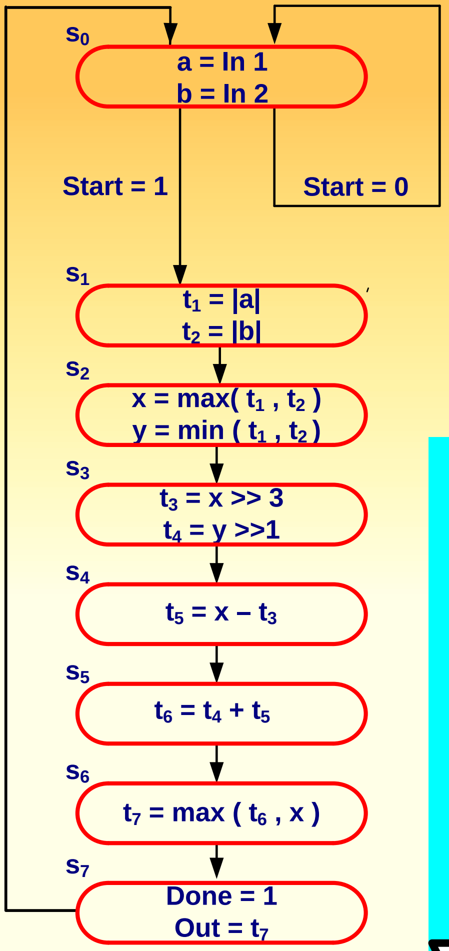


Non-shared design

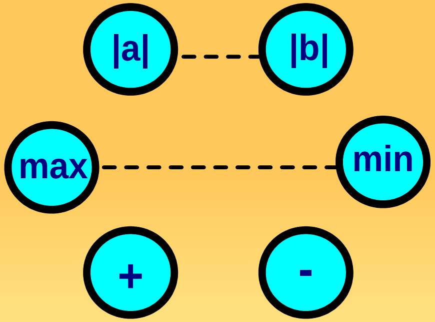


Shared design

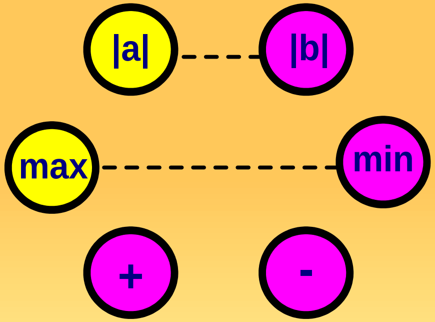
Operator-merging for SRA



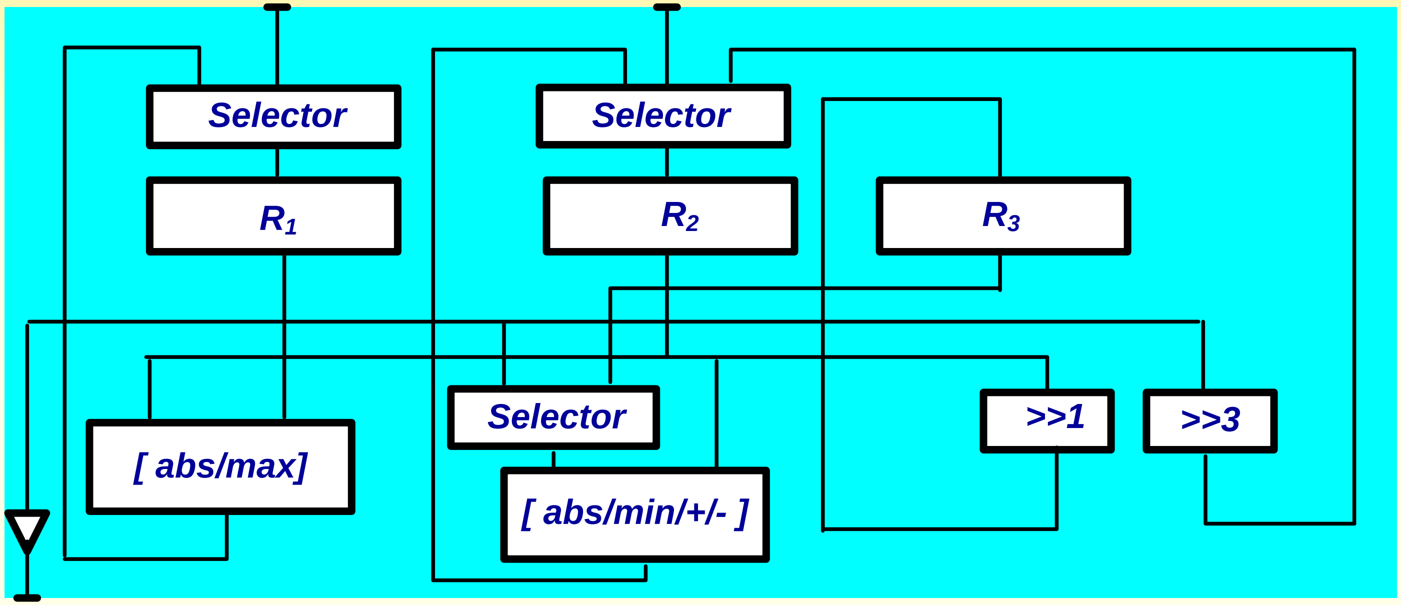
Square-root approximation



Compatibility graph



Partitioned compatibility graph

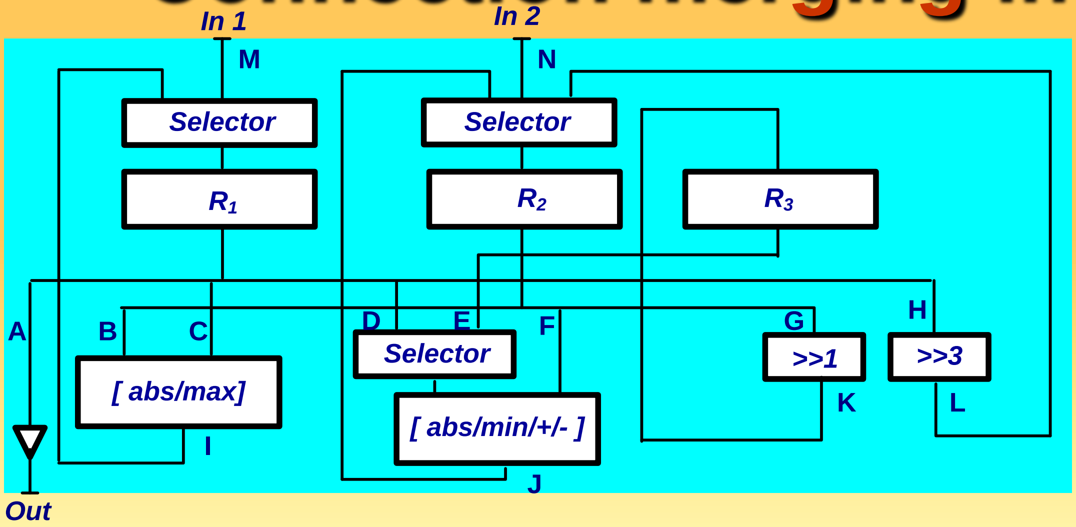


Datapath after variable and operator merging

Bus sharing (connection merging)

- Group connections that are not used concurrently
- Each group forms a bus
- Connection merging reduces number of wires
- Clustering algorithm work well

Connection merging in SRA datapath



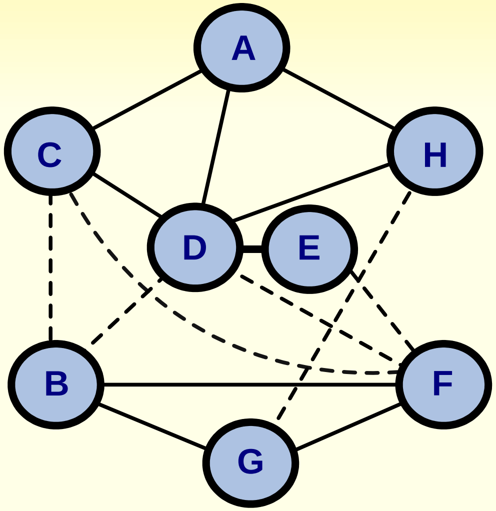
Datapath after variable and operator merging

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

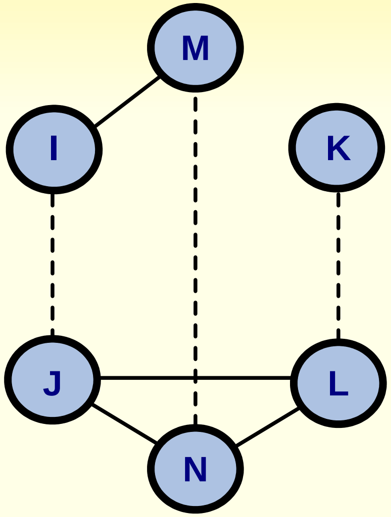
Bus assignment

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
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							

Connectivity usage table

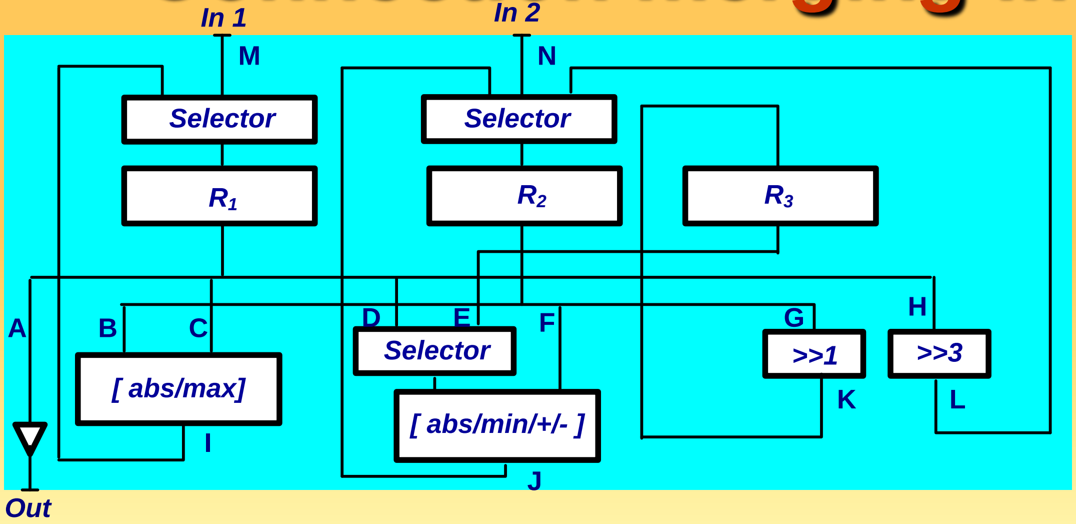


Compatibility graph for input buses



Compatibility graph for output buses

Connection merging in SRA datapath



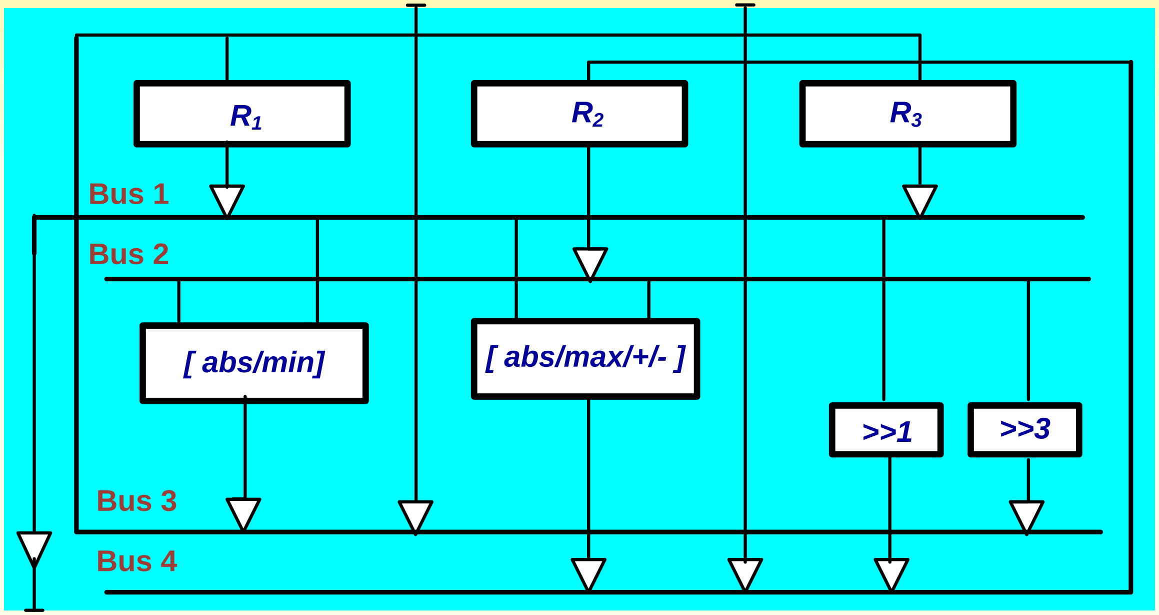
Datapath after variable and operator merging

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

Bus assignment

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
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							

Connectivity usage table

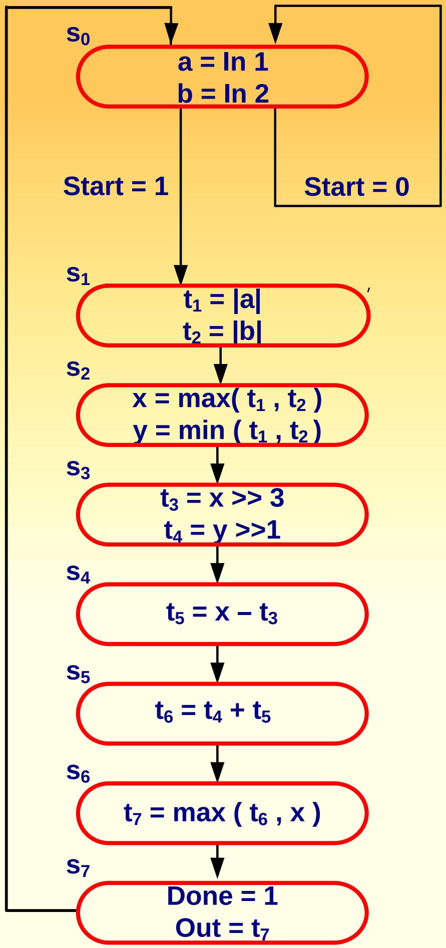


Datapath after variable, operator and connectivity merging

Register merging into Register files

- Group register with non-overlapping accesses
- Each group assigned to one register file
- Register grouping reduces number of ports, and therefore number of buses
- Use some clustering algorithms

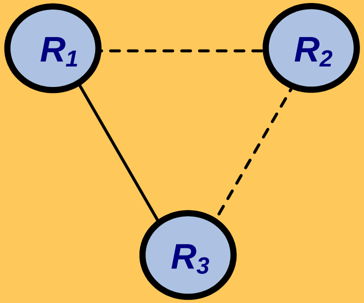
Register merging



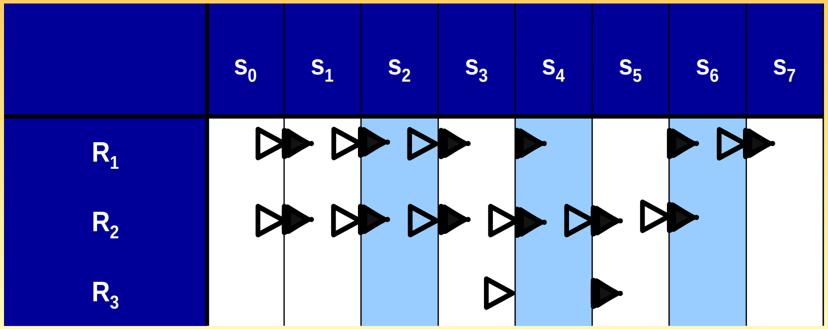
Square-root approximation

- $R_1 = [\ a,\ t_1,\ x,\ t_7\ ]$
- $R_2 = [\ b,\ t_2,\ y,\ t_3,\ t_5,\ t_6\ ]$
- $R_3 = [\ t_4\ ]$

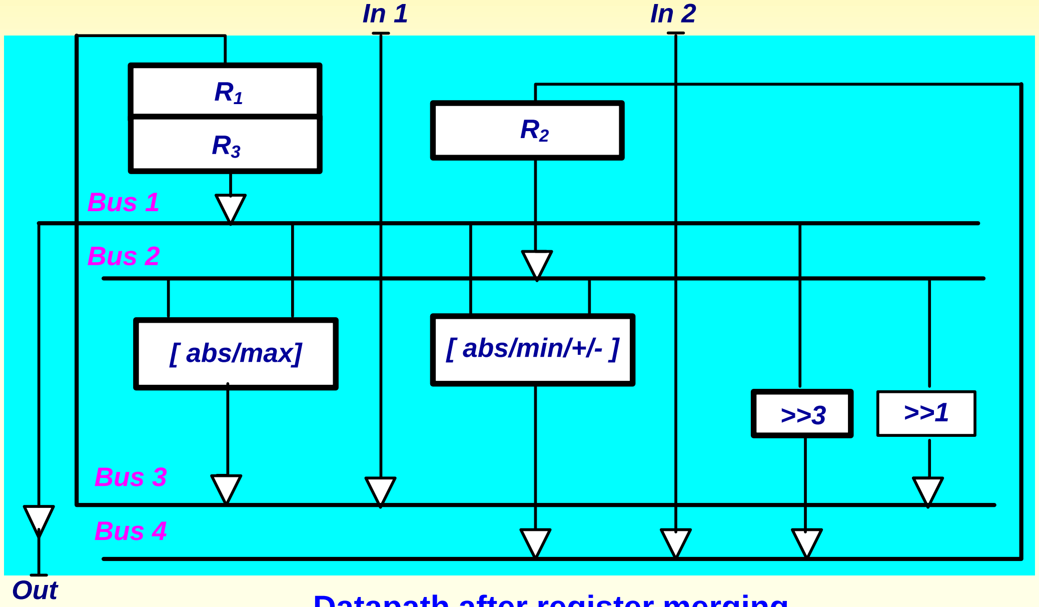
Register assignment



Compatibility graph


s0
s1
s2
s3
s4
s5
s6
s7
R1
R2
R3

Register access table

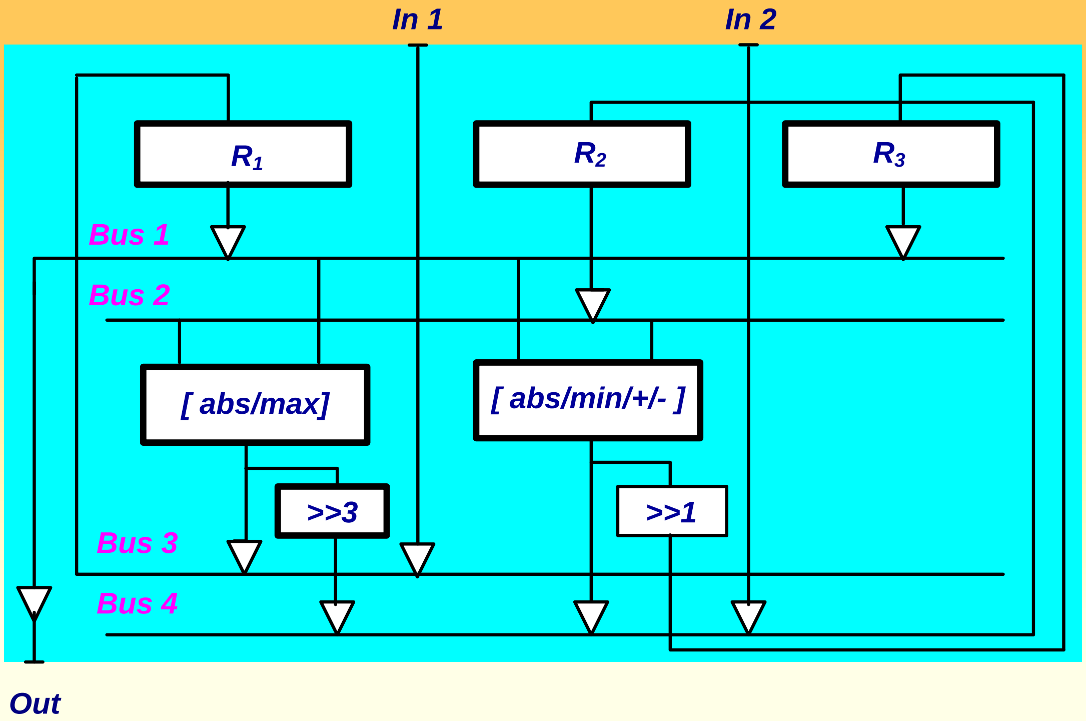


Datapath after register merging

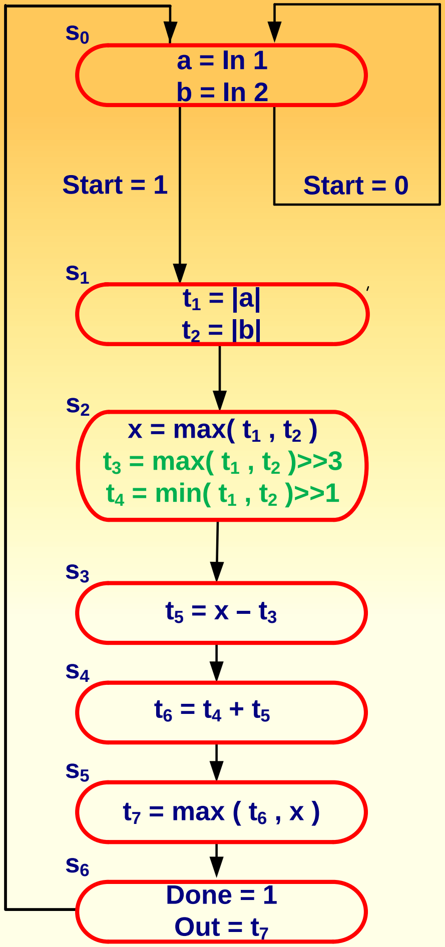
Chaining and multi-cycling

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

SRA datapath with chained units



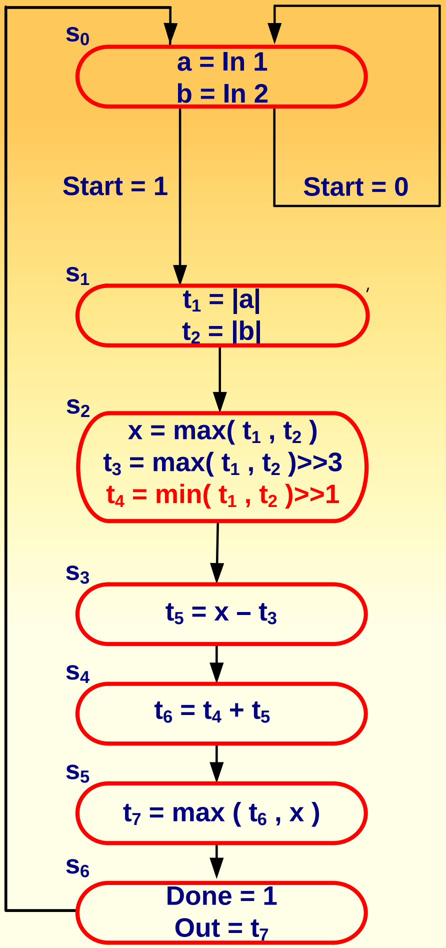
Datapath schematic



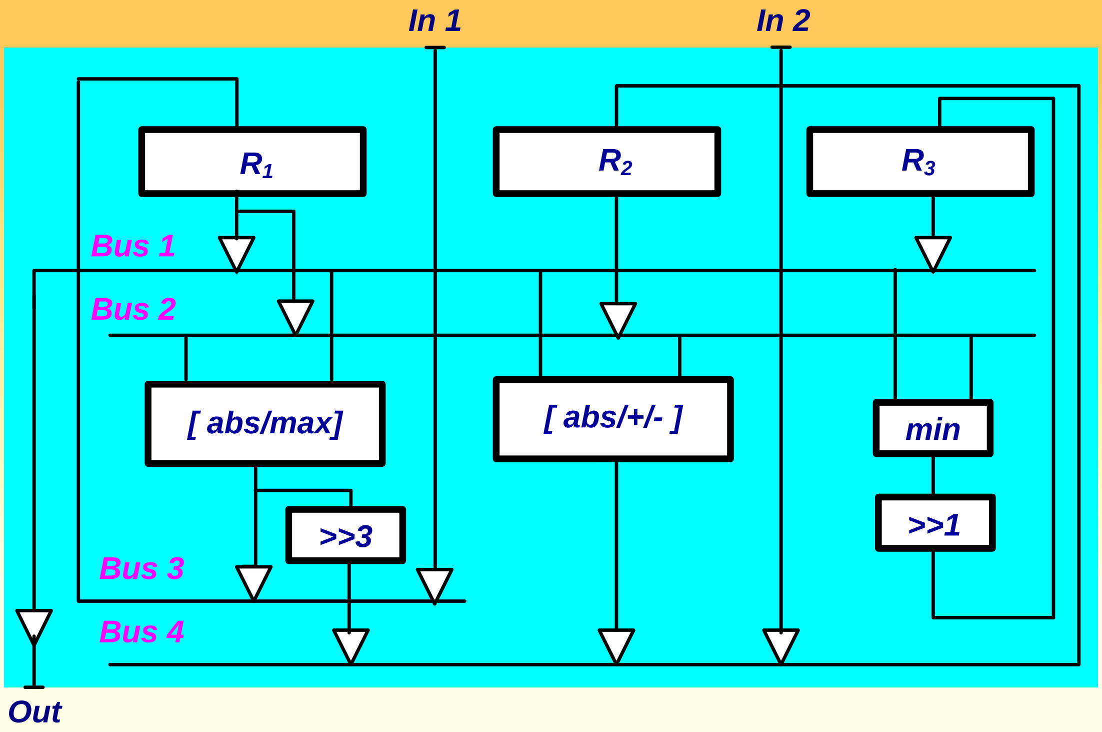
Square-root approximation

- $R_1 = [\ a, t_1, x, t_7\ ]$
- $R_2 = [\ b, t_2, y, t_3, t_5, t_6\ ]$
- $R_3 = [\ t_4\ ]$

SRA datapath with multi-cycle units



Square-root approximation



Datapath schematic

- $R_1 = [\ a, t_1, x, t_7\ ]$
- $R_2 = [\ b, t_2, y, t_3, t_5, t_6\ ]$
- $R_3 = [\ t_4\ ]$

Pipelining

- Pipelining improves performance at a very small additional cost

- Pipelining divides design into stages and uses all stages concurrently for different data (assembly line principle)

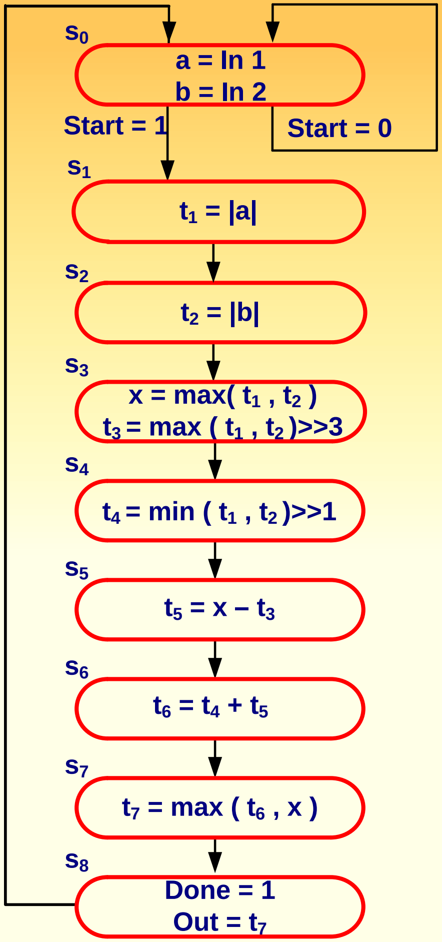
- Pipelining principles works on several levels:

(a) Unit pipelining

(b) Control pipelining

(c) Datapath pipelining

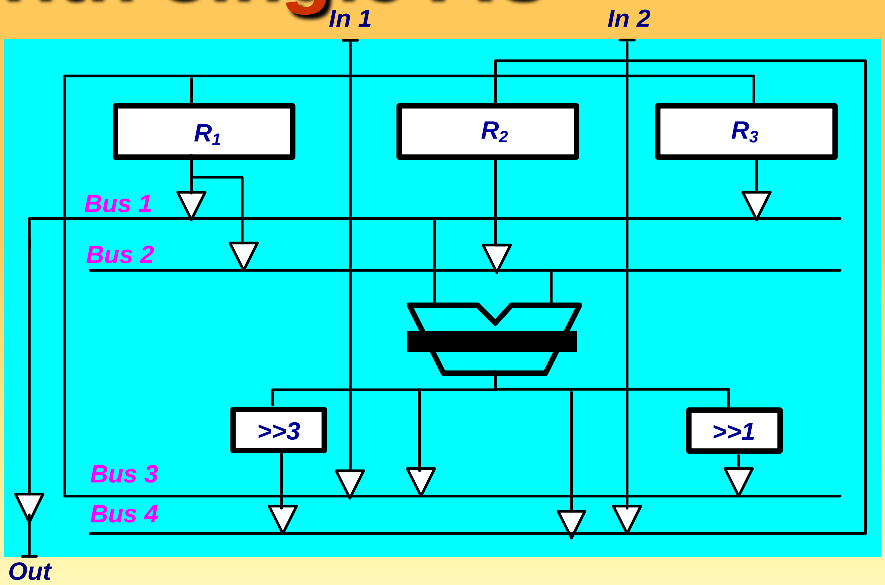
SRA datapath with single AU



Square-root approximation

for single AU

Datapath schematic

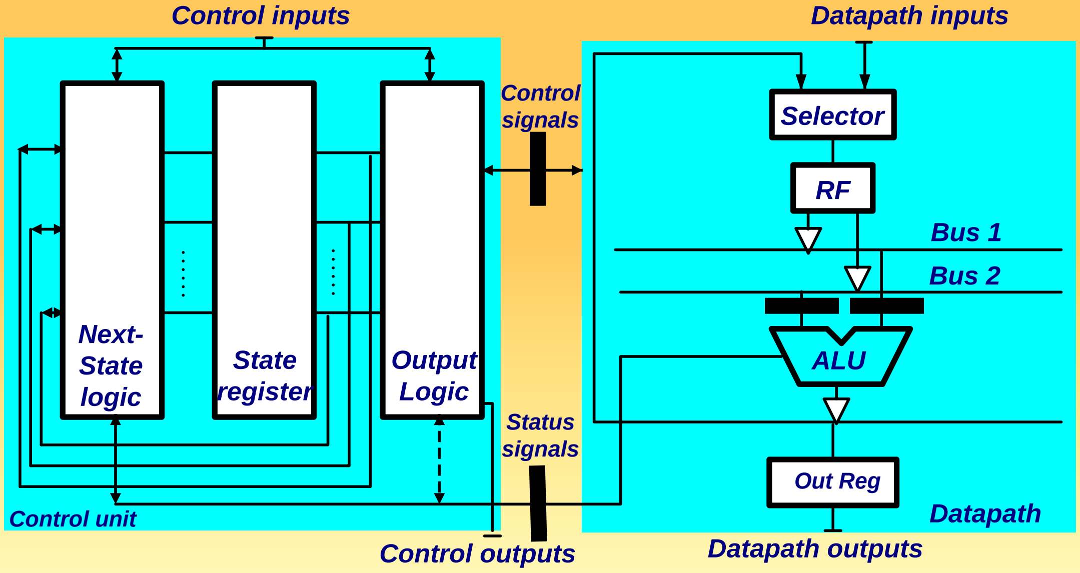


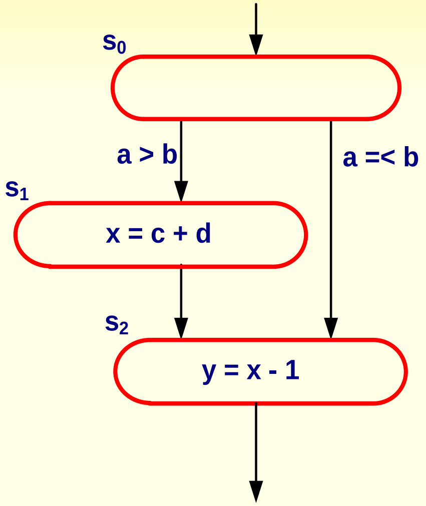
Timing diagram

	s_0	s_1	s_2		s_3	s_4	s_5		s_6		s_7		s_8
Read R_1		a			t_1	t_1	x				x		$t_7 t_7$
Read R_2			b		t_2	t_2	t_3		t_5		t_6		
Read R_3									t_4				
AU stage 1		|a|	|b|		max	min	-		+		max		
AU stage 2			|a|	|b|		max	min	-		+		max	
shifters						>>3	>>1						
Write R_1	a		t_1			x						t_7	
Write R_2	b			t_2		t_3		t_5		t_6			
Write R_3							t_4						
Outport													t_7

Pipelined FSMD implementation

Standard FSMD implementation





Example

Timing diagram

Read SReg	s_0				s_1		s_2				
Write CReg	s_0				s_1		s_2				x
Read CReg		s_0				s_1		s_2			t_6
Read RF		a,b				c,d		x			
Write ALUIn		a,b				c,d		x			
Read ALUIn			a,b				c,d		x		
ALU			a>b				c+d		x-1		
Write RF							x		y		
Write Status			a>b								
Read Status				a>b							
Write SR				s_1/s_2		s_2					

Summary

We introduced RTL design:

- FSMD model
- RTL specification with
  - FSMD
  - CDFG
- Procedure for synthesis from RTL specification
- Scheduling of basic blocks
- Design Optimization through
  - Register sharing
  - Functional unit sharing
  - Bus sharing
  - Unit chaining
  - Multi-clocking
- Design Pipelining
  - Unit pipelining
  - Control pipelining
  - Datapath pipelining