COMPSYS701:Advanced Digital Design

Advanced Register Transfer Level Design

Zoran Salcic

Department of Electrical, Computer, and
Software Engineering

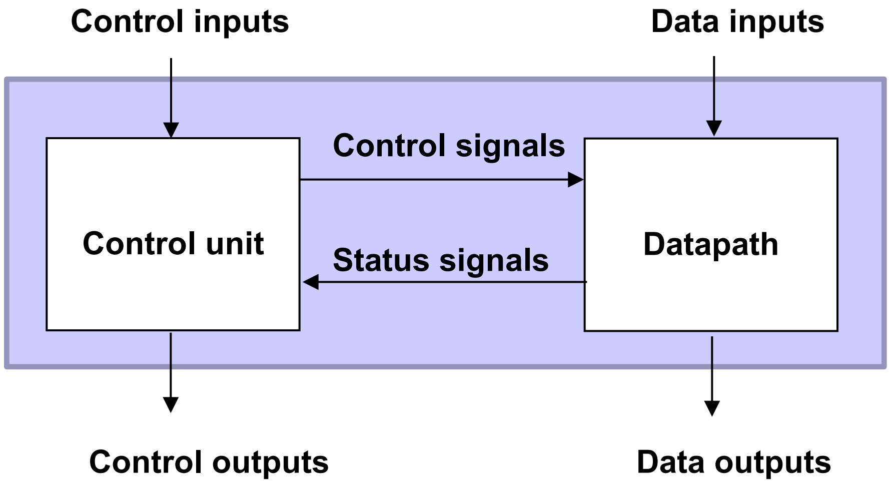
Semester 1 2024, University of Auckland

Outline

- Register transfer level (RTL) design model
- Finite state machines with datapath (FSMD)
- Algorithmic state machine (ASM) charts as graphical presentation of FSMDs
- Synthesis from ASMs
- Variable sharing
- Operation sharing
- Chaining and multicycling
- Pipelining (operation, datapath, control unit)

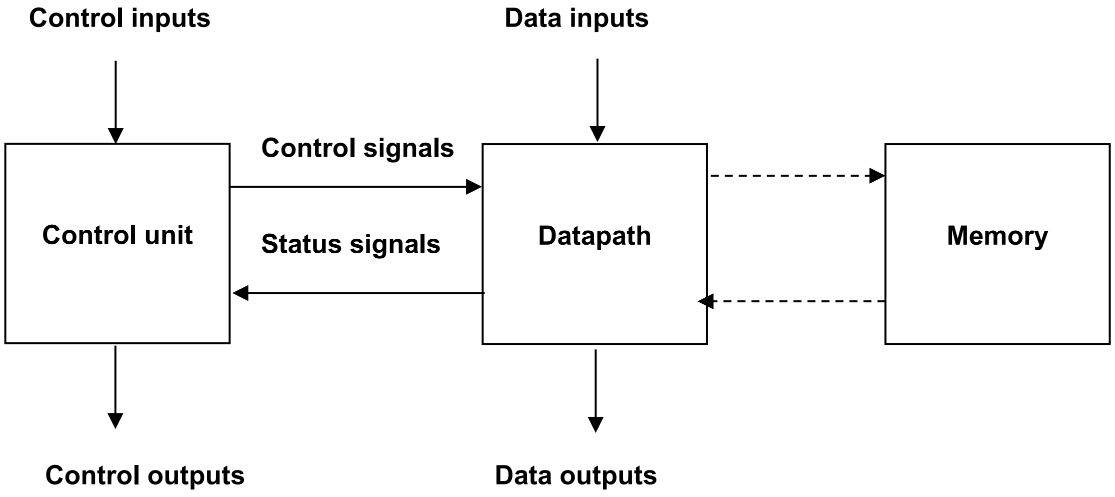
Register Transfer Level (RTL) design model

- The general design model for RTL design consists of control unit and datapath
- Two types of I/O ports:
- data ports (inputs and outputs) to exchange data with the outside environment
- control ports to control the operations performed by the datapath and receive information about the status of selected registers in the datapath



RTL design model - memories

- The datapath often receives operands from memory and writes results of operations to memory
- Often cascaded – almost never nested



Register Transfer Level (RTL) design model

- Time divided into intervals called control states or steps
- Register transfer description used to specify for each control state
    - the conditions to be tested
    - all register transfers to be executed and
    - next control state to be entered

RTL design model - summary

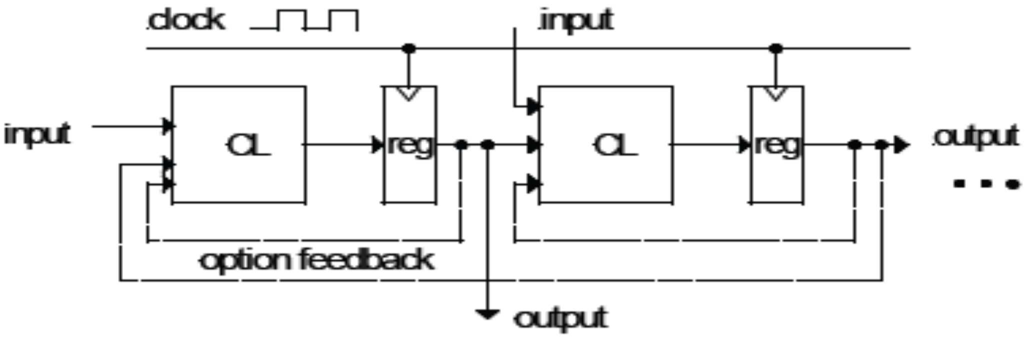
- The datapath takes the operands from storage units, performs computation in the combinatorial units, and returns the results to storage units during each state.
- Control unit controls selection of data sources, operations and data destinations by setting proper values of datapath control signals.
- The datapath indicates that data value has been properly stored in a particular storage unit or when a particular relation between data values in the datapath is satisfied.
- Control unit operates on a set of input control signals: external control inputs and internal status signals
- Control unit produces two types of output signals:
  - external signals used to indicate to the environment that the circuit has reached a certain state or finished a particular operation and
  - datapath control signals that select the operation for each component in the datapath

Datapaths

- Used in all standard processor, ASIC and FPGA implementations to perform complex numerical computation and manipulations
- Consist of temporary storage and processing elements (functional units)
- The variable values and constants are stored in storage components (registers and memories)
- They are fetched from storage components after the active (e.g. rising) edge of the clock signal
- They are transformed in combinational components (processing elements) between two active edges of the clock
- The results are stored back into the storage components at the next active edge of the clock signal

Datapaths

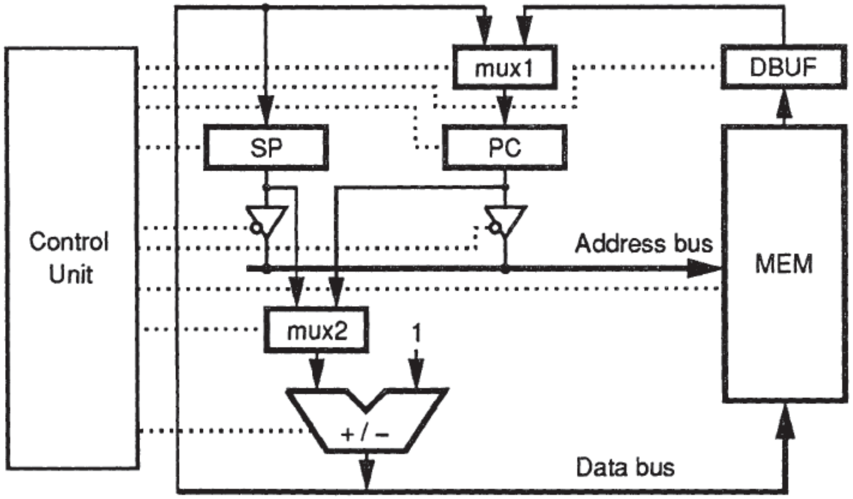
- Data stored in registers and processed by processing elements
- Movement of data – Register Transfer Operations (RTOs)
- RTOs are described by 3 basic components:
    - Set of registers in the system
    - Operations performed on data
    - Control that supervises the sequence of operations



Behavioural vs structural description/design

- Part of the processor design example:
  - Instruction that implements “a conditional jump to subroutine”, if IR(5)= 0, PC is incremented; otherwise (fetches new value of subroutine address MEM[PC] from memory, stores the return address on the stack, up-dates the SP and continues with the execution of the subroutine)
- Synthesis transforms behavioural into structural representation

```
If IR(5)=‘0’ then
   PC:=PC+1;
else
   DBUF :=MEM[PC];
   MEM(SP) :=PC+1;
   SP :=SP-1;
   PC :=DBUF;
end if;
```



(b)

(a) Behavioural and (b) Structural description

Finite State Machine with Datapath (FSMD)

- FSMD can completely specify the behaviour of an arbitrary digital design on an abstract level (formal model)
- We start with the definition of an FSM, which can be defined as a quintuple $< S, I, O ,f, h>$

S is a set of states,

I and O are the sets of input and output symbols, and

f and h are the functions that define the next state and the FSM output:

$f: S \times I \rightarrow S$

$h: S \times I \rightarrow O$ *(Mealy machine);* $h: S \rightarrow O$ *(Moore machine)*

They are often specified by (1) a table in which the next state and output symbols are given for each state and each input symbol or (2) a directed graph (states represented by vertices/nodes and transitions represented by directed arcs/edges)

Finite State Machine with Datapath (FSMD)

- Each state, input and output symbol is defined by a cross-product of Boolean variables:

  I = A1 x A2 x … x Ak

  S = Q1 x Q2 x … x Qm

  O = Y1 x Y2 x … x Yn

  Ai is an input signal, Qi is a state register bit output, and

  Yi is an output signal

- This model can be extended by adding the set of datapath variables, inputs and outputs. The variables set

  V= V1 x V2 x …. X Vq

  defines the state of the datapath by defining the values of all variables in each state.

Finite State Machine with Datapath (FSMD)

- The set of FSMD inputs can be partitioned into a set of FSM control inputs I_C and set of datapath inputs I_D thus

  $I = I_C \times I_D$, where $I_C = A1 \times A2 \times \ldots \times Ak$ and $I_D = B1 \times B2 \times \ldots \times Bp$

- Similarly, the output set is

  $O = O_C \times O_D$, where $O_C = Y1 \times Y2 \times \ldots \times Yn$ *and* $O_D = Z1 \times Z2 \times \ldots \times Zr$

- While *Ai, Qi and Yi* represent Boolean variables, Bi, Vi and Zi represent Boolean vectors, which in turn represent integers, floating-point numbers and characters.

- The size of datapath variables and ports makes specification of functions f and h in tabular form very difficult – in FSMD they are specified with arithmetic expressions.

Finite State Machine with Datapath (FSMD)

- The set of all possible expressions over the set of variables V is

  $EXPR(V) = K\ U\ V\ U\ \{(e_i\ op\ e_j) \mid e_i, e_j\ from\ EXPR(V), op\ is\ an\ operator\}$
  K are all constants of the same type as variables

- Values of the status signals can be defined as

  $STAT = \{statk = e_i\ relop\ e_j \mid e_i, e_j\ from\ EXPR(V), relop\ from\ \{\underline{<}, <, =, /=, \underline{>}, >\}\}$ representing all relations between variables or expressions of variables.

Finite State Machine with Datapath (FSMD)

- Function $f: (S \times V) \times I \rightarrow S \times V$ can be simplified by separating it into two parts, f_C and f_D, where function f_C defines the next state of the control unit

  $f_C: S \times I_C \times STAT \rightarrow S$

  while function f_D defines the values of datapath variables in the next state:

  $f_D: S \times V \times I_D \rightarrow V$

- For each state s_i we compute new value for each variable v_j in the datapath by evaluating e_j which belongs to EXPR(V).
- Function f_D is essentially decomposed into a set of functions f_{Di}, where each f_{Di} assigns one expression e_k to each variable v_j in the datapath state s_i.

Finite State Machine with Datapath (FSMD)

- Similarly, the output function h: S x V x I → O is decomposed into two different functions, h_C and h_D
- They define external control outputs O_C and external datapath outputs O_D, respectively.
- An FSMD can be specified in tabular form (state transition table):

Present state	Next state		Control Output	Datapath output	Datapath variables	
	Input1	Input2…	CO1…	DO1….	Var1	Var2 ….
s_0	s_1	s_0	0	Data1	Temp+Data1	x
s_1	s_2	s_4	1			
.						
.						
.						

Finite State Machine with Datapath (FSMD)

- As practical datapath may store hundreds of variables that seldom change their values, variable assignment statements could be used to represent change of those variables that takes place in a particular state (Similar simplification can be done for the next-state column).
- If variable assignment statements are used for control output and data output ports, they do not retain their values beyond the current state since the values are not stored in registers or memory.
- State action table:

Present state	Next state	Control and datapath actions
	Condition, state	Condition, actions
s_0	Inp = 0, s_0	Out1 = 0, Out2 = 1
s_1	Inp = 1, s_1	Var1 = Inport, Var2 = 0
.	s_2	.
.	.	.
.	.	.
	.	

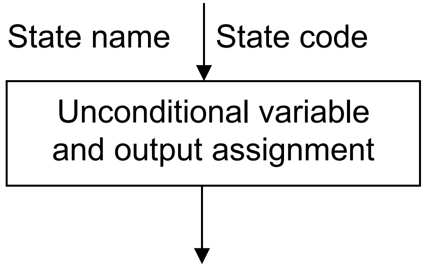
Algorithmic State Machine (ASM) Charts

- Alternative form for specifying FSMDs – equivalent to state-action table
- ASM represents an FSMD using
  - a **state box**, a state name added during state assignment phase
  - a **decision box**, describes the condition under which the FSMD will execute specific action
  - a **conditional output box**, describes variable or output assignments that are executed under conditions specified in one or more decision boxes
  - the **ASM block**, a complex structure that contains one state box and a number of decision and or conditional output boxes

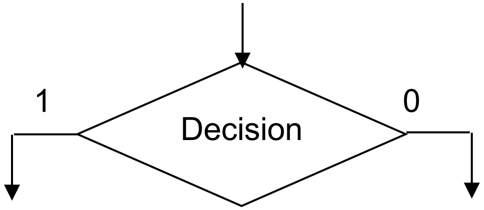
Algorithmic State Machine (ASM) Charts

State box

Moore type

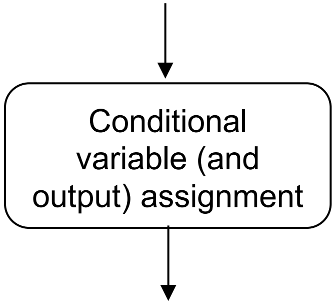


Decision box

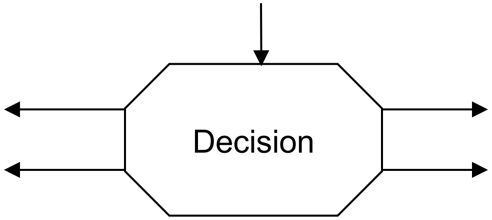


Condition box

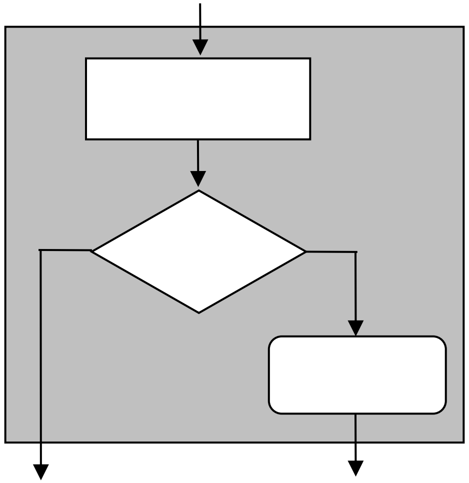
Mealy type



Vector decision box



ASM Block



One state box + all decision boxes and conditional output boxes connected between the state box exit and entry point to the same or other state boxes

Algorithmic State Machine (ASM) Charts

- When specifying an FSMD with a ASM chart, two rules must be satisfied:
    - The chart must define a unique next state for each state and set of conditions
    - Every path defined by a network of condition boxes must lead to another state

- ASM charts can model a state based (Moore) FSMD or an input based (Mealy) FSMD
- State-based model usually has more states than input-based model
- ASM charts easily convert into state-action tables

Example: Square Root Approximation (SRA) unit

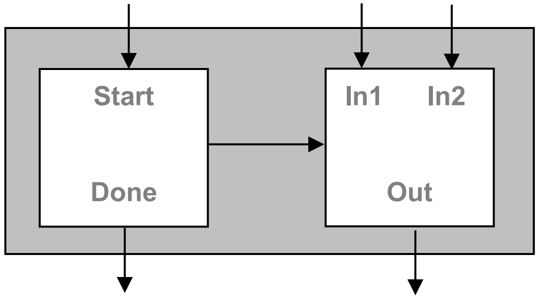
Uses the following formula:

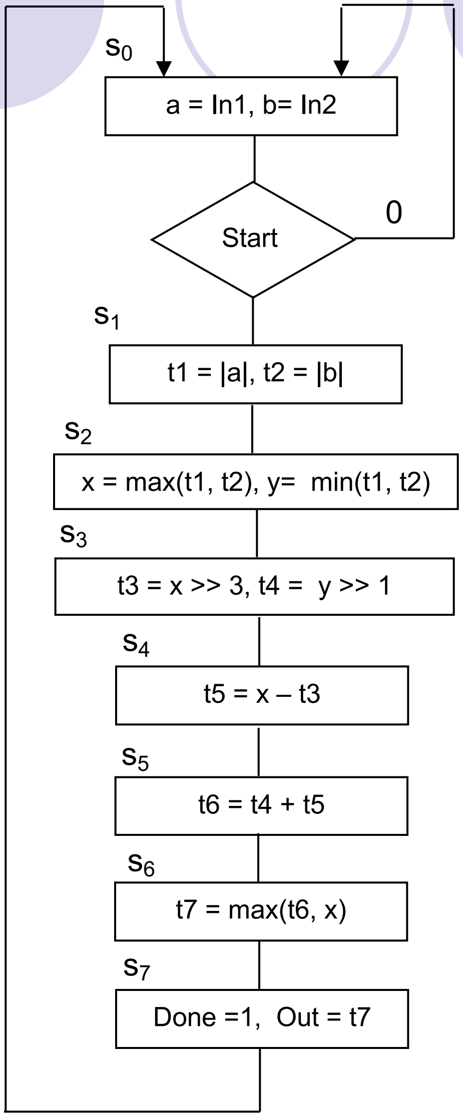
$$sqrt(a^2 + b^2) \sim max((0.875x+0.5y), x)$$

where

$x = max(|a|, |b|)$

$y = min(|a|, |b|).$





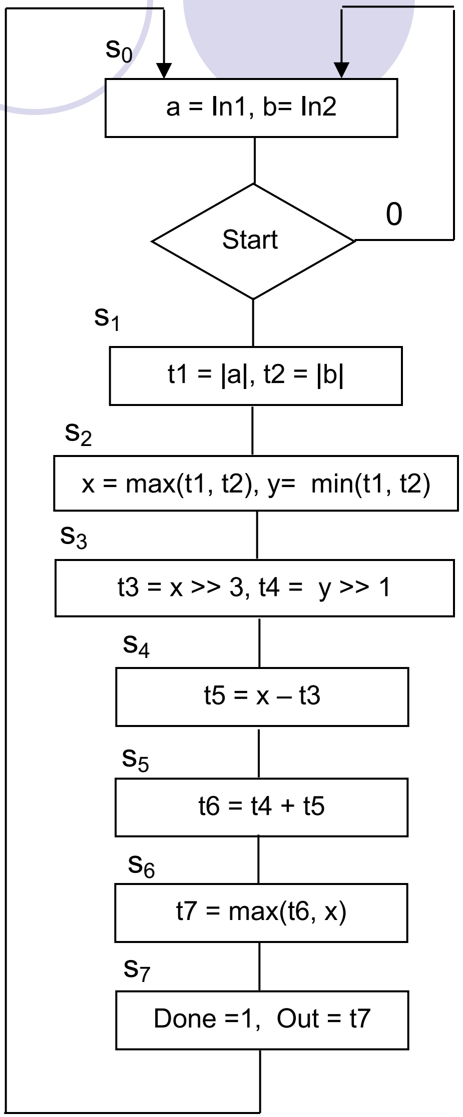
SRA unit – variable usage and reg/memory sharing

- Variable usage and register/memory sharing - variable alive from the first state that follows the rising edge of the clock signal which assigns its new value and in all states in which this new value is used for the last time

State Variable	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
a	x	x						
b	x	x						
t1		x	x					
t2		x	x					
x			x	x	x	x	x	
y			x	x				
t3				x	x			
t4				x	x	x		
t5					x	x		
t6						x	x	
t7							x	x
alive	2	4	4	4	4	4	3	1

Max 4 variables alive in any state

Must be grouped and assigned to registers or memory locations



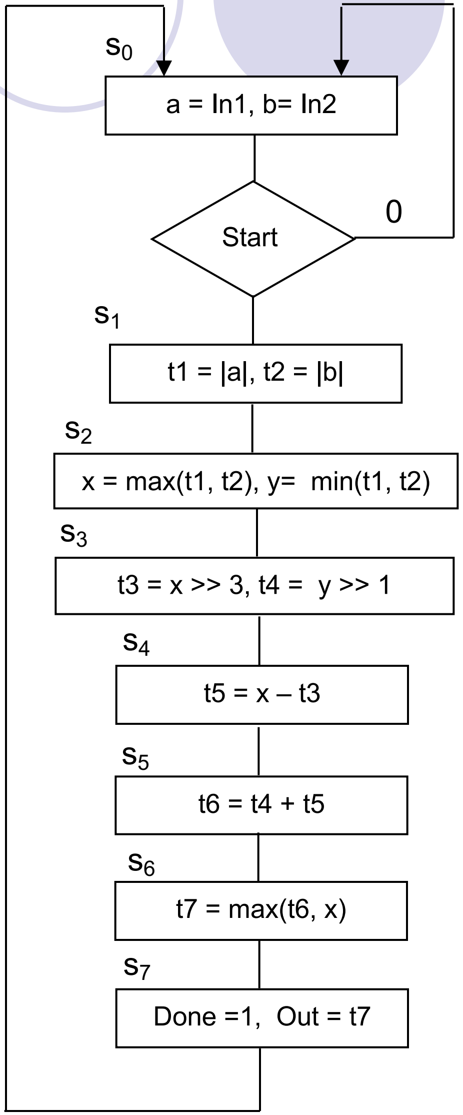
SRA unit – operation usage

State Operation	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		2
>>			2					2
sub				1				1
add					1			1
No of operations	2	2	2	1	1	1		

Straightforward solution: Datapath with 2 abs units, 2 shifters, and one min, max, sub and add unit each

Some units can me merged to make a functional unit for each group, e.g.:

- add/sub
- 3-bit and 1-bit shifter
- min and max unit



SRA unit – minimizing wiring

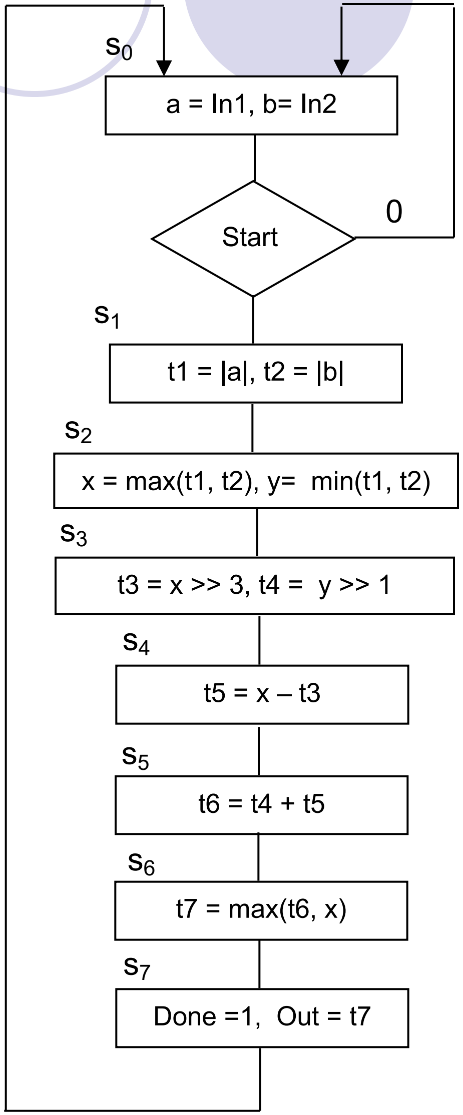
Minimizing wiring – merging connections into buses

	a	b	t1	t2	x	y	t3	t4	t5	t6	t7
abs1	**I**		**O**								
abs2		**I**		**O**							
min			**I**	**I**	**O**						
max			**I**	**I**	**I**	**O**				**I**	**O**
>>3					**I**		**O**				
>>1						**I**		**O**			
sub					**I**		**I**		**O**		
add								**I**	**I**	**O**	

SRA requires 14 input connections and 9 output connections

Maximum of 6 connections needed in state s2

The task of RT synthesis is to group connections and assign one bus to each group in order to minimize connection cost.- this is called **bus sharing**



s_0

a = In1, b= In2

Start

0

s_1

t1 = |a|, t2 = |b|

s_2

x = max(t1, t2), y= min(t1, t2)

s_3

t3 = x >> 3, t4 = y >> 1

s_4

t5 = x – t3

s_5

t6 = t4 + t5

s_6

t7 = max(t6, x)

s_7

Done =1, Out = t7

Register (Variable) Sharing

- A register can be shared only by those variables with non-overlapping lifetimes – the task is to determine the lifetime of each variable.

- Lifetime: includes the state in which it is assigned a new value (write state), all states in which it is used (read states), and all states on the path between the write state and last read state.

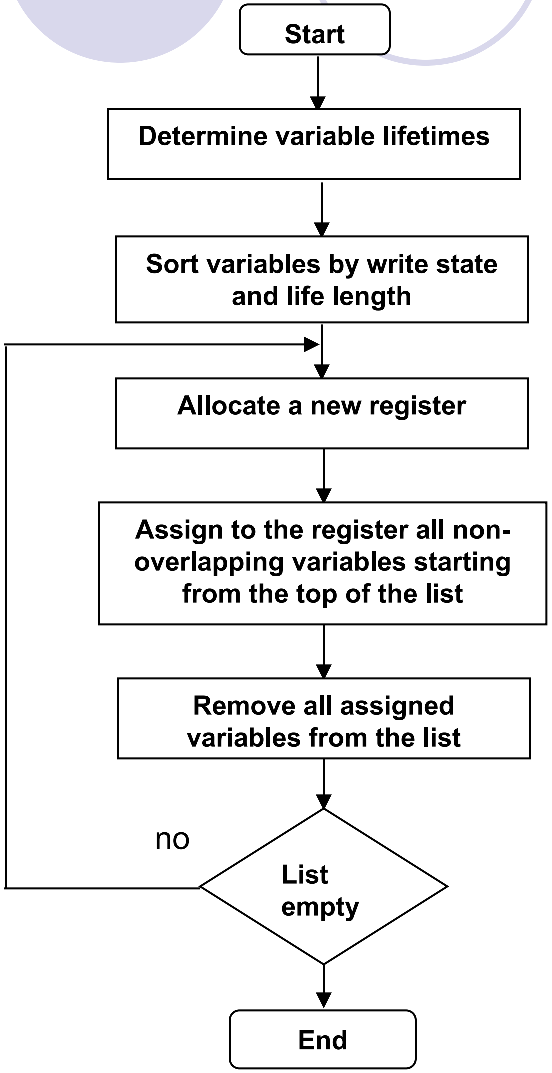
- Once the lifetime of each variable has been determined, we can group variables that have non-overlapping lifetimes and assign each group to a single register/memory.

- The goal is to do it with as few registers as possible – partition variables into the smallest number of groups while ensuring that every variable belongs to only one of these groups

Register (Variable) Sharing

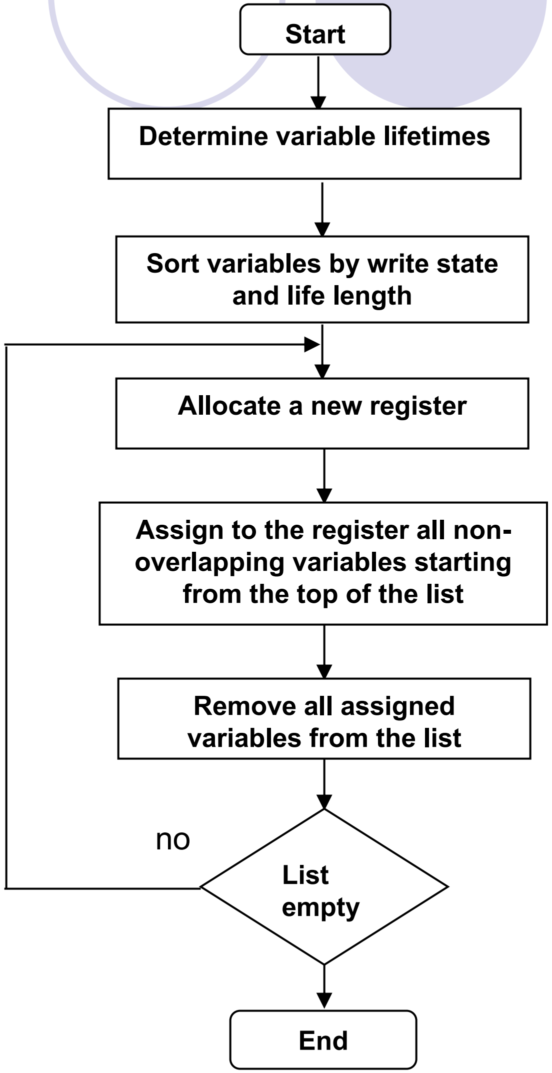
Left-edge algorithm

If two variables have the same write state, the priority is given to the variable with the longer lifetime


Start
Determine variable lifetimes
Sort variables by write state and life length
Allocate a new register
Assign to the register all non-overlapping variables starting from the top of the list
Remove all assigned variables from the list
no
List empty
End

Register Sharing – RSA Example

state	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
variable								
a	x	x						
b	x	x						
t1		x	x					
t2		x	x					
x			x	x	x	x	x	
y			x	x				
t3				x	x			
t4				x	x	x		
t5					x	x		
t6						x	x	
t7							x	x

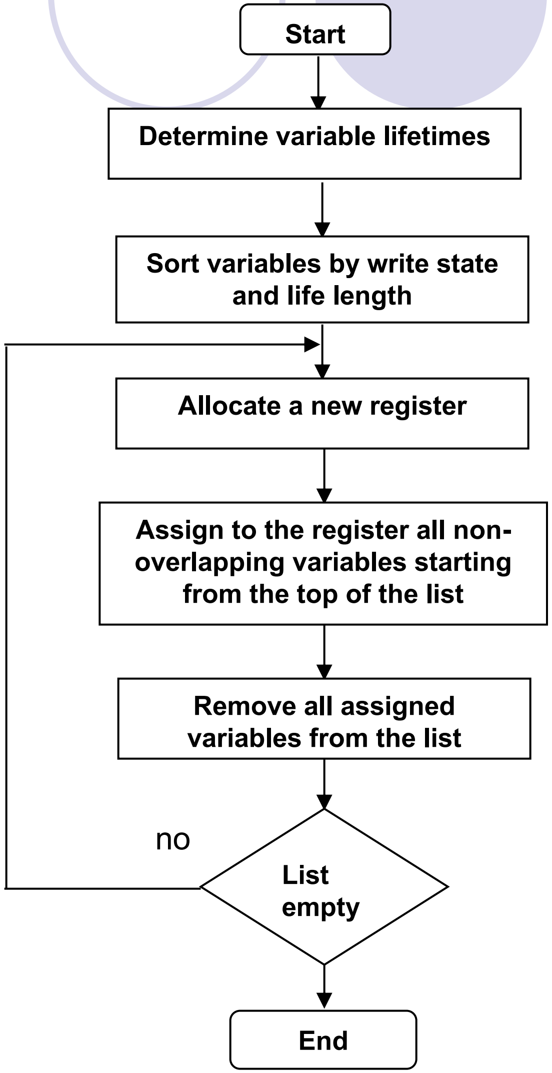

Start
Determine variable lifetimes
Sort variables by write state and life length
Allocate a new register
Assign to the register all non-overlapping variables starting from the top of the list
Remove all assigned variables from the list
no
List empty
End

Register assignments:

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

Register Sharing – RSA Example

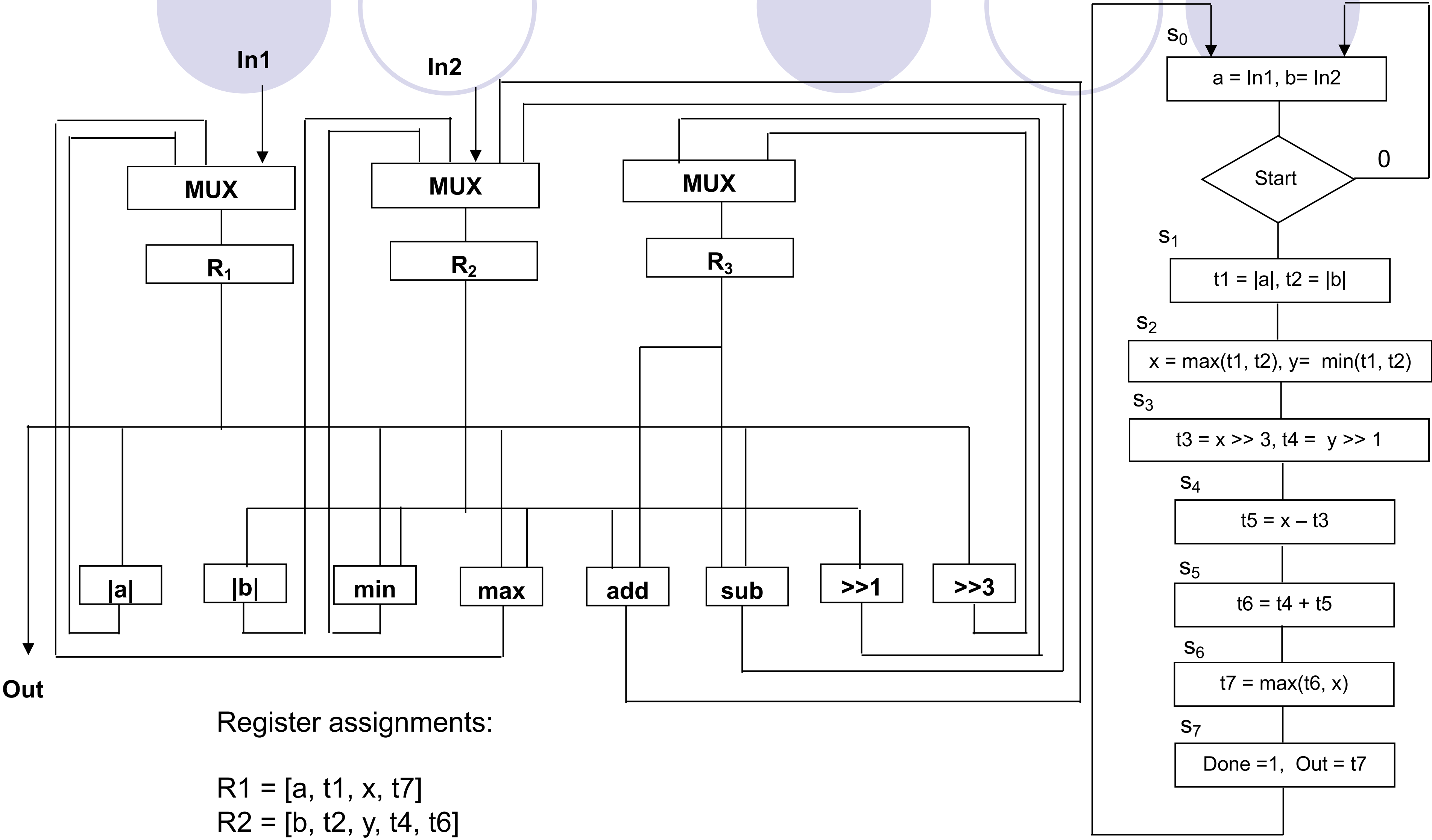
state	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
variable								
a	x	x						
b	x	x						
t1		x	x					
t2		x	x					
x			x	x	x	x	x	
y			x	x				
t3				x	x			
t4				x	x	x		
t5					x	x		
t6						x	x	
t7							x	x


Start
Determine variable lifetimes
Sort variables by write state and life length
Allocate a new register
Assign to the register all non-overlapping variables starting from the top of the list
Remove all assigned variables from the list
no
List empty
End

Register assignments:

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

Register Sharing – RSA Example



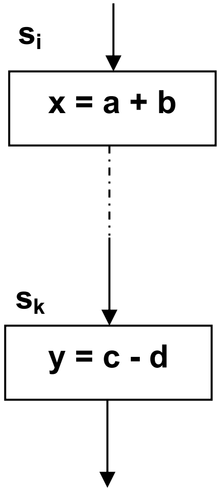
Register assignments:

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

Operation Sharing

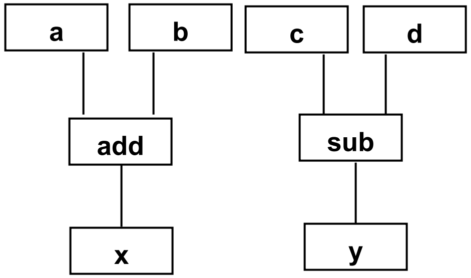
The goal – to minimize the number of functional units in a datapath

- It is possible because within any given state, the datapath will not perform all operations
- Operations can be grouped, although every grouping does not lead to reduction of the cost

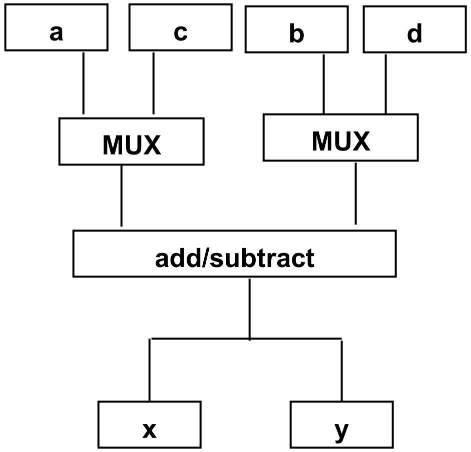


Functional units and multiplexers should be grouped into multi-functional units if the cost of their grouping is lower.

For this purpose we can use the graph-partitioning algorithm.



Non-shared design



Shared design

Register Merging

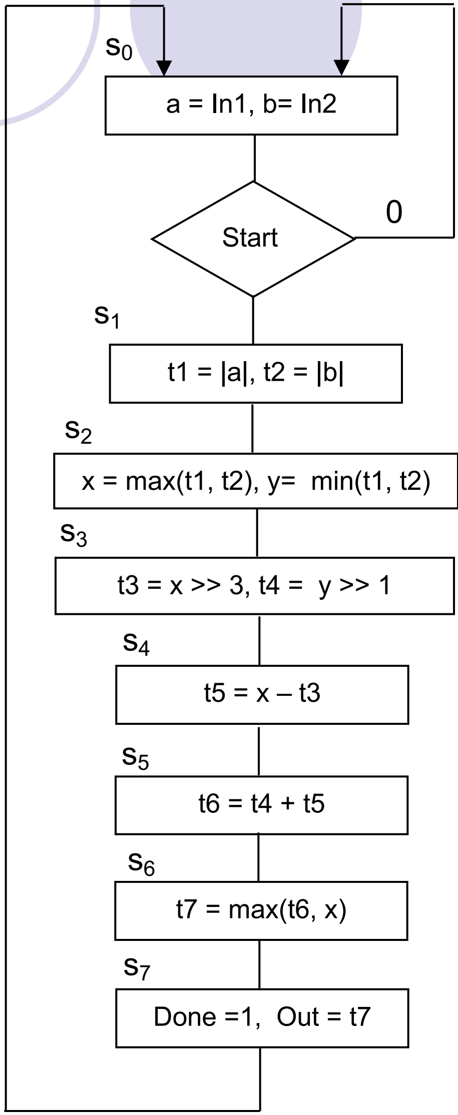
- Registers with non-overlapping access times can be merged into register files to share register input and output ports
  - This increases register-to-register delay (address decoding logic)
  - The same approach as for variable, operator and connection sharing can be used.
  - Initially we create a register access table from which we can generate compatibility graph.
  - Graph-partitioning algorithm can be used to group compatible registers into register files.

Register Merging

- Register access table
- Dividing line between the states represents the rising edge of the clock which loads data into registers
  - W - data will be written to the register at that particular edge
  - R – data will be used in that particular state

	s_0		s_1		s_2		s_3		s_4		s_5		s_6		s_7	
R_1		W	R	W	R	W	R		R				R	W	R	
R_2		W	R	W	R	W	R	W			R	W	R			
R_3								W	R	W	R					

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



s_0

a = In1, b= In2

Start

0

s_1

t1 = |a|, t2 = |b|

s_2

x = max(t1, t2), y= min(t1, t2)

s_3

t3 = x >> 3, t4 = y >> 1

s_4

t5 = x – t3

s_5

t6 = t4 + t5

s_6

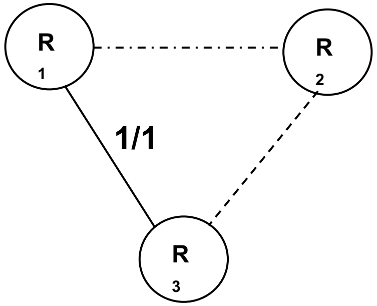
t7 = max(t6, x)

s_7

Done =1, Out = t7

Register Merging

	s_0		s_1		s_2		s_3		s_4		s_5		s_6		s_7	
R_1		W	R	W	R	W	R		R				R	W	R	
R_2		W	R	W	R	W	R	W			R	W	R			
R_3								W	R	W	R					



Compatibility graph

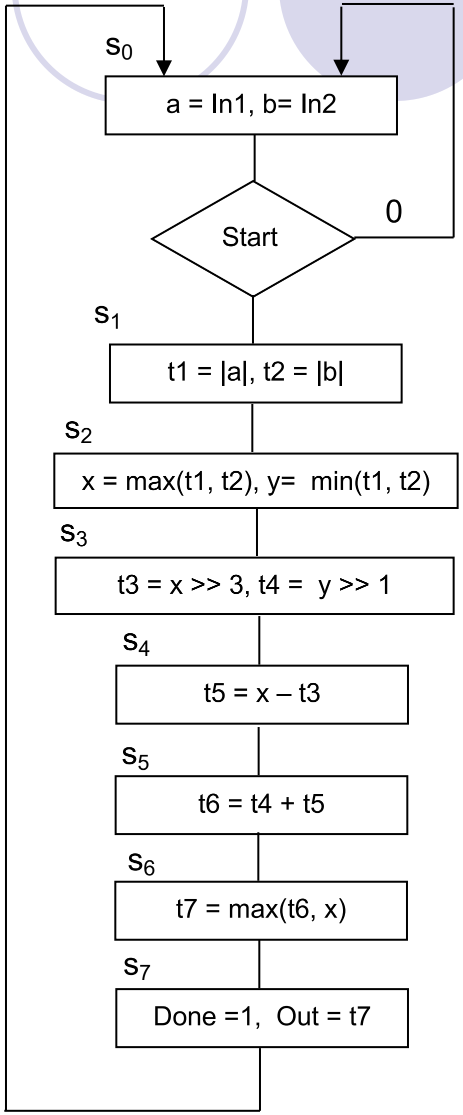
Register assignments:

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

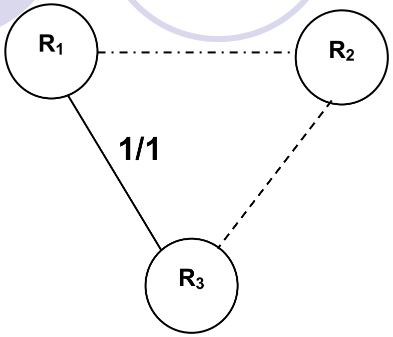
R_1 and R_2 incompatible as they are accessed concurrently in the number of states

R_2 and R_3 incompatible as they are accessed concurrently in s_3 and s_5

R_1 and R_3 incompatible as they **are read in the same cycle (state 4), but can be made compatible by introducing an additional state**



Register Merging



Compatibility graph

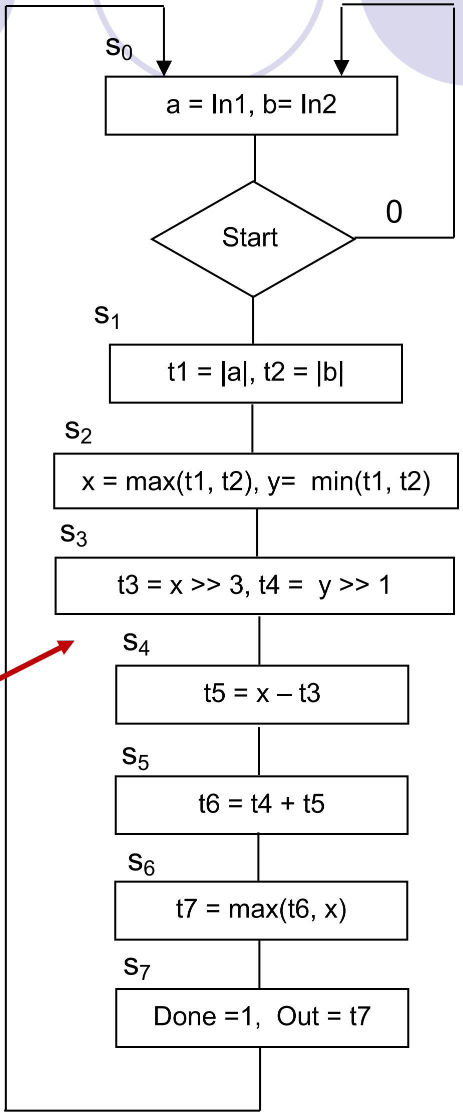
Register assignments:

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

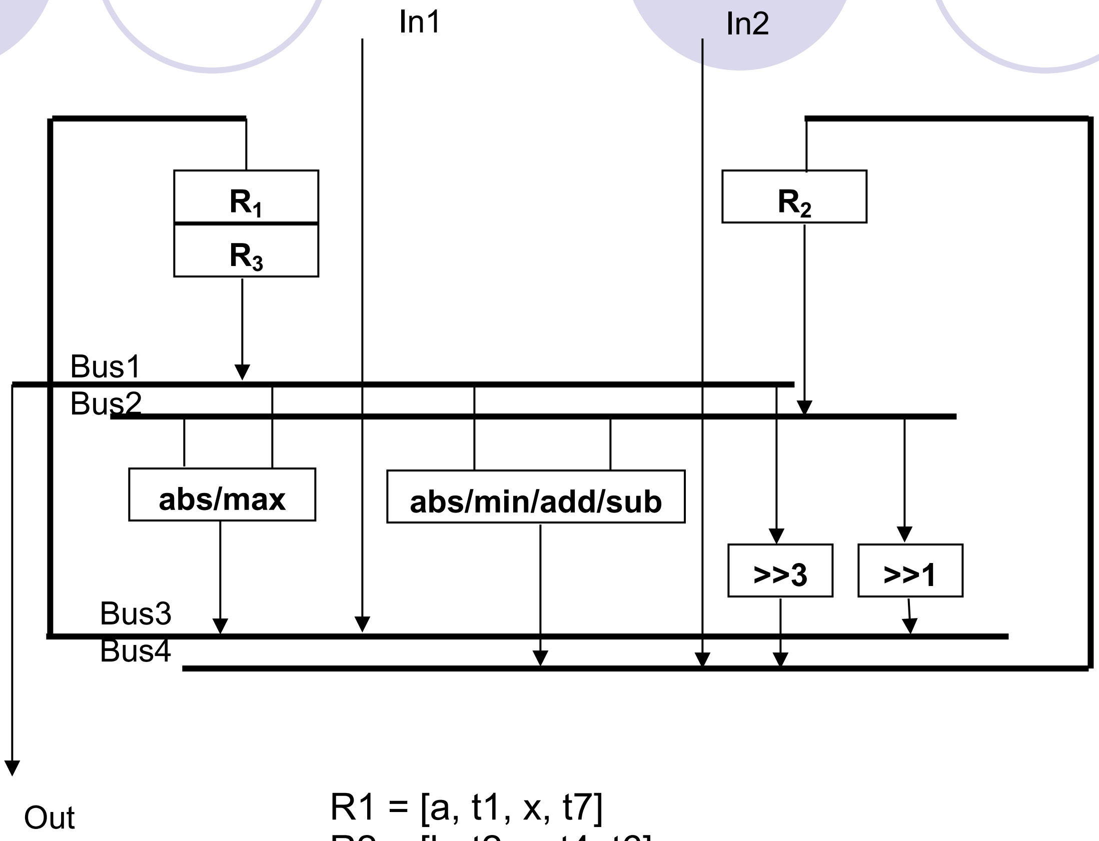
R_1 and R_2 incompatible as they are accessed concurrently in the number of states

R_2 and R_3 incompatible as they are accessed concurrently in s_3 and s_5

R_1 and R_3 become compatible with addition of one state S_{34} (e.g. $t_5=t_3$) and then S_4 ($t_5=x-t_5$)



Register Merging



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

Chaining

- In the previous examples registers are clocked by a clock signal whose clock cycle is equal to the worst register-to-register delay.

- If the total delay of two functional units is shorter than the clock cycle, it is possible to connect them in series and perform two operations in a single clock cycle.

- This technique of connecting functional units in series is called chaining.

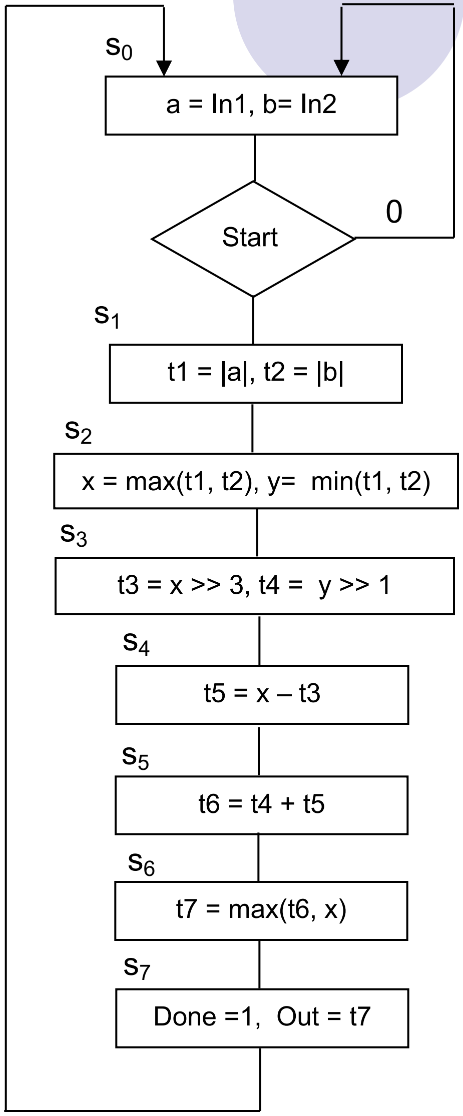
- The ASM chart will contain two or more operators on the right-hand side of assignment statements in the same state.

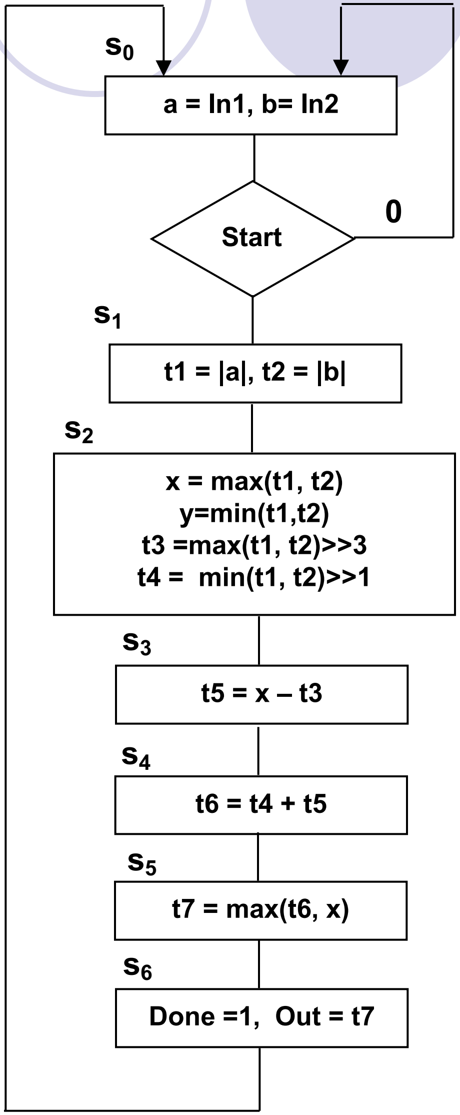
Chaining – RSA Example

- Two states (S2 and S3) merged into one (S2)
- Shifting has no delay
- As the ASM chart has only 7 states this modified datapath performs 12.5% faster
- Additional link to R3 had to be created to store the results concurrently to registers

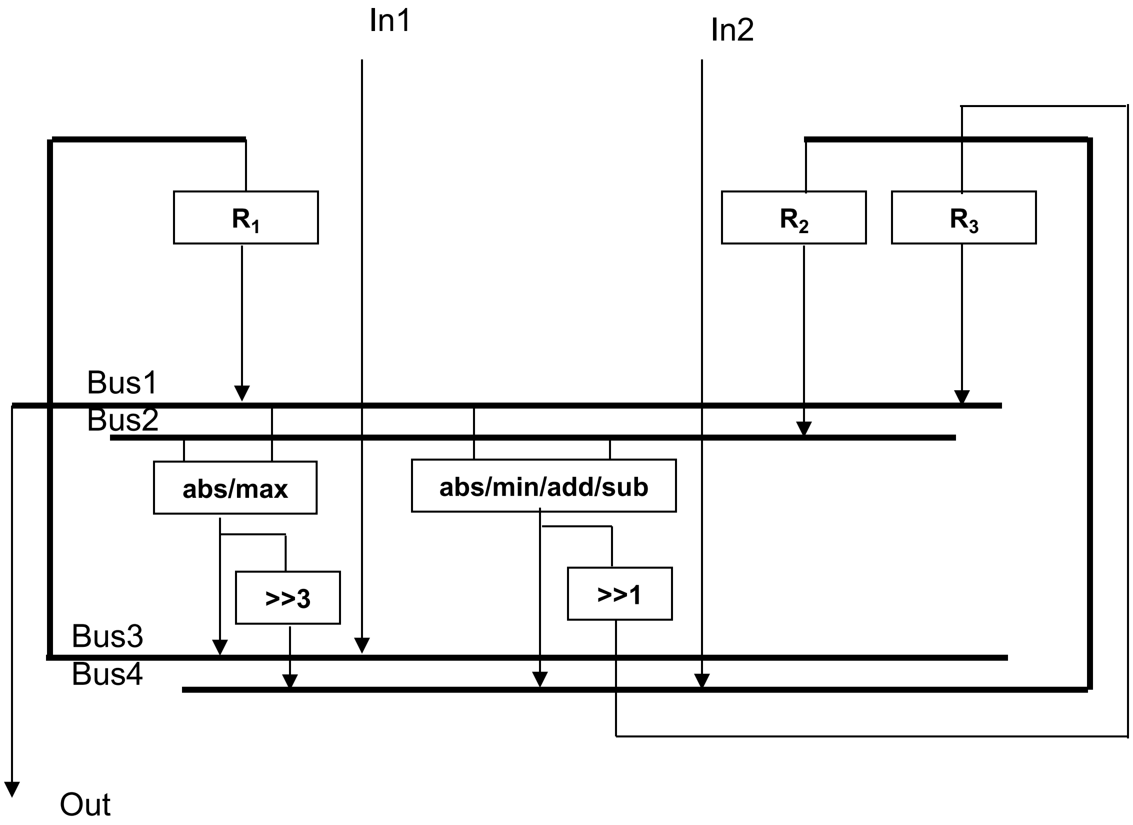
Register assignments:

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





Chaining – RSA Example



Multicycling

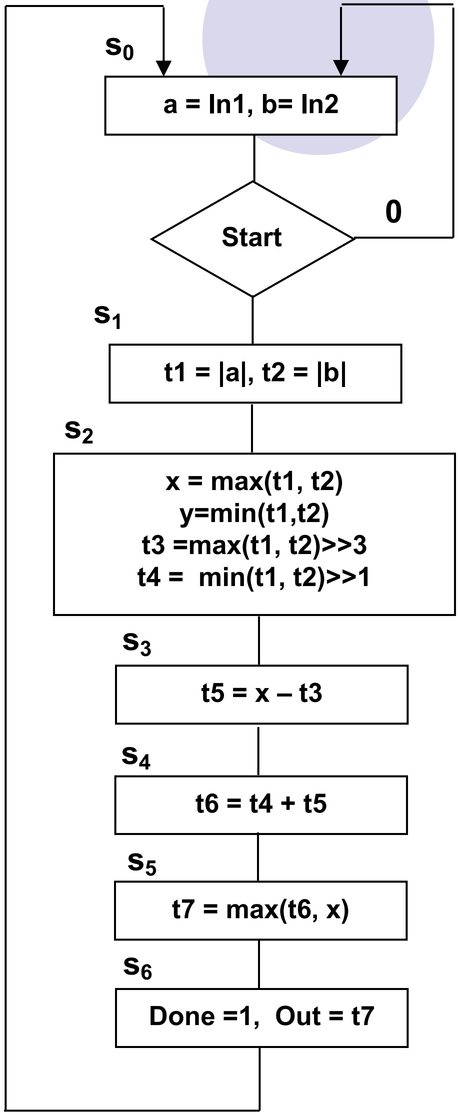
- If we sometimes use slower units that require more than one cycle to generate their results, we apply multi-cycling technique.

- Such units can only be in non-critical paths through the ASM chart.

- Example: Variable t4 assigned a new value in state s2, but will not be used until state s4. We have 2 cycles to perform this operation.

- t4 will be assigned at some later state

- t4 is assigned new value based on computation which started at some earlier state



s_0

a = In1, b= In2

Start

0

s_1

t1 = |a|, t2 = |b|

s_2

x = max(t1, t2)
y=min(t1,t2)
t3 =max(t1, t2)>>3
t4 = min(t1, t2)>>1

s_3

t5 = x – t3

s_4

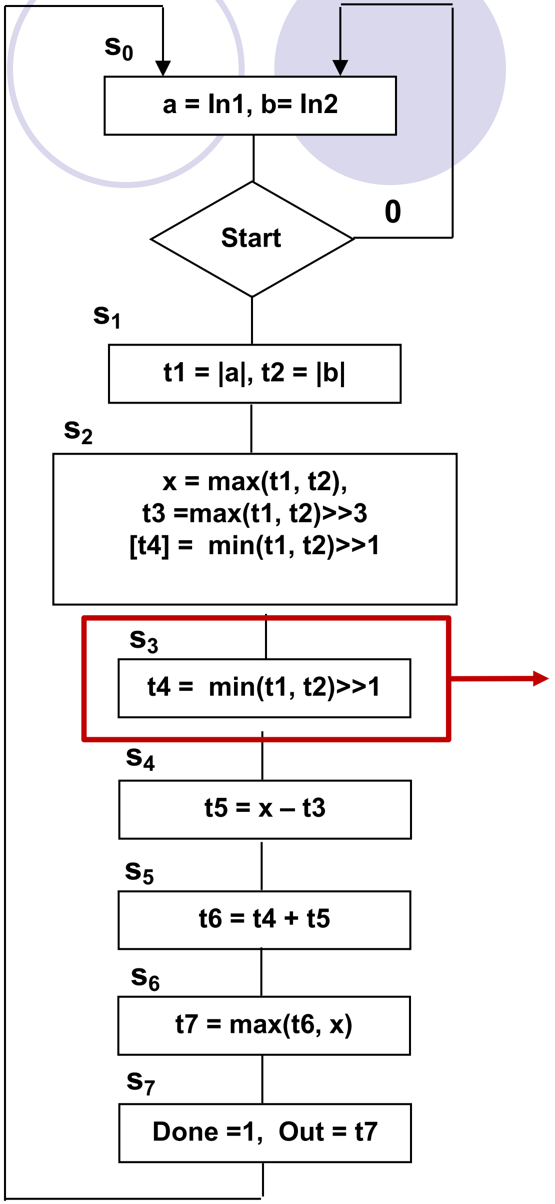
t6 = t4 + t5

s_5

t7 = max(t6, x)

s_6

Done =1, Out = t7



s_0

a = In1, b= In2

Start

0

s_1

t1 = |a|, t2 = |b|

s_2

x = max(t1, t2),
t3 =max(t1, t2)>>3
[t4] = min(t1, t2)>>1

s_3

t4 = min(t1, t2)>>1

s_4

t5 = x – t3

s_5

t6 = t4 + t5

s_6

t7 = max(t6, x)

s_7

Done =1, Out = t7

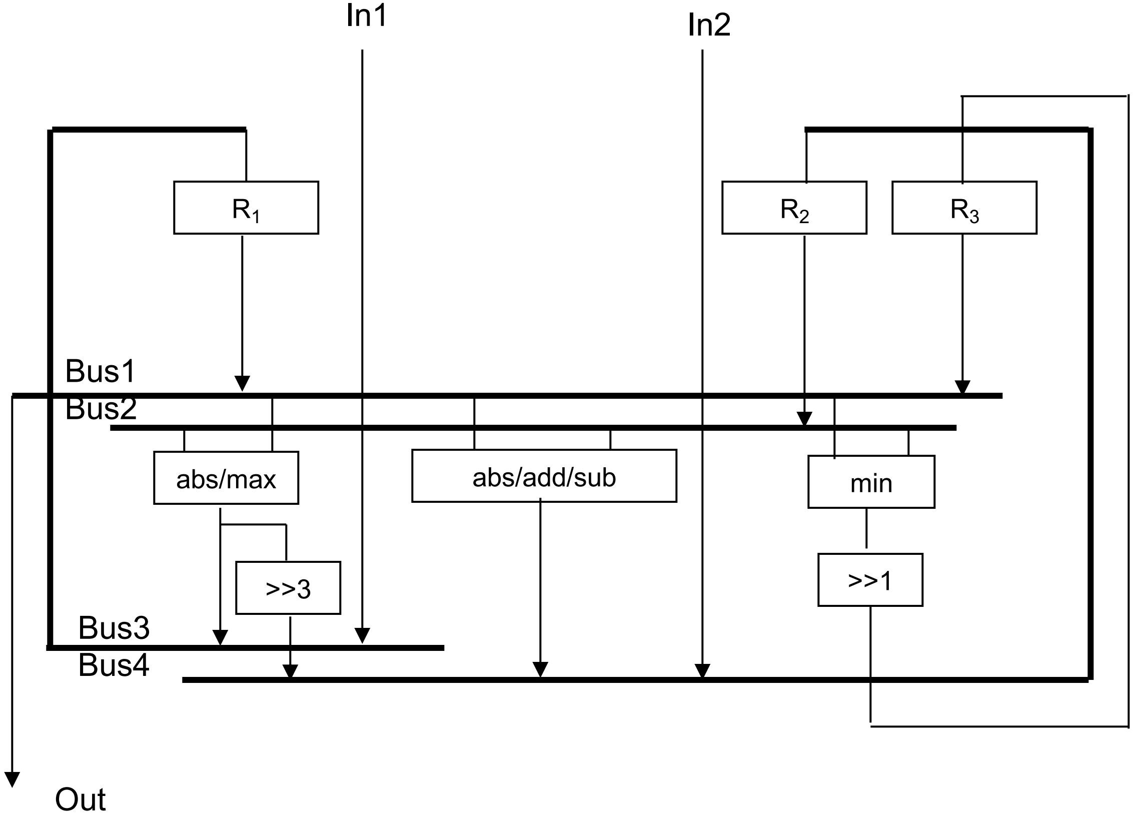
Multicycling

Register assignment:

R1 = [a, t1, x, t7]

R2 = [b, t2, y, t3, t5, t6]

R3 = [t4]

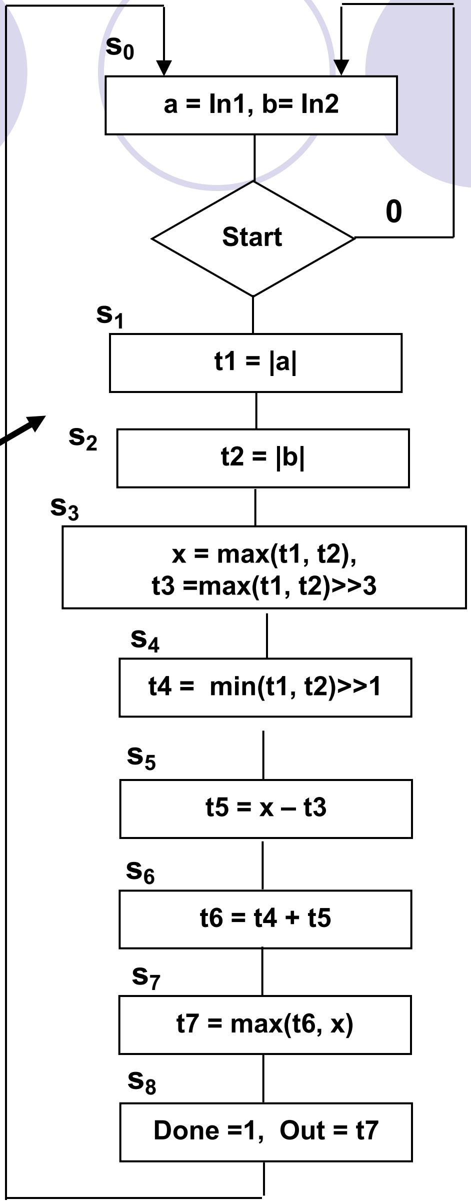


Operation pipelining

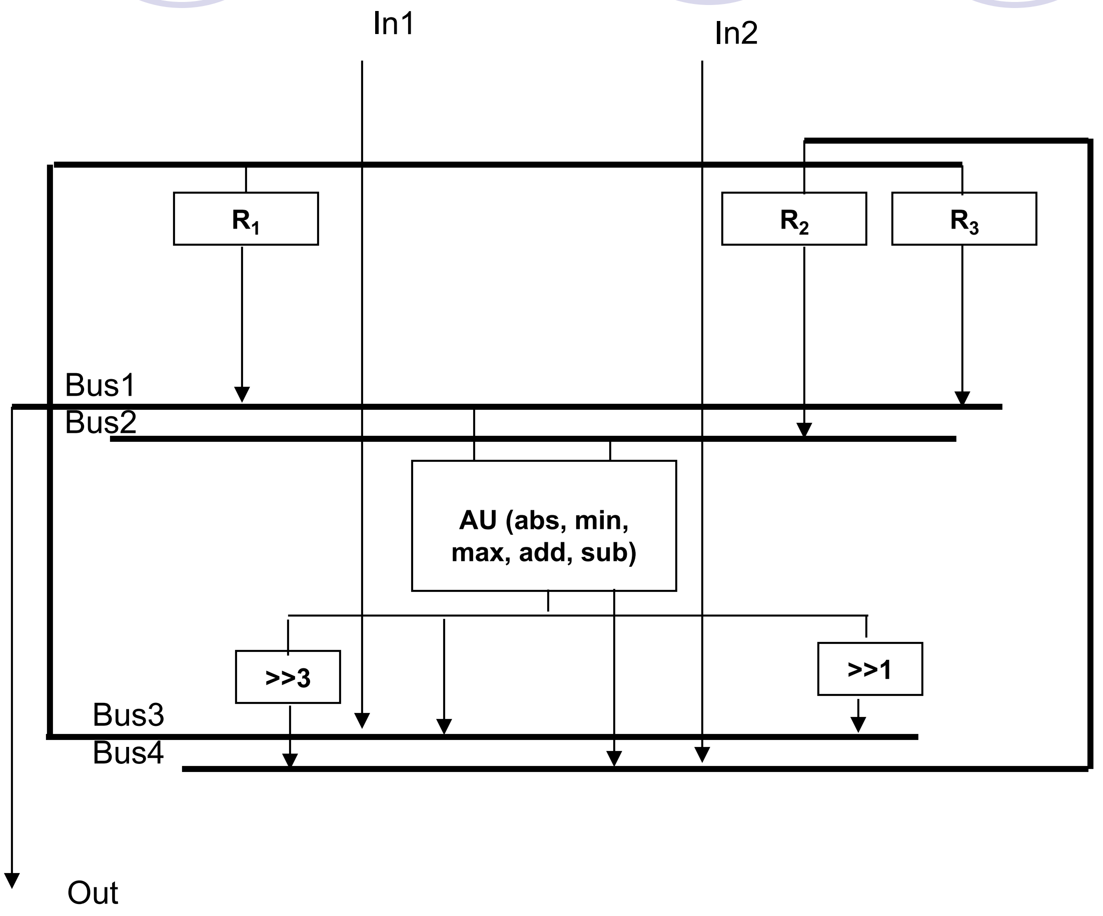
- So far we concentrated on techniques that reduce datapath implementation cost.
- Now we concentrate on those that increase the performance of the datapath.
- Pipelining - the technique is using simultaneous processing on a number of stations (functional units)/multiple stages.
- The technique can be applied on functional units, datapaths or control.
- Multiple latches are introduced into functional unit design to separate its stages, each stage operating on different set of operands.
- Generally, if there are n stages in pipeline, we can reduce time to generate results to 1/n times the non-pipelined execution time, with the exception of the first n-1 results

Operation pipelining – RSA Example

- Arithmetic unit performs abs, min, max, add and sub.
- |a| and |b| have to be performed in separate clock cycles
- Two-stage arithmetic unit (AU) – stages separated with three data latches and one control latch.



Operation pipelining – RSA Example



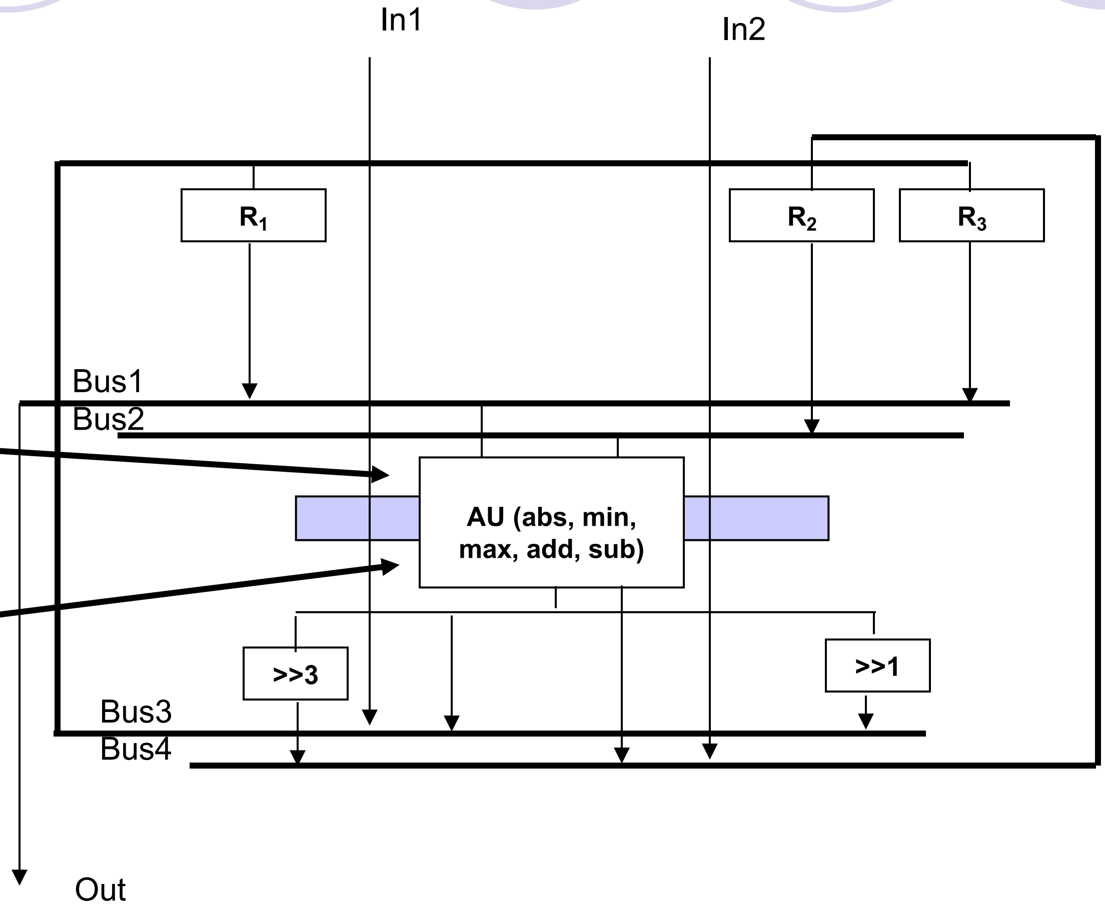
Operation pipelining – RSA Example

Divide AU into two pipeline stages

AU Stage 1

AU Stage 2

with half propagation delay of original AU



Operation pipelining – RSA Example

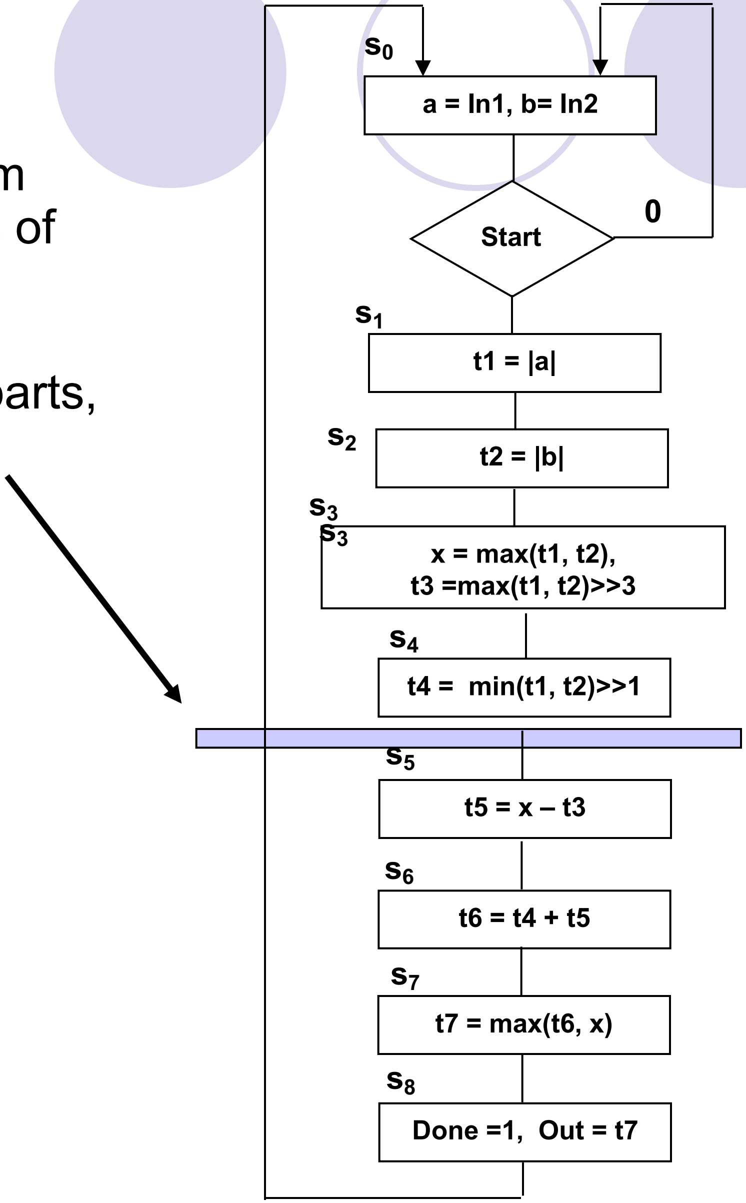
- The algorithm requires 13 states and clock cycles to complete, but two of these cycles are equal to one cycle in non-pipelined case.

- The pipelined implementation outperforms any of the presented non-pipelined designs described above.

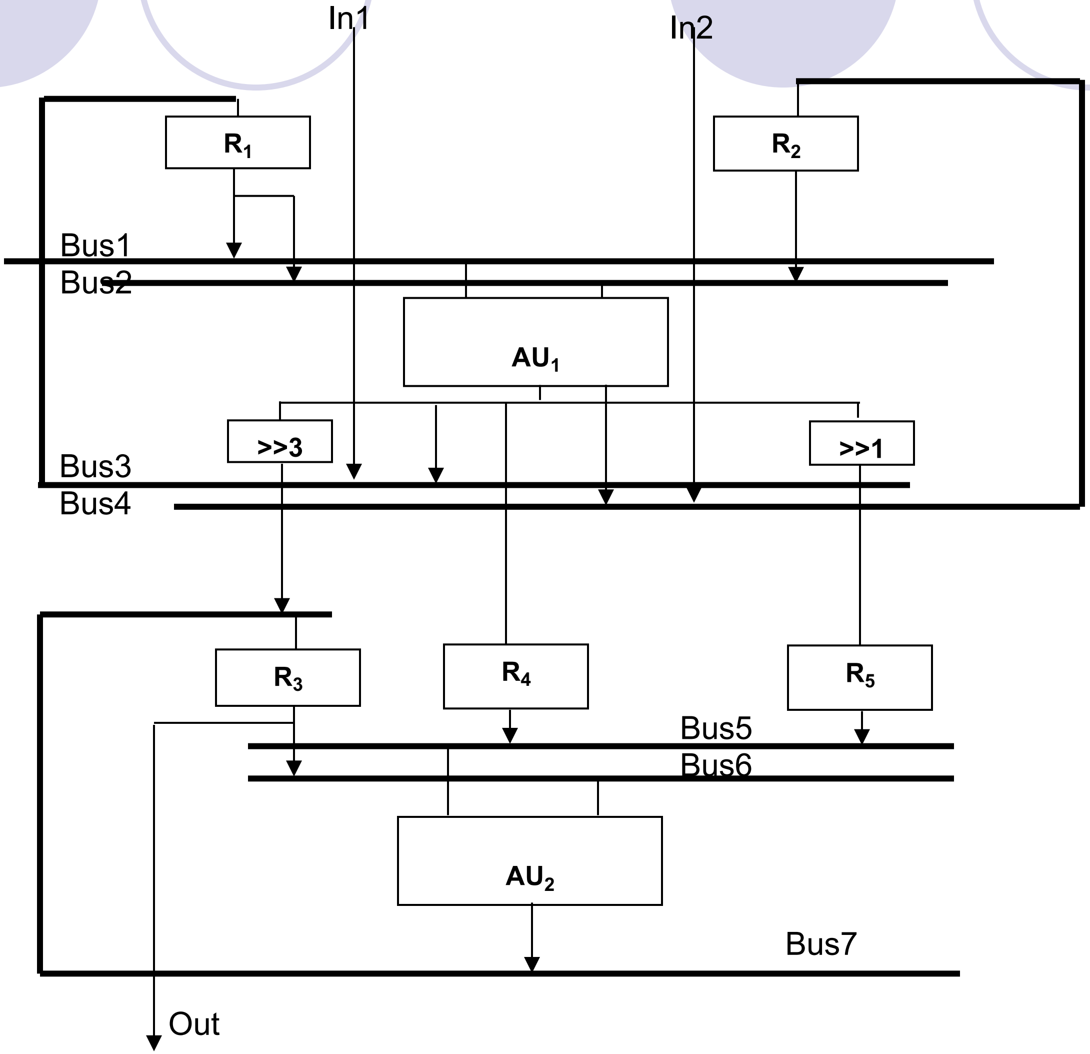
	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}
Read R_1		a			t1	t1	x				x		t7
Read R_2			b		t2	t2	t3		t5		t6		
Read R_3									t4				
AU stage 1		|a|	|b|		max	min	sub		add		max		
AU stage 2			|a|	|b|		max	min	sub		add		max	
Shifters						>>3	>>1						
Write R_1	a		t1			x						t7	
Write R_2	b			t2		t3		t5		t6			
Write R_3							t4						
Out													t7

Datapath pipelining – RSA Example

- Datapath can be pipelined to perform the same operation on different sets of operands.
- The ASM chart can be divided into several equal (or near equal) -size parts, which can be executed in different datapath stages.
- The assignment of variables to the stages is:
  - Stage 1:
    R1 = [a, t1]
    R2 = [b, t2]
    AU1 = [abs, min, max]
  - Stage 2:
    R3 = [t3, t5, t6, t7]
    R4 = [x]
    R5 = [t4]
    AU2 = [add, sub, max]



Datapath pipelining – RSA Example



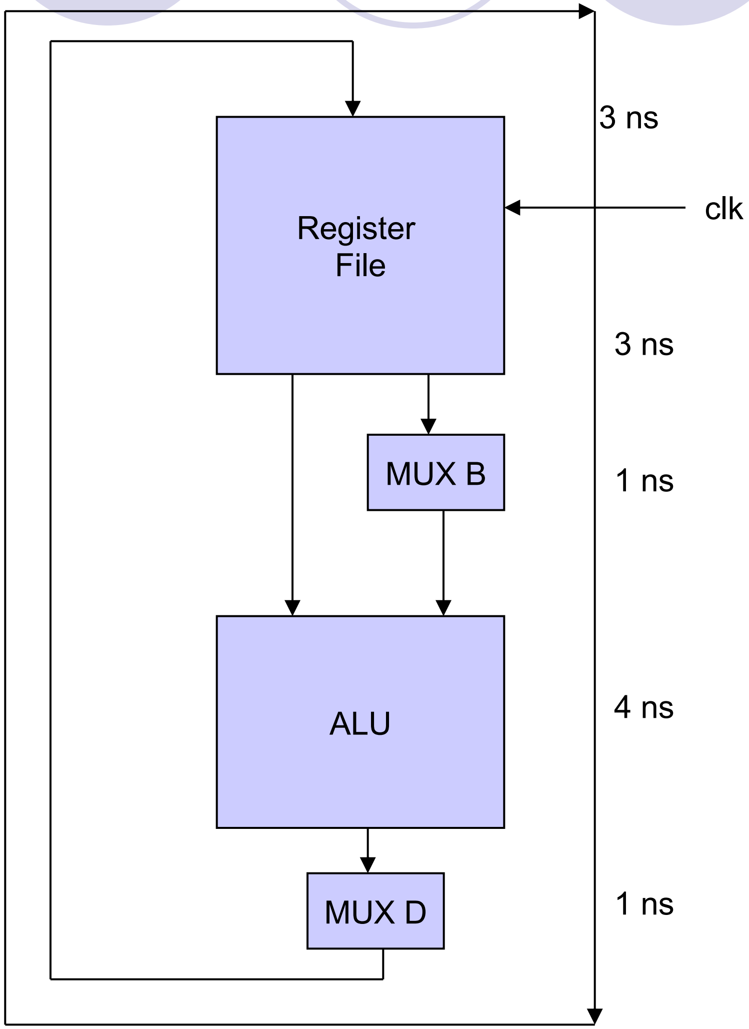
Datapath pipelining – RSA Example

	nth pair					(n+1)th pair				
	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
Read R_1		a		t1	t1					
Read R_2			b	t2	t2					
AU_1		|a|	|b|	min	max					
Shifters				>>1	>>3					
Write R_1	a	t1								
Write R_2	b		t2							
Read R_3						t3	t5	t6	t7	
Read R_4						x		x		
Read R_5							t4			
AU_2						sub	add	max		
Write R_3					t3	t5	t6	t7		
Write R_4					x					
Write R_5				t4						
	(n-1)th pair					nth pair				

- Enables computation on different sets of operands concurrently
- Can be combined with component (operation) pipelining

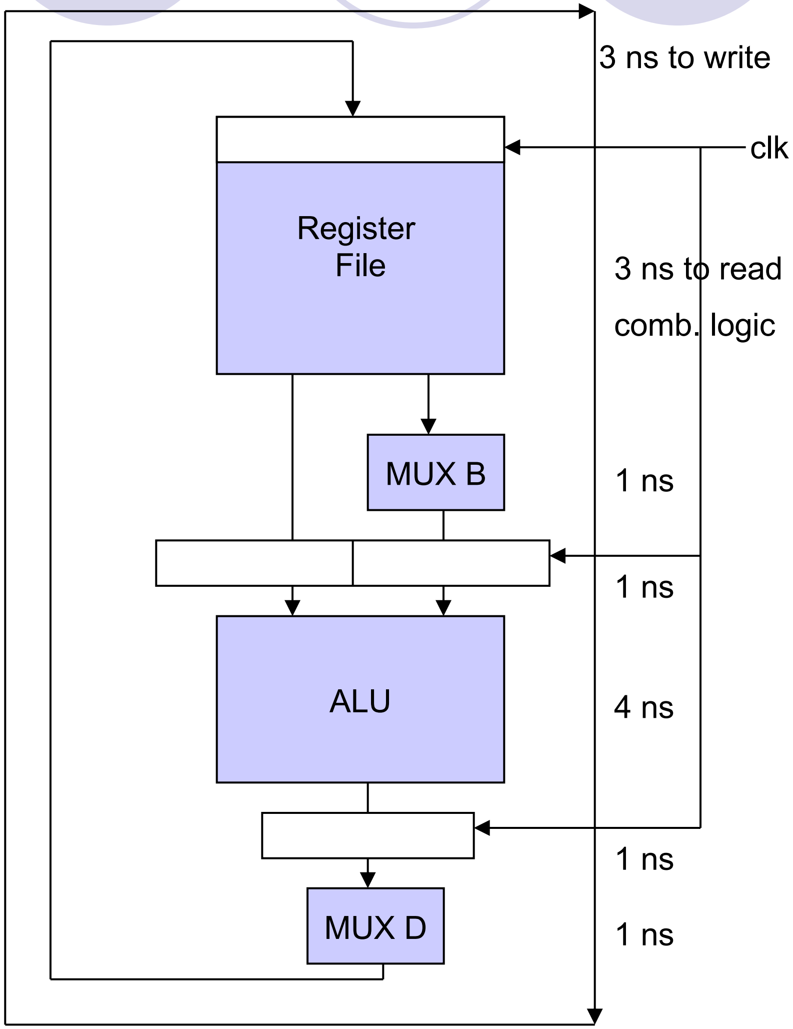
Datapath pipelining – General datapath example

- Timing problems due to propagation times through combinational logic and register set-up times
- Total delay 12 ns = minimum clock cycle
- Max frequency ~ 83 MHz



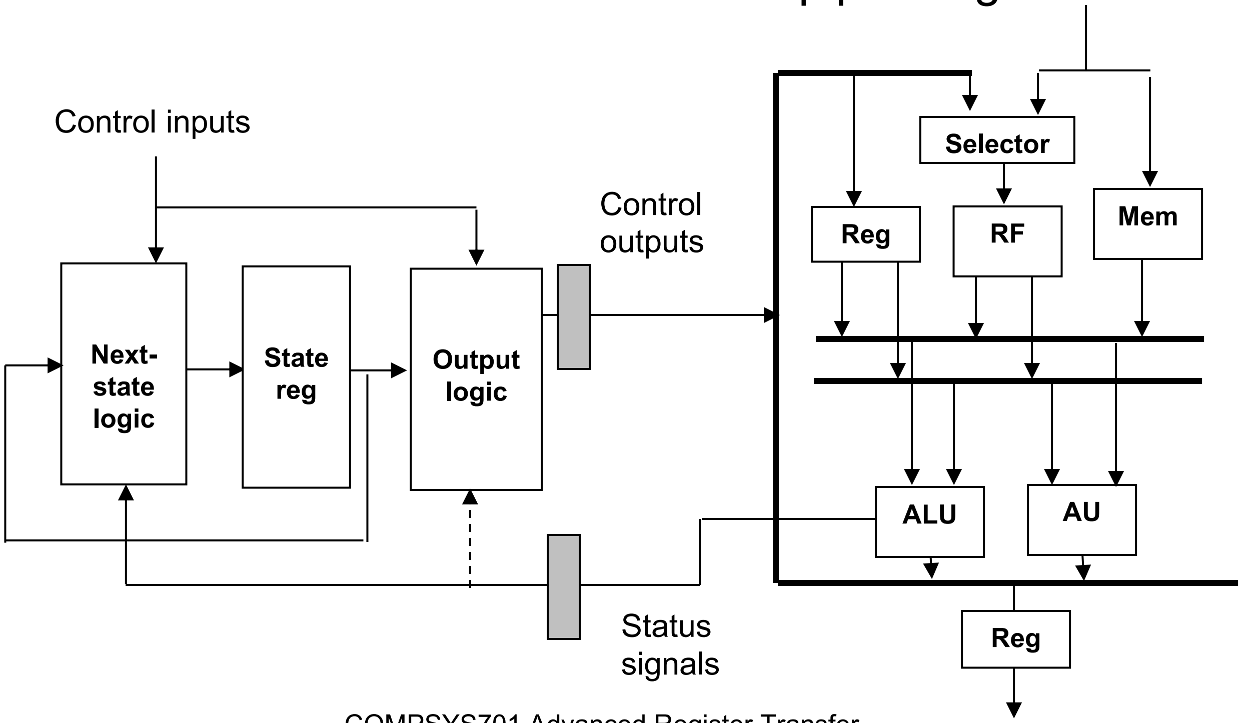
Datapath pipelining – General datapath example

- Delay can be broken with registers (“assembly line”-like) into three stages:
- Operands fetch (OF)
- Execute operation (EX)
- Write back (WB)
- Can process 3 times as many microoperations in a given time as the conventional datapath
- Pipeline + registers + stages
- Max delay 5 ns → 200 MHz
- Pipeline has to fill and empty from time to time

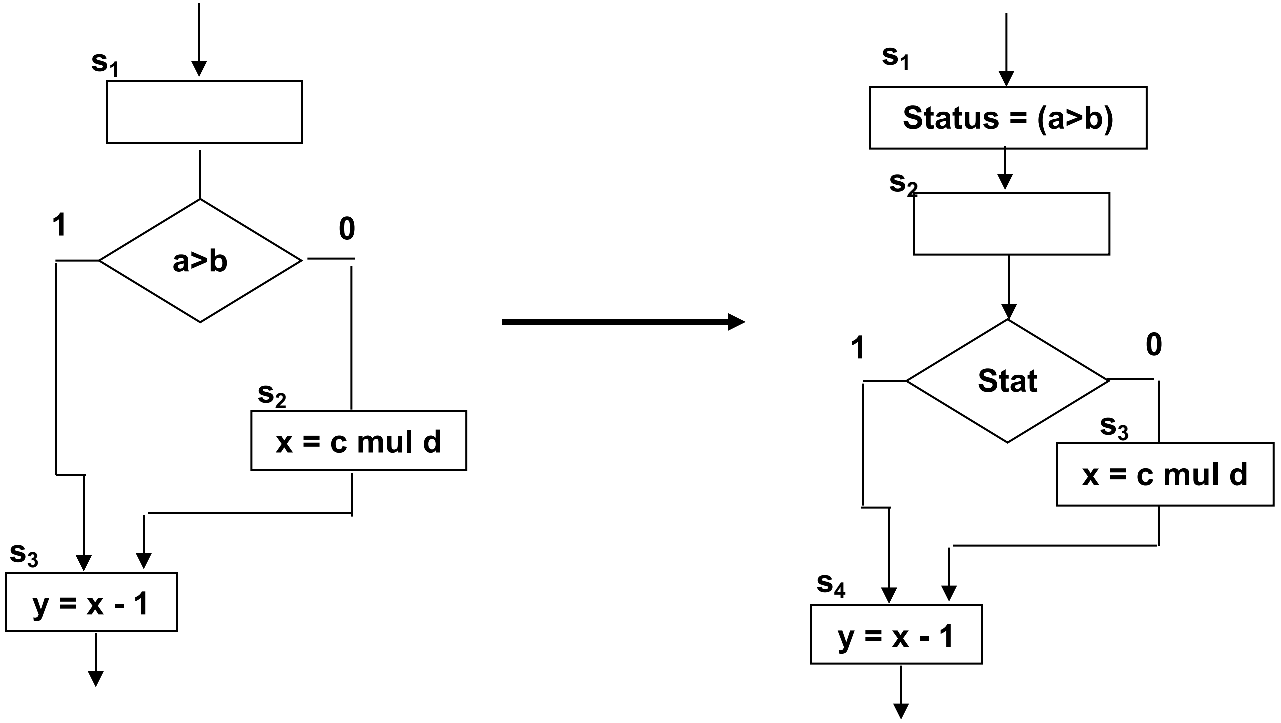


Control pipelining

- The longest critical path in the circuit is still in control unit (state register update), especially if the next state depends on the status signals
- The critical path can be divided into pieces with the registers inserted between them
- Status register has one flip-flop for each status signal
- Control register contains one flip-flop for each control signal
- Pipeline registers between storage elements and functional units
- ASM chart has to be constructed to reflect pipelining decisions



Control pipelining – Example of modified ASM



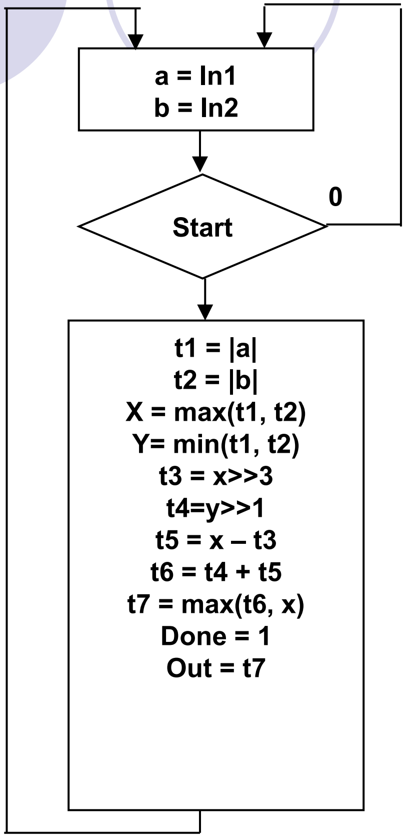
Implementing Algorithms in Digital Circuits – Scheduling

- To transform an ordinary algorithm or a flowchart into an ASM chart
  - its execution must be partitioned into a series of time intervals and states, and
  - each variable must be assigned to a particular state – schedule
- The variable access and operations in the states are under either resource (resource constraints, RC) or time constraints (time constraints, TC)
- First step is to convert a program or a flowchart (or ASM chart) into a control/dataflow graph (CDFG) that shows explicitly control dependencies among statements and data dependencies among variables

Implementing Algorithms in Digital Circuits – Scheduling

- CDFG contains the same sequence of assignment statements as a flowchart – nodes represent operators and edges results generated by the operators
- RC and TC scheduling algorithms usually use As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling algorithm to determine operation priority and range for scheduling
- ASAP and ALAP assume that
  - each operation takes one clock cycle to execute
  - unlimited number of functional units or resources are available in each state
- As such they may be considered as constrained only by data dependencies

Scheduling – Conversion of an ASM into CDFG

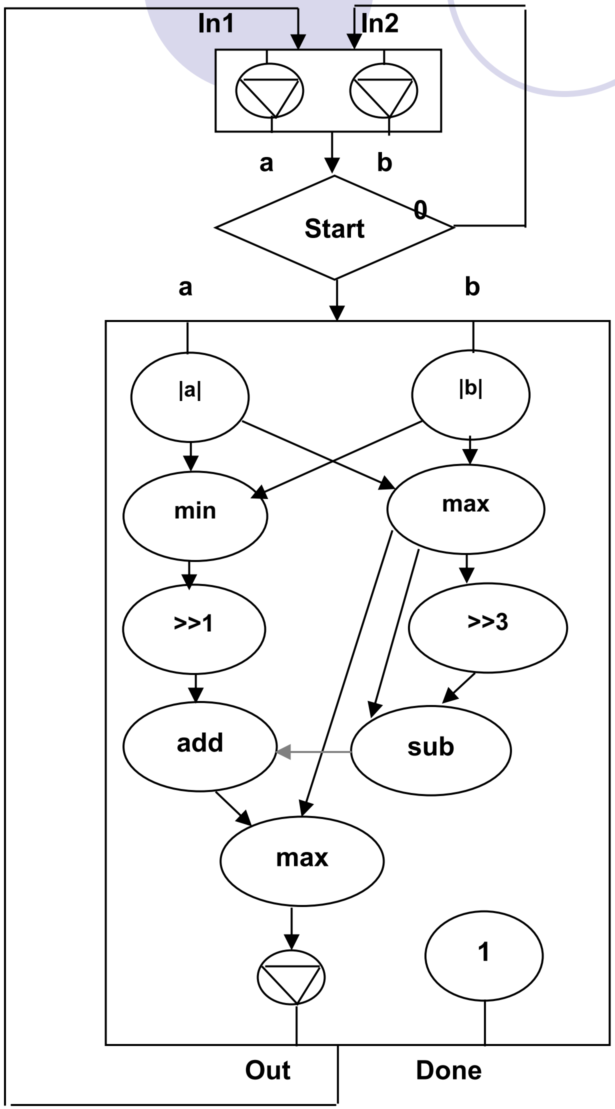


a = In1
b = In2

Start

0

t1 = |a|
t2 = |b|
X = max(t1, t2)
Y= min(t1, t2)
t3 = x>>3
t4=y>>1
t5 = x – t3
t6 = t4 + t5
t7 = max(t6, x)
Done = 1
Out = t7



ASAP Schedule

- ASAP schedules each operation into earliest state in which all operands are available
- It scans the CDFG from the top to the bottom and assigns to each state all nodes in the graph whose predecessors or parent nodes have been already assigned into previous states.
- It generates a schedule with the minimum number of states (shortest execution time).

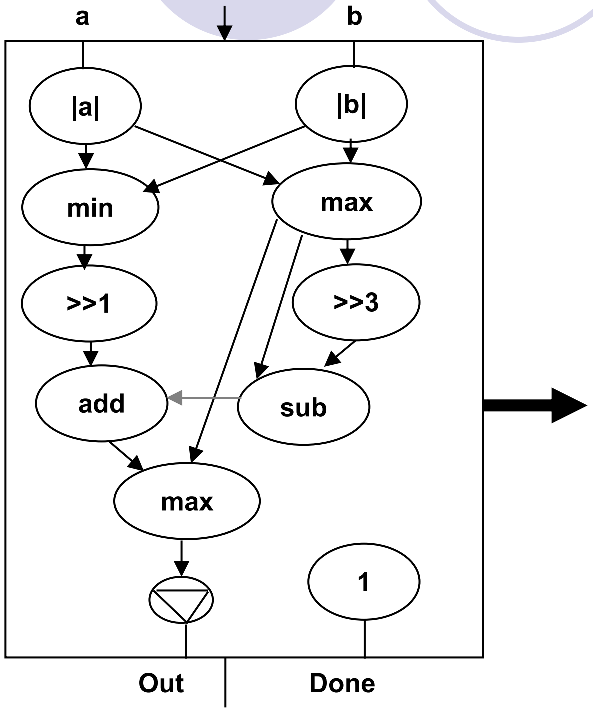
ALAP Schedule

- ALAP schedules each operation into the last possible state before its result is needed if it is given the length of the final schedule in the number of states as a constraint

- It scans the CDFG from the bottom to the top and assigns to each state all the nodes whose successor or children nodes have been already assigned into later states.

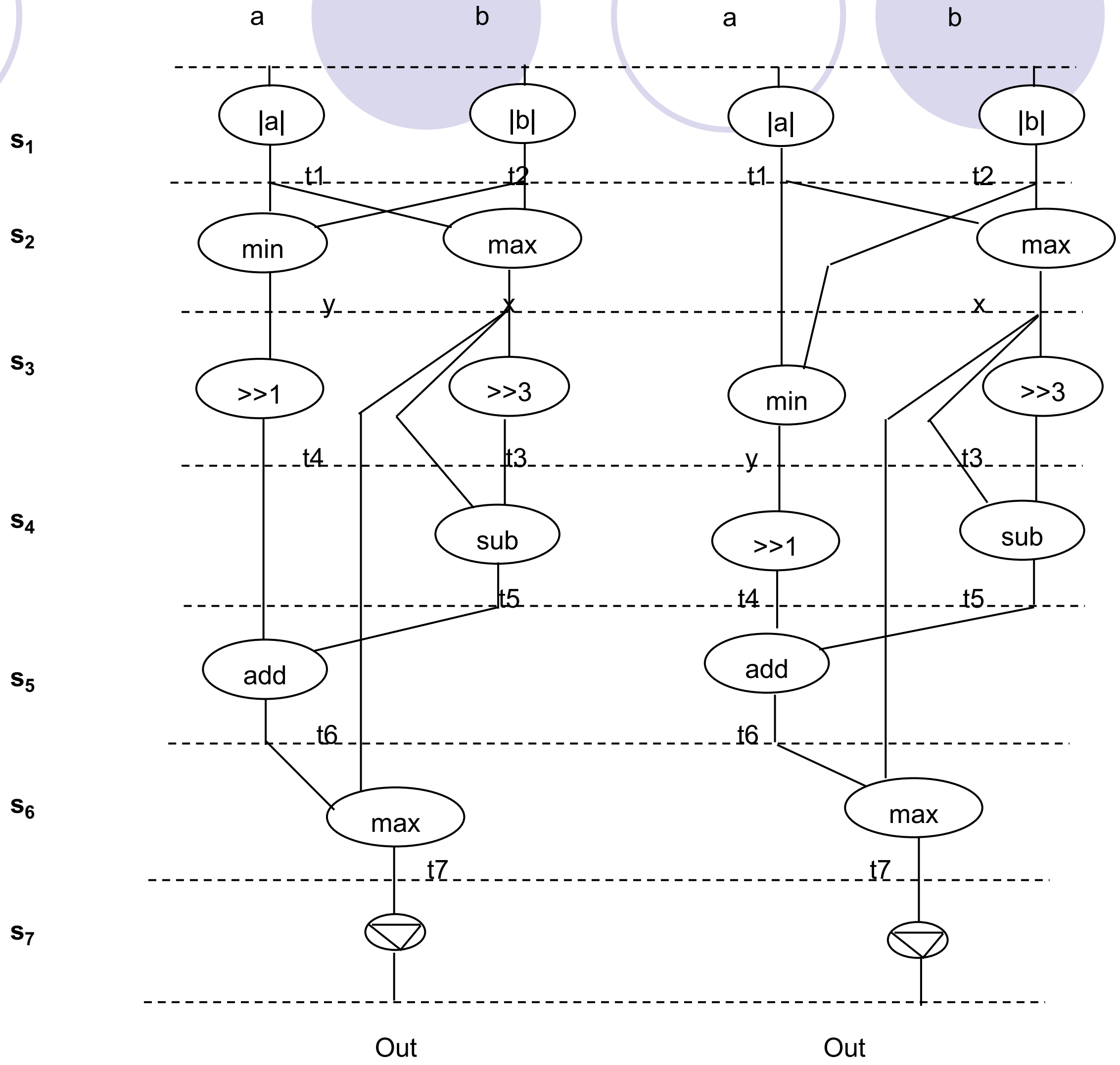
- If the given schedule has equal the number of states obtained by ASAP algorithm, the ALAP algorithm schedules all the operations on the longest critical path through the CDFG into the same states as the ASAP algorithm

Scheduling – RSA Example



All operations except “min” and “>>1” are on critical path.

They are scheduled as early as possible in ASAP and as late as possible in ALAP schedule.



(a) ASAP schedule

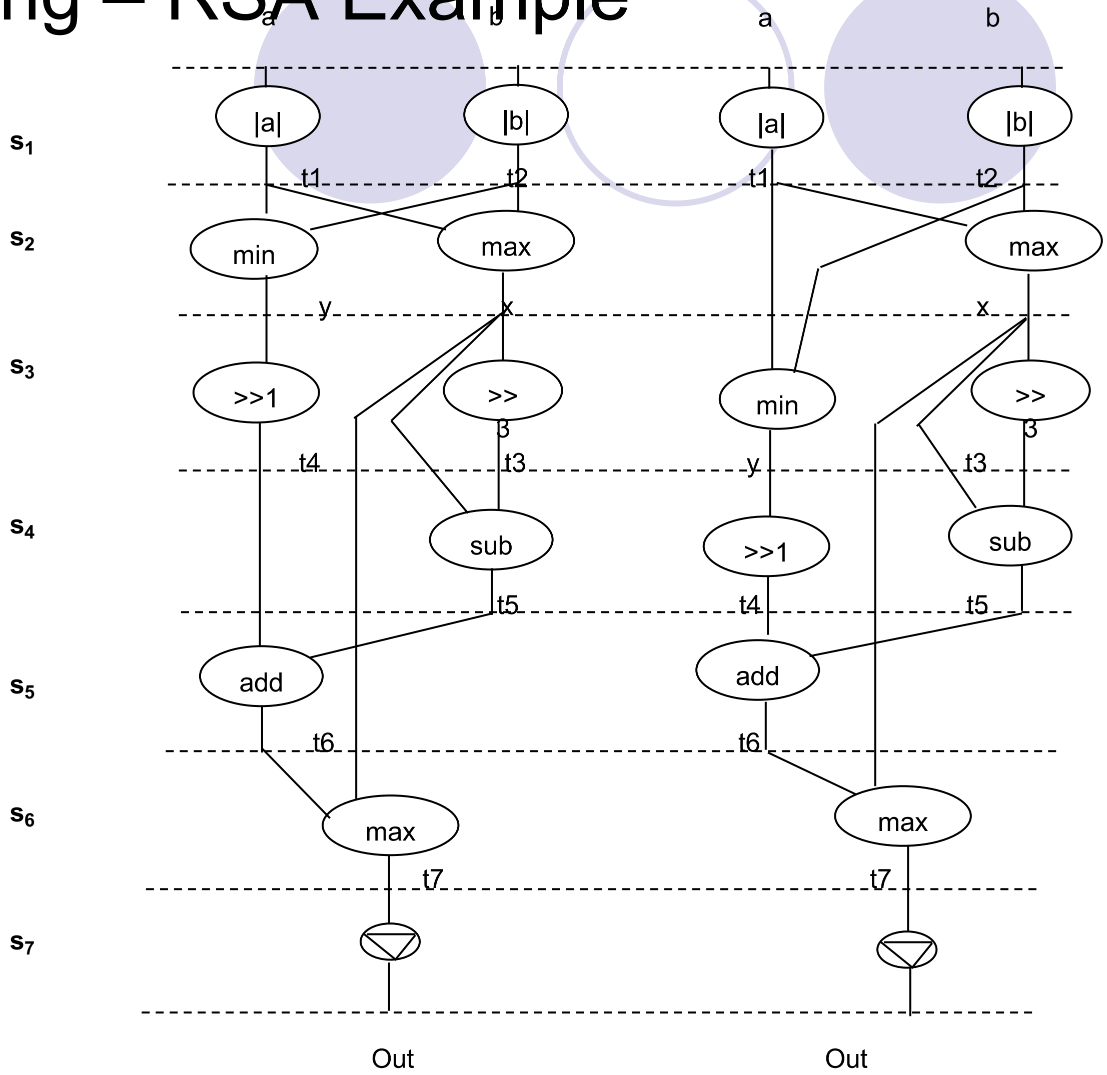
(b) ALAP schedule

Scheduling – RSA Example

Lower priority can be given to operations that are not on the critical path as they can be delayed without affecting the execution time of the entire CDFG.

Operation's priority scheduling can be measured by different metrics, e.g. its **operation mobility** – if it is in state s_i in ASAP schedule and s_k in ALAP schedule, its mobility M(op) = k – i.

An alternative measure of priority is **operation urgency** – the distance between the state it is available for scheduling and its ALAP schedule. If an operation is available in state s_j, but is not scheduled until state s_k in its ALAP schedule, the operation urgency U(op) = k - j.

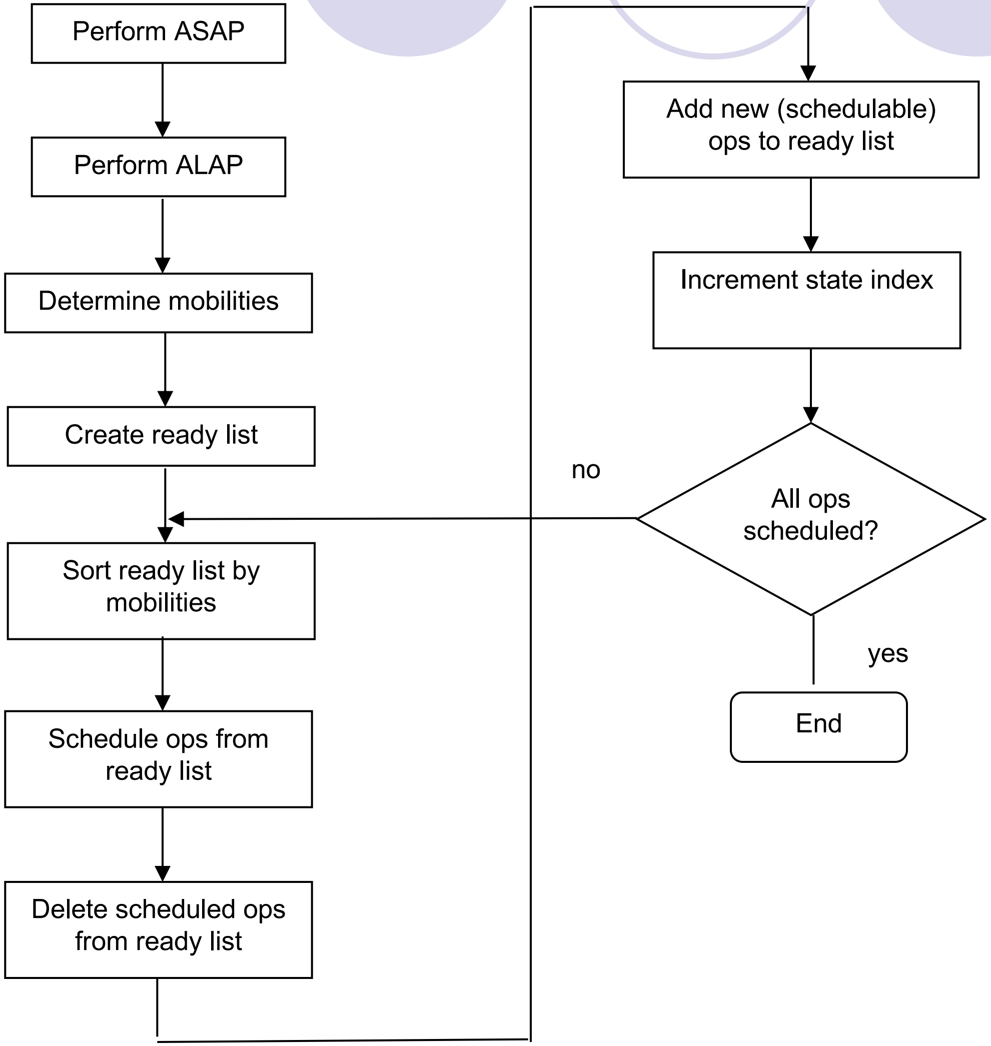


(a) ASAP schedule **(b) ALAP schedule**

RC scheduling: List-scheduling algorithm

- Operations on the ready list are sorted by their mobilities – those with zero mobility on the top of the list, and those with the highest mobility on the bottom –

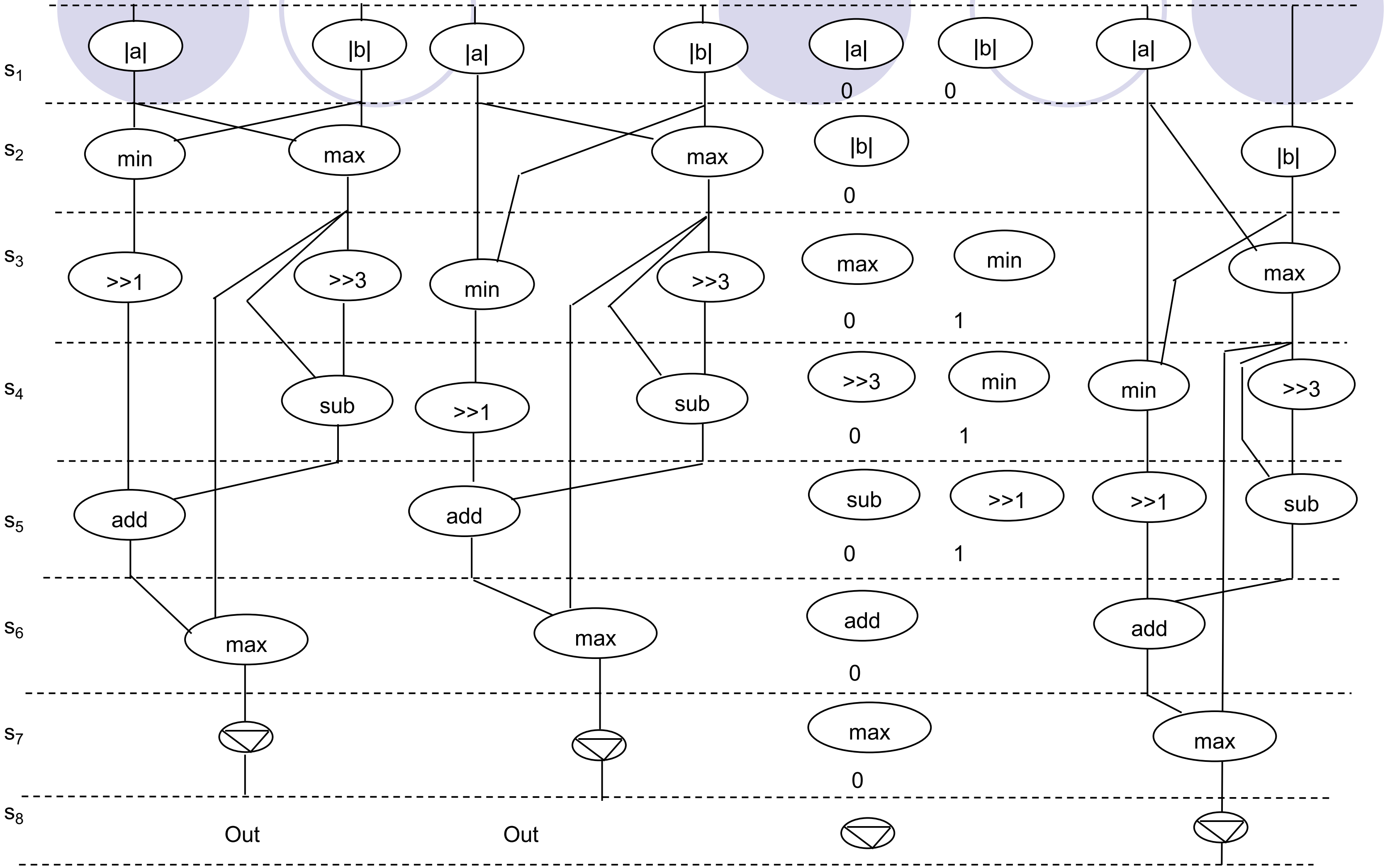
- If two operations have the same mobility, priority is given to the operation with the lower urgency number. If the urgency numbers are the same, priority is assigned randomly.


Perform ASAP
Perform ALAP
Determine mobilities
Create ready list
Sort ready list by mobilities
Schedule ops from ready list
Delete scheduled ops from ready list
Add new (schedulable) ops to ready list
Increment state index
All ops scheduled?
no
yes
End

RC scheduling: List-scheduling algorithm

- In each state:
  - assign the highest priority operations from the ready list to the available functional units, one at a time
  - then delete all the assigned operations from the ready list
  - insert new operations, that become schedulable, into the list in the positions that correspond to their mobilities and urgencies
- Example:
  - Assumption that there is only one arithmetic unit which can perform abs, min, max, add and sub in addition to two shifters.
  - The goal of RC scheduling is to schedule in each state as many operations as it can, given the fixed number of available functional units.
  - In this example we have eight states, but also only one functional unit instead of two when using ASAP or ALAP algorithm.

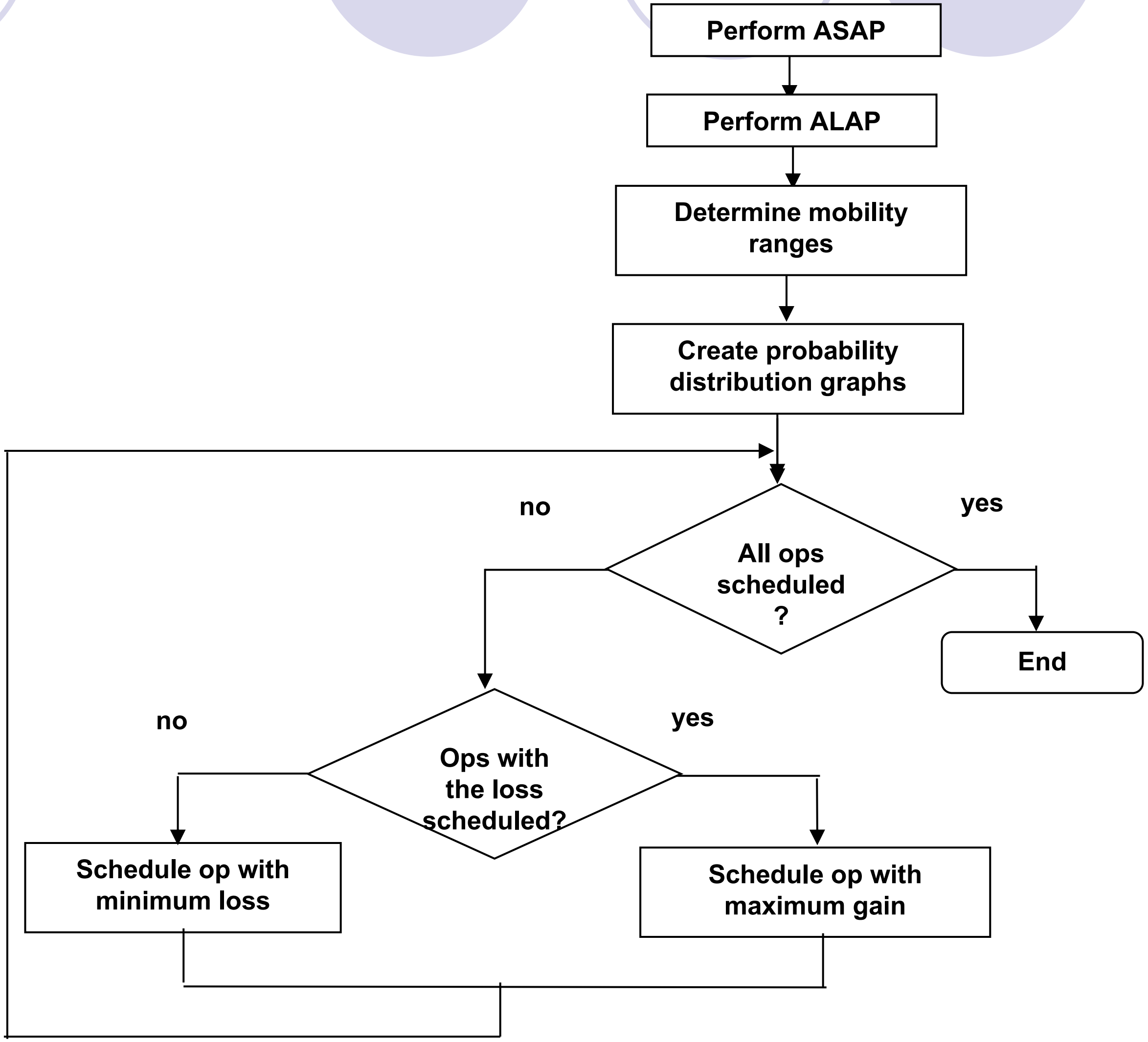
RC scheduling: RSA Example



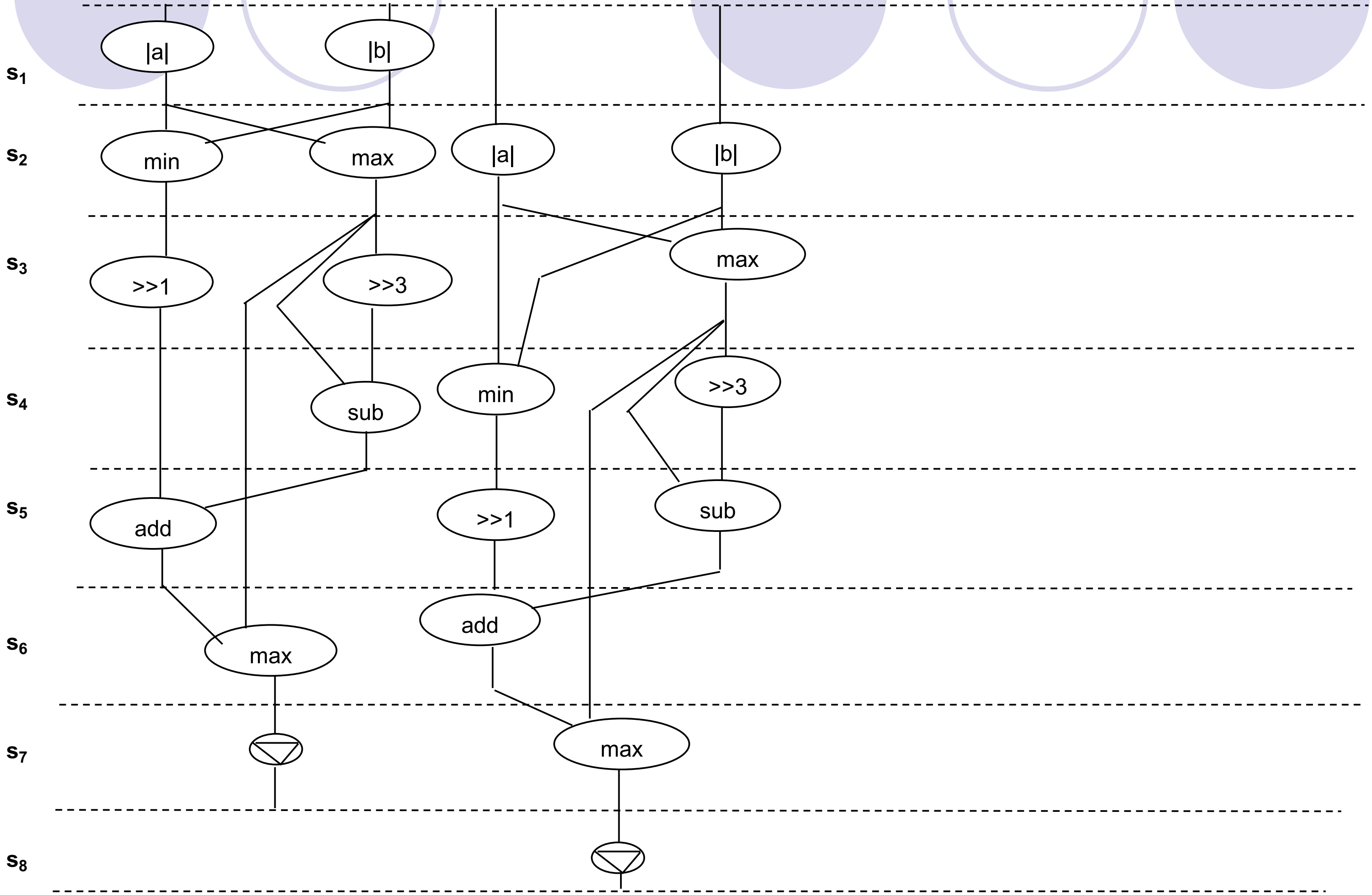
(a) ASAP schedule **(b) ALAP schedule** **(c) Ready list with mobilities** **(d) RC schedule**

TC scheduling

- In TC scheduling the primary goal is performance - to execute a given algorithm in a fixed amount of time.
- Schedule contains a particular number of states, while attempting to minimize the number of functional units in the datapath.
- The goal is achieved by creating a probability distribution graph and by scheduling operations, one at the time, so that the largest sum of probabilities for each operator and each state is minimal


Perform ASAP
Perform ALAP
Determine mobility ranges
Create probability distribution graphs
All ops scheduled ?
no
yes
End
Ops with the loss scheduled?
no
yes
Schedule op with minimum loss
Schedule op with maximum gain

TC scheduling RSA Example with eight states



(a) ASAP schedule **(b) ALAP schedule**

TC scheduling - RSA Example

Initial probability distribution graph

State	AU Units				Probability Sum/state	Shift units		Probability Sum/state
S_1	|a|	|b|			1.0			
S_2			max	min	1.83			
S_3					0.83	>>1	>>3	0.83
S_4	sub				0.83			0.83
S_5		add			1.0			0.33
S_6			max		1.0			
S_7					0.5			

TC scheduling - RSA Example

Distribution graph after max, add and sub were scheduled

State	AU Units				Probability Sum/state	Shift units		Probability Sum/state
S_1	|a|	|b|			1.0			
S_2			min		1.33			
S_3	max				1.33	>>1	>>3	0.83
S_4					0.33			0.83
S_5	sub				1.00			0.33
S_6	add				1.00			
S_7	max				1.00			

TC scheduling - RSA Example

Distribution graph after max, min, add, sub, >>3, and >>1 were scheduled

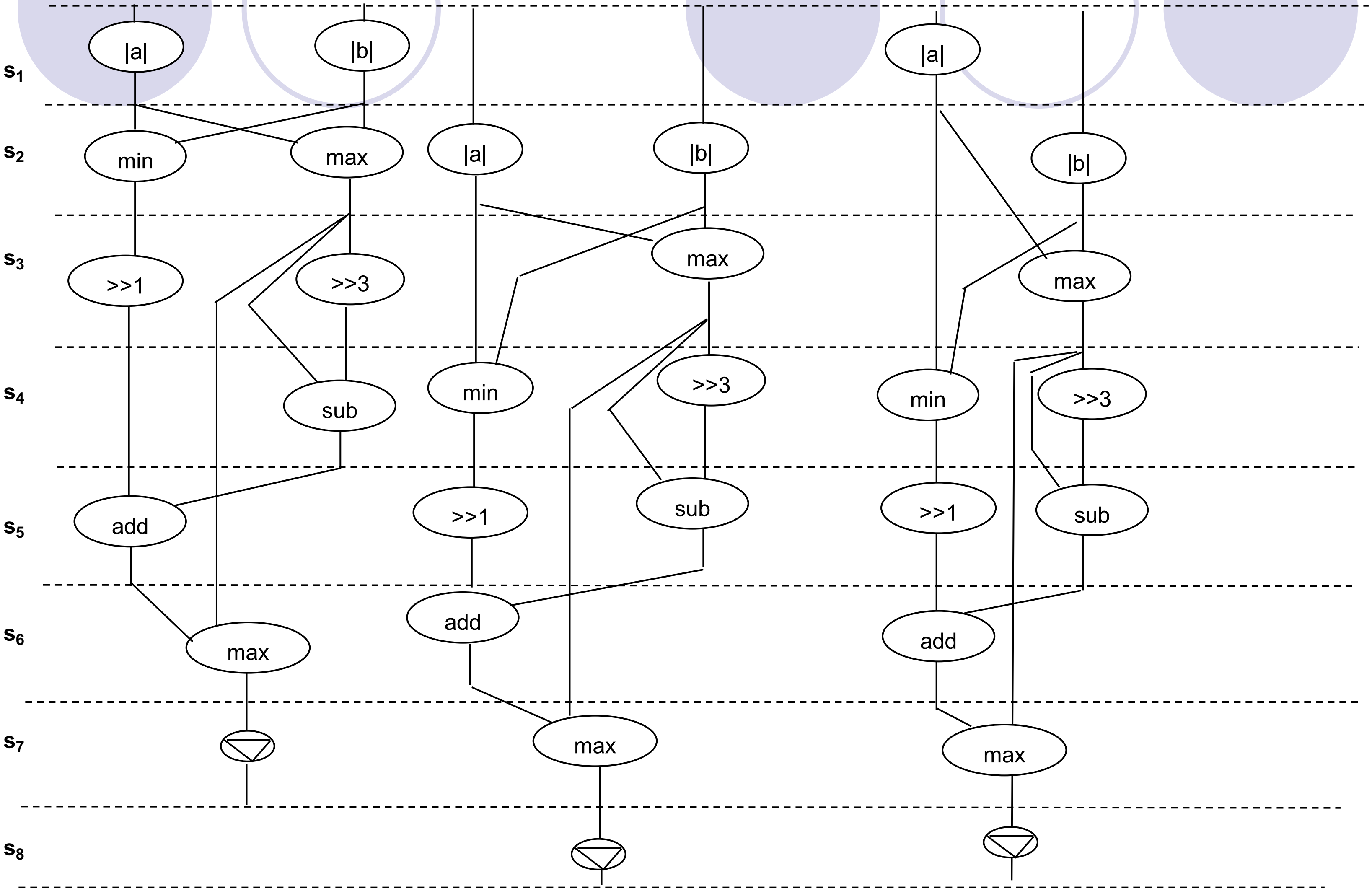
State	AU Units			Probability Sum/state	Shift units		Probability Sum/state
S_1	|a|	|b|		1.0			
S_2				1.00			
S_3	max			1.00			
S_4	min			1.00	>>3		1.00
S_5	sub			1.00	>>1		1.00
S_6	add			1.00			
S_7	max			1.00			

TC scheduling - RSA Example

Final schedule

State	AU Units		Probability Sum/state	Shift units		Probability Sum/state
S_1	|a|		1.0			
S_2	|b|		1.00			
S_3	max		1.00			
S_4	min		1.00	>>3		1.00
S_5	sub		1.00	>>1		1.00
S_6	add		1.00			
S_7	max		1.00			

TC scheduling - RSA Example final schedule



(a) ASAP schedule **(b) ALAP schedule** **(d) TC schedule (Final)**