



**EECS151/251A**

**Spring 2019**

**Digital Design and Integrated  
Circuits**

Instructors:

John Wawrzynek

**Lecture 24**

# Outline

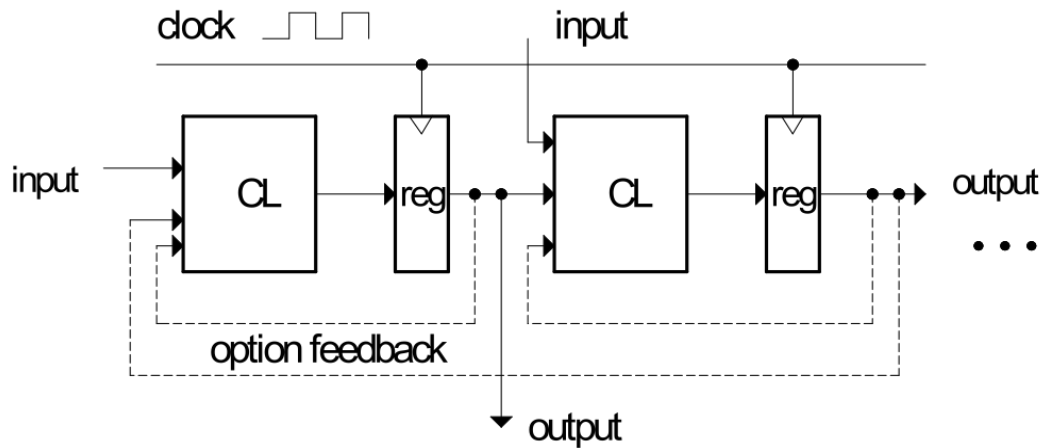


- ❑ *Register Transfer Language*
- ❑ *List Processor Example*
- ❑ *Design Optimization*
- ❑ *Resource Utilization Charts*
- ❑ *Modulo scheduling*



# Register Transfer Language

# Register Transfer Level Review



- ❑ At the high-level we view these systems as a collection of state elements and CL blocks.
- ❑ “RTL” is a commonly used acronym for “Register Transfer Level” description.
- ❑ It follows from the fact that all synchronous digital system can be described as a set of state elements connected by combinational logic blocks.
- ❑ Though not strictly correct, some also use “RTL” to mean the Verilog or VHDL code that describes such systems. 4



# Register Transfer “Language” Descriptions

- ❑ We introduce a language for describing the behavior of systems at the register transfer level.
- ❑ Can view the operation of digital synchronous systems as a set of data transfers between registers with combinational logic operations happening during the transfer.
- ❑ We will avoid using “RTL” to mean “register transfer language.”

*RT Language comprises a set of register transfers with optional operators as part of the transfer.*

*Example:*

```
regA ← regB  
regC ← regA + regB  
if (start==1) regA ← regC
```

*My personal style:*

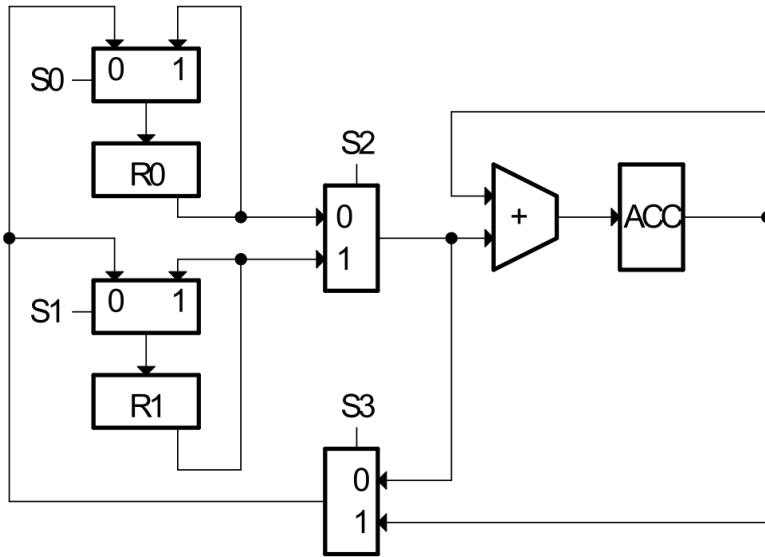
*use “;” to separate transfers that occur on separate cycles.*

*Use “,” to separate transfers that occur on the same cycle.*

*Example (2 cycles):*

```
regA ← regB, regB ← 0;  
regC ← regA;
```

# Example of Using RT Language



$ACC \leftarrow ACC + R0, R1 \leftarrow R0;$   
 $ACC \leftarrow ACC + R1, R0 \leftarrow R1;$   
 $R0 \leftarrow ACC;$

•  
•  
•

- *In this case: RT Language description is used to sequence the operations on the datapath.*
- *It becomes the high-level specification for the controller.*
- *Design of the FSM controller follows directly from the RT Language sequence. FSM controls movement of data by controlling the multiplexor control signals.*

## Example of Using RT Language

- Sometimes RT Language is used as a starting point for designing *both* the datapath and the control:

- example:

regA  $\leftarrow$  IN;

regB  $\leftarrow$  IN;

regC  $\leftarrow$  regA + regB;

regB  $\leftarrow$  regC;

- From this we can deduce:

- IN must fanout to both regA and regB
- regA and regB must output to an adder
- the adder must output to regC
- regB must take its input from a mux that selects between IN and regC

- *What does the datapath look like:*

- *The controller:*



## List Processor Example

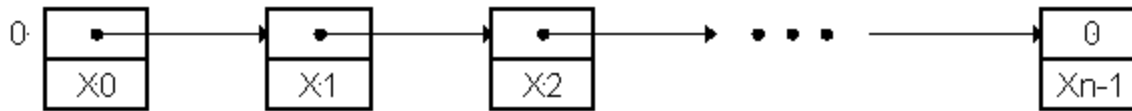


# List Processor Example

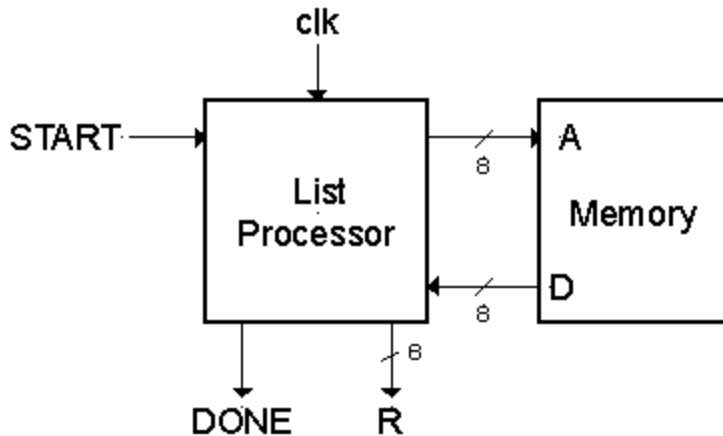
- ❑ RT Language gives us a framework for making high-level optimizations.
  
- ❑ General design procedure outline:
  1. *Problem, Constraints, and Component Library Spec.*
  2. *“Algorithm” Selection*
  3. *Micro-architecture Specification*
  4. *Analysis of Cost, Performance, Power*
  5. *Optimizations, Variations*
  6. *Detailed Design*

# 1. Problem Specification

- Design a circuit that forms the sum of all the 2's complement integers stored in a linked-list structure starting at memory address 0:

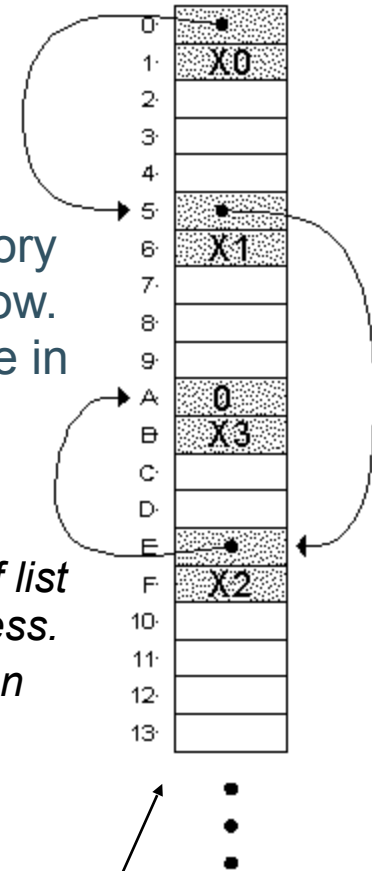


- All integers and pointers are 8-bit. The link-list is stored in a memory block with an 8-bit address port and 8-bit data port, as shown below. The pointer from the last element in the list is 0. At least one node in list.



I/Os:

- START* resets to head of list and starts addition process.
- DONE* signals completion
- R* holds the final result



*Note: We don't assume nodes are aligned on 2 Byte boundaries.*

# 1. Other Specifications

- ❑ Design Constraints:
  - Usually the design specification puts a restriction on cost, performance, power or all. We will leave this unspecified for now and return to it later.
- ❑ Component Library:

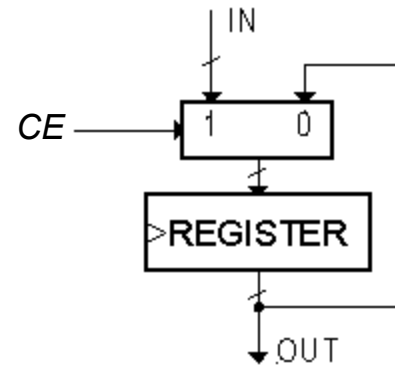
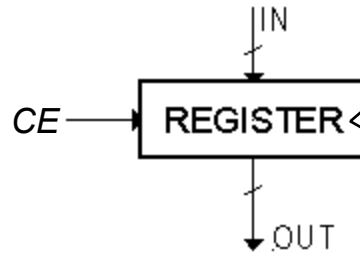
component	delay
simple logic gates	0.5ns
n-bit register	clk-to-Q=0.5ns setup=0.5ns
n-bit 2-1 multiplexor	1ns
n-bit adder	$(2 \log(n) + 2)$ ns
memory	10ns read (asynchronous read)
zero compare	$0.5 \log(n)$

*(single ported memory)*

Are these reasonable?

## Review of Register with “Clock Enable”

- ❑ Register with Clock Enable:



*Functional description only. Transistor level circuit has lower input delay.*

- ❑ Allows register to be either be loaded on selected clock posedge or to retain its previous value.
- ❑ Assume both data and CE require setup time = 0.5ns.
- ❑ Assume no reset input.

# “Algorithm” Selection

- In this case the memory only allows one access per cycle, so the algorithm is limited to sequential execution. If in another case more input data is available at once, then a more parallel solution may be possible.

*Assume datapath state registers NEXT and SUM.*

- NEXT holds a pointer to the node in memory.
- SUM holds the result of adding the node values to this point.

```
If (START==1) NEXT←0, SUM←0, DONE←0;
repeat {
    SUM←SUM + Memory[NEXT+1];
    NEXT←Memory[NEXT];
} until (NEXT==0);
R←SUM, DONE←1;
```

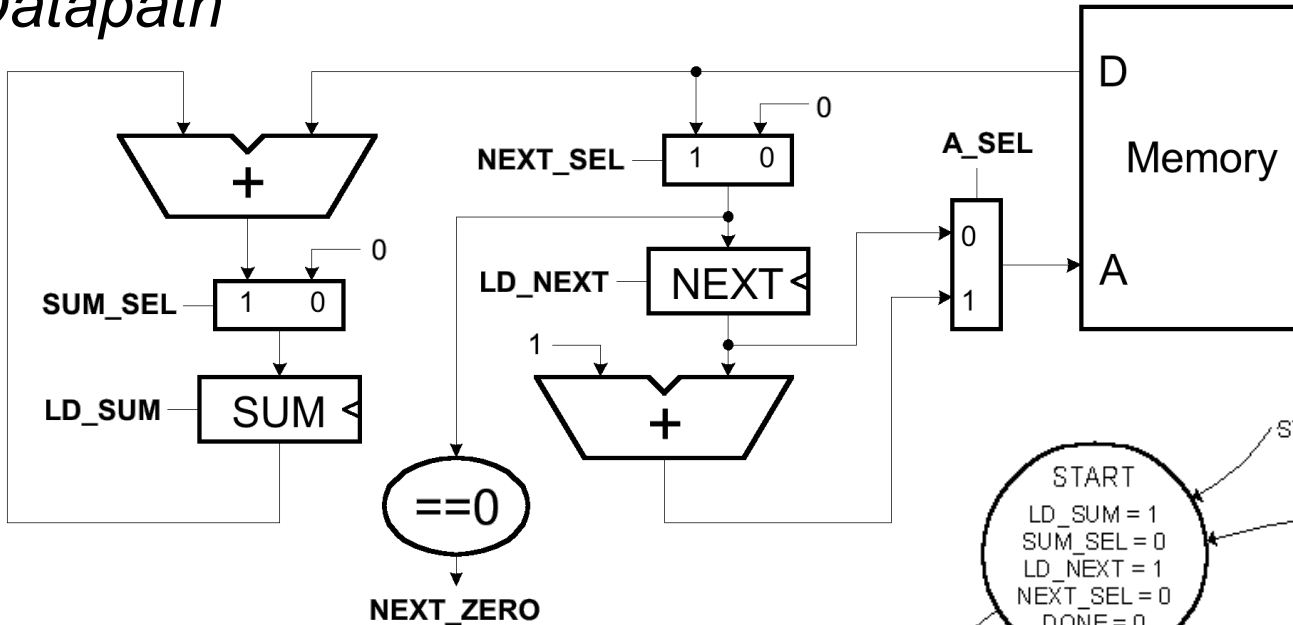
*This RT Language “code” becomes the basis for DP and controller.*



# 3. Micro-Architecture #1

Direct implementation of RTL description:

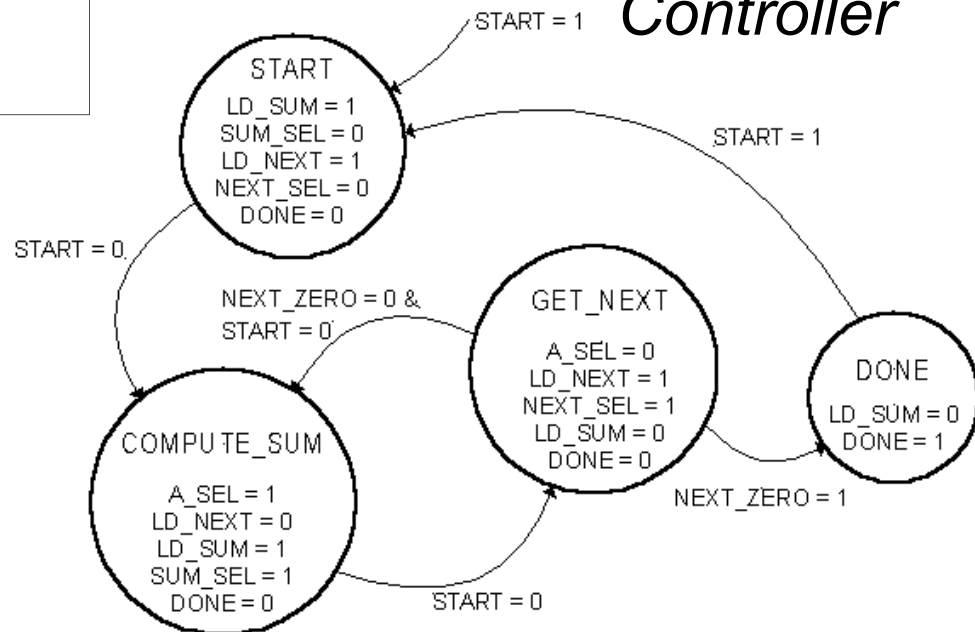
## Datapath



```

If (START==1) NEXT←0, SUM←0;
repeat {
    SUM←SUM + Memory[NEXT+1];
    NEXT←Memory[NEXT];
} until (NEXT==0);
R←SUM, DONE←1;
    
```

## Controller

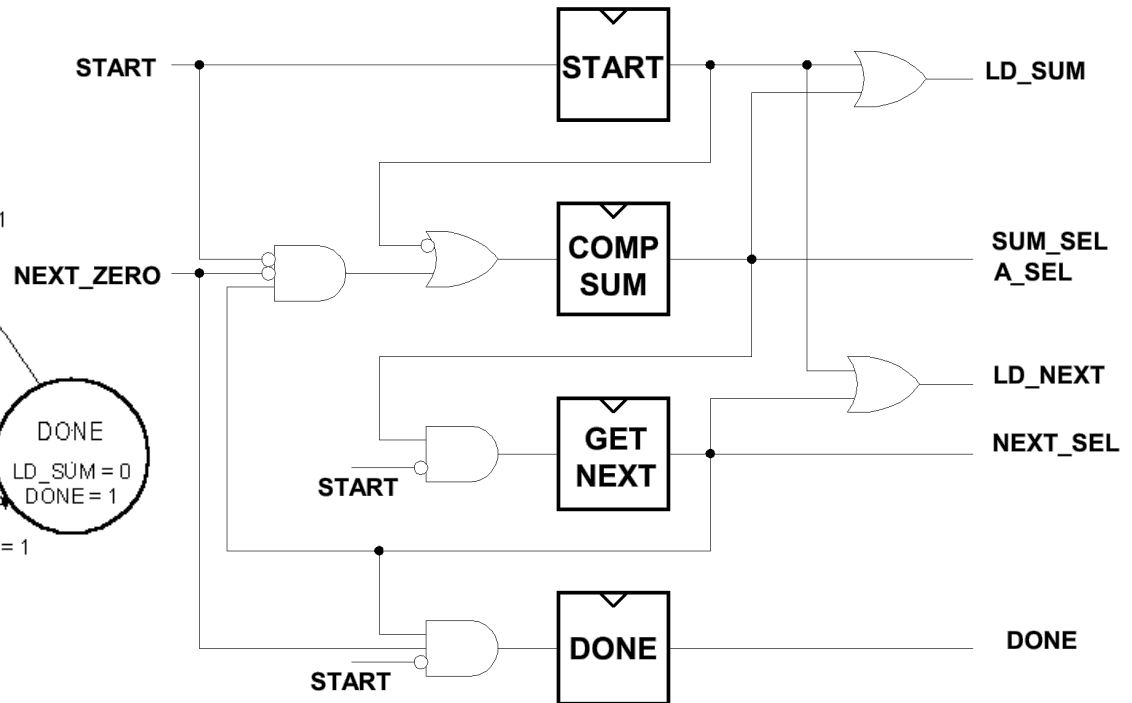
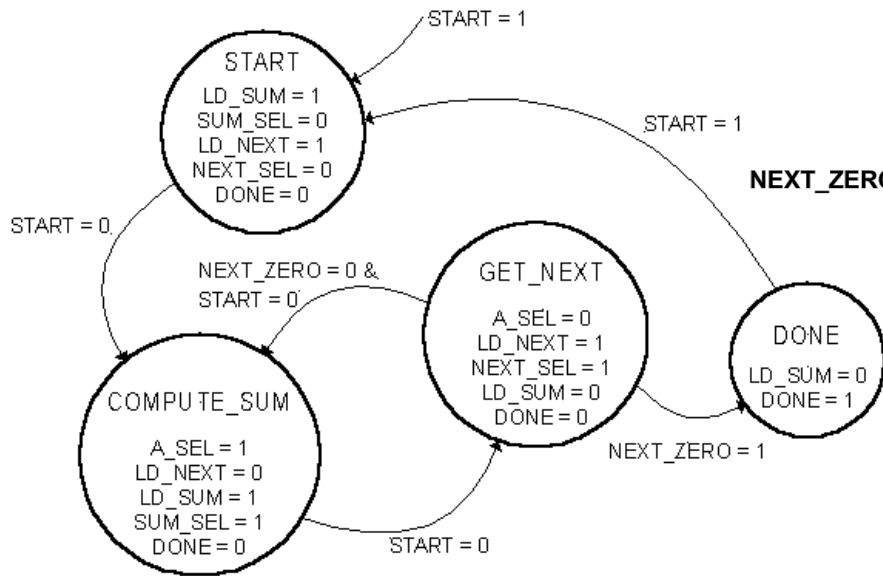


## 4. Analysis of Cost, Performance, and Power

- ❑ Skip Power for now.
- ❑ Cost:
  - How do we measure it? # of transistors? # of gates? # of CLBs?
  - Depends on implementation technology. Often we are just interested in comparing the *relative* cost of two competing implementations. (Save this for later)
- ❑ Performance:
  - 2 clock cycles per number added.
  - What is the minimum clock period?
  - The controller might be on the critical path. Therefore we need to know the implementation, and controller input and output delay.

# Possible Controller Implementation

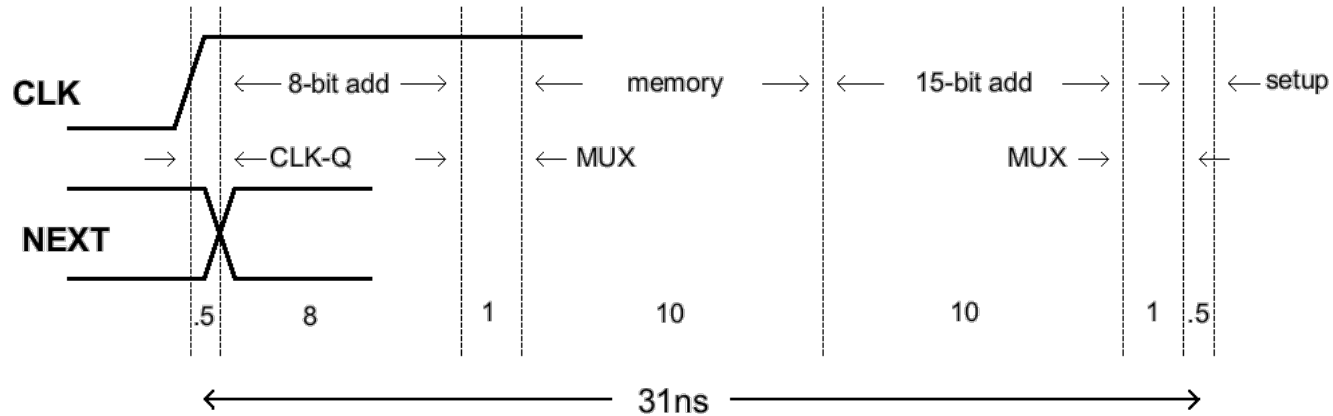
## One-hot FSM



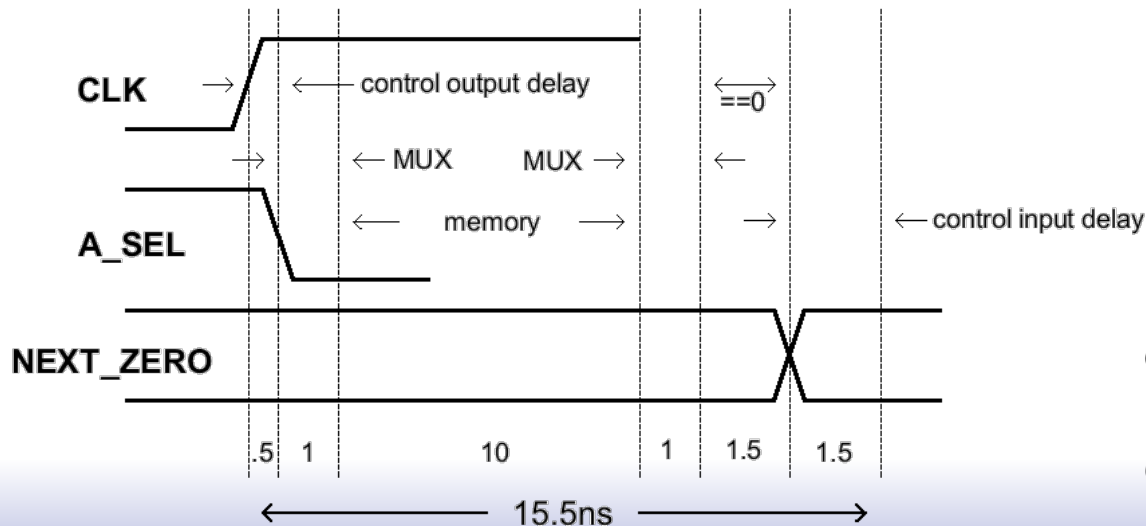
- Based on this, what is the controller input and output delay?

# 4. Analysis of Performance

COMPUTE\_SUM state



GET\_NEXT state



*Other paths exist for each cycle in the loop. These are the worst case.*

## 4. Analysis of Performance

### □ Detailed timing:

clock period (T) = max (clock period for each state)

T > 31ns, F < 32 MHz

### □ Observation:

COMPUTE\_SUM state does most of the work. Most of the components are inactive in GET\_NEXT state.

GET\_NEXT does: Memory access + ...

COMPUTE\_SUM does: 8-bit add, memory access, 15-bit add +  
...

### □ Conclusion:

**Move one of the adds to GET\_NEXT.**





# List Processor Optimization

## 5. Optimization

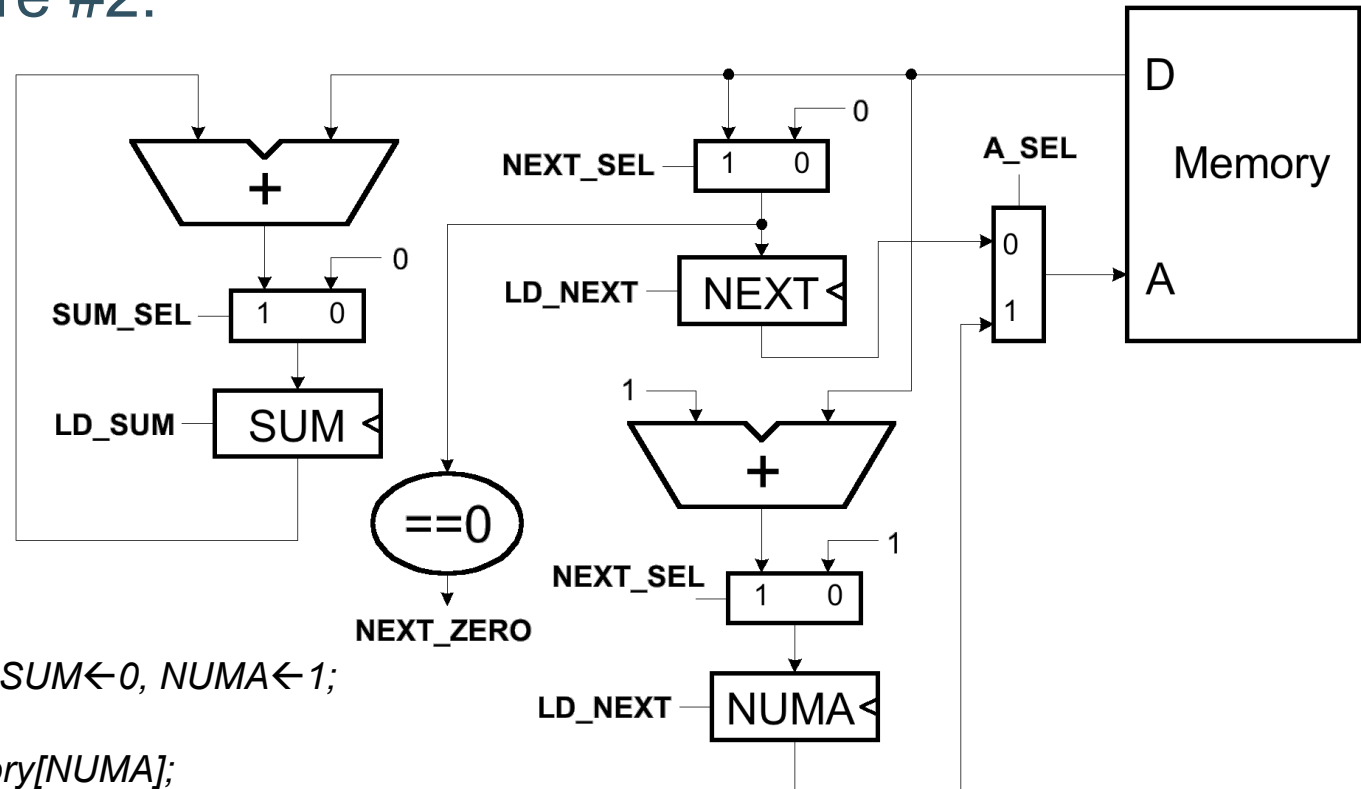
Add new register named NUMA, for *address of number to add*.

- Update code to reflect our change (note still 2 cycles per iteration):

```
If (START==1) NEXT←0, SUM←0, NUMA←1;
repeat {
    SUM←SUM + Memory[NUMA];
    NUMA←Memory[NEXT] + 1, NEXT←Memory[NEXT] ;
} until (NEXT==0);
R←SUM, DONE←1;
```

## 5. Optimization

### Architecture #2:

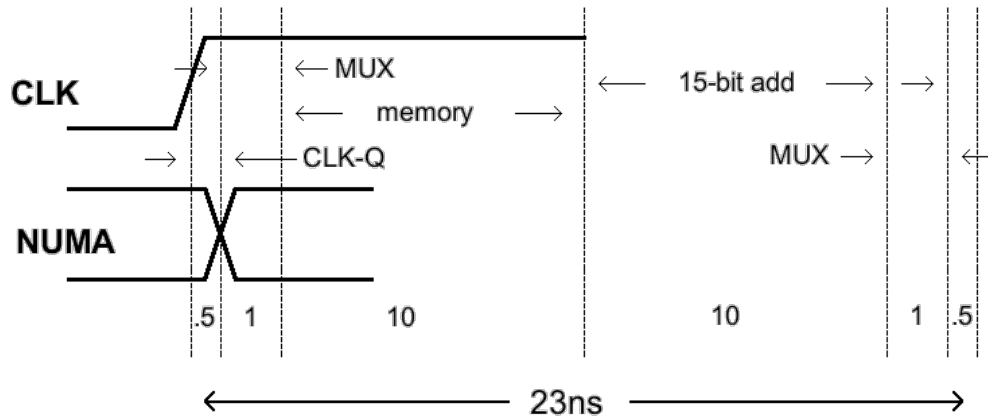


```
If (START==1) NEXT←0, SUM←0, NUMA←1;  
repeat {  
    SUM←SUM + Memory[NUMA];  
    NUMA←Memory[NEXT] + 1, NEXT←Memory[NEXT];  
} until (NEXT==0);  
R←SUM, DONE←1;
```

### Incremental cost: addition of another register and mux.

# 5. Optimization, Architecture #2

## COMPUTE\_SUM state

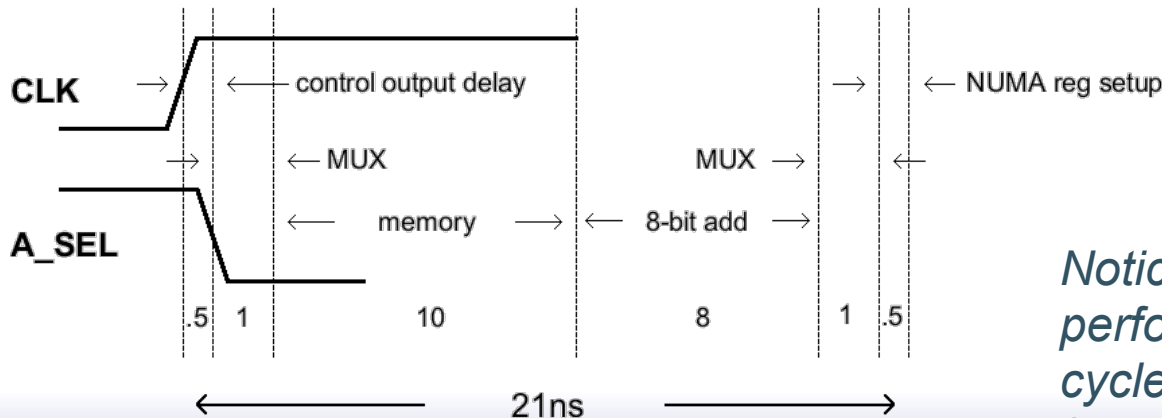


- New timing:  
Clock Period (T) = max (clock period for each state)

$$T > 23\text{ns}, F < 43\text{Mhz}$$

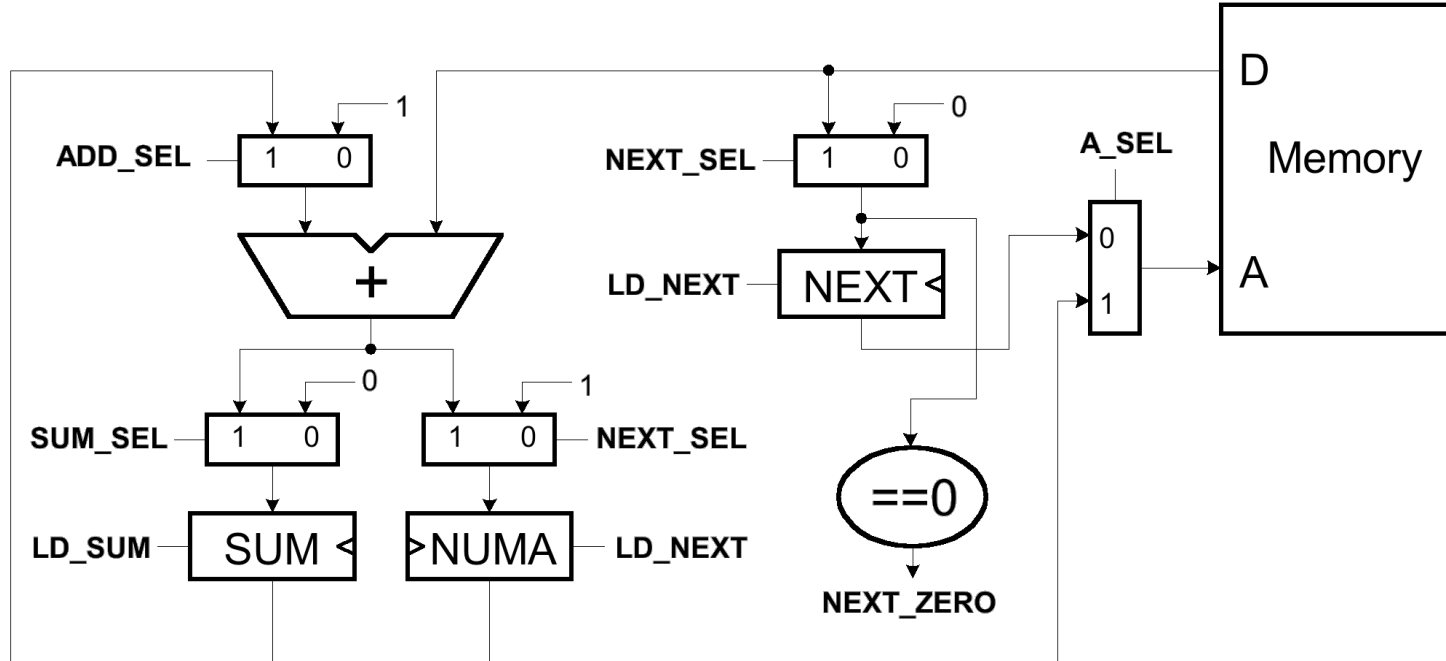
- Is this worth the extra cost?
- Can we lower the cost?

## GET\_NEXT state



*Notice that the circuit now only performs one add on every cycle. Why not share the adder for both cycles?*

## 5. Optimization, Architecture #3



- ❑ Incremental cost:
  - Addition of another mux and control (**ADD\_SEL**). Removal of an 8-bit adder.
- ❑ Performance:
  - No change.
- ❑ Change is definitely worth it.





## Resource Utilization Charts

# Resource Utilization Charts

- ❑ One way to visualize these (and other possible) optimizations is through the use of a *resource utilization charts*.
- ❑ These are used in high-level design to help schedule operations on shared resources.
- ❑ Resources are listed on the y-axis. Time (in cycles) on the x-axis.
- ❑ Example:

<b>memory</b>	fetch A1		fetch A2				
<b>bus</b>		fetch A1		fetch A2			
<b>register-file</b>		read B1		read B2			
<b>ALU</b>			A1+B1		A2+B2		
<i>cycle</i>	1	2	3	4	5	6	7

- ❑ Our list processor has two shared resources: memory and adder

# List Example Resource Scheduling

- Unoptimized solution: 1.  $SUM \leftarrow SUM + Memory[NEXT+1]$ ; 2.  $NEXT \leftarrow Memory[NEXT]$ ;

<b>memory</b>	← fetch x	fetch next	fetch x	fetch next
<b>adder1</b>	next+1		next+1	
<b>adder2</b>	sum		sum	
	1	2	1	2

- *Optimized solution:* 1.  $SUM \leftarrow SUM + Memory[NUMA]$ ;  
2.  $NEXT \leftarrow Memory[NEXT]$ ,  $NUMA \leftarrow Memory[NEXT]+1$ ;

<b>memory</b>	fetch x	fetch next	fetch x	fetch next
<b>adder</b>	sum	numa	sum	numa

- *How about the other combination:* *add x register*

<b>memory</b>	fetch x	fetch next	fetch x	fetch next
<b>adder</b>	numa	sum	numa	sum

1.  $X \leftarrow Memory[NUMA]$ ,  $NUMA \leftarrow NEXT+1$ ;
2.  $NEXT \leftarrow Memory[NEXT]$ ,  $SUM \leftarrow SUM+X$ ;

- *Does this work? If so, a very short clock period. The fetch and the add from each cycle would be independent.  $T = \max(T_{mem}, T_{add})$  instead of  $T_{mem} + T_{add}$ .*

## List Example Resource Scheduling

- Schedule one loop iteration followed by the next:

Memory	$next_1$		$x_1$		$next_2$		$x_2$		
adder		$numa_1$		$sum_1$		$numa_2$		$sum_2$	

- How can we overlap iterations?  $next_2$  depends on  $next_1$ .

– “slide” second iteration into first (4 cycles per result):

Memory	$next_1$		$x_1$	$next_2$		$x_2$			
adder		$numa_1$		$sum_1$	$numa_2$		$sum_2$		

– or further:

Memory	$next_1$	$next_2$	$x_1$	$x_2$	$next_3$	$next_4$	$x_3$	$x_4$	
adder		$numa_1$	$numa_2$	$sum_1$	$sum_2$	$numa_3$	$numa_4$	$sum_3$	$sum_4$

The repeating pattern is 4 cycles. Not exactly the pattern what we were looking for. But does it work correctly?

# List Example Resource Scheduling

- In this case, first spread out, then pack.

Memory	$next_1$			$x_1$		
adder		$numa_1$			$sum_1$	

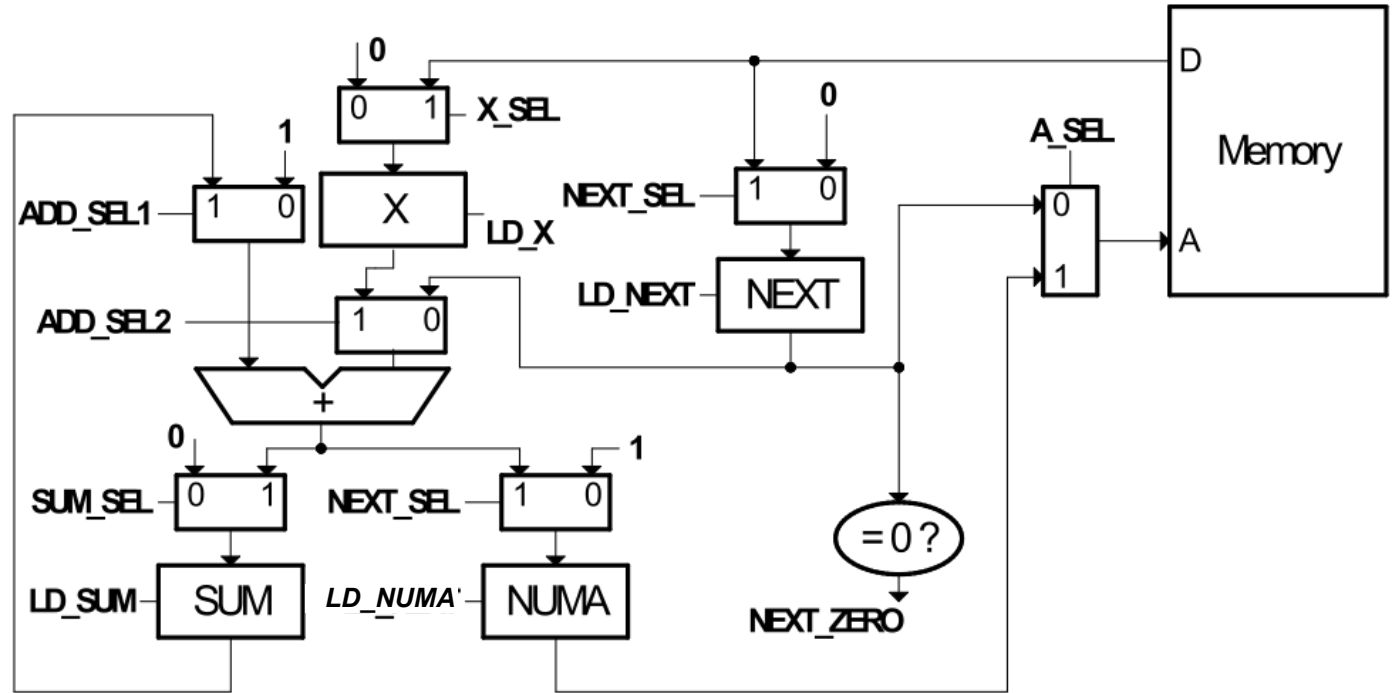
Memory	$next_1$		$next_2$	$x_1$	$next_3$	$x_2$	$next_4$	$x_3$	
adder		$numa_1$		$numa_2$	$sum_1$	$numa_3$	$sum_2$	$numa_4$	$sum_3$

1.  $X \leftarrow \text{Memory}[\text{NUMA}], \text{NUMA} \leftarrow \text{NEXT} + 1;$
2.  $\text{NEXT} \leftarrow \text{Memory}[\text{NEXT}], \text{SUM} \leftarrow \text{SUM} + X;$

- Three different loop iterations active at once.
- Short cycle time (no dependencies within a cycle)
- full utilization (only 2 cycles per result)
- Initialization:  $x=0, numa=1, sum=0, next=\text{memory}[0]$
- Control states (out of the loop)
  - two to start: initialize next, clear sum, set numa, clear x; get  $next_2$
  - two to finish:

## 5. Optimization, Architecture #4

### □ Datapath:



### □ Incremental cost:

- Addition of another register & mux, adder mux, and control.

### □ Performance: find max time of the four actions

1.  $X \leftarrow \text{Memory}[\text{NUMA}]$ ,  $\text{NUMA} \leftarrow \text{NEXT} + 1$ ;  $0.5 + 1 + 10 + 1 + 0.5 = 13\text{ns}$

same for all  $\Rightarrow T > 13\text{ns}, F < 77\text{MHz}$

2.  $\text{NEXT} \leftarrow \text{Memory}[\text{NEXT}]$ ,  $\text{SUM} \leftarrow \text{SUM} + X$ ;

## Other Optimizations

### □ Node alignment restriction:

- If the application of the list processor allows us to restrict the placement of nodes in memory so that they are aligned on even multiples of 2 bytes.
  - NUMA addition can be eliminated.
  - Controller supplies “0” for low-bit of memory address for NEXT, and “1” for X.
- Furthermore, if we could use a memory with a 16-bit wide output, then could fetch entire node in one cycle:

$\{\text{NEXT}, X\} \leftarrow \text{Memory}[\text{NEXT}], \text{SUM} \leftarrow \text{SUM} + X;$

⇒ execution time cut in half (half as many cycles)

# List Processor Conclusions

- ❑ Through careful optimization:
  - clock frequency increased from 32MHz to 77MHz
  - little cost increase.
- ❑ “Scheduling” was used to overlap and to maximize use of resources.
- ❑ Essentially through pipelining the operations (the extra added registers - NUMA, X - act as pipeline registers.
- ❑ Questions:
  - Consider the design process we went through:
    - Could a computer program go from RTL description to circuits automatically?
    - Could a computer program derive the optimizations that we did?
    - It is the goal of “High-Level Synthesis” to do similar transformations and automatic mappings. “C-to-gates” compilers are an example.



# Modulo Scheduling

- Review of list processor scheduling:

Memory	next <sub>1</sub>			x <sub>1</sub>		
adder		numa <sub>1</sub>			sum <sub>1</sub>	

- How did we know to “spread” out the schedule of one iteration to allow efficient packing?

Memory	next <sub>1</sub>		next <sub>2</sub>	x <sub>1</sub>	next <sub>3</sub>	x <sub>2</sub>	next <sub>4</sub>	x <sub>3</sub>	
adder		numa <sub>1</sub>		numa <sub>2</sub>	sum <sub>1</sub>	numa <sub>3</sub>	sum <sub>2</sub>	numa <sub>4</sub>	sum <sub>3</sub>

- The goal of *modulo scheduling* is to find the schedule for one *characteristic section* of the computation. This is the part the control loops over.
- The entire schedule can then be derived, by repeating the characteristic section and repeating it with some pieces omitted for starting and ending.

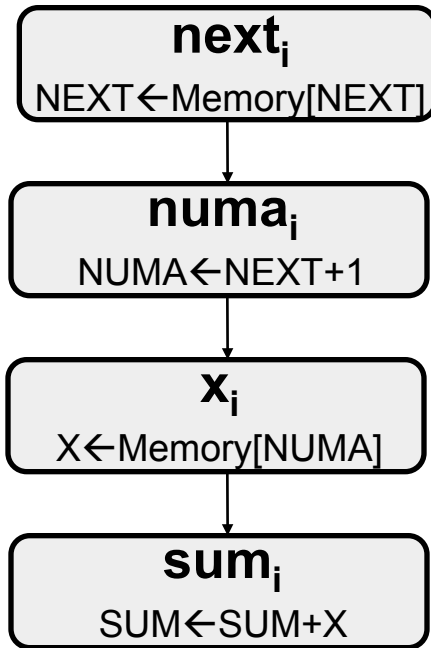
# Modulo Scheduling Procedure

1. Calculate *minimal length of characteristic section*.

The maximum number of cycles that any one resource is used during one iteration of the computation (assuming a resource can only be used once per cycle).

2. Schedule one iteration of the computation on the characteristic section wrapping around when necessary. Each time the computation wraps around, decrease the iteration subscript by one.
3. If iteration will not fit on minimal length section, increase section by one and try again.

# Modulo Scheduling List Processor



- Assuming a single adder and a single ported memory. Minimal schedule section length = 2. Because both memory and adder are used for 2 cycles during one iteration.

memory	next <sub>i</sub>	
adder		numa <sub>i</sub>

memory	next <sub>i</sub>	X <sub>i-1</sub>
adder		numa <sub>i</sub>

wrap-around,  
decrease subscript

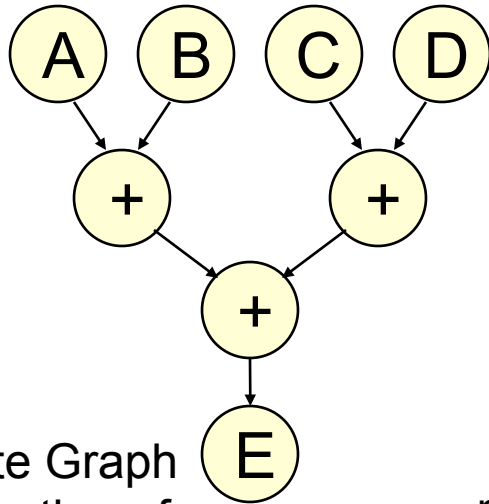
memory	next <sub>i</sub>	X <sub>i-1</sub>
adder	sum <sub>i-2</sub>	numa <sub>i</sub>

wrap-around,  
decrease subscript

- Finished schedule for 4 iterations:

Memory	next <sub>1</sub>		next <sub>2</sub>	x <sub>1</sub>	next <sub>3</sub>	x <sub>2</sub>	next <sub>4</sub>	x <sub>3</sub>	
adder		numa <sub>1</sub>		numa <sub>2</sub>	sum <sub>1</sub>	numa <sub>3</sub>	sum <sub>2</sub>	numa <sub>4</sub>	sum <sub>3</sub>

# Another Scheduling Example



Compute Graph  
(one iteration of a  
repeating calculation)

- Assume A, B, C, D, E stored in a dual port memory.
- Assume a single adder.
- Minimal schedule section length = 3.  
(Both memory and adder are used for 3 cycles during one iteration.)

memory port 1	load A	load C	
memory port 2	load B	load D	store E
adder	E =	A + B	C + D

Repeating schedule:

load A	load C		load A	load C		load A	load C	
load B	load D	store E	load B	load D	store E	load B	load D	store E
E =	A + B	C + D	E =	A + B	C + D	E =	A + B	C + D