EECS150 - Digital Design
Lecture 8 – High Level Design (Part 2) Multipliers (part 1): Design Example

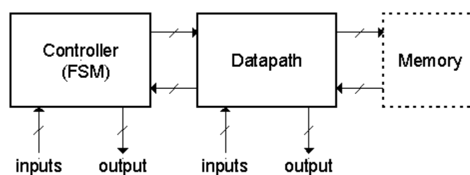
Sep. 24, 2013
Prof. Ronald Fearing
Electrical Engineering and Computer Sciences
University of California, Berkeley

(slides courtesy of Prof. John Wawrzynek)

http://www-inst.eecs.berkeley.edu/~cs150

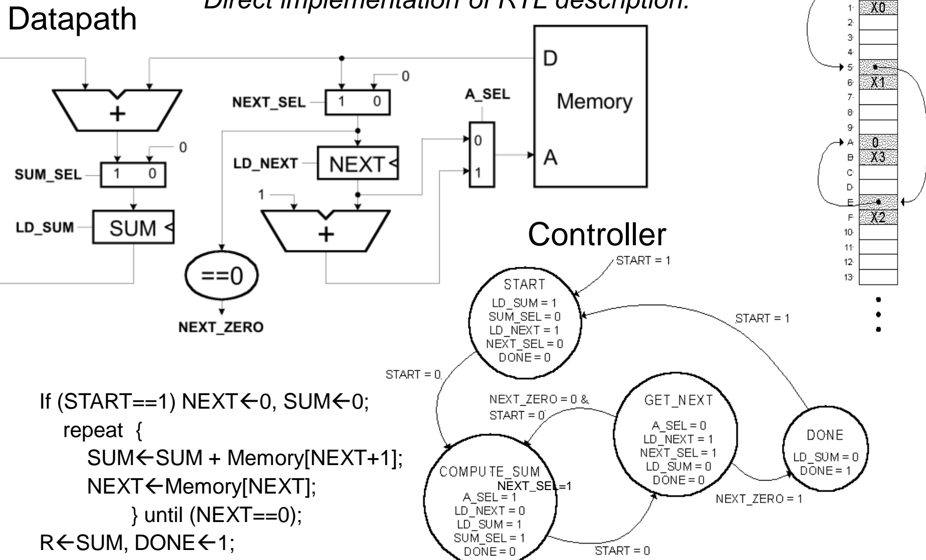
Outline

- Recap: high level design intro



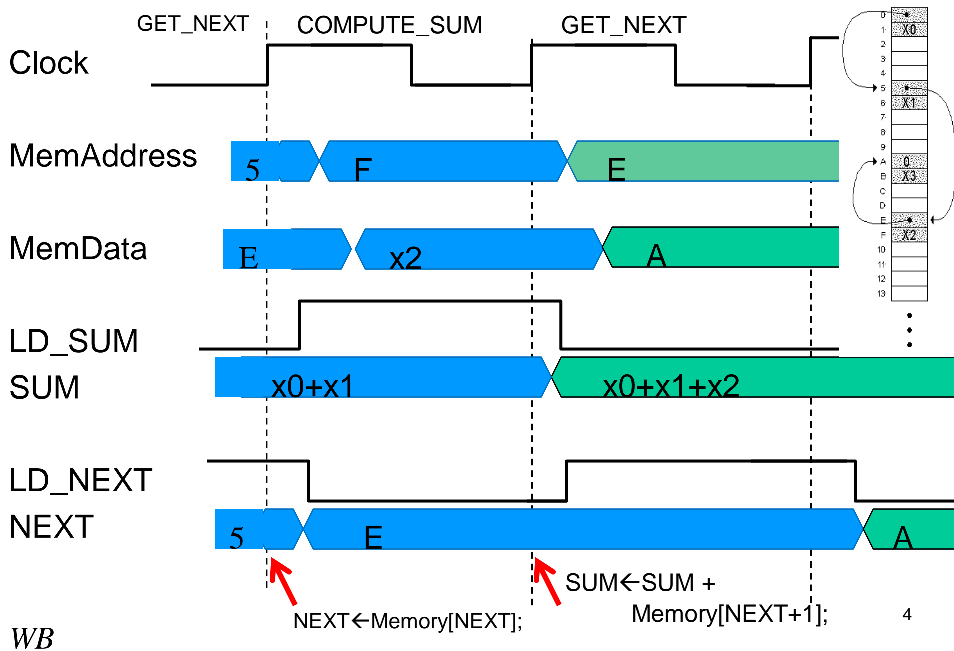
- List processor, conclusion and optimization
- Multiplier example

List Processor Architecture #1

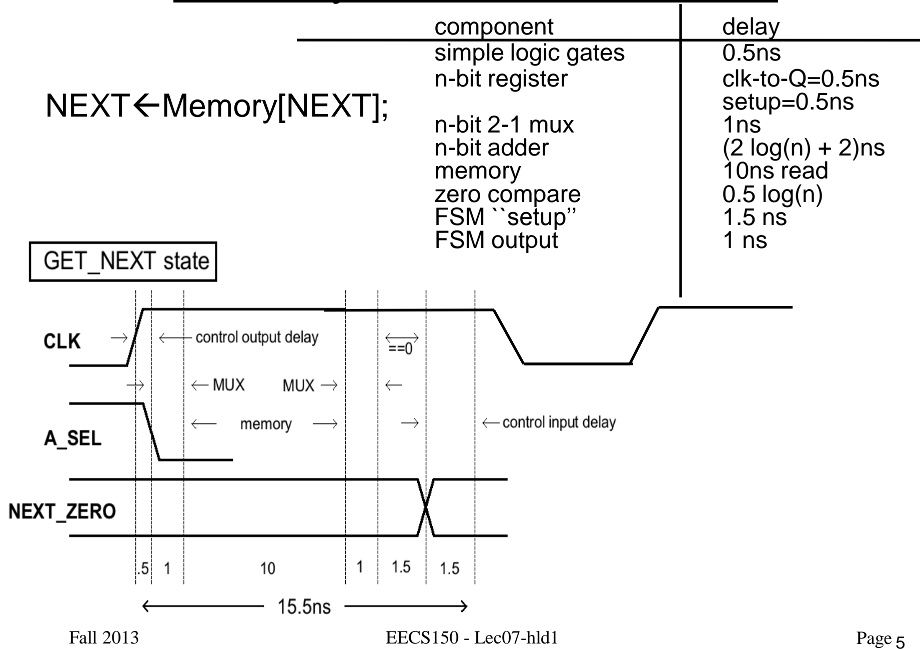


Fall 2013 EECS150 - Lec07-hld1 Page 3

operational timing diagram for list processor



4. Analysis of Performance



4. Analysis of Performance

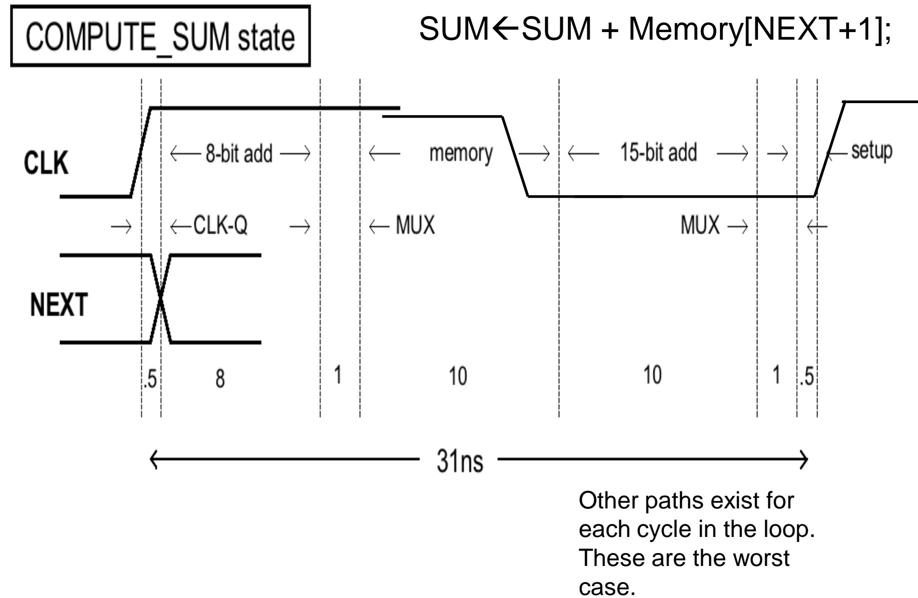
SUM←SUM + Memory[NEXT+1];

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

?

6

4. Analysis of Performance



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

4. Analysis of Performance

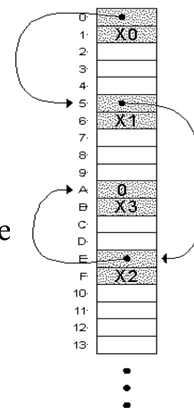
- Detailed timing:
  clock period (T) = max (clock period for each state)
  T > 31ns, F < 32 MHz
- Observation:
  COMPUTE_SUM state does most of the work. Most of the components are inactive in GET_NEXT state.
  GET_NEXT does: Memory access + …
  COMPUTE_SUM does: 8-bit add, memory access, 15-bit add + …
- Conclusion:
  Move one of the adds to GET_NEXT.

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):

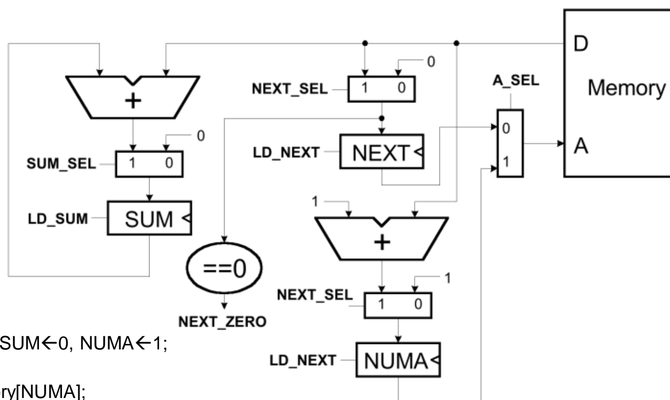
```
If (START==1) NEXT←0, SUM←0, NUMA←1;
repeat {
        SUM←SUM + Memory[NUMA];
        NUMA←Memory[NEXT] + 1,
        NEXT←Memory[NEXT] ;
        } until (NEXT==0);
R←SUM, DONE←1;
```

} one cycle



5. Optimization

- Architecture #2:

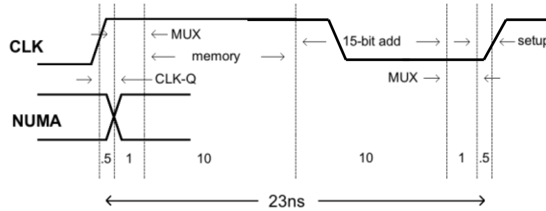



```
If (START==1) NEXT←0, SUM←0, NUMA←1;
  repeat {
    SUM←SUM + Memory[NUMA];
    NUMA←Memory[NEXT] + 1, NEXT←Memory[NEXT] ;
    } until (NEXT==0);
R←SUM, DONE←1;
```

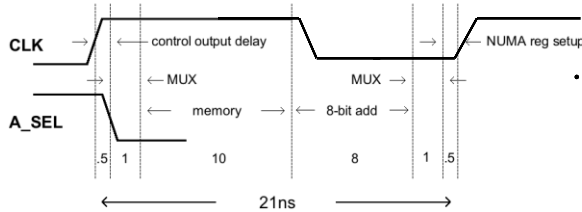
- Incremental cost: addition of another register and mux.

5. Optimization, Architecture #2

COMPUTE_SUM state



GET_NEXT state



- New timing:

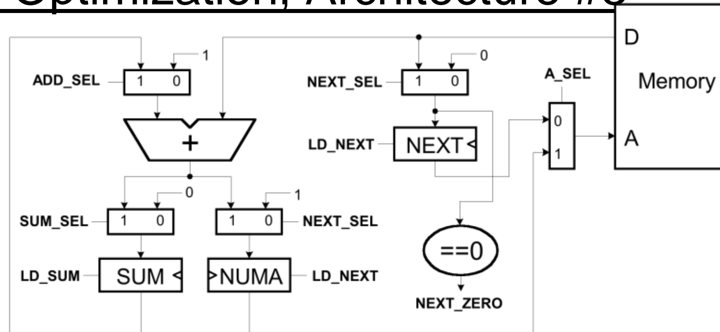
Clock Period (T) = max (clock period for each state)

T > 23ns, F < 43Mhz

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

- Notice that the circuit now only performs one add operation per cycle, but have two adders. Why not share 1 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.

[Except wiring is more limit than computation…]

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.
- CPU Example:

memory	fetch A1		fetch A2				
bus		fetch A1		fetch A2			
register-file		read B1		read B2			
ALU			A1+B1		A2+B2		
cycle	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←SUM + Memory[NEXT+1]; 2. NEXT←Memory[NEXT];

memory	fetch x	fetch next	fetch x	fetch next
adder1	next+1		next+1	
adder2	sum		sum	
	1	2	1	2

- Optimized solution: 1. SUM←SUM + Memory[NUMA];
  2. NEXT←Memory[NEXT], NUMA←Memory[NEXT]+1;

memory	fetch x	fetch next	fetch x	fetch next
adder	sum	numa	sum	numa

- How about the other combination: add x register

memory	fetch x	fetch next	fetch x	fetch next
adder	numa	sum	numa	sum

1. X←Memory[NUMA], NUMA←NEXT+1;
2. NEXT←Memory[NEXT], SUM←SUM+X;

- Does this work? If so, a very short clock period. Each cycle could have *independent* fetch and add. $T = \max(T_{mem}, T_{add})$ 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	

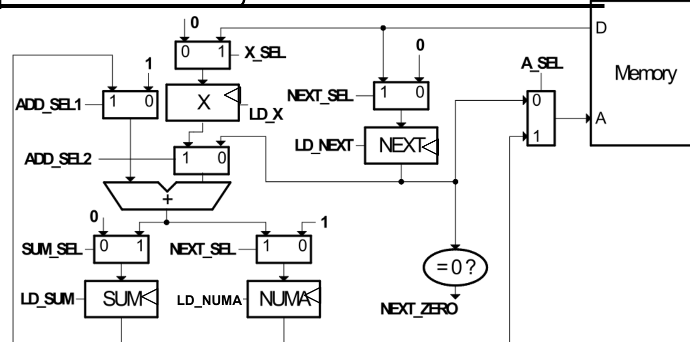
Memory	$next_1$		$next_2$	x_1	$next_3$	x_2	$next_4$	x_3	
adder		$numa_1$		$numa_2$	sum_1	$numa_3$	sum_2	$numa_4$	sum_3

1. X←Memory[NUMA], NUMA←NEXT+1;
2. NEXT←Memory[NEXT], SUM←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=memory[0]
- Extra control states (out of the loop)
  - one to initialize next, clear sum, set numa
  - one to finish off. 2 cycles after next==0.

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←Memory[NUMA], NUMA←NEXT+1; 0.5+1+10+1+0.5 = 13ns
   same for all ➔ T>13ns, F<77MHz
2. NEXT←Memory[NEXT], SUM←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:

{NEXT, X} ← Memory[NEXT], SUM ← SUM + X;

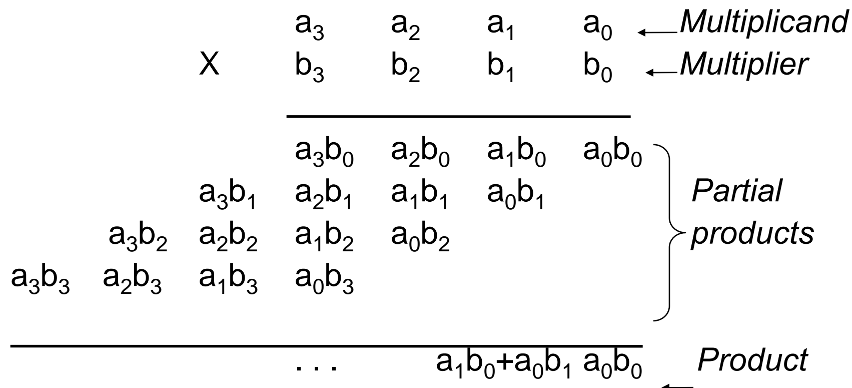
➔ execution time cut in half (half as many cycles)

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.
- 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.

Multiplier Example

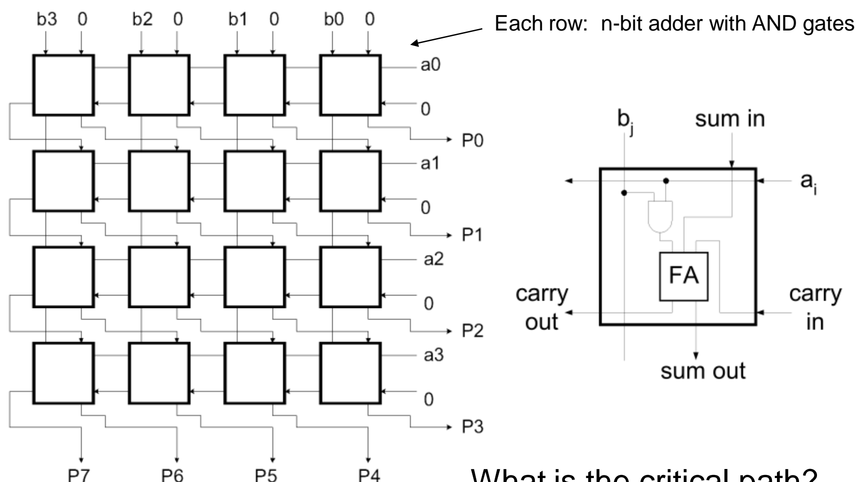
Multiplication



Many different circuits exist for multiplication.
Each one has a different balance between speed (performance) and amount of logic (cost).

Array Multiplier

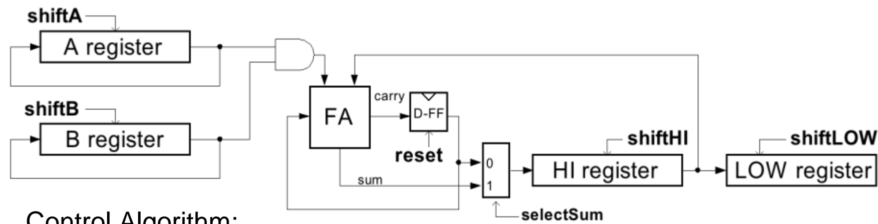
Single cycle multiply: Generates all n partial products simultaneously.



What is the critical path?

Bit-Serial Multiplier

- Example, Bit-serial multiplier (n^2 cycles, one bit of result per n cycles):



- Control Algorithm:

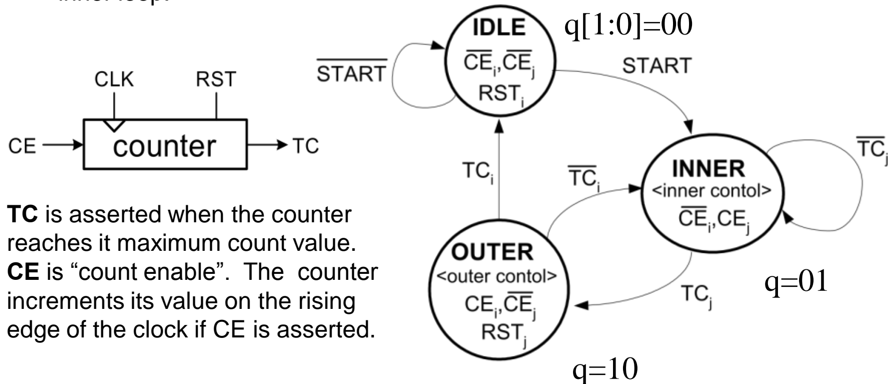
```
repeat n cycles {   // outer (i) loop
       repeat n cycles{    // inner (j) loop
              shiftA, selectSum, shiftHI
       }
       shiftB, shiftHI, shiftLOW, reset
}
```

Note: The occurrence of a control signal x means x=1. The absence of x means x=0.

Controller using Counters

- **State Transition Diagram:**
  - Assume presence of two binary counters. An “i” counter for the outer loop and “j” counter for inner loop.

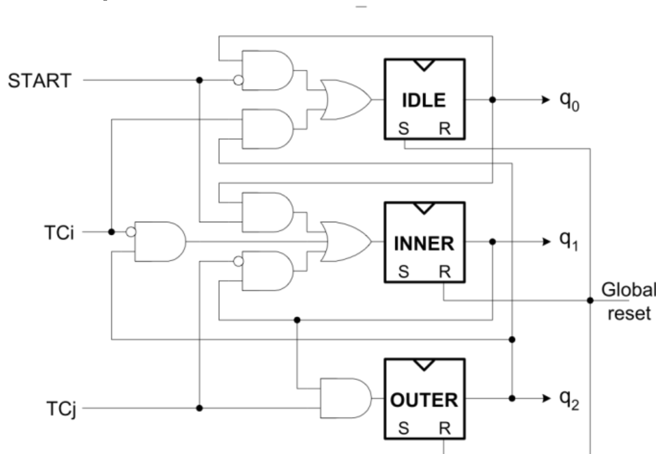
```
repeat n cycles {   // outer (i) loop
       repeat n cycles{    // inner (j) loop
              shiftA, selectSum, shiftHI
       }
       shiftB, shiftHI, shiftLOW, reset
}_
```



TC is asserted when the counter reaches it maximum count value. **CE** is “count enable”. The counter increments its value on the rising edge of the clock if CE is asserted.

Controller using Counters

- **Controller circuit implementation:**



- Outputs:

$CE_i = q_2$
$CE_j = q_1$
$RST_i = q_0$
$RST_j = q_2$

shiftA = q_1
shiftB = q_2
shiftLOW = q_2
shiftHI = $q_1 + q_2$
reset = q_2
selectSUM = q_1

Conclusions

- List processor: scheduling of resources and minimizing combinational delay stages
- binary multiplier: full parallel vs complete serial