List Processor Example

- 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:

  ![Diagram of a list processor](image)

  - 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 above. The pointer from the last element in the list is 0.

  I/Os:
  - START resets to head of list and starts addition process.
  - DONE signals completion
  - R, Bus that holds the final result

Architecture #1

Direct implementation of RTL description:

- T > 32ns, F < 31 MHz

Cycle Optimized Version

- Architecture #2:
  - Incremental cost: addition of another register.

- T > 24ns, F < 41.67 Mhz
Cycle Optimized with reduced cost

- **Datapath:**
  
  - Incremental cost:
    - Addition of another mux and control. Removal of an 8-bit adder.
  
  - Performance:
    - mux adds 1ns to cycle time. 25ns, 40MHz.
  
- Is the cost savings worth the performance degradation?

Resource Utilization Charts

- One way to visualize these (and other possible) optimizations is through the use of a resource utilization chart.
- 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:

<table>
<thead>
<tr>
<th>cycle</th>
<th>memory</th>
<th>bus</th>
<th>register-file</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fetch A1</td>
<td>fetch A1</td>
<td>read B1</td>
<td>A1+B1</td>
</tr>
<tr>
<td>2</td>
<td>fetch A2</td>
<td>fetch A2</td>
<td>read B2</td>
<td>A2+B2</td>
</tr>
</tbody>
</table>

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, sum1, sum1
    - adder2: sum, sum, numa, numa

- 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, numa, numa, numa

- 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

- First schedule one loop iteration:

<table>
<thead>
<tr>
<th>Memory</th>
<th>x1</th>
<th>next1</th>
<th>numa1</th>
<th>x2</th>
<th>numa2</th>
<th>sum1</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa</td>
<td>numa2</td>
<td>numa2</td>
<td>sum</td>
<td>numa3</td>
<td>numa3</td>
</tr>
</tbody>
</table>

- How can we overlap iterations? next2 depends on next1.
  - “slide” second iteration into first:

    | Memory | x1 | next1 | numa1 | numa2 | numa3 | x2 | numa4 | sum2 |
    |--------|----|-------|-------|-------|-------|----|-------|------|
    | adder  | numa| numa3 | numa3 | sum2 | numa4 | numa4 | sum2 |

- or further:

    | Memory | x1 | next1 | next2 | numa1 | numa2 | numa3 | numa4 | x2 | numa5 | sum3 |
    |--------|----|-------|-------|-------|-------|-------|-------|----|-------|------|
    | adder  | numa| numa3 | numa3 | numa3 | numa4 | numa4 | numa4 | sum| numa4 | numa5 |

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.

<table>
<thead>
<tr>
<th>Memory</th>
<th>next, numa, x, sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa, sum</td>
</tr>
</tbody>
</table>

1. X<i>Memory[NUMA], NUMA<i>NEXT+1;  
2. NEXT<i>Memory[NEXT], SUM<i>SUM+X;  

- Three different loop iterations active at once.
- Short cycle time (no dependencies within a cycle)  
- Full utilization
- Initialization: x=0, numa=1, sum=0, next=memory[0]
- Extra control states (out of the loop)  
  - one to initialize next  
  - one to finish off. 2 cycles after next==0.

5. Optimization, Architecture #4

- Datapath:

<table>
<thead>
<tr>
<th>Memory</th>
<th>next, numa, x, sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa, sum</td>
</tr>
</tbody>
</table>

- Incremental cost:
  - Addition of another register & mux, adder mux, and control.
- Performance: find max time of the four actions  
  1. X<i>Memory[NUMA], 0.5+1+10+1+1+0.5 = 14ns  
  2. NEXT<i>Memory[NEXT], same for all T=14ns, F=71MHz

Controller Timing

- In these analyses we assumed that the FSM would not effect the timing and therefore the performance. Is this correct?
- Output timing. We assumed that control signals would arrive at the datapath clk-to-Q delay after the clock edge.  
  - This delay could be more, depending on complexity of output CL.  
  - Simpler (less delay) for one-hot-encoding.
- Next-state timing. Absent inputs, next-state is only a function of present state. Have entire clock period to compute next-state. For simple controller and relatively complex datapath, the controller will not be the critical path.  
  - What about input? In our example, FSM takes input from next_zero signal.  
  - How does this effect timing? A potentially critical timing path is formed:  
    - =0? block → next-state logic  
    - In this case, =0? block is fast (1.5ns), next-state logic is simple ⇒ not the limiting path.

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 31MHz to 71MHz
  - little (if any) 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?

Modulo Scheduling

- Review of list processor scheduling:
  - We “spread” out the schedule of one iteration to allow efficient packing?
  - 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 or repeating it with missing pieces.

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.
  - Both memory and adder are used for 2 cycles during one iteration.
    - Assuming a single adder and a single ported memory. Minimal schedule section length = 2.
      - Both memory and adder are used for 2 cycles during one iteration.
        - memory
          - next
          - numa
          - x
          - numa + x
        - adder
          - x
          - numa + x
        - memory
          - next
          - numa
          - wrap-around, decrease subscript
          - adder
            - numa
            - wrap-around, decrease subscript
          - memory
            - next
            - numa
            - wrap-around, decrease subscript
            - adder
              - numa
              - wrap-around, decrease subscript

Finished schedule for 4 iterations:

- Memory
  - next
    - numa
    - x
    - numa + x
  - adder
    - numa
    - numa + x
  - Memory
    - next
      - numa
      - x
      - numa + x
    - adder
      - numa
      - numa + x
    - Memory
      - next
        - numa
        - x
        - numa + x
      - adder
        - numa
        - numa + x
      - Memory
        - next
          - numa
          - x
          - numa + x
        - adder
          - numa
          - numa + x
Scheduling Example

- 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.)

Repeating schedule:

<table>
<thead>
<tr>
<th>Load</th>
<th>Load</th>
<th>Load</th>
<th>Load</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>B</td>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>E = A + B</td>
<td>C + D</td>
<td>E = A + B</td>
<td>C + D</td>
<td>E = A + B</td>
</tr>
</tbody>
</table>

Memory Port 1:
- load A
- load B
- load C
- load D
- store E

Memory Port 2:
- load A
- load B
- load C
- load D
- store E

Adder:
- E = A + B
- E = C + D