1. Problem Specification

• Design a circuit that forms the sum of all the 2’s complements 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.

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

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:
  
<table>
<thead>
<tr>
<th>Component</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-bit register</td>
<td>clk-t0-C=0.5ns setup=0.5ns</td>
</tr>
<tr>
<td>2-bit s-1 multiplexor</td>
<td>1ns</td>
</tr>
<tr>
<td>n-bit adder</td>
<td>(2 log(n) + 2) ns</td>
</tr>
<tr>
<td>memory</td>
<td>10ns read (asynchronous read)</td>
</tr>
<tr>
<td>zero compare</td>
<td>0.5 log(n)</td>
</tr>
</tbody>
</table>

Are these reasonable?

New Component

• Register with Load Enable:

  • Allows register to be loaded on selected clock posedge or to retain its previous value.

2. Algorithm Specification

• 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.

If (START==1) NEXT=0, SUM=0;
    do |
        SUM=SUM + Memory[NEXT+1];
        NEXT=Memory[NEXT];
    while (NEXT==0)
    R=SUM, DONE=1;

Note: “until” replaced by “do while”

3. Architecture #1

Direct implementation of RTL description:
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. Usually we are 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?
  – Detailed timing next page:

4. Analysis of Performance

• Detailed timing:
  clock period (T) = max (clock period for each state)
  T > 32ns, F < 31 MHz

Assumes that the controller delay does not limit the performance.

• Conclusion:
  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 + …

  Move one of the adds to GET_NEXT.

5. Optimization

• Architecture #2:

• Incremental cost:
  – addition of another clearable, load_enabled register.

5. Optimization, Architecture #2

• New timing:
  Clock Period (T) = max (clock period for each state)
  T > 24ns, F < 41.67MHz

• Is this worth the extra cost?
• Can we lower the cost?
• Notice that the circuit now only performs one add on every cycle. Why not share the adder
  for both cycles?
5. Optimization, Architecture #3

- 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:
- memory
- bus
- register-file
- ALU

List Example Resource Scheduling
- Unoptimized solution: 1. SUM = SUM + Memory[MEMORY]; 2. NEXT = Memory[MEMORY];
- memory
- fetch x
- fetch next
- fetch x
- fetch next
- adder
- next+1
- next+1
- adder2
- sum
- sum
- 1
- 2
- 2
- 2
- 2

- Optimized solution: 1. SUM = SUM + Memory[MEMORY]; 2. NEXT = Memory[MEMORY]+1;
- memory
- fetch x
- fetch next
- fetch x
- fetch next
- adder
- sum
- numa
- numa
- 1
- 2
- 2
- 2
- 2

- How about the other combination: add x register
- memory
- fetch x
- fetch next
- fetch x
- fetch next
- adder
- numa
- numa
- numa
- numa
- 1
- 2
- 2
- 2
- 2

- 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:
- Memory next, x
- adder numa, numa, numa, numa
- How can we overlap iterations? next depends on next.
  - “slide” second iteration into first:
  - or further:
- The repeating pattern is 4 cycles. Need extra registers.

List Example Resource Scheduling
- In this case, first spread out, then pack.
- Memory next, x
- adder numa, numa, numa, numa

List Example Resource Scheduling
- 1. X = Memory[MEMORY], NUMA = NEXT+1; 2. NEXT = Memory[MEMORY], SUM = 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.

List Example Resource Scheduling
- 1. X = Memory[MEMORY], NUMA = NEXT+1; 2. NEXT = Memory[MEMORY], SUM = SUM + X;
- Incremental cost:
  - Addition of another register & mux, adder mux, and control.
- Performance: find max time of the four actions
  - 1 = min(t_{memory} + t_{numa} + t_{add}) instead of t_{memory} + t_{add}.

5. Optimization, Architecture #4

- Datapath:
- Incremental cost:
  - Addition of another register & mux, adder mux, and control.
- Performance: find max time of the four actions
  - same for all T = 1 cycles, f = 1GHz

Spring 2002 EECS150 - Lec23-HL2 Page 13
Spring 2002 EECS150 - Lec23-HL2 Page 14
Spring 2002 EECS150 - Lec23-HL2 Page 15
Spring 2002 EECS150 - Lec23-HL2 Page 16
Spring 2002 EECS150 - Lec23-HL2 Page 17
Spring 2002 EECS150 - Lec23-HL2 Page 18
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 appear 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)

Concluding 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?