Outline

• Recap: high level design intro

• List processor, conclusion and optimization
• Multiplier example
List Processor Architecture #1

Direct implementation of RTL description:

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

operational timing diagram for list processor
4. Analysis of Performance

<table>
<thead>
<tr>
<th>component</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple logic gates</td>
<td>0.5ns</td>
</tr>
<tr>
<td>n-bit register</td>
<td>clk-to-Q=0.5ns</td>
</tr>
<tr>
<td>n-bit 2-1 mux</td>
<td>setup=0.5ns</td>
</tr>
<tr>
<td>n-bit adder</td>
<td>1ns</td>
</tr>
<tr>
<td>memory</td>
<td>(2 \log(n) + 2)ns</td>
</tr>
<tr>
<td>zero compare</td>
<td>10ns read</td>
</tr>
<tr>
<td>FSM &quot;setup&quot;</td>
<td>0.5 log(n)</td>
</tr>
<tr>
<td>FSM output</td>
<td>1.5 ns</td>
</tr>
</tbody>
</table>

NEXT ← Memory[NEXT];

SUM ← SUM + Memory[NEXT+1];

<table>
<thead>
<tr>
<th>component</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple logic gates</td>
<td>0.5ns</td>
</tr>
<tr>
<td>n-bit register</td>
<td>clk-to-Q=0.5ns</td>
</tr>
<tr>
<td>n-bit 2-1 mux</td>
<td>setup=0.5ns</td>
</tr>
<tr>
<td>n-bit adder</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</td>
</tr>
<tr>
<td>zero compare</td>
<td>0.5 log(n)</td>
</tr>
</tbody>
</table>
4. Analysis of Performance

- Detailed timing:
  \[ \text{clock period (T)} = \max (\text{clock period for each state}) \]
  \( T > 31 \text{ns}, F < 32 \text{ MHz} \)

- Observation:
  \[
  \begin{align*}
  \text{COMPUTE}_\text{SUM} \text{ state does most of the work. Most of the components are inactive in GET}_\text{NEXT} \text{ state.} \\
  \text{GET}_\text{NEXT} \text{ does: Memory access + ...} \\
  \text{COMPUTE}_\text{SUM} \text{ does: 8-bit add, memory access, 15-bit add + ...}
  \end{align*}
  \]

- 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):

\[
\begin{align*}
\text{If (START==1) NEXT} & \leftarrow 0, \text{SUM} \leftarrow 0, \text{NUMA} \leftarrow 1; \\
\text{repeat} & \\
\text{SUM} & \leftarrow \text{SUM} + \text{Memory[NUMA]}; \\
\text{NUMA} & \leftarrow \text{Memory[NEXT]} + 1, \\
\text{NEXT} & \leftarrow \text{Memory[NEXT]}; \\
\} & \text{until (NEXT==0)}; \\
R & \leftarrow \text{SUM}, \text{DONE} \leftarrow 1;
\end{align*}
\]

5. Optimization

• Architecture #2:

\[
\begin{align*}
\text{If (START==1) NEXT} & \leftarrow 0, \text{SUM} \leftarrow 0, \text{NUMA} \leftarrow 1; \\
\text{repeat} & \\
\text{SUM} & \leftarrow \text{SUM} + \text{Memory[NUMA]}; \\
\text{NUMA} & \leftarrow \text{Memory[NEXT]} + 1, \\
\text{NEXT} & \leftarrow \text{Memory[NEXT]}; \\
\} & \text{until (NEXT==0)}; \\
R & \leftarrow \text{SUM}, \text{DONE} \leftarrow 1;
\end{align*}
\]

• Incremental cost: addition of another register and mux.
5. Optimization, Architecture #2

- New timing:
  \[ \text{Clock Period (T)} = \max (\text{clock period for each state}) \]
  
  \[ T > 23\text{ns}, F < 43\text{Mhz} \]

- 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:

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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];

<table>
<thead>
<tr>
<th>Resource</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>fetch x</td>
<td>fetch next</td>
</tr>
<tr>
<td>adder1</td>
<td>next+1</td>
<td>sum</td>
</tr>
<tr>
<td>adder2</td>
<td></td>
<td>numa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Resource</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>fetch x</td>
<td>fetch next</td>
</tr>
<tr>
<td>adder</td>
<td>sum</td>
<td>numa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- How about the other combination: add x register

<table>
<thead>
<tr>
<th>Resource</th>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>fetch x</td>
<td>fetch next</td>
</tr>
<tr>
<td>adder</td>
<td>numa</td>
<td>sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th>Memory</th>
<th>next₁</th>
<th>x₁</th>
<th>next₂</th>
<th>x₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>numa₁</td>
<td></td>
<td></td>
<td>numa₂</td>
<td></td>
</tr>
<tr>
<td>sum₁</td>
<td></td>
<td></td>
<td>sum₂</td>
<td></td>
</tr>
</tbody>
</table>

- How can we overlap iterations? next₂ depends on next₁.
  - “slide” second iteration into first (4 cycles per result):

<table>
<thead>
<tr>
<th>Memory</th>
<th>next₁</th>
<th>x₁</th>
<th>next₂</th>
<th>x₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>numa₁</td>
<td></td>
<td></td>
<td>numa₂</td>
<td></td>
</tr>
<tr>
<td>sum₁</td>
<td></td>
<td></td>
<td>sum₂</td>
<td></td>
</tr>
</tbody>
</table>

-or further:

<table>
<thead>
<tr>
<th>Memory</th>
<th>next₁</th>
<th>next₂</th>
<th>x₁</th>
<th>x₂</th>
<th>next₃</th>
<th>x₃</th>
<th>next₄</th>
<th>x₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>numa₁</td>
<td>numa₂</td>
<td></td>
<td>sum₁</td>
<td>sum₂</td>
<td>numa₃</td>
<td>numa₄</td>
<td>sum₃</td>
<td>sum₄</td>
</tr>
</tbody>
</table>

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>next₁</th>
<th>x₁</th>
<th>numa₁</th>
<th>sum₁</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>next₁</th>
<th>next₂</th>
<th>x₁</th>
<th>x₂</th>
<th>next₃</th>
<th>x₃</th>
<th>next₄</th>
<th>x₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>numa₁</td>
<td>numa₂</td>
<td></td>
<td>sum₁</td>
<td>sum₂</td>
<td>numa₃</td>
<td>numa₄</td>
<td>sum₃</td>
<td>sum₄</td>
</tr>
</tbody>
</table>

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. \[X \leftarrow \text{Memory}[\text{NUMA}],\]
   \[
   0.5 + 1 + 10 + 1 + 0.5 = 13\text{ns}
   \]
   \[\text{NUMA} \leftarrow \text{NEXT} + 1;\]
   \[\text{same for all } \Rightarrow T > 13\text{ns}, F < 77\text{MHz}\]

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:

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

\[\Rightarrow \text{execution time cut in half (half as many cycles)}\]
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

\[ \begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
  a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
  a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
  a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
  a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\hline
  \ldots & & a_1b_0 + a_0b_1 & a_0b_0 \\
\end{array} \]

\text{Partial products}

\[ \text{Product} \]

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 (n² 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.

TC is asserted when the counter reaches its 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:

  \[ \begin{align*}
  CE_i &= q_2 \\
  CE_j &= q_1 \\
  RST_i &= q_0 \\
  RST_j &= q_2 \\
  \text{shiftA} &= q_1 \\
  \text{shiftB} &= q_2 \\
  \text{shiftLOW} &= q_2 \\
  \text{shiftHI} &= q_1 + q_2 \\
  \text{reset} &= q_2 \\
  \text{selectSUM} &= q_0 \\
\end{align*} \]

Conclusions

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