Outline

- Register Transfer Language
- List Processor Example
- Design Optimization
- Resource Utilization Charts
Introduction

- **High-level Design** Specifies:
  - How data is moved around and operated on.
  - The architecture (sometimes called micro-architecture):
    - The organization of state elements and combinational logic blocks
    - Functional specification of combinational logic blocks

- **Optimization**
  - Deals with the task of modifying an architecture and data movement procedure to meet some particular design requirement:
    - performance, cost, power, or some combination.

- Most designers spend most of their time on high-level organization and optimization
  - modern CAD tools help fill in the low-level details and optimization
  - gate-level minimization, state-assignment, etc.
  - A great deal of the leverage on effecting performance, cost, and power comes at the high-level.
One Standard High-level Pattern (template)

- **Controller**
  - accepts external and control input, generates control and external output and sequences the movement of data in the datapath. (puppeteer)

- **Datapath**
  - is responsible for data manipulation. Usually includes a limited amount of storage. (puppet)

- **Memory**
  - optional block used for long term storage of data structures.

*Standard model for CPUs, micro-controllers, many other digital sub-systems.*

*Usually not nested.*

*S sometimes cascaded:*
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.
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:

\[
\begin{align*}
\text{regA} &\leftarrow \text{regB} \\
\text{regC} &\leftarrow \text{regA + regB} \\
\text{if (start==1) regA} &\leftarrow \text{regC}
\end{align*}
\]

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):

\[
\begin{align*}
\text{regA} &\leftarrow \text{regB, regB} \leftarrow 0; \\
\text{regC} &\leftarrow \text{regA;}
\end{align*}
\]
Example of Using RT Language

- 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.

\[
\begin{align*}
\text{ACC} & \leftarrow \text{ACC} + \text{R0}, \text{R1} \leftarrow \text{R0}; \\
\text{ACC} & \leftarrow \text{ACC} + \text{R1}, \text{R0} \leftarrow \text{R1}; \\
\text{R0} & \leftarrow \text{ACC};
\end{align*}
\]
**Example of Using RT Language**

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

example:

```
regA ← IN;
regB ← IN;
regC ← regA + regB;
regB ← 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:**

<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, setup=0.5ns</td>
</tr>
<tr>
<td>n-bit 2-1 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>

*(single ported memory)*

Are these reasonable?
Review of Register with “Clock Enable”

- Register with Clock Enable:
  - 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.

Functional description only. Transistor level circuit has lower input delay.
“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.
If (START==1) NEXT←0, SUM←0; 
repeat 
  SUM←SUM + Memory[NEXT+1];
  NEXT←Memory[NEXT];
} until (NEXT==0);
R←SUM, DONE←1;
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

Based on this, what is the controller input and output delay?
4. Analysis of Performance

Other paths exist for each cycle in the loop. These are the worst case.
4. Analysis of Performance

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

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

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

```cpp
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 \((\text{START}==1)\) \(\text{NEXT} \leftarrow 0, \text{SUM} \leftarrow 0, \text{NUMA} \leftarrow 1\);

\[
\text{repeat} \{ \\
\quad \text{SUM} \leftarrow \text{SUM} + \text{Memory[NUMA]}; \\
\quad \text{NUMA} \leftarrow \text{Memory[NEXT]} + 1, \text{NEXT} \leftarrow \text{Memory[NEXT]};
\}
\text{until (NEXT==0);} \\
\text{R} \leftarrow \text{SUM}, \text{DONE} \leftarrow 1;
\]

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

- 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?

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:

<table>
<thead>
<tr>
<th>memory</th>
<th>fetch A1</th>
<th>fetch A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register-file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle</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>memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bus</td>
<td>fetch A1</td>
<td>fetch A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>register-file</td>
<td>read B1</td>
<td>read B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>A1+B1</td>
<td>A2+B2</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. \( \text{SUM} \leftarrow \text{SUM} + \text{Memory}[\text{NEXT}+1] \);  
  2. \( \text{NEXT} \leftarrow \text{Memory}[\text{NEXT}] \);

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

- **Optimized solution:**
  1. \( \text{SUM} \leftarrow \text{SUM} + \text{Memory}[\text{NUMA}] \);
  2. \( \text{NEXT} \leftarrow \text{Memory}[\text{NEXT}], \ \text{NUMA} \leftarrow \text{Memory}[\text{NEXT}]+1 \);

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

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

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

- **Does this work?** If so, a very short clock period. Each cycle could have independent fetch and add. \( T = \max(T_{\text{mem}}, T_{\text{add}}) \) instead of \( T_{\text{mem}} + T_{\text{add}} \).
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>adder</td>
<td>numa₁</td>
<td>sum₁</td>
<td>numa₂</td>
<td>sum₂</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>adder</td>
<td>numa₁</td>
<td>sum₁</td>
<td>numa₂</td>
<td>sum₂</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>next₄</th>
<th>x₃</th>
<th>x₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa₁</td>
<td>numa₂</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>numb₁</th>
<th>sum₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numb₁</td>
<td></td>
<td>sum₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>next₁</th>
<th>next₂</th>
<th>x₁</th>
<th>next₃</th>
<th>x₂</th>
<th>next₄</th>
<th>x₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numb₁</td>
<td>numb₂</td>
<td>sum₁</td>
<td>numb₃</td>
<td>sum₂</td>
<td>numb₄</td>
<td>sum₃</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]
- Control states (out of the loop)
  - one to initialize next, clear sum, set numa, clear x
  - two to finish off after next==0.

30
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. $X \leftarrow \text{Memory[NUMA]}$, $\text{NUMA} \leftarrow \text{NEXT}+1$; $0.5+1+10+1+0.5 = 13\text{ns}$
  2. $\text{NEXT} \leftarrow \text{Memory[NEXT]}$, $\text{SUM} \leftarrow \text{SUM}+X$; same for all $\Rightarrow T>13\text{ns}, F<77\text{MHz}$
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:

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

⇒ 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 for a 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.