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 Template

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

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

- **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.
- Sometimes cascaded:

RTL

- 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.
Register Transfer “Language” Descriptions

• 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:
  
  ```plaintext
  regA ← regB
  regC ← regA + regB
  if (start==1) regA ← regC
  ```

• 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):
  
  ```plaintext
  regA ← regB, regB ← 0;
  regC ← regA;
  ```

Example of Using RT Language

```
ACC ← ACC + R0, R1 ← R0;
ACC ← ACC + R1, R0 ← R1;
R0 ← ACC;
```

• In this case: RT Language description is used to sequence the operations on the datapath (dp).

• 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.
Example of Using RT Language

- Sometimes RT Language is used as a starting point for designing both the datapath and the control:
- example:
  
  \[
  \begin{align*}
  \text{regA} & \leftarrow \text{IN}; \\
  \text{regB} & \leftarrow \text{IN}; \\
  \text{regC} & \leftarrow \text{regA} + \text{regB}; \\
  \text{regB} & \leftarrow \text{regC};
  \end{align*}
  \]

- 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

- 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.

  ![Diagram of a linked-list structure](image)

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>

*Are these reasonable?*
Review of Register with “Load Enable”

- Register with Load Enable:

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

  - Assume both data and LD require setup time = 0.5ns.

  - Assume no reset input.

  ![](image)

  Functional description only. Transistor level circuit has lower input delay.

---

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.
  - 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;
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.
3. Architecture #1

Direct implementation of RTL description:

Datapath

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

Controller

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

### COMPUTE_SUM state

<table>
<thead>
<tr>
<th>State</th>
<th>8-bit add</th>
<th>memory</th>
<th>16-bit add</th>
<th>MUX</th>
<th>setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK-Q</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

- 31ns

### GET_NEXT state

<table>
<thead>
<tr>
<th>State</th>
<th>control-output delay</th>
<th>MUX</th>
<th>memory</th>
<th>control input delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>control-output delay</td>
<td>MUX</td>
<td>memory</td>
<td>control input delay</td>
</tr>
<tr>
<td>NEXT_ZERO</td>
<td>15.5ns</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

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

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

- Observation:
  \[ \text{COMPUTE\_SUM state does most of the work. Most of the} \]
  \[ \text{components are inactive in GET\_NEXT state.} \]
  \[ \text{GET\_NEXT does: Memory access + …} \]
  \[ \text{COMPUTE\_SUM does: 8-bit add, memory access, 15-bit add + …} \]

- Conclusion:
  \[ \text{Move one of the adds to GET\_NEXT.} \]

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):

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

• 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 on every cycle. Why not share the adder for both cycles?
5. Optimization, Architecture #3

- **Incremental cost:**
  - Addition of another mux and control (\texttt{ADD\_SEL}). Removal of an 8-bit adder.
- **Performance:**
  - No change.
- **Change is definitely worth it.**

---

Resource Utilization Charts

- One way to visualize these (and other possible) optimizations is through the use of a \textit{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>resource</th>
<th>cycle</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>1</td>
<td>fetch A1</td>
<td>fetch A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bus</td>
<td>2</td>
<td>fetch A1</td>
<td></td>
<td>fetch A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>register-file</td>
<td>3</td>
<td>read B1</td>
<td></td>
<td></td>
<td>read B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>4</td>
<td>A1+B1</td>
<td></td>
<td></td>
<td>A2+B2</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></th>
<th>memory</th>
<th></th>
<th>fetch next</th>
<th>fetch x</th>
<th>fetch next</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adder2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>memory</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td></td>
<td>suma</td>
<td></td>
<td>numa</td>
</tr>
</tbody>
</table>

1. \( \text{SUM} \leftarrow \text{SUM} + \text{Memory}[\text{NEXT}+1] \); 2. \( \text{NEXT} \leftarrow \text{Memory}[\text{NEXT}] \);

- 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></th>
<th>memory</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa</td>
<td></td>
<td>numa</td>
<td>numa</td>
</tr>
</tbody>
</table>

1. \( \text{X} \leftarrow \text{Memory}[\text{NUMA}] \), \( \text{NUMA} \leftarrow \text{NEXT}+1 \); 2. \( \text{NEXT} \leftarrow \text{Memory}[\text{NEXT}] \), \( \text{SUM} \leftarrow \text{SUM}+\text{X} \);

- 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 \( T_{\text{mem}} + T_{\text{add}} \).

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

- How about the other combination: \( \text{add x register} \)

<table>
<thead>
<tr>
<th></th>
<th>memory</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa</td>
<td></td>
<td>numa</td>
<td>numa</td>
</tr>
</tbody>
</table>

1. \( \text{X} \leftarrow \text{Memory}[\text{NUMA}] \), \( \text{NUMA} \leftarrow \text{NEXT}+1 \); 2. \( \text{NEXT} \leftarrow \text{Memory}[\text{NEXT}] \), \( \text{SUM} \leftarrow \text{SUM}+\text{X} \);

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>suma₁</td>
<td>numa₂</td>
<td>suma₂</td>
</tr>
</tbody>
</table>

- How can we overlap iterations? \( \text{next}₂ \) depends on \( \text{next}₁ \).
  - “slide” second iteration into first (4 cycles per result):

<table>
<thead>
<tr>
<th>Memory</th>
<th>next₁</th>
<th>next₂</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa₁</td>
<td>numa₂</td>
<td>suma₁</td>
<td>numa₃</td>
<td>numa₄</td>
<td>suma₃</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.
- Three different loop iterations active at once.
- Short cycle time (no dependencies within a cycle)
- Full utilization (only 2 cycles per result)
- Initialization: \(x=0, \text{numa}=1, \text{sum}=0, \text{next}=	ext{memory}[0]\)
- Extra control states (out of the loop)
  - one to initialize next, clear sum, set numa
  - one to finish off. 2 cycles after next==0.

<table>
<thead>
<tr>
<th>Memory</th>
<th>next(_1)</th>
<th>(x)_1</th>
<th>numa(_1)</th>
<th>sum(_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td></td>
<td></td>
<td>numa(_1)</td>
<td>sum(_1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>next(_2)</th>
<th>(x)_2</th>
<th>numa(_2)</th>
<th>sum(_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td></td>
<td></td>
<td>numa(_2)</td>
<td>sum(_2)</td>
</tr>
</tbody>
</table>

1. \(X\leftarrow\text{Memory[NUMA]}, \text{NUMA}\leftarrow\text{NEXT}+1;\)
2. \(\text{NEXT}\leftarrow\text{Memory[NEXT]}, \text{SUM}\leftarrow\text{SUM}+X;\)

5. Optimization, Architecture #4

- Datapath:
  - Addition of another register & mux, adder mux, and control.
- Performance: find max time of the four actions

\[
\begin{align*}
1. & \ X \leftarrow \text{Memory[NUMA]}, & 0.5+1+10+1+1+0.5 = 14\text{ns} \\
& \text{NUMA} \leftarrow \text{NEXT}+1; & \text{same for all} \Rightarrow T>14\text{ns}, F<71\text{MHz} \\
2. & \text{NEXT} \leftarrow \text{Memory[NEXT]}, & \text{SUM} \leftarrow \text{SUM}+X; \\
\end{align*}
\]