Discussion Session 8
Behnam Khaleghi
University of California, San Diego
bkhaleghi@ucsd.edu
Outline

• Register-Transfer Level (RTL) Design
Register-Transfer Level (RTL) Design

• Develop a design that finds out the index of the maximum element of a 32-element array “Arr” stored in a memory
• Array elements are 8-bits.
  • So the memory is 32x8
• The design should include HLSM, and the corresponding datapath and control unit implementation
Register-Transfer Level (RTL) Design

So, at the high level,

• The circuit will have input $S$, denoting \texttt{Start} of the operations
• Output $Y$ shows the index of the max element, so it should have 5 bits to represent $2^5 = 32$ indexes
• We also need 1 bit to show the end of (search) operations

From algorithmic perspective,

• We need one variable, say it “$i$”, to hold the loop counter
  • Note we need to iterate over all 32 elements
• We also need one variable, “$i_m$”, to keep the index of the (current) maximum element
HLSM

- Initial state: wait until active command (S) is received
- Also, we initialize the required variables

\[
S' \\
\text{Wait} \\
i = 1 \\
i_m = 0
\]

- We assume that the max element is in index 0, so compare the remaining elements (starting from index 1) with that
HLSM

• The next state performs comparison between the current index, and the current maximum

\[ i = 1 \]
\[ i_m = 0 \]
• Depending on the result of comparison, we do or do not update the max index. Either, we need to update the loop counter.
• We need to check if the loop iterated reached the upper bound, 32.
• Otherwise, we go to make another comparison; with the next element.
Datapath Components

• Next, we need to decide which components we need according to the HLSM
Datapath Components

- Two 6-bit registers to hold $i$ and $i_m$
Datapath Components

- A 8-bit comparator that return whether X is greater than Y
- As mentioned earlier, the array elements are 8 bit
Datapath Components

- An adder to increment loop iterator
Datapath Components

- Another comparator to check if \( X = 31 \)
  - A constant equality comparator is basically an AND gate and some NOTs.
  - \( A = 31 \rightarrow A_4A_3A_2A_1A_0 = 11111 \)
  - \( i \) starts with 1, so we can use 5-bit registers for \( i \) and \( i_{\text{max}} \).
Datapath Components

- And finally, a memory (register file) to read the array elements
  - It is given, we are just reading with that (i.e., not part of our design)
  - We assume it has two read ports, i.e., gets two addresses (indexes) and outputs the corresponding elements
  - Note that we don’t need to read $\text{Arr}[i_m]$ every time, so with some more complexity (e.g., register to hold the $\text{Arr}[i_m]$), we could use a single-port memory (which is way more cost-efficient). Think about it.
Datapath Components

- Putting all together
  - Adder for $i + 1 \rightarrow$ can be made up only from Has
  - 5-bit registers for $i$ and $i_{\text{max}}$
  - 8-bit comparator
  - An AND gate (to check $i == 31$)
  - Some additional multiplexers to share the logics

![Adder Diagram](attachment://adder.png)

![Register Diagram](attachment:register.png)

![Comparator Diagram](attachment:comparator.png)
Datapath Components

- Need to connect the component together
- Let’s begin with the adder
- Inputs of the adder are always $i$ and $1 \rightarrow$ no multiplexer
Datapath Components

• Now register of “i”: sometimes loaded by 1 (in Wait state, and sometimes by i+1, i.e., output of the adder)
• It does not always loaded (e.g., in Comp/Done states), so needs a load enabler bit
Datapath Components

- Among the remaining, let’s connect register $i_m$
  - $i_m$ becomes either 0 (state Wait) or $i$ (state S_G)
  - We assume that the register has a clear input, so we don’t need a mux here
  - Connection from Reg_i to Reg_i_max doesn’t mean that $\text{Reg}_i_{\text{max}} = \text{Reg}_i$
Datapath Components

- The values of $i$ (Reg$_i$) and $i_{\text{max}}$ act as address for the (read-only) memory.

- A general dual-port memory has:
  - Address ports 1 and 2, showing the address (element) to be read.
  - Since we have 32 elements, the width of the address ports is 5.
  - Data_out 1 and 2, to output the required elements.
  - Data For the cases we want to write in the memory, the data address points to the memory address (index) we need to write into. Data_in is the value to be written there.
  - Read/Write = 0 allows read (so we will not have a write, event though the data and its address is ready). Write = 1 allows write. We cannot have simultaneous read and write.
  - Enable = 1 enables read/write.
  - In this example we just have read operations.
    - Much simpler memory.

Dual-port Memory

Address 1
Address 2
Data Address
Data_in
Read/Write
Enable

Data_out 1
Data_out 2
Datapath Components

• The values of \( i \) (\text{Reg}_i) and \( i_{\text{max}} \) act as address for the (read-only) memory.
Datapath Components

- Outputs of the memory are used in the comparator
Datapath Components

- We also need to check the $i == 31$
Datapath Components

- “e” and “g” will help the FSM/control unit to determine the states
- Why we need FSM?
  - To keep track which state we are in, so can determine the controlling signals used in the datapath: L_i_max, C_i_max, S_m, L_i, mem_en

![Datapath Diagram]

- \( S_m \)
- \( L_i \)
- \( L_i_{\text{max}} \)
- \( C_i_{\text{max}} \)
- \( Arr[0] \)
- \( Arr[1] \)
- \( Arr[31] \)
- \( \text{mem}_\text{en} \)
- \( \text{Reg}_i \)
- \( \text{Reg}_i_{\text{max}} \)

\[ A > B \]

- \( g + e \)
Control Unit

- The control unit includes as many states as the HLSM
- The FSM’s inputs come from the datapath or primary inputs
- \( g, e, \) and \( S \)
Control Unit

- FSM needs to keep track of the states of the HLSM
Control Unit

• At each state, the FSM needs to specify all the controlling bits
• Example for the “Wait” state:
  • \( i = 1 \), so \( \text{Reg}_i \) should be loaded
  • \( \text{L}_i = 1 \) and \( \text{S}_m = 1 \) (to pass 1 into \( \text{Reg}_i \))
  • \( i_m = 0 \), so \( \text{Reg}_i\_\text{max} \) should be cleared
  • \( \text{C}_i\_\text{max} = 1 \), and \( \text{L}_i\_\text{max} = 0 \)
  • No memory read is required, so \( \text{mem}_{\text{en}} = 0 \)
Control Unit

• Note that $f = 1$ shows the end of operations and is the primary output of the circuit
  • We can produce it in the control unit
  • We don’t need to produce all of the outputs in the datapath
• We needed to include $f = 0$ in all previous states
  • Rather, at each state, we can just show the “1” signals
Control Unit

- We needed to include $f = 0$ in all previous states?
  - Rather, at each state, we can just show the “1” signals
- In the last state, the value of $S_m$ doesn’t matter
  - Load of the $\text{Reg}_i$ is disabled
  - We can make it don’t care for better simplification of the FSM (Kmap)
The Entire System

• Do we miss something?
The Entire System

- Yes! (at least) the main output: index of the max element
  - It’s simply the output of the Reg_i_max
- Clock connected to the control unit
FSM Implementation

- 5 states (3 flip-flops) and 3 variables
  - $s_2, s_1, s_0, S, g, e$
- Determining the next state requires 6-variable K-map
FSM Implementation

• Leverage Boolean algebra

• Example:

\[ s_2(t + 1) = 1 @ \text{Done}(t + 1) = S_G(t) \cdot e + S_LE(t) \cdot e \]

\[ = e(s'_2s'_1s'_0 + s'_2s_1s_0) = es'_2s_1 \]

\[ \rightarrow s_2(t + 1) = e \cdot s'_2(t) \cdot s_1(t) \]
FSM Implementation

\[ s_1(t + 1) = 1 @ S_G(t + 1) + S_LE(t + 1) \]
\[ = \text{Comp}(t) \cdot g + \text{Comp}(t) \cdot g' = \text{Comp}(t) = s_2' \cdot s_1' \cdot s_0 \]
\[ \rightarrow s_1(t + 1) = s_2'(t) \cdot s_1'(t) \cdot s_0(t) \]

\[ s_0(t + 1) = 1 @ \text{Comp}(t + 1) + S_LE(t + 1) \]
\[ = \text{Wait}(t) \cdot S + S_G(t) \cdot e' + S_LE(t) \cdot e' + \text{Comp}(t) \cdot g' \]
\[ = s_2's_1's_0S + s_2's_1's_0e' + s_2's_1's_0e' + s_2's_1's_0g' \]
\[ = s_2'(s_1'(s_0S + s_0g) + s_1'e') \]
FSM Implementation

- The less “1” in the states, the simpler FSM implementation becomes
- \( s_0(t + 1) = 1 @\text{Comp}(t + 1) \)
  - Eliminated the \( S\_LE(t+1) \) term
  - Though adds one more flip-flop
    - One more variable multiplied
    - Potentially increases the size some of gates by 1

<table>
<thead>
<tr>
<th>State</th>
<th>s3</th>
<th>s2</th>
<th>s1</th>
<th>s0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Comp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S_G</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S_LE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Done</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>