Chapter 2: Custom single-purpose processors

Outline

- Introduction
- Combinational logic
- Sequential logic
- Custom single-purpose processor design
- RT-level custom single-purpose processor design
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis

Introduction

- Processor
  - Digital circuit that performs a computation task
  - Controller and datapath
  - General-purpose: variety of computation tasks
  - Single-purpose: one particular computation task
  - Custom single-purpose: non-standard task

- A custom single-purpose processor may be
  - Fast, small, low power
  - But, high NRE, longer time-to-market, less flexible

CMOS transistor on silicon

- Transistor
  - The basic electrical component in digital systems
  - Acts as an on/off switch
  - Voltage at “gate” controls whether current flows from source to drain
  - Don’t confuse this “gate” with a logic gate

Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
CMOS transistor implementations

- Complementary Metal Oxide Semiconductor
- We refer to logic levels
  - Typically 0 is 0V, 1 is 5V
- Two basic CMOS types
  - nMOS conducts if gate=1
  - pMOS conducts if gate=0
  - Hence “complementary”
- Basic gates
  - Inverter, NAND, NOR

Basic logic gates

- Inverter: $F = \overline{x}$
- NAND: $F = \overline{x \cdot y}$
- NOR: $F = \overline{x + y}$
- XOR: $F = x \oplus y$
- OR: $F = x + y$
- AND: $F = x \cdot y$
- XNOR: $F = \overline{x \oplus y}$
Combinational logic design

A) Problem description

\[ y = 1 \text{ if } a \text{ is to } 1, \text{ or } b \text{ and } c \text{ are } 1. \text{ } z = 1 \text{ if } b \text{ or } c \text{ is } 1, \text{ but not both, or if all are } 1. \]

B) Truth table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

C) Output equations

\[ y = a'b'c + ab'c' + ab'c + abc' + abc \]
\[ z = a'b'c + ab'c + ab'c + abc' + abc \]

D) Minimized output equations

\[ y = a + bc \]
\[ z = ab + b'c + bc' \]

E) Logic Gates

Combinational components

- **O** = 10 if S=0..00 11 if S=0..01 ...
- \( O(\log n-1) = 1 \) if \( S^<1..11 \)

<table>
<thead>
<tr>
<th>Input(s)</th>
<th>Omit 1s</th>
<th># of 0s</th>
<th># of #'s</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( B )</td>
<td>( S )</td>
<td>( \bar{S} )</td>
</tr>
</tbody>
</table>

O = \( A \oplus B \)
sum = \( A + B \)
less = \( 1 \) if \( A < B \)

<table>
<thead>
<tr>
<th>Input(s)</th>
<th>Output(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( B )</td>
</tr>
<tr>
<td>( C_i )</td>
<td>( \bar{C}_i )</td>
</tr>
</tbody>
</table>

With enable input \( e \) all \( O \)'s are 0 if \( e=0 \)
With carry-in input \( C_i \) sum = \( A + B + C_i \)
May have status outputs carry, zero, etc.
Sequential components

Q = 0 if clear=1,
   1 if load=1 and clock=1,
   Q(previous) otherwise.

Q = 1 if
   - Count shifted
   - 1 stored in msh
Q = 0 if clear=1,
   Q(prev)+1 if count=1 and clock=1.

Sequential logic design

A) Problem Description
You want to construct a clock divider. Slow down your pre-existing clock so that you output a 1 for every four clock cycles

B) State Diagram

C) Implementation Model

D) State Table (Moore-type)

- Given this implementation model
  - Sequential logic design quickly reduces to combinational logic design
Sequential logic design (cont.)

E) Minimized Output Equations

\[
I_1 = Q_1'Q_0a + Q_1a' + Q_1Q_0'
\]

\[
I_0 = Q_0a' + Q_0'a
\]

\[
x = Q_1Q_0
\]

F) Combinational Logic

Custom single-purpose processor basic model

controller

external control inputs

datapath control inputs

external control outputs

datapath control outputs

datapath

data inputs

external data outputs

controller and datapath

controller

next-state and control logic

state register

datpath

datpath control inputs

registers

functional units

a view inside the controller and datapath
Example: greatest common divisor

- First create algorithm
- Convert algorithm to “complex” state machine
  - Known as FSMD: finite-state machine with datapath
  - Can use templates to perform such conversion

```
0: int x, y;
1: while (1) {  
  2:   while (!go_i);
  3:   x = x_i;
  4:   y = y_i;
  5:   while (x != y) {  
    6:     if (x < y)
    7:        y = y - x;
    else
    8:        x = x - y;
  9:   }   
 10:  d_o = x;
}
```

State diagram templates

- Assignment statement: `a = b`
- Loop statement: `while (cond) {
  loop-body-statements
} next statement`
- Branch statement: `if (c1) c1 stmts
  else if c2 c2 stmts
  else other stmts`
Creating the datapath

- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component
  - control input and output

Creating the controller’s FSM

- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations
Splitting into a controller and datapath

Controller implementation model

Controller state table for the GCD example

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \text{ sel} )</td>
<td>( y \text{ ld} )</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>X</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
Completing the GCD custom single-purpose processor design

- We finished the datapath
- We have a state table for the next state and control logic
  - All that’s left is combinational logic design
- This is not an optimized design, but we see the basic steps

RT-level custom single-purpose processor design

- We often start with a state machine
  - Rather than algorithm
  - Cycle timing often too central to functionality
- Example
  - Bus bridge that converts 4-bit bus to 8-bit bus
  - Start with FSMD
  - Known as register-transfer (RT) level
  - Exercise: complete the design
RT-level custom single-purpose processor design (cont’)

Optimizing single-purpose processors

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
  - original program
  - FSMD
  - datapath
  - FSM
Optimizing the original program

- Analyze program attributes and look for areas of possible improvement
  - number of computations
  - size of variable
  - time and space complexity
  - operations used
    - multiplication and division very expensive

Optimizing the original program (cont’)

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Optimized Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: int x, y;</td>
<td>0: int x, y, r;</td>
</tr>
<tr>
<td>1: while (1) {</td>
<td>2: while (1) {</td>
</tr>
<tr>
<td>2:   while (!go_i);</td>
<td>while (!go_i);</td>
</tr>
<tr>
<td>3:   x = x_i;</td>
<td>// x must be the larger number</td>
</tr>
<tr>
<td>4:   y = y_i;</td>
<td>if (x_i &gt;= y_i) {</td>
</tr>
<tr>
<td>5:   while (x != y) {</td>
<td>x = x_i;</td>
</tr>
<tr>
<td>6:       if (x &lt; y)</td>
<td>if (x &gt; y)</td>
</tr>
<tr>
<td>7:         y = y - x;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>8:         x = x - y;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>9:   d_o = x;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

GCD(42, 8) - 9 iterations to complete the loop
x and y values evaluated as follows: (42, 8), (34, 8), (26, 8), (18, 8), (10, 8), (2, 8), (2, 4), (2, 2).

GCD(42, 8) - 3 iterations to complete the loop
x and y values evaluated as follows: (42, 8), (8, 2), (2, 0).
Optimizing the FSMD

• Areas of possible improvements
  – merge states
    • states with constants on transitions can be eliminated, transition taken is already known
    • states with independent operations can be merged
  – separate states
    • states which require complex operations \((a*b*c*d)\) can be broken into smaller states to reduce hardware size
  – scheduling

---

Optimizing the FSMD (cont.)

<table>
<thead>
<tr>
<th>Original FSMD</th>
<th>Optimized FSMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>eliminate state 1 – transitions have constant values</td>
<td>eliminate state 1-J – transition from state 1-J can be done directly from state 9</td>
</tr>
<tr>
<td>merge state 2 and state 2J – no loop operation in between them</td>
<td>eliminate state 5J and 6J – transitions from each state can be done from state 7 and state 8, respectively</td>
</tr>
<tr>
<td>merge state 3 and state 4 – assignment operations are independent of one another</td>
<td>merge state 5 and state 6 – transitions from state 6 can be done in state 5</td>
</tr>
</tbody>
</table>

---

Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
Optimizing the datapath

- Sharing of functional units
  - one-to-one mapping, as done previously, is not necessary
  - if same operation occurs in different states, they can share a single functional unit
- Multi-functional units
  - ALUs support a variety of operations, it can be shared among operations occurring in different states

Optimizing the FSM

- State encoding
  - task of assigning a unique bit pattern to each state in an FSM
  - size of state register and combinational logic vary
  - can be treated as an ordering problem
- State minimization
  - task of merging equivalent states into a single state
    - state equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state
Summary

- Custom single-purpose processors
  - Straightforward design techniques
  - Can be built to execute algorithms
  - Typically start with FSMD
  - CAD tools can be of great assistance