ECE 465
Controller or “Action” FSMs

Shantanu Dutt
UIC
Motivating Example 1

- Computation needed: \( r_4 = \frac{(a+b)(c+d)}{x} \)
- Computation needs to be pipelined
- Functional units with different delays
- No sharing of FUs between operations (simpler situation than sharing)
- Pipeline can move at the rate of 3 cc’s due to Stage 3 having the max delay of 3 cc’s. This essentially means that Stage 1 has to wait 2 cc’s and Stage 2 has to wait 1 cc in order to synchronize with Stage 3, so that all stages move in lock step at the rate of 3 cc (this is the only way to do this when there is no sharing)
- We assume here that the module furnishing input data a, b, c, d, x provides a new data set every 3 cc.
- How to synchronize the movement of data and loading of registers across the stages so that the computation is correctly done w/o new data to any FU clobbering earlier data that is still being processed?
- Need a control or action FSM to orchestrate this
Motivating Example 1 (contd.)

O/p labeling convention in Moore control FSM states: signals not specified in a state are inactive

```
[ra <- a; rb <- b; rc <- c;
rd<- d; r1 <- a+b, r2 <- c+d,
r3 <- r1*r2, r4 = r3/rx2,
rx1 <= x, rx2 <- rx1, rx3 <- rx2;]
```

(achieved at the beginning of the next cc). Note: Junk values stored in r3 in 1st iteration and in r4 in 1st and 2nd iteration

NOP (cc counting state)

NOP (cc counting state)
Motivating Example 1 (contd.)

Reset

[repeat for ra to rx’s;
r1 <- a+b, r2 <- c+d, rx1 <= x, rx2 <= rx1]
RTL specification

No junk data solution

NOP (cc counting state)

NOP (cc counting state)

NOP (cc counting state)

NOP (cc counting state)

[r1 <- a+b, r2 <- c+d, rx1 <= x, rx2 <= rx1]

[r1 <- a+b, r2 <- c+d, r3 <- r1*r2, rx1 <= x, rx2 <= rx1]

[r1 <- a+b, r2 <- c+d, r3 <- r1*r2, r4 <- r3/rx2, rx1 <= x, rx2 <= rx1]

[r1 <- a+b, r2 <- c+d, r3 <- r1*r2, r4 <- r3/rx2, rx1 <= x, rx2 <= rx1]

[r1 <- a+b, r2 <- c+d, r3 <- r1*r2, r4 <- r3/rx2, rx1 <= x, rx2 <= rx1]
Motivating Example 2

- Computation: \( z = a \times b + (c+d) \)
- Mapping to h/w w/ constraints: use only 1 (X) (2 cc) and 1 (+) (1 cc)
- Need to share the + FU, so need explicit scheduling of operations (start time assignment)
- Non-overlapped pipelining
- Inputs are loaded in this example

\[
\begin{align*}
\text{lda} & \rightarrow a \\
\text{ldb} & \leftarrow \text{ldb} \\
\text{mux1} & \rightarrow \text{mux} \\
\text{mux2} & \leftarrow \text{mux2} \\
\end{align*}
\]

i) Non-overlapped pipelined scheduling

Controller FSM:

\[
\begin{align*}
\text{lda} = 1, & \quad \text{ldb} = 1, \\
\text{ldc} = 1, & \quad \text{ldd} = 1, \\
\text{mux1} = 1, & \quad \text{mux2} = 1, \\
\text{demux} = 1, & \quad \text{ldy} = 1 \\
\end{align*}
\]

Note: A register is loaded at the +ve/-ve edge (in a +ve/-ve edge triggered system) of the cc after the one in which its load signal is asserted.
Motivating Example 3

- Same problem of example 3 with the change the pipelining is overlapped, i.e., different iterations are occurring simultaneously in the pipelined datapath (this is “regular” pipelining).

ii) Overlapped pipelined scheduling

Controller FSM:

Reset

cc 2i

ldc=1, ldd=1, mux1=1, mux2=1, demux=1, ldz=1

[y ← c+d, x ← a x b] ((c1, c2))

For 4 iterations, the overlapped schedule takes 9 cc’s versus 12 cc’s by the non-overlapped sched.

- Overlap. sched: Time for n iterations = 2n+1
  Throughput = n/(2n+1) ~ 0.5 outputs/cc
- Nonoverlap. sched: Time for n iterations = 3n
  Throughput = n/3n ~ 0.33 outputs/cc

⇒ ~ 34% throughput improvement using an overlapped schedule
Part I: Control of Datapaths via Basic Components, Control Operations & Connections

Shantanu Dutt
UIC
(a) $2^n:1$ Mux: $sel = j \implies o/p = I_j$

(b) $n:2^n$ DCD:
- $addr = j \land enb = 1 \implies sel = 1$
- $sel = 0, i = j \land enb = 0 \implies sel = 0$, for all $j$

(c) Tri-state buffer:
- $enb = 1 \implies o/p = I$
- $enb = 0 \implies o/p = Z$

(d) Bus connection via tri-state buffers:
- exactly one $enb = 1 \implies Bus = I_j$
- $enb = 0$, for all $i \implies Bus = Z$

(high impedance, not connected to any signal)
(a) +ve edge-trigg. D-FF w/o enable

(b) Timing diagram for +ve edge triggered D-FF (w/o enable)

(c) +ve edge-trigg. D-FF w/ enable

(d) 4-bit register with load/write and reset control
1) Basic Operations:

1) Load a register (individual/separate register, e.g., the Program Counter \([PC]\) in a processor, or one or more registers in a register file)

e.g.
\[
R1 \leftarrow \text{BusA} \quad \text{/* load reg. } \text{R1} \text{ with BusA content */}
\]

- To perform \(R1 \leftarrow \text{BusA}\) correctly, the "load-\(R1\)" signals needs to be "activated" (generally, set to '1') by a controller FSM.

- If the FSM sets \(\text{load-}R1 = 1\) in clock cycle \(i\), then \(R1 \leftarrow \text{BusA}\) (i.e. loading of \(R1\)) is accomplished at the two edge (assuming a two edge system/register) of clock cycle \(i+1\).

\[\text{CLK} \quad \text{---} \quad \text{load-}R1 = 1 \quad \text{---} \quad R1 \leftarrow \text{BusA} \]

\[\text{CLK} \quad \text{---} \quad \text{CC}i \quad \text{---} \quad \text{CC}i+1 \quad \text{---} \quad \text{CC}i+2 \quad \text{---} \quad \text{CC}i+3 \quad \text{---} \quad \text{CC}i+4 \quad \text{---} \quad \text{CC}i+5 \quad \text{---} \]
2) Connecting data to a bus from one of multiple sources

E.g. Write-Bus ← Adder_output or Read-Bus ← register address of 32-register register file.

- One of multiple sources is selected to connect to a bus via either a mux (as shown above) or tri-state buffers, by activating the right control signals.

- In the above e.g. to accomplish Write-Bus ← Adder_output, "alu-set" control signal is set to '00' by a controller flip-flop, read is accomplished in the same cycle in which control signal is activated.
- Other combinations of the above two operations can be used for various types of connections and regular loadings.

- E.g., loading $R_1$ from one of multiple sources.

- $R_1 \leftarrow Bus A$ / Bus is one source out of 5 sources.

- $R_1 \leftarrow Bus A$ / $R_1$-Input-set

- $R_1 \leftarrow Bus A$ / $R_1$-Input

- $R_1 \leftarrow Bus A$ / $R_1$-Output

- $R_1 \leftarrow Bus A$ / $R_1$-Input-set

- $R_1 \leftarrow Bus A$ / $R_1$-Input

- $R_1 \leftarrow Bus A$ / $R_1$-Output

- $R_1 \leftarrow Bus A$ / $R_1$-Input-set

- $R_1 \leftarrow Bus A$ / $R_1$-Input
B) Secondary Operations: 1) Register w/ various shift/rotate/load options

Controller FSM:

- \( \text{oper} = 00 \) means load-reg from D-bus (I/P Bus)
- \( \text{oper} = 01 \) means LSR
- \( \text{oper} = 10 \) means ASR
- \( \text{oper} = 11 \) means ROR (rotate right)

- \( \text{en}_\text{reg} = 0 \) means inactive
- \( \text{en}_\text{reg} = 1 \) means active

- \( \text{reset}_\text{reg} = 0 \) means inactive
- \( \text{reset}_\text{reg} = 1 \) means active
The above shift register is really a combination of:
1. Connection to I or (bus) of R1, from one of multiple
   sources, including a function unit (Fu), etc., a combinational
   circuit.

2. Loading of R1.

---

Take simple case of LSR, only for R1.

---

**Note:** Entries in boxes are avoided internally in the logic.

Inside the R1 "box".

---

Let's LSR logic:

[Diagram with labels and connections related to shift register logic]
2) Similarly other secondary operations like count-up, count-down of a counter can be accomplished by a combination of the two basic operations (selective has connection) and appropriate FUs (combinational cycle in the data path).

CLK

 incr-opn = 10
 emb-n = 1

 dec-opn = 00
 emb-n = 1

This block is used to count a set of a whose count is EKB1 plus RI in the previous cycle via EKB1.
Part II: Controller FSM Design Issues—Correctness and Efficiency

Shantanu Dutt

UIC
CU FSM Design Strategies: Design Correctness

- For a major operation (e.g., fetch & decode), trace the datapath through the computer system (processor, memory, I/O, etc.) needed for executing this operation.

- Break up the datapath into subpaths s.t. each subpath starts from a register read (explicit or implicit), has an optional operation, and, except possibly the last subpath, ends with a register write.

- Each subpath $P_i$ needs to be implemented by a 1 or more states (> 1 states in the case of multi-cc operations) that are disjoint from states implementing subpaths on which $P_i$ is data/control dependent, or those that are data/control dependent on $P_i$.

- Control signals to implement a subpath are set in the corresponding state(s).

- A subpath is dependent on the data produced as a register write in the previous subpath. This data dependency is reflected by their corresponding states following each other in the same order as the subpaths.
CU FSM Design Strategies: Design Correctness (contd.)

- Need to perform data-dependent operations sequentially one after the other (as dictated by the dependency) in disjoint sets of states.

(a) Datapath

(b) Incorrect assertion of control signals

(c) Correct assertion of control signals (default value is inactive or 0)

(d) FSM when f2 delay is 3ccs

Counting States
(can also do using a multipurpose counter)
CU FSM Design Strategies: Design Correctness (contd.)

- Connections to the input of the FU (functional unit, i.e., some combinational logic) need to be maintained for the entire period of the required operation.

- In multiple-cc operations, the write control signal for writing to the destination register should only be valid in the last cc of the operation.
CU FSM Design Strategies: Design Correctness (contd.)

- Feedback connection from a register back to itself via one or more FUs and without other intermediate registers is OK if registers are edge-triggered or operate on a 2-phase non-overlapped clocking system—there is no race condition in such a situation.

- For operations that can be completed within a clock cycle the reading from a register and writing back to it (via one or more FUs) can be accomplished in 1cc without a race condition problem.
While subpaths of one operation need to be implemented in separate states, subpaths of 2 or more operations (at most one subpath from each operation) may be combined into one state if: (a) there is no data dependency between them (data produced by one is not needed by the other), and (b) there are no resource conflicts—the same resource is not needed for more than one purpose in any clock cycle. Resources include registers, FUs, buses and other connections, memory.

Implementing subpaths of different operations simultaneously in one state, speeds up the overall speed over all operations and reduces the number of states required in the Control Unit.
CU FSM Design Strategies: *Time Efficiency–Concurrency (contd.)*

(a) Datapath

(b) Correct but time-inefficient

(c) Concurrent implementation of the 2 operations in 1 cc; possible, since no data dependency and no resource conflicts

(d) FSM when f2 delay is 3ccs; in the 1st state f1 and f2 start concurrently/simultaneously
MULTIPLICATION

Serial Multiplication

Add-and-shift (A&S) multiplication:
Manual Example:

\[
\begin{array}{c}
1001 = 9 \\
1011 = 11 \\
\hline
00011011 = P \\
\hline
+ 1001 \\
\hline
00011011 = P1 \\
\hline
+ 1001 \\
\hline
00011011 = P2 \\
\hline
+ 0000 \\
\hline
00011011 = P3 \\
\hline
+ 1001 \\
\hline
01100011 = P4 = P = 99
\end{array}
\]
If the additions are done one at a time, we obtain a sequence of partial products \( P_0, P_1, \ldots, P_n \). Each partial product \( P_{i+1} \) is obtained as

\[
P_0 = 0
\]

\[
P_{i+1} = P_i + 2^i(x_i \cdot Y)
\]

Thus

\[
P_n = \sum_{i=0}^{n-1} 2^i(x_i \cdot Y) = Y \cdot \sum_{i=0}^{n-1} 2^i x_i = Y \cdot X
\]

where \( X \) is the multiplier and \( Y \) the multiplicand.
A&S multiplication (contd.):

The same effect as shifting the multiplicand left \(2^i(x_i.Y)\) can be achieved by keeping the multiplicand fixed at the left-most position and shifting the partial product \(P_i\) right.

Example:

\[
\begin{array}{c}
1001 = 9 \\
1011 = 11 \\
\hline
00000000 = P_0 \\
1001 \\
\hline
10010000 \\
SHR: 01001000 = P_1 \\
1001 \\
\hline
11011000 \\
SHR: 01101100 = P_2 \\
0000 \\
\hline
01101100 \\
SHR: 00110110 = P_3 \\
1001 \\
\hline
11000110 \\
SHR: 01100011 = P_4 = P = 99
\end{array}
\]

In this case, the partial products obtained are:

\[
P_0 = 0, \quad P_{i+1} = 2^{-1}(P_i + 2^n(x_i.Y))
\]

However, \(P_n\) is the same: \(P_n = \sum_{i=0}^{n-1} 2^i(x_i.Y)\)
A&S multiplication (contd.):

Hardware:

![Diagram of A&S multiplication hardware](image)

Algorithm:
Initialize $AC = 0$; $Q = $ Multiplier; $M = $ Multiplicand.
Do the following steps $n$ times

\{
If LSB of $Q$ is 1 then $AC = AC + M$ else $AC = AC$;
Shift $C_{out}$-AC-$Q$ register combination right by 1 bit
\}

Final product is in AC-$Q$ register
NOTE: Overflows are tolerated in the additions—$C_{out}$ is fed to the MSB of AC when right shifting
Structural VHDL allows the designer to represent a system in terms of components and their interconnections. This module discusses the constructs available in VHDL to facilitate structural descriptions of designs.

Acknowledgement (this and next 3 slides): [http://www.people.vcu.edu/~rhklenke/tutorials/vhdl/modules/m11_23/sld001.htm](http://www.people.vcu.edu/~rhklenke/tutorials/vhdl/modules/m11_23/sld001.htm). The next 3 slides appear in sld037.htm to sld039.htm of this module.
Behavioral VHDL
RASSP Education & Facilitation
Module 12
Version 2.01

Copyright © 1995-1998 RASSP E&F

All rights reserved. This information is copyrighted by the RASSP E&F Program and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the RASSP E&F Program is prohibited. All information contained herein may be duplicated for non-commercial educational use provided this copyright notice is included. No warranty of any kind is provided or implied, nor is any liability accepted regardless of use.

FEEDBACK:
The RASSP E&F Program welcomes and encourages any feedback that you may have including any changes that you may make to improve or update the material. You can contact us at feedback@rasp.scrs.org or http://rasp.scrs.org/module-request/FEEDBACK/feedback-on-modules.html.

Acknowledgement (this and next 7 slides): http://www.people.vcu.edu/~rhklenke/tutorials/vhdl/modules/m12_23/sld001.htm. The next 7 slides appear in sld050.htm to sld056.htm of this module.
The final example is of an RTL level datapath for an unsigned 8 bit multiplier. It illustrates the use of complex components in a structural description and the use of multiple levels of hierarchy in that the components are themselves structural descriptions of lower level components. This is the entity description for the datapath. The register controls (enable, mode) will go to the control unit when it is added.
Now we are going to develop a behavioral description of the controller for the unsigned 8 bit multiplier. This is a flow chart of the algorithm that the controller uses.
This is state diagram of the controller state machine. The outputs for each state aren’t shown for clarity. Notice that there is also a “count” variable that must be included in the state machine to count the number of iterations through the loop. The count variable is actually implemented as another state variable.
Appendix: VHDL Codes for Datapath (Structural) and Controller FSM (Behavioral) for Unsigned 8-bit Multiplication
This shows the component declarations and binding indications for the datapath.
This is the component instantiations for the datapath. The mapping of individual bits of the D and Q input and output of the shift registers is necessary to reverse the inputs and outputs. Recall that the shift register was a “shift up” type where the scan_in input goes to D(0) and D(0) to D(6) go to D(1) to D(7) when in shift mode. What is needed for the multiplier is a “shift down” type register where scan_in goes to D(7), etc. It should have been possible (we believe) to use the syntax
\[ d \Rightarrow \text{multiplier}(0 \text{ to } 7) \]
in the shift_reg8_str PORT MAP to accomplish the same thing, but the QuickVHDL compiler gave a “warning” and the simulator crashed, so we don’t know if it really should work.
The state machine actually has two state variables, the current state of the control state machine (e.g., initialize, shift, add), and the present loop count. The loop count is a state variable in that it has a present value and a next value, and it is updated in the clock process. However, the value of count only affects the next control state the machine goes to and doesn’t affect the outputs. The implementation is actually more like two state machines in the same architecture.
This the entity description for the unsigned 8 bit multiplier control unit. It hooks to the datapath via the control signals listed.
This is the beginning of the architecture of the control unit. Note that the constrained subtype of integer is for synthesis - unconstrained integers are hard to synthesize! Also note the state variables are enumerated types. This allows the synthesis tools to encode the state variable using different schemes.

Also included here is the clock process. Note that it is edge triggered and that both present_state and present_count are updated on the clock edge. Also note the asynchronous reset signal.
This is the state transition process for the state machine. Note the default assignment of `next_state = present_state` which is only really required for (some) synthesis tools.
This is the output process. Note that the outputs are only dependent on the present_state variable (Moore machine).