Limitations of FSMs (A Practical Introduction to HW/SW Codesign, P. Schaumont)

The Finite State Machine controller in an FSMD is non-programmable

A microprogrammed architecture is obtained by substituting the FSM for a **programmable controller**

The advantage of a programmable architecture is *flexibility* to implement **multiple** functionalities

Here we cover the design of micro-programmed controllers and datapaths, including advantages/disadvantages

FSMs are a convenient way of capturing control and decision making

FSM graphs, in fact, resemble Control Dependence Graphs

Yet FSMs are not a **universal solution** for control, and they suffer from several modeling weaknesses, particularly when dealing with complex control requirements

Limitations of FSMs

FSM graphs are a *flat* model (no **hierarchy**)

They are like a C program written completely in a single function

Realistic systems do not use flat control, they need a **control hierarchy**

There have been proposals for hierarchical modeling mechanisms for FSM, e.g., **Statecharts**, but they have not found widespread use

The most obvious problem of a flat FSM model is ***state explosion***, which occurs when multiple independent activities interfere in a single model

Assume that a single finite state machine has to model 2 different activities each of which can be in one of three states

The resulting FSM, called a *product state-machine*, needs **9 states** to represent the overall model

Due to *conditional state transitions*, one state machine can remain in a single state while the other state machine proceeds to the next state

Limitations of FSMs

As shown below, *A1*, *A2* and *A3* become intermediate states

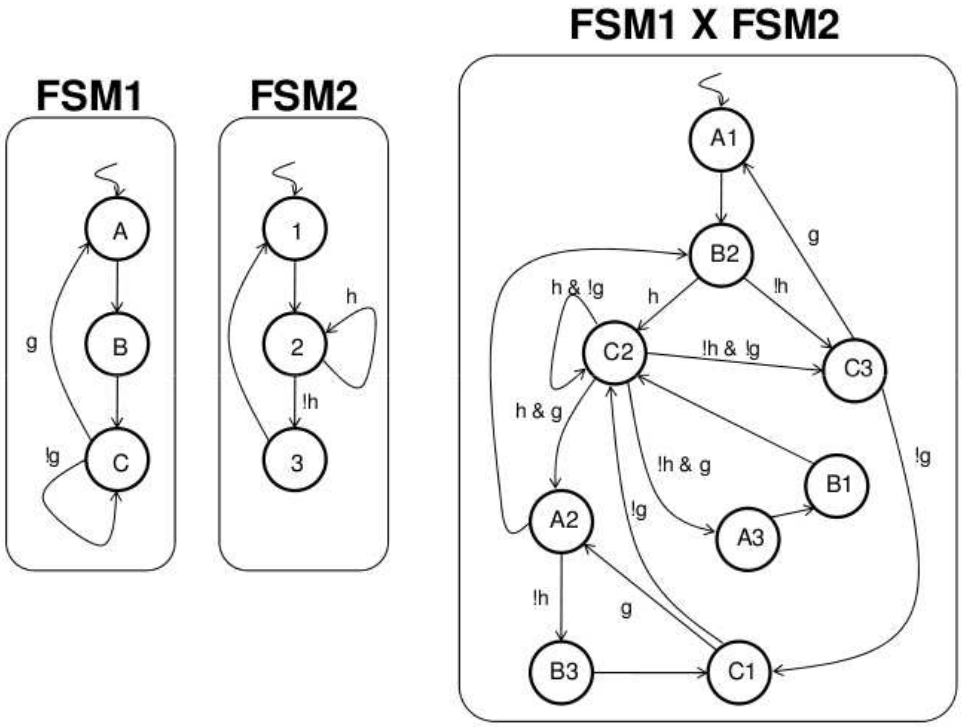


Fig. 5.1 State explosion in FSM when creating a product state-machine

The resulting number of state transitions is even higher, i.e., for *n* independent state transition in each state machine, we can have upto 2^n state transition conditions

Limitations of FSMs

FSM graphs have trouble expressing **exceptions** and **global conditions**

An exception can be defined as a *global condition* that will override all other conditions, and that brings the FSM into an 'exception state'

The purpose of an exception is to **abort** the regular flow of control and to transfer control to a dedicated exception-handler

An exception may have internal causes, such as an overflow condition in a datapath, or external causes, such as an interrupt

To model this, we need to introduce state transitions ***out of all states***, and re-work all state transition conditions to reflect the proper priority

The effect of these modifications on the graph is a drastic increase in *complexity*, with a spaghetti-like result

Consider adding an exception input called ***exc*** to the above FSM

Requires an immediate transition to state *A1*

Limitations of FSMs

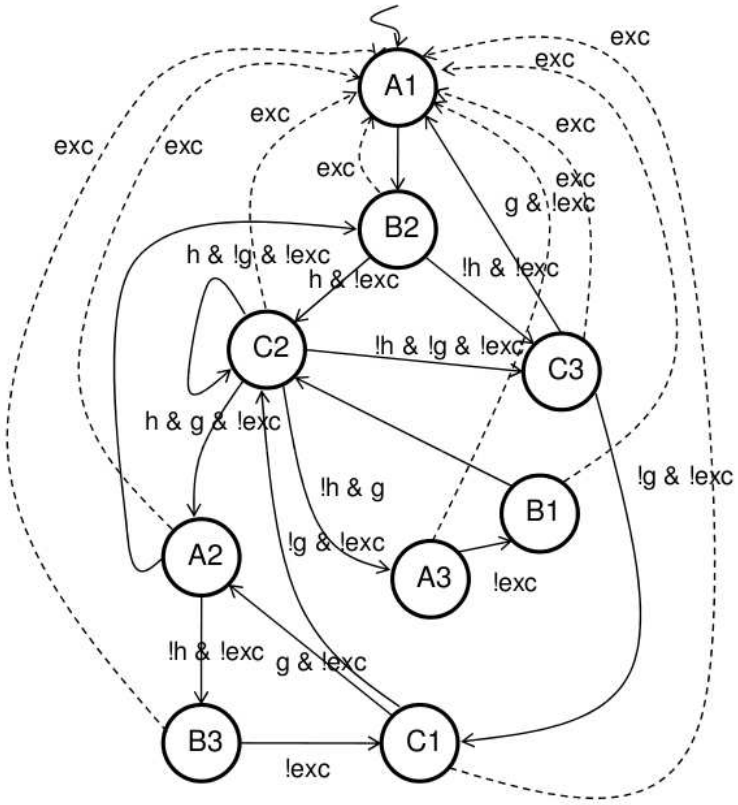


Fig. 5.2 Adding a single global exception deteriorates the readability of the FSM significantly

Perhaps the biggest issue from the viewpoint of hardware-software codesign, a FSM is a **non-flexible** model

Once the states and state transitions are defined, the control flow of the FSM is **fixed and hardwired**

Limitations of FSMs

To deal with *flexibility*, *exceptions*, and *hierarchical modeling*, designers use other techniques, e.g., **Microprogramming**, for specifying and implementing control

Microprogramming was orginally introduced in the 1950's by Maurice Wilkes

Objective was to create a *programmable instruction-set* for mainframes

Became very popular in the 1970's and throughout the 1980's as a means to develop complex microprocessors

Currently (2008), microprogramming is less popular and *flexibility* is almost always implemented on microprocessors, in **software**

However, newer architectures, such as **FPGAs** and **ASIPs**, suggest that *flexibility* is **not** the exclusive domain of software

We investigate microprogramming because it illustrates how hardware circuit design can incorporate *flexibility* and full *customizability*

Microprogrammed Control

A micro-programmed machine next to an FSMD

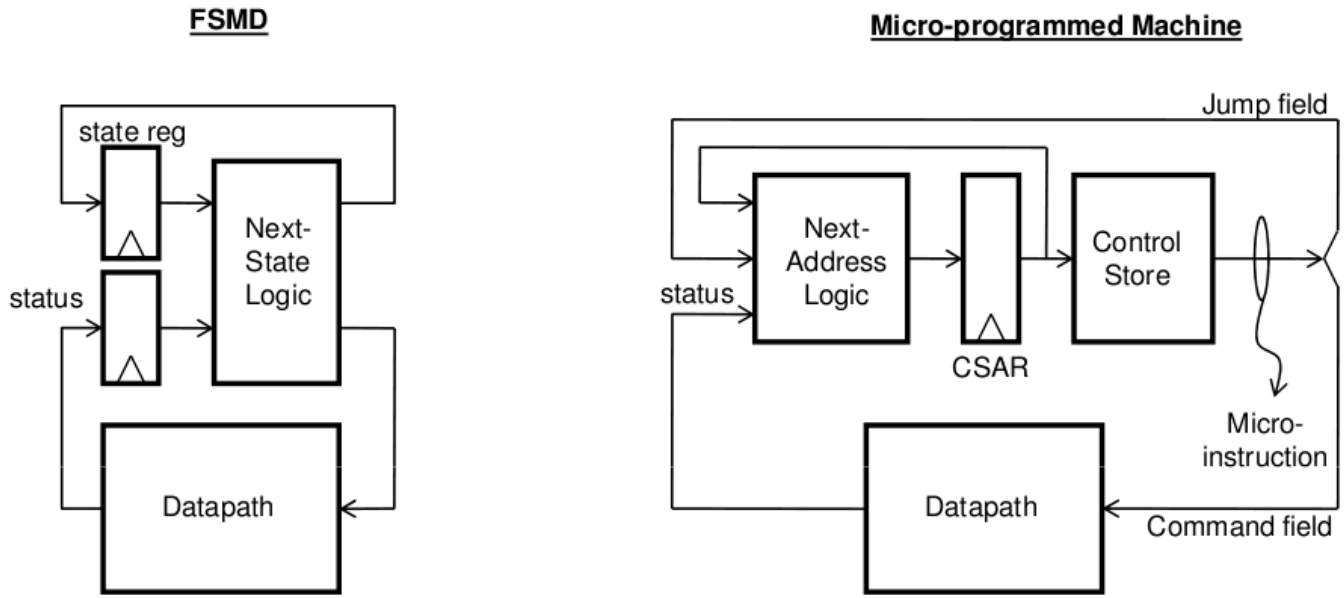


Fig. 5.3 In contrast to FSM-based control, microprogramming uses a flexible control scheme

The fundamental idea of microprogramming is to **replace** the *next-state logic* of a FSM with a **programmable memory**, called the **control store**

The **control store** holds micro-instructions, and is addressed using a register called **CSAR** (Control Store Address Register)

Microprogrammed Control

This register is the equivalent of a *program counter* in a microprocessor

The next-value of CSAR is determined by the next-address logic, using

- The current value of CSAR
- The current micro-instruction
- The value of status *flags* evaluated by the datapath

The default next-state value is (CSAR + 1)

In addition, the next-address-logic also implements **conditional** and **absolute** jumps

The next-address logic, the CSAR, and the control store implement the equivalent of an *instruction-fetch cycle* in a microprocessor

From the figure, each micro-instruction takes a single clock cycle to execute

Microprogrammed Control

Within a single clock cycle, the following activities occur:

- The CSAR provides an address to the control store which retrieves a micro-instruction

  The micro-instruction is split in two parts: a **command-field** and a **jump-field**

  The *command-field* serves as a command for the datapath

  The *jump-field* 'points' to the next-address logic

- The datapath executes the command encoded in the micro-instruction, and returns status information to the next-address logic

- The next-address logic combines datapath states, micro-instruction jump-field and status returned from the datapath

  The next-address logic will eventually update the CSAR

  The **critical path** of the micro-programmed machine is determined by the delay through the control store, the next-address logic, and the datapath

Principles of Microprogramming

While the micro-programmed controller is more complicated than the FSM, it also addresses the problems of FSMs very effectively

- The micro-programmed controller **scales** well with complexity

  For example, a 12-bit CSAR will allow a control store with up the 4096 locations, and therefore a micro-program with 4096 steps

  An equivalent FSM diagram with 4096 states, on the other hand, would be horrible to draw!

- A micro-programmed machine deals very well with **control hierarchies**

  Small modifications to the microprogrammed machine show above allow *pushing* and *popping* of the CSAR for sub-routine calls

- A micro-programmed machine can deal efficiently with **exception handling**, since global exceptions are managed directly by the next-address logic

  For example, the presence of a global exception can feed a hard-coded value into the CSAR, immediately transferring control to an exception-handler

Principles of Microprogramming

Therefore, exception handling does **not affect every instruction** of a micro-program in the same way as it affects every state of a FSM

- Micro-programs are **flexible** and very easy to change after the micro-programmed machine is designed

  Simply *changing the contents* of the control store is sufficient to change the program of the machine

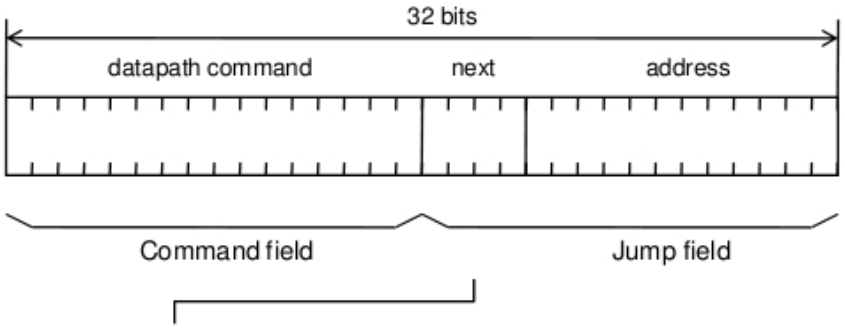
  Here, there is a clear distinction between the architecture of the machine and the functionality implemented using that architecture

Micro-Instruction Encoding

An interesting design problem is deciding on the format of micro-instructions in the control store

A sample format for a 32-bit micro-instruction word is shown below

Micro-Instruction Encoding



next

0000	Default	CSAR = CSAR + 1
0001	Jump	CSAR = address
0010	Jump if carry	CSAR = cf ? address : CSAR + 1
1010	Jump if no carry	CSAR = cf ? CSAR + 1 : address
0100	Jump if zero	CSAR = zf ? address : CSAR + 1
1100	Jump if not zero	CSAR = zf ? CSAR + 1 : address

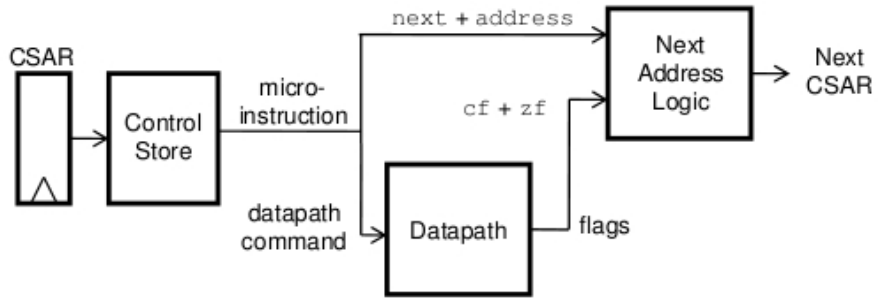


Fig. 5.4 Sample format for a 32-bit micro-instruction word

Micro-Instruction Encoding (Jump Field)

Of the 32-bit micro-instruction word, **16 bits** are reserved for the datapath and **16 bits** are reserved for the next-address logic

The *next-address* field holds an absolute target address, pointing to a location in the control store

The address is 12 bit, which allows a control store as large as 4096 locations

The *next* field encodes the operation that will lead to the next value of CSAR

As mentioned, the default operation is to increment CSAR

For such instructions, the address field remains **unused**

The *next* field allows various jump instructions can be encoded

An **absolute jump** transfers the value of the address field into CSAR

A **conditional jump** will use the value of a *flag* to conditionally update the CSAR (or just increment it)

Micro-Instruction Encoding (Command Field)

As indicated above, allocating 12 bits for the *address* field is wasteful

Only about 1 in every 5 instructions is a *jump*

There are ways to optimize this so that the *address* field bits are used for another purpose when the micro-instructions is ***not*** a jump instruction

Command Field

There are two approaches at micro-instruction encoding

- Horizontal microcode, where **no** encoding is done (or just a minimal amount)
- Vertical microcode, where the **maximal** amount of instruction encoding is done

A **wide** (horizontal) micro-instruction word allows each control bit of the data path to be stored separately

A **narrow** micro-instruction word, on the other hand, will require the creation of *symbolic instructions*, which are **encoded** groups of control-bits for the datapath

Therefore, a **few** bits of the micro-instruction define the value of **many** control bits in the data-path

Micro-Instruction Encoding (Command Field)

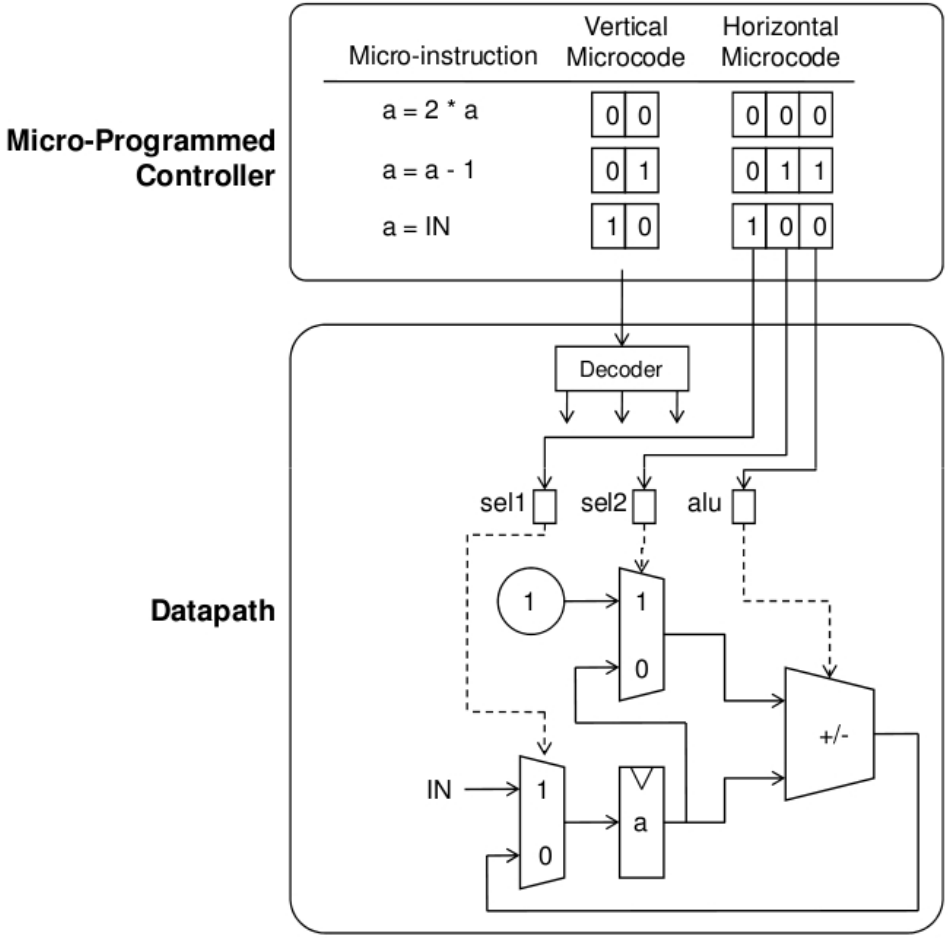


Fig. 5.5 Example of vertical versus horizontal micro-programming

Here, we create a micro-programmed machine with three instructions on reg *a*

Micro-Instruction Encoding (Command Field)

The three instructions do one of the following

- Double the value in *a*,
- Decrement the value in *a*, or
- Initialize the value in *a*

The datapath shown along the bottom of the figure contains two multiplexers and a programmable adder/subtractor

The controller on top shows two possible encodings for the three instructions: a horizontal encoding, and a vertical encoding

- For horizontal, the control store includes each of the control bits in the datapath directly (**3 bits**)
- For vertical, the micro-instructions are encoded with a **two-bit** micro-instruction word, and a *decoder* is used

So what is the design trade-off between horizontal and vertical microprograms?

Micro-Instruction Encoding (Command Field)

Vertical micro-programs have a better **code density**, which is beneficial for the size of the control store.

From the figure, the vertically-encoded version of the microprogram will be only **2/3rds** of the size of the horizontally-encoded version

On the other hand, vertical micro-programs use an **additional level of encoding**, and need **decoding** before it can drive the control bits of the datapath

Thus, the machine with the vertically encoded micro-program may have a **longer critical path**

In practice, designers use a *combination* of vertical and horizontal encoding concepts, so that the resulting digital structure is compact yet efficient

Consider for example the value of the *next* field of the micro-instruction word

There are *six different types* of jump instructions, which would imply that a vertical micro-instruction needs **no** more then **three** bits to encode these six jumps

Micro-Instruction Encoding (Command Field)

Yet, **four bits** have been used, indicating that there is some redundancy

The encoding was chosen to simplify the design of the next-address logic

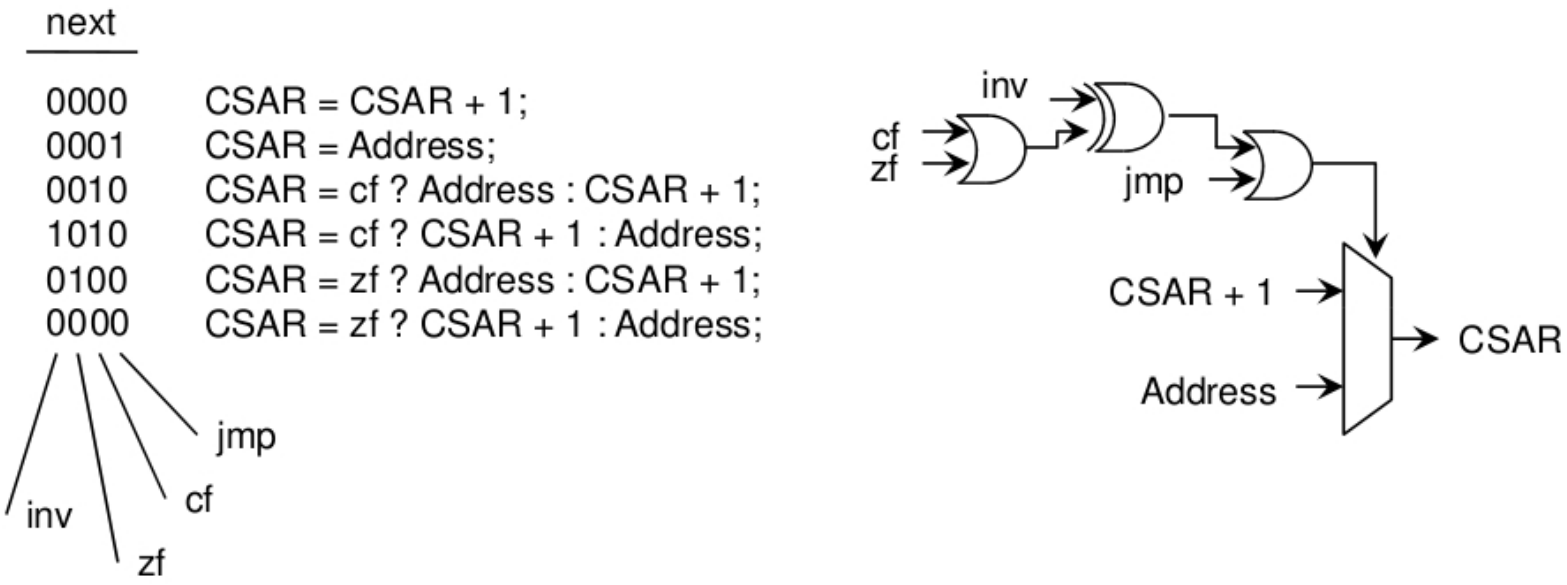


Fig. 5.6 Example of vertical versus horizontal micro-programming

Another reason to leave 'room' in the encoding is to allow future upgrades

For example, it is quite easy to add an additional conditional jump that uses an arbitrary combination of *cf* and *zf*