Microprogramming: architectures and control

Lecture 06 on Dedicated systems

Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2019-20

Table of Contents

outline:

- limitations of FSMs
- microprogramming: origins, microprogrammed interpreters
- microprogrammed control: architecture, benefits
- microinstruction encoding
- microprogrammed datapath
- writing microprograms
- examples:
  - microprogrammed architecture
  - microinstruction encoding
  - microprogram for GCD computation

FSM models are well suited to capture the control flow and decision making of algorithms, however, they lack *hierarchy*; this gives rise to severe limitations when dealing with complex control systems

state explosion
: the size of the state space of a *product FSM* is the product of the state space sizes of the component FSMs; even worse, if these have independent transition conditions, the number of conditions of the product FSM grows exponentially

exception handling
: exceptions are conditions that require transition to an exception handling state regardless of the current state of the machine when they occur; adding exception transitions to a given transition diagram often obfuscates the main course of control

runtime flexibility
: the hardwired control flow defined by an FSM cannot be changed in any other way than replacing the implemented FSM with another one; the motivation for greater model flexibility is tied to that for greater hardware flexibility

a more flexible control is obtained by *microprogramming* it

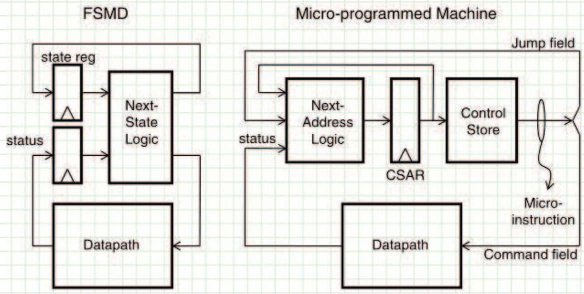
- the fixed schedule by an FSM controller is replaced by the variable one that is determined by a *microprogram*, composed of *microinstructions*, each of which is translated into datapath control signals

the first idea of microprogramming was proposed by Maurice Wilkes, in 1951, but it found wide application starting from the sixties, to become dominant in the subsequent decade with the diffusion of CISC architectures (*Complex Instruction-Set Computer*)

- in this idea, the microprogram is an *interpreter*, resident in a small control store
- the microinterpreter fetches machine instructions from main memory (that here are seen as higher-level instructions) and executes each of them by a *microroutine*, that is a sequence of low-level *microinstructions*
  - the microarchitecture of a CISC processor thus takes the shape of a "processor inside the processor", with a fixed microprogram which, during every machine cycle, fetches an instruction and executes the microroutine that is determined by decoding the opcode of the fetched instruction

starting from the eighties, RISC architectures (*Reduced Instruction-Set Computer*) have competed with CISC ones, to become dominant eventually

- microprogramming is still a very useful technique to increase the flexibility of hardware design

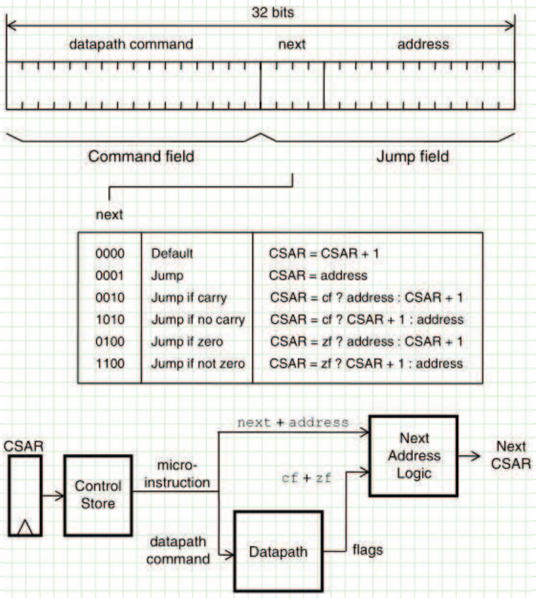


Schaumont, Figure 6.3 - In contrast to FSM-based control, microprogramming uses a flexible control scheme

CSAR (*Control Store Address Register*): analogue of the conventional Program Counter

- clock cycle determined by the critical path through the microprogrammed architecture microprogrammed control solves the problems of FSMs:

- it scales very well with complexity, for example a 12-bit CSAR may address a 4K-instruction control store, whereas a 4K-state FSM would be extremely complicated
- with small additions to the architecture in figure 6.3 hierarchical control may be easily implemented, for example, by adding a register or a stack to save and restore the CSAR, one may define microinstructions for microsubprogram call/return
- exception handling is also easy to deal with, by the next-address logic which would feed the CSAR with the hard-coded address of an exception handling microroutine
- the hardware flexibility advantage is evident, as datapath control may be modified by just rewriting the control store

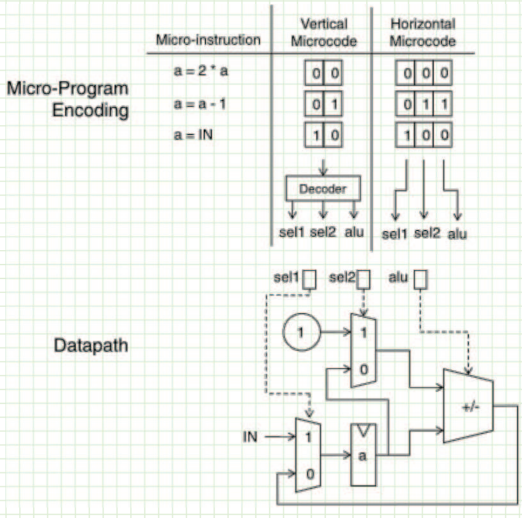


Schaumont, Figure 6.4 - Sample format for a 32-bit micro-instruction word

microinstruction format and encoding is driven by design trade-offs; a sample encoding is as follows

- assumption: 32-bit micro-instruction size, half for the datapath command, the other half for the next-address logic; we start with the latter
- 12-bit address field → up to 4K microinstructions in the control store
- 4-bit next field: selects how to compute the next address to be loaded onto CSAR, see table in the figure

the format in figure 6.4 is not optimal, as the address field is only used for jump instructions—it may be used for other purposes with other instructions



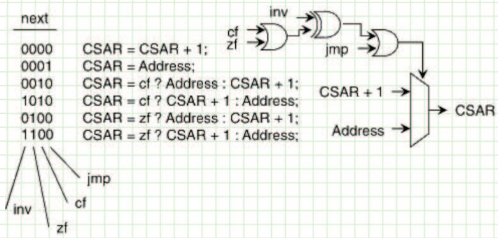
Schaumont, Figure 6.5 - Example of vertical versus horizontal micro-programming

another space-time trade-off is presented by the alternative for the command field:

horizontal encoding: each datapath control bit is assigned a distinct bit

vertical encoding: shortest encoding of datapath control bits

a combined solution is often adopted, *e.g.* the encoding in the next field:



Schaumont, Figure 6.6 - CSAR encoding

the datapath of a microprogrammed machine consists of three elements:

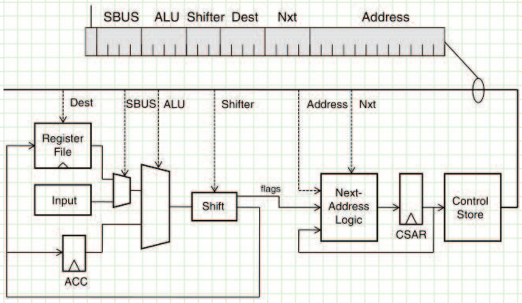
- computation units
- storage units (register file)
- communication buses

each of these elements may contribute a few control bits to the microinstruction word, for example:

- multi-function computation units: function selection bits
- storage units: address bits and read/write command bits
- communication buses: source/destination control bits

the datapath may also generate status flags for the microprogrammed controller here is an example of microprogrammed control of a datapath that includes: an ALU with shifter unit, a register file with eight entries, an accumulator register, and an input port

mixed horizontal/vertical encoding: overall horizontal, for each unit in the datapath takes a distinct portion of the control word, vertical encoding of each unit control signals in that portion



Schaumont, Figure 6.7 - A micro-programmed datapath

the shifter also generates flags, which are used by the microprogrammed controller to implement conditional jumps

control word fields:

- Nxt, Address: used by the microprogrammed controller; the other fields are used by the datapath
- ALU: up to 16 ALU operations may be encoded
- SBUS: source operand selection for the ALU operation, out of entries in the register file and input port, the other source operand is the accumulator
- Dest: destination selection for the ALU+shifter operation, out of entries in the register file and accumulator
- Shifter: shift function selection, up to eight functions

the datapath fetches and executes a microinstruction every clock cycle

table 6.1 presents an example of microinstruction encoding for the given architecture (first part):

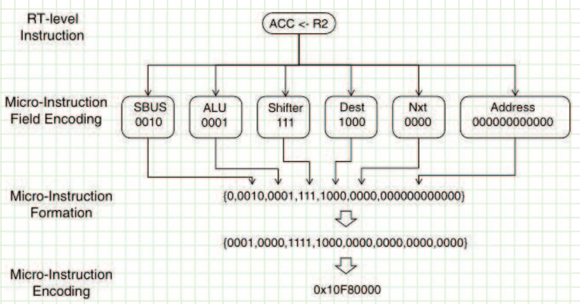
Field	Width	Encoding			
SBUS	4	Selects the operand that will drive the S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	Input
		0100	R4	1001	Address/Constant
ALU	4	Selects the operation performed by the ALU			
		0000	ACC	0110	ACC | S-Bus
		0001	S-Bus	0111	not S-Bus
		0010	ACC + S-Bus	1000	S-Bus + 1
		0011	ACC – S-Bus	1001	ACC + 1
		0100	S-Bus – ACC	1010	0
		0101	ACC & S-Bus	1011	1

table 6.1 (second part):

Field	Width	Encoding			
Shifter	3	Selects the function of the programmable shifter			
		000	logical SHL(ALU)	100	arith SHL(ALU)
		001	logical SHR(ALU)	101	arith SHR(ALU)
		010	rotate left ALU	111	ALU
		011	rotate right ALU		
Dest	4	Selects the target that will store S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	ACC
		0100	R4	1111	unconnected
Nxt	4	Selects next-value for CSAR			
		0000	CSAR + 1	1010	cf ? CSAR + 1 : Address
		0001	Address	0100	zf ? Address : CSAR + 1
		0010	cf ? Address : CSAR + 1	1100	zf ? CSAR + 1 : Address

using the encoding defined in table 6.1, a microinstruction is formed by selecting a function for each module in the datapath and a next address for the Address field (with a suitable don't care value for this whenever Nxt is null)

by way of example, let's see how an RTL instruction, such as ACC ← R2, is translated to a microinstruction



Schaumont, Figure 6.8 - Forming micro-instructions from register-transfer instructions

- the source operand is in R2: SBUS = 0010
- the ALU passes the S-Bus input to the output: ALU = 0001
- The shifter passes the ALU output unmodified: Shifter = 111
- the output of the shifter updates the accumulator: Dest = 1000
- CSAR gets the default increment: Nxt = 0000 and Address is a don't care, e.g. all zeroes

complex control operations, such as loops and if-then-else statements, can be expressed as a combination (or sequence) of RTL instructions

as an example, let's develop a micro-program that computes the GCD of two input numbers using Euclid's algorithm

the microprogram is written in a symbolic RTL notation that immediately translates to microinstructions in a similar way as in the previous example

	Command Field	||	**Jump Field**
	IN → R0		
	IN → ACC		
Lcheck:	(R0 – ACC)	||	JUMP_IF_Z Ldone
	(R0 – ACC) << 1	||	JUMP_IF_C Lsmall
	(R0 – ACC) → R0	||	JUMP Lcheck
Lsmall:	ACC – R0 → ACC	||	JUMP Lcheck
Ldone:			JUMP Ldone

Schaumont, Listing 6.1 – Micro-program to evaluate a GCD

recommended readings:

Schaumont, Ch. 6, Sect. 6.1-6.4

for further consultation:

Schaumont, Ch. 6, Sect. 6.6-6.8