Microprogrammed Control

Review Notes

Yul Williams, D.Sc.

A Special Note

- Most of the content of this file were excerpted from the text: “Computer Systems Organization and Architecture” by John Carpinelli.
- The balance of the content is provided as original material from the course instructor.

Goals

- At the conclusion of this session, the student should be able to:
- Understand the basic purpose and operation of a microsequencer
- State the 3 methods of designing microsequencers
- Design and implement a microsequencer using the 3 methods

Outline

- Microsequencers
- Basic Microsequencer Design
- Microinstruction Formats
- A Very Simple Microsequencer
- Sequence Generation and Mapping Logic Design
- Horizontal Microcode
- Vertical Microcode
- Direct Control Signal Generation
- Summary

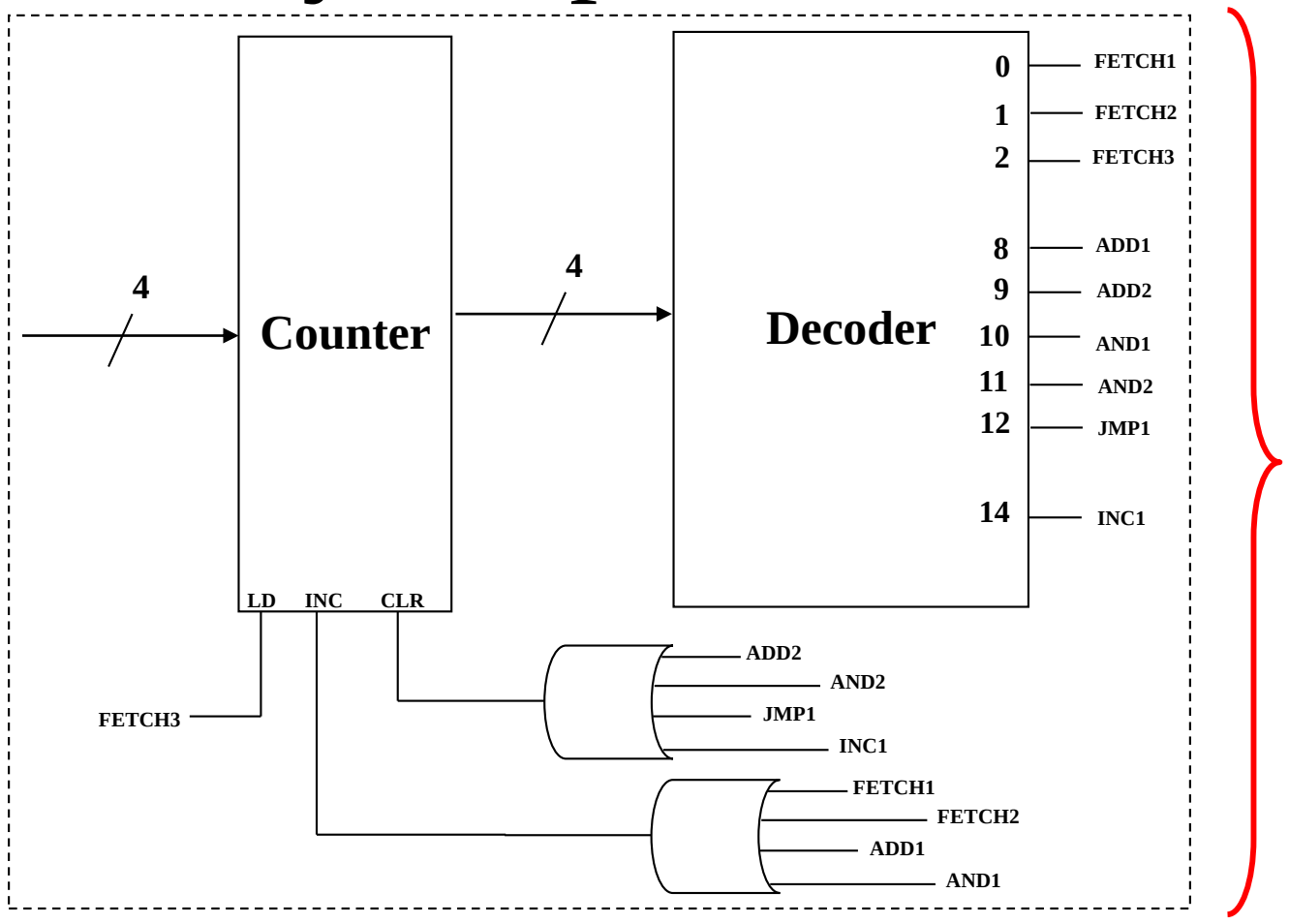
Microsequencers

What is a Microsequencer?

- A control mechanism that is used to govern the operations of a CPU
- It operates differently than the hardwired approach demonstrated in the Very Simple CPU design

Hardwired Control Unit for The Very Simple CPU

We are replacing this…



Does the Logic Look Familiar?

- As you viewed the previous hard-wired control logic design, the rationale for spending the time learning logic design techniques may be readily apparent.
- Even when the hardware logic is not used to represent the functionality, it will be represented in the software logic, where applicable.

Generating Control Signals

- Its control signals are store in microcode memory (ROM-based) approach. This memory is often referred to as the lookup ROM
- This approach does not depend on combinational and sequential logic to generate the proper control signals

Lookup ROM

- Unlike the hardwired control approach, the control signals are asserted when the correct lookup ROM address is accessed
- This method is widely used in complex microprocessors to reduce the design complexity of the control unit

Microsequencer for the Very Simple CPU

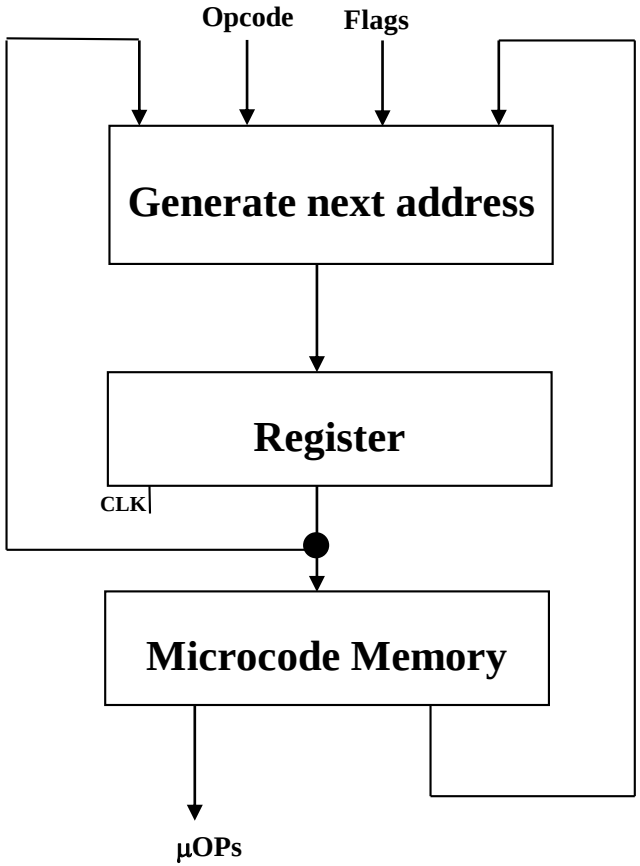
- In order to compare the hardwired approach with the microsequenced approach, we will design a microsequencer for the Very Simple CPU model that was constructed in the last class session
- We will examine how to store control signal values in the lookup ROM
- We will also examine how to optimize on the number of control words that are necessary for our design

Basic Microsequencer Design

Microsequencer Operations

- Microsequencers are designed as finite state machines (FSMs)
- Remember that we are considering only the control aspect of the CPU. All other parts remain static
- The following diagram provides a generic view of a microsequencer

Generic Microsequencer Organization



The Register

- The register is used to store one state of the CPU state diagram
- The value held in the register is used to address the microcode memory
- When the microcode memory receives an address, it outputs a microinstruction that corresponds to the address

Microcode

- Microcode is the collection of all of the microinstructions for the CPU
- Sometimes called a microprogram

Microinstructions

- Microinstructions are composed of several bit fields
- Each bit field can be divided into two broad groups
  - Micro-operations (the “micro” part of microsequencer)
  - Address generation (the “sequencer” part of microsequencer)

Micro-operations

- These signals are output from the microsequencer to the rest of the CPU
- They are provided as input to combinational logic to generate CPU control signals
- They may also directly produce control signals
- When viewing a state diagram, these are the state outputs

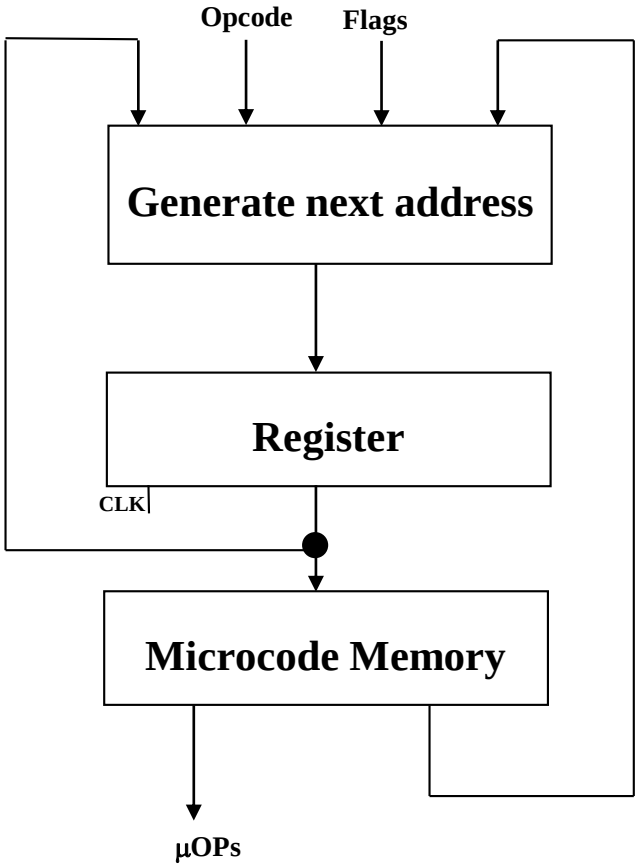
Address Generator

- The address generator outputs are combined with the instruction opcode and flag bits as input to combinational logic to produce the address of the next microinstruction
- On a state diagram, this is the equivalent of changing states
- A transition occurs based on the current state and its input values

Common Address Generation Schemes

- Current address + 1
- Absolute address
- Mapping function
- Subroutine return address
- Note – Every microsequencer uses the mapping address to go from the last state of the fetch routine to the correct execute routine

Microsequencer Operation



The Generate next address block all possible next addresses and passes the selected address to the register.

Microcode routines typically occupy sequential locations in microcode memory. This improves readability and helps in the debugging process. A microsequencer typially uses a parallel adder to generate the values of the current address plus 1 as a possible next address.

Another possible address is an absolute address supplied by the microcode memory. This is necessary for branching microinstructions.

Microsequencers must be capable of accessing the correct execution routines. Mapping logic is used to perform this function. The opcode of the fetched instruction is input to the mapping hardware which converts it into the address of the first microinstruction of the instructions execute routine. The mapping hardware is used only at the end of the fetch cycle.

Micro-subroutines

- Microsequencers may have subroutines
- When several instruction have common sequences of microinstructions, it is more efficient to make the sequences into subroutines
- When a subroutine is called, the address is specified by the microcode memory as an absolute address
- The return address (current address + 1) is stored in the microsubroutine register (hardware stack)

Microinstruction Formats

Generic Microinstruction Format

Microinstruction Format

- Each microsequencer may have its own format
- The order of the fields may vary between differing CPUs
- They all (in general) have representations of the select, address and micro-operations fields

The SELECT Field

- The SELECT field specifies the source of the address for the next microinstruction
- It does not specify the actual address itself. It only specifies the source of the address

The ADDR Field

- The ADDR field specifies an absolute address
- This address is used when performing an absolute jump instruction
- All microinstructions may not use this field. This is especially true for microinstructions that explicitly specify another source for the next address

The MICRO-OPERATIONS Field

- There are 3 different methods to specify micro-operations:
  - Horizontal microcode
  - Vertical microcode
  - Direct generation of control signals

Horizontal Microcode

- Horizontal microcode is a scheme whereby each microinstruction that is performed by the CPU is listed in a horizontal bit pattern
- First, every micro-operation performed by the CPU is listed
- A bit is then assigned in the micro-operations field in the microinstruction for each micro-operation
- If a CPU performs a total of 50 different micro-operations, the micro-operations field would use 50 bits for every microinstruction

Vertical Microcode

- One way to reduce the number of bits in the micro-operations field is to use vertical microcode
- In vertical microcode, the micro-operations are grouped into fields
- Each micro-operation is assigned a unique encoded value within the field

Vertical Microcode cont'd

- 16 microinstructions, for example, may be encoded using 4 bits from the micro-operations field with the microinstruction
- Even though fewer bits are necessary, a decoder is required in order to generate the actual micro-operation signals

Direct Generation of Control Signals

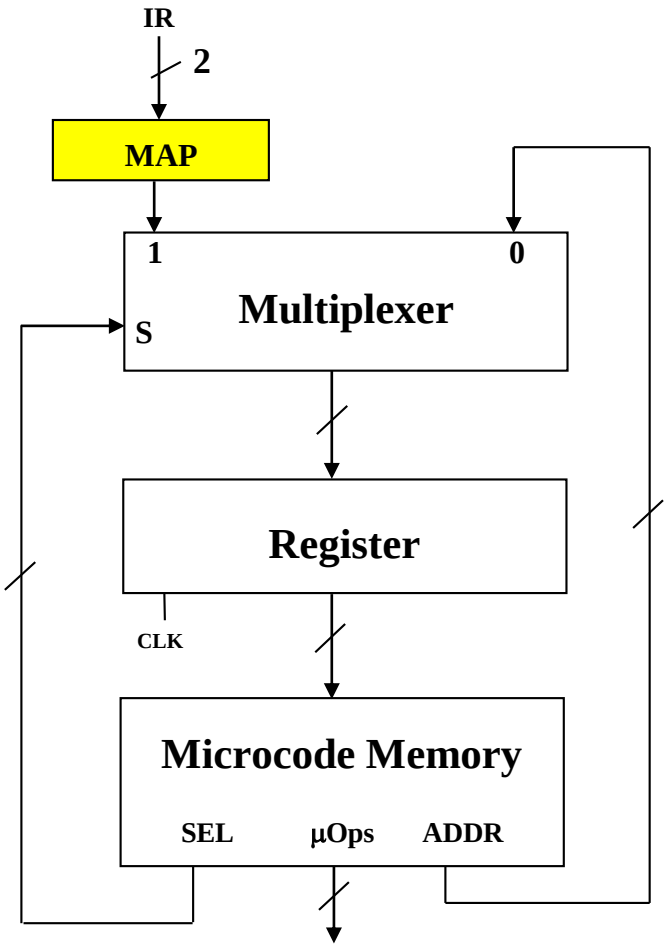
- The direct generation of control signals method store the values of the control signals directly in the microinstruction
- No excessively long micro-operations fields or decoders are required for this method
- This method is less readable and thus harder to debug

A Very Simple Microsequencer

Microsequencer Design for the Very Simple CPU

- We will use a microsequencer instead of hardwired control logic to generate the proper signals
- Key points:
  - There is no change in flow of data within the CPU, only in the manner that the control signals are generated
  - The CPU does not need to be designed from scratch! The instruction set, FSM, data paths and ALU are all the same

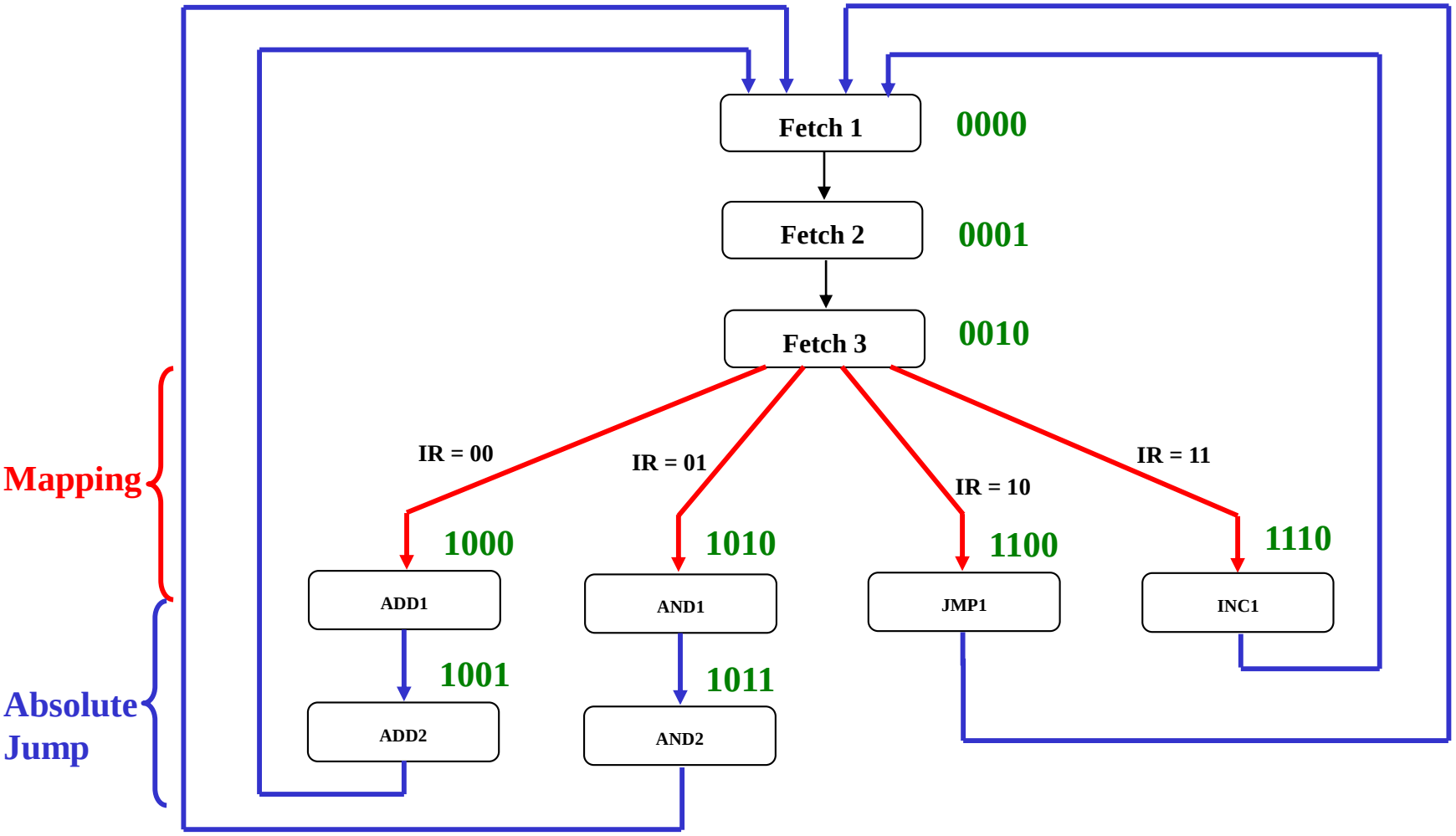
Very Simple CPU Microsequencer



Next Address Generation

- In this design, there are only 2 possible next addresses:
  - The opcode mapping
  - The absolute jump
- In the following diagram of the CPU FSM, FETCH3 goes to 1 of 4 possible execute routines. This must be implemented with a mapping input
- Each of the other states must go to another specific state. This is handled via absolute jumps
- Let’s take a look…

Next Address Generation for CPU



Selecting the Next Address

- Since there are two possible sources of next addresses, a mechanism must be used to make a selection
- This mechanism is the multiplexer or MUX
- The SEL (select) line is generated by the microsequencer and used to choose the proper next address
- In every state, this CPU knows (with absolute certainty) the source of its next address. This is not the case with all CPUs

Sequence Generation and Mapping Logic Design

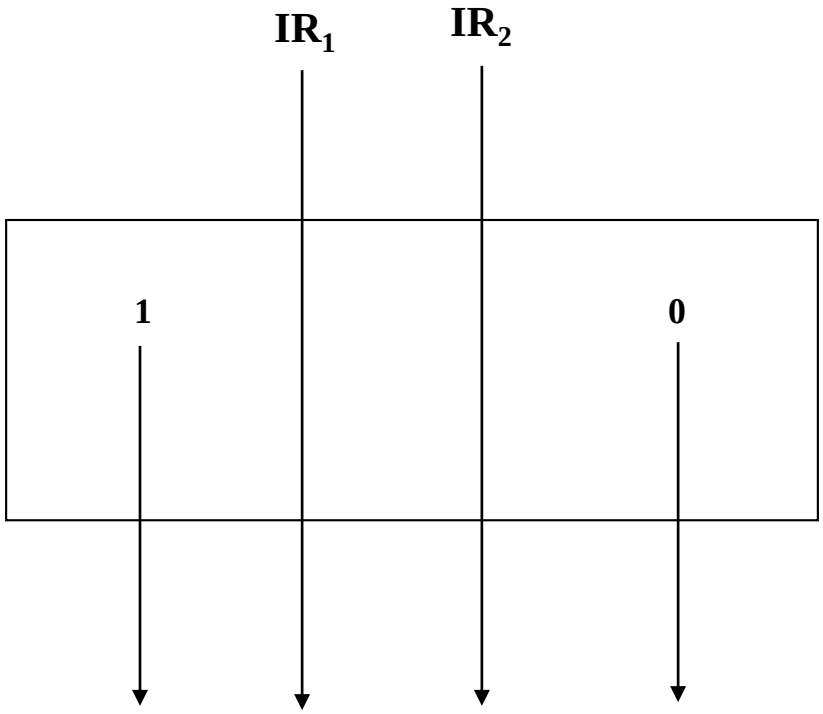
Generating the Correct Sequences

- This step is to design the portion of the microcode that sequences through the states of the FSM
- In order to do this, we assign each state of the FSM to an address in the microcode memory
- We want to assign the addresses of the first state in each execute routine

Microsequencer Mapping Function

- We will use the same mapping function from the Very Simple CPU design
- The mapping function 1 IR[1..0] 0 is used to produce the addresses
  - 1000 : ADD1
  - 1010 : AND1
  - 1100 : JMP1
  - 1110 : INC1
- The required hardware for this mapping is a few wires

Microsequencer Mapping Function



Microsequencer State Addresses

State	Address
FETCH1	0000
FETCH2	0001
FETCH3	0010
ADD1	1000
ADD2	1001
AND1	1010
AND2	1011
JMP1	1100
INC1	1110

Microcode Preparation

- The next step is to set up the microcode to sequence through states properly
- To go unconditionally to another state, the microsequencer places the address of the target state into the ADDR and uses the SEL to select the absolute address
- When SEL = 0, the address is selected from the ADDR field of the microinstruction

Microcode Preparation

- All states in the FSM use SEL = 0, except for the FETCH3 state, which maps to the proper execute routine
- When SEL = 1, the contents held in the ADDR field are not used in the microinstruction
- The next address is selected from the mapping function (MAP)

Partial Microcode Listing

State	Address	SEL	ADDR
FETCH1	**0000**	**0**	**0001**
FETCH2	**0001**	**0**	**0010**
FETCH3	**0010**	**1**	**xxxx**
ADD1	**1000**	**0**	**1001**
ADD2	**1001**	**0**	**0000**
AND1	**1010**	**0**	**1011**
AND2	**1011**	**0**	**0000**
JMP1	**1100**	**0**	**0000**
INC1	**1110**	**0**	**0000**

Horizontal Microcode

Mnemonics and Micro-operations

Mnemonic	Micro-operation
ARPC	AR <-- PC
ARDR	AR <-- DR[5..0]
PCIN	PC <-- PC + 1
PCDR	PC <-- DR[5..0]
DRM	DR <-- M
IRDR	IR <-- DR[7..6]
PLUS	AC <-- AC + DR
AND	AC <-- AC ^ DR
ACIN	AC <-- AC + 1

Preliminary Horizontal Microcode

State	Address	SEL	ARPC	ARDR	PCIN	PCDR	DRM	IRDR	PLUS	AND	ACIN	ADDR
FETCH1	0000	0	1	0	0	0	0	0	0	0	0	0001
FETCH2	0001	0	0	0	1	0	1	0	0	0	0	0010
FETCH3	0010	1	0	1	0	0	0	1	0	0	0	xxxx
ADD1	1000	0	0	0	0	0	1	0	0	0	0	1001
ADD2	1001	0	0	0	0	0	0	0	1	0	0	0000
AND1	1010	0	0	0	0	0	1	0	0	0	0	1011
AND2	1011	0	0	0	0	0	0	0	0	1	0	0000
JMP1	1100	0	0	0	0	1	0	0	0	0	0	0000
INC1	1110	0	0	0	0	0	0	0	0	0	1	0000

Optimizing Horizontal Microcode

- Examine the microcode in the table to see if any of the micro-operations are identical
- If so, let's combine them into a single micro-operation
- ARDR and IRDR match in every state, so, they may be combined into a new micro-operation called AIDR
- Let's look at the resulting table

Optimized Horizontal Microcode

State	Address	SEL	ARPC	AIDR	PCIN	PCDR	DRM	PLUS	AND	ACIN	ADDR
FETCH1	0000	0	1	0	0	0	0	0	0	0	0001
FETCH2	0001	0	0	0	1	0	1	0	0	0	0010
FETCH3	0010	1	0	1	0	0	0	0	0	0	xxxx
ADD1	1000	0	0	0	0	0	1	0	0	0	1001
ADD2	1001	0	0	0	0	0	0	1	0	0	0000
AND1	1010	0	0	0	0	0	1	0	0	0	1011
AND2	1011	0	0	0	0	0	0	0	1	0	0000
JMP1	1100	0	0	0	0	1	0	0	0	0	0000
INC1	1110	0	0	0	0	0	0	0	0	1	0000

Control Signal Generation

- Once we have completed the microcode for the microsequencer, the actual control signals must be generated
- Our methodology is logically OR the micro-operations together to produce the control signals
- This yields the following results…

Control Signals for Microsequencer

Signal	Value
ARLOAD	ARPC v AIDR
PCLOAD	PCDR
PCINC	PCDIN
DRLOAD	DRM
ACLOAD	PLUS v AND
ACINC	ACIN
IRLOAD	AIDR
ALUSEL	AND
MEMBUS	DRM
PCBUS	ARPC
DRBUS	AIDR v PCDR v PLUS v AND
READ	DRM

Horizontal Microcode Pros and Cons

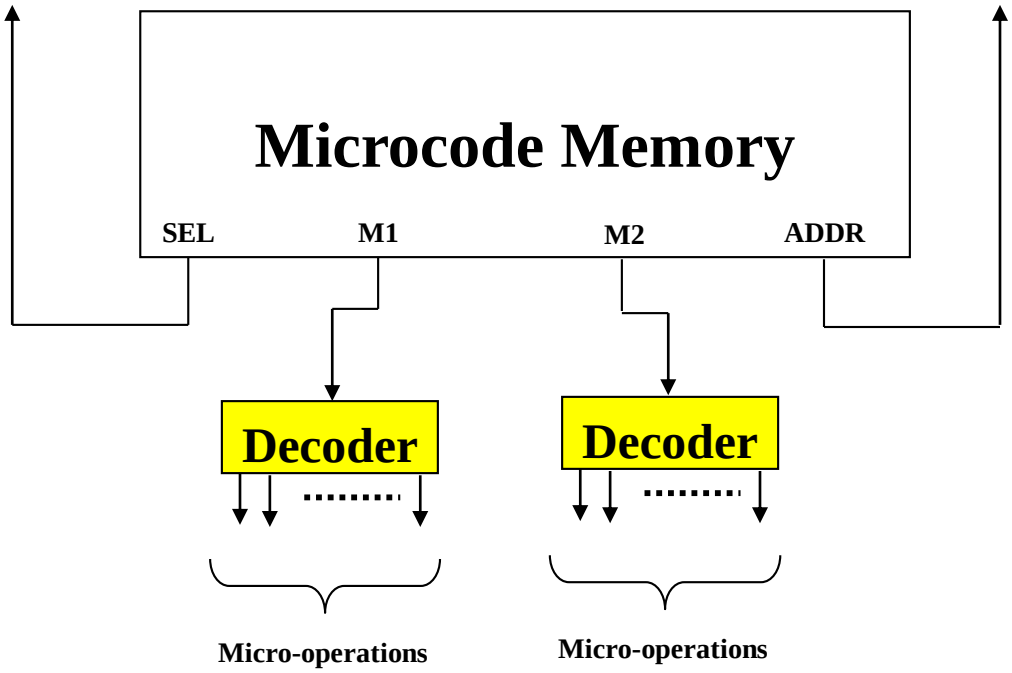
- Pros
  - Straightforward design methodology
  - A single bit is used to represent a micro-operation
- Cons
  - Each micro-operation requires a long bit pattern
  - Very few bits are active in each pattern

Vertical Microcode

Vertical Microsequencer Design

- Micro-operations are grouped into fields
- Each micro-operation has a unique pattern representation within that field
- A field of 4 bits, for instance, may represent up to 16 unique micro-operations
- The output of the microcode memory must be routed to a decoder that determines the micro-operations to execute

Vertical Microcode Generation of Micro-operations



Generic Representation of micro-operation generation using vertical microcoding

We're Going Vertical!!!

- What actually needs changing from the horizontal model? The answer is "not much"!
- We keep the same instruction set, state diagram, data paths and the ALU
- We also keep the sequencing hardware, mapping logic and the SEL and ADDR part of the microcode.
- We also keep the same micro-operations used in the horizontal model
- So what changes …? Let's see!!!

Allocating Micro-operations

- Notice in our generic vertical microcoded microsequencer, that there are two micro-operations fields. We are not limited to two fields.
- We have to allocate all of our micro-operations into the fields using a specific methodology
- Our methodology will be based on 4 heuristics
  - Def: Heuristic – encouraging the student to discover for himself. -- Webster
  - Def: Heuristic – your guess is a good as anyone else's guess. -- Yul

Heuristics for Micro-operation Allocation

- Whenever 2 micro-operations occur in the same state, assign them to different fields
- Include a NOP in each field if necessary
- Distribute the remaining micro-operations to make the best use of micro-operation field bits
- Aggregate micro-operations that modify the same registers in the same fields

Field Formation

- First, let’s examine the micro-operations for simultaneous operations
- DRM and PCIN both occur during FETCH2 and must be assigned to different fields
- PCIN and PCDR both modify PC
- After all of the juggling, the following table shows a possible assignment for the fields M1 and M2.

Initial Mapping

M1	M2
NOP	NOP
DRM	PCIN
ACIN	PCDR
PLUS	ARPC
AND	AIDR

Since each field contains 5 micro-operations, 3 bits are required to represent the range of unique micro-operations. This is a total of 6 bits. If we make some attempts to optimize the arrangement of micro-operations between the fields, we may be able to reduce the number of bits required to represent all of the micro-operations.

Optimized Mapping

	M1
Value	**Micro-operation**
000	**NOP**
001	**DRM**
010	**ARPC**
011	**AIDR**
100	**PCDR**
101	**PLUS**
110	**AND**
111	**ACIN**

Requires 3 bits

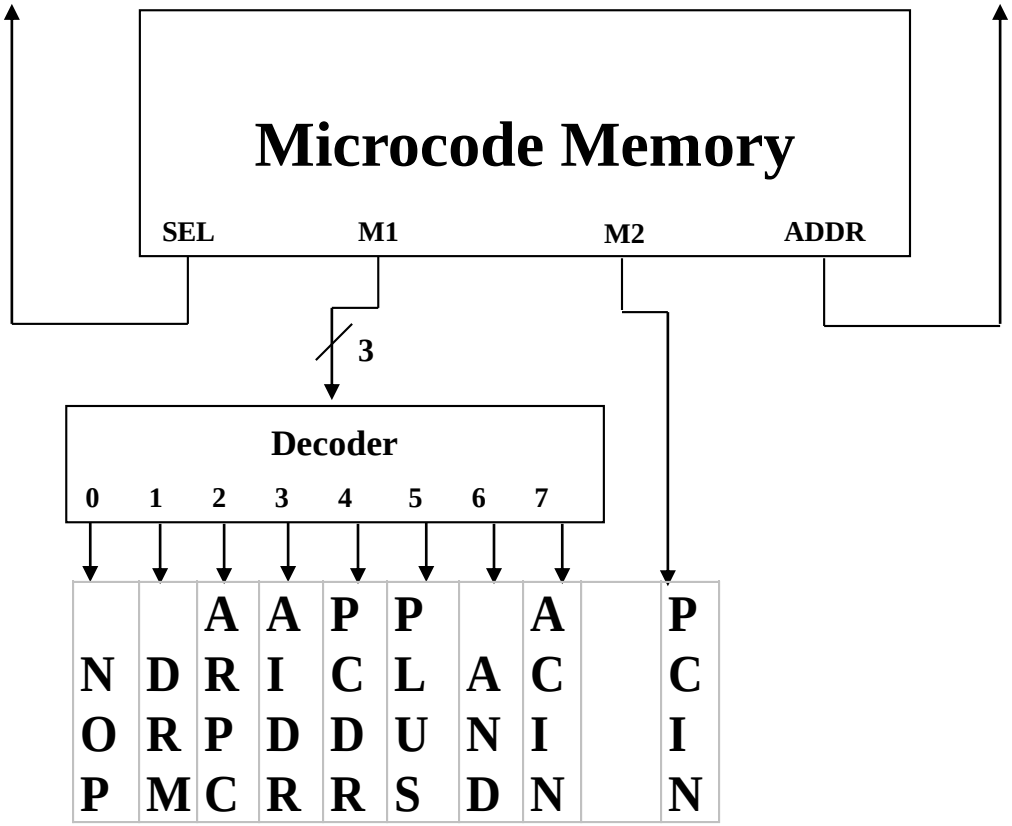
	M2
Value	**Micro-operation**
0	**NOP**
1	**PCIN**

Requires 1 bit

Vertical Microcode Listing

State	Address	SEL	M1	M2	ADDR
FETCH1	**0000**	**0**	**010**	**0**	**0001**
FETCH2	**0001**	**0**	**001**	**1**	**0010**
FETCH3	**0010**	**1**	**011**	**0**	**xxxx**
ADD1	**1000**	**0**	**001**	**0**	**1001**
ADD2	**1001**	**0**	**101**	**0**	**0000**
AND1	**1010**	**0**	**001**	**0**	**1011**
AND2	**1011**	**0**	**110**	**0**	**0000**
JMP1	**1100**	**0**	**100**	**0**	**0000**
INC1	**1110**	**0**	**111**	**0**	**0000**

Generating the Micro-operations



Vertical Microcode Pros and Cons

- Pros
  - Straightforward design methodology
  - More efficient use of bits in micro-operation fields
- Cons
  - Extra hardware is required to decode the micro-operations

Direct Control Signal Generation

Direct Control Signal Generation

- Instead of outputting micro-operations from the microcode memory and then generating the control signals, we can directly output the control signals from the microcode memory
- In place of each of the micro-operations, the microsequencer can include 1 bit for each control signal
- For each word in the microcode memory, each control bit is set to 1 if active and 0 if inactive
- Let's look at the following table to illustrate this point

Direct Control Signal from Microcode

State	Address	SEL	ARLOAD	PCLOAD	PCINC	DRLOAD	ACLOAD	ACINC	IRLOAD	ALUSEL	MEMBUS	PCBUS	DRBUS	READ	ADDR
FETCH1	0000	0	1	0	0	0	0	0	0	0	0	1	0	0	0001
FETCH2	0001	0	0	0	1	1	0	0	0	0	1	0	0	1	0010
FETCH3	0010	1	1	0	0	0	0	0	1	0	0	0	1	0	xxxx
ADD1	1000	0	0	0	0	1	0	0	0	0	1	0	0	1	1001
ADD2	1001	0	0	0	0	0	1	0	0	0	0	0	1	0	0000
AND1	1010	0	0	0	0	1	0	0	0	0	1	0	0	1	1011
AND2	1011	0	0	0	0	0	1	0	0	1	0	0	1	0	0000
JMP1	1100	0	0	1	0	0	0	0	0	0	0	0	1	0	0000
INC1	1110	0	0	0	0	0	0	1	0	0	0	0	0	0	0000

Optimization of Control Signals

- After examining the previous table, we can observe that the control signals DRLOAD, MEMBUS and READ are identical across all the states
- These signals will be consolidated into the DMR signal
- The optimized signals are shown in the following table

Optimized Direct Control Signal from Microcode

State	Address	SEL	ARLOAD	PCLOAD	PCINC	DMR	ACLOAD	ACINC	IRLOAD	ALUSEL	PCBUS	DRBUS	ADDR
FETCH1	0000	0	1	0	0	0	0	0	0	0	1	0	0001
FETCH2	0001	0	0	0	1	1	0	0	0	0	0	0	0010
FETCH3	0010	1	1	0	0	0	0	0	1	0	0	1	xxxx
ADD1	1000	0	0	0	0	1	0	0	0	0	0	0	1001
ADD2	1001	0	0	0	0	0	1	0	0	0	0	1	0000
AND1	1010	0	0	0	0	1	0	0	0	0	0	0	1011
AND2	1011	0	0	0	0	0	1	0	0	1	0	1	0000
JMP1	1100	0	0	1	0	0	0	0	0	0	0	1	0000
INC1	1110	0	0	0	0	0	0	1	0	0	0	0	0000

Direct Control Signal Generation Pros and Cons

- Pros
  - Straightforward design methodology
  - Faster to construct
  - Does not require micro-operation decoding
  - Most efficient use of bits in microcode memory
- Cons
  - Extremely hard to debug

Summary

- A microsequencer is used to generate control signals for a CPU
- It may be used instead of dedicated hardware
- The 3 basic methods of designing microsequencers include horizontal microcoding, vertical microcoding and direct signal generation from microcode memory