



UiO : **Department of Informatics**
University of Oslo

IN3160 IN4160
Microcode



Yngve Hafting 2020



Messages

- Monday 27.3 14:15-16:00, Logo:
 - Guest Lecture in FPGA Design by Espen Tallaksen, Founder and CEO EmLogic
- Master presentation Tuesday 28.3 at 14:15 @ROBIN (4th floor south)
 - For all potential applicants to master in Robotics and Intelligent systems
 - Pizza serveres for den som er meldt på
 - <https://nettskjema.no/a/329253>

In this course you will learn about the **design of advanced digital systems**. This includes programmable logic circuits, a hardware design language and system-on-chip design (processor, memory and logic on a chip). Lab assignments provide practical experience in how real design can be made.

After completion of the course you will:

- understand important **principles for design** and testing of digital systems
- understand the relationship between behaviour and different construction criteria
- **be able to describe advanced digital systems at different levels of detail**
- be able to perform simulation and synthesis of digital systems.

Course Goals and Learning Outcome

<https://www.uio.no/studier/emner/matnat/ifi/IN3160/index-eng.html>

Goals for this lesson:

- Know the principles used in microcoded state-machines
- Be able to describe how microcoded state machines can lead to microprocessors

Microcoded FSMs

- **FSM coded using memory (asynch. mem.)**

- Can be used for any FSM
- Input and state decides memory output

- **Single ROM solution**

- Both Mealy and Moore possible *depending on decoding...*
 - General solution is a Mealy machine (*Moore is a special case*).
- **ROM decoding added to critical path for downstream modules.**

- **Single ROM with output synchronization**

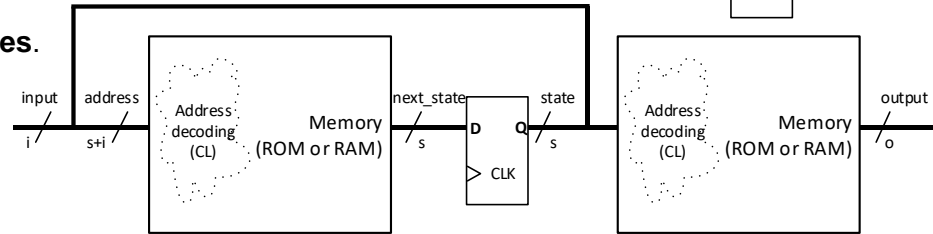
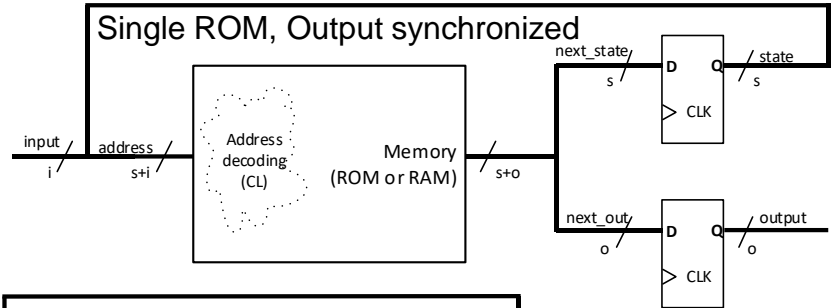
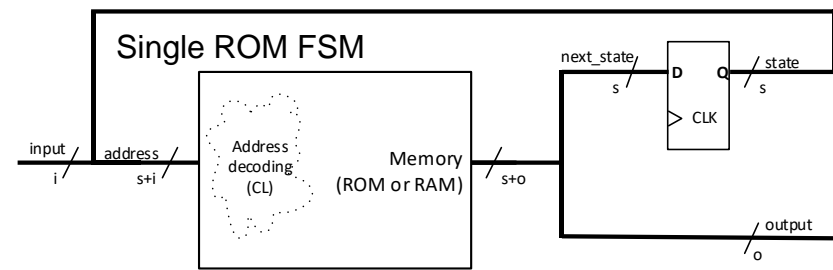
- No hazards, but output is delayed by one clock cycle

- **Dual ROM Moore machine**

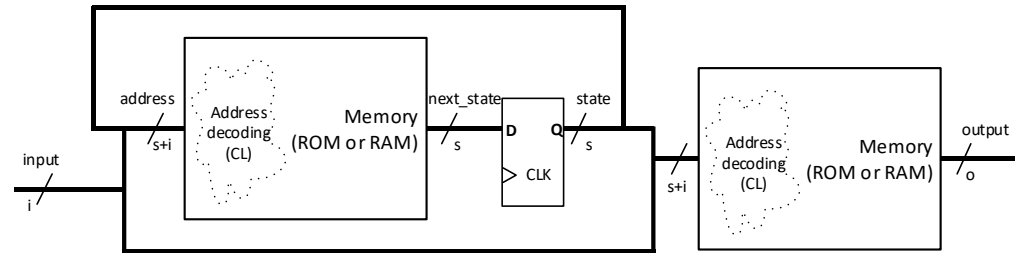
- Separate state and output decoding
 - Easier to comprehend
- Requires the least amount of storage
 - least impact on downstream critical without synchronizers.

- **Dual ROM Mealy machine**

- Both memories has the same address decoding
 - *No gains in terms of storage or critical path*



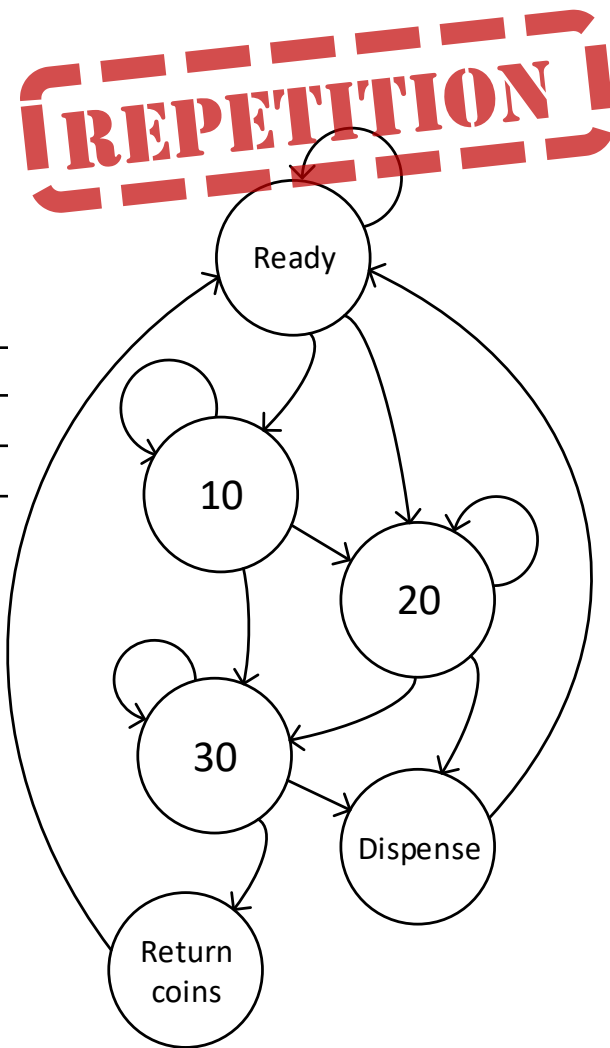
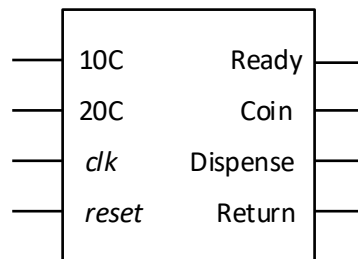
Dual ROM, Moore FSM



Dual ROM, Mealy FSM

Example: Vending machine

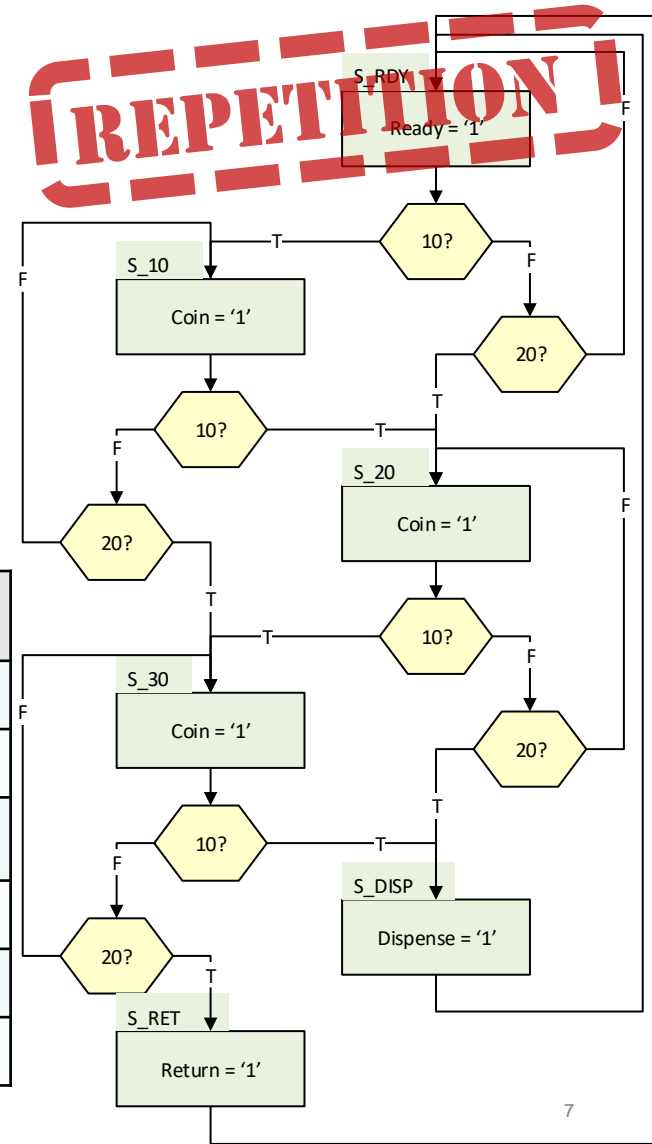
- Specification:
 - We want to design a vending machine that sells drinks for 40c.
 - The machine accepts 20c and 10c coins (all others will be rejected mechanically).
 - If 40c are inserted a drink shall be dispensed
 - If more than 40c is inserted all coins are returned
 - The machine has two lights
 - One to show that it is ready
 - One to show that further coins are needed



ASM diagram & State and output table

- If possible- simplify early.
 - Both state and output tables and ASM charts can be used to find redundancy

State	10c	20c	No coin	Ready	Coin	Dispense	Return
S_RDY	S_10	S_20	Self	1	0	0	0
S_10	S_20	S_30	Self	0	1	0	0
S_20	S_30	S_DISP	Self	0	1	0	0
S_30	S_DISP	S_RET	Self	0	1	0	0
S_DISP	S_RDY	S_RDY	S_RDY	0	0	1	0
S_RET	S_RDY	S_RDY	S_RDY	0	0	0	1



Example: Single ROM, extended State and output table

- Moore machine implementation:

- One address for every unique combination of inputs and state

- Pro's

- **Can be implemented using fixed hardware**
 - ROM + a few state registers
- Reprogrammable

- Con's

- A lot of duplicated data in ROM
 - Output the same for all states
Here: 3x output data in legal states...
- *Illegal states need a plan..*
 - Here: input = "11", state = "110", "111"
=> 14 illegal states, 18 legal

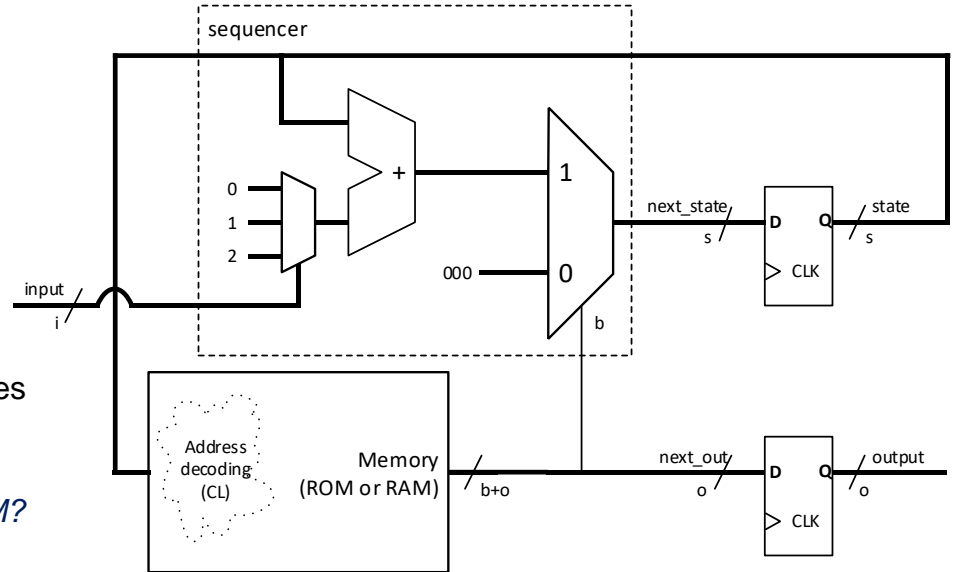
Memory		State	Next state			Output			
Address (s+i)	Data (n_s+o)	State	No coin	10c	20c	Ready	Coin	Disp-ense	Re- turn
000 00	000 1000	S_RDY	Self			1	0	0	0
000 01	001 1000			S_10					
000 10	010 1000				S_20				
001 00	001 0100	S_10	Self			0	1	0	0
001 01	010 0100			S_20					
001 10	011 0100				S_30				
010 00	010 0100	S_20	Self						
010 01	011 0100			S_30					
010 10	100 0100				S_DISP				
011 00	011 0100	S_30	Self			0	0	1	0
011 01	100 0100			S_DISP					
011 10	101 0100				S_RET				
100 00	000 0010	S_DISP	S_RDY			0	0	1	0
100 01	000 0010			S_RDY					
100 10	000 0010				S_RDY				
101 00	000 0001	S_RET	S_RDY			0	0	0	1
101 01	000 0001			S_RDY					
101 10	000 0001				S_RDY				

- Example resource usage:

- 3 state registers (+ 4 output registers if synchronized.)
- 5 bit address = 32 lines, 7 bit data => 224 bit ROM

Example: Adding a sequencer *can* reduce storage

- We reduce the address space from 2^{s+i} to 2^s
 - (ie. Memory has as many instructions as states)
- Here:
 - In branchable states:
Input decides if we jump 0, 1 or 2 states
 - Non branchable state => fixed next state
 $next_state \leftarrow S_RDY$
 - Adding 1 branch bit and sequencing logic reduces address space from 32 to 8 and data word size from 7 to 5.
 - Can we make Mealy with this reduced size ROM?*



Memory		State	Next state			Output			
Address (s)	Data (b+o)	State	No coin	10c	20c	Ready	Coin	Dispense	Return
000	1 1000	S_RDY	Self	S_10	S_20	1	0	0	0
001	1 0100	S_10	Self	S_20	S_30	0	1	0	0
010	1 0100	S_20	Self	S_30	S_DISP				
011	1 0100	S_30	Self	S_DISP	S_RET				
100	0 0010	S_DISP	S_RDY			0	0	1	0
101	0 0001	S_RET	S_RDY			0	0	0	1

VHDL

microcode

example (1/2):

- Entity as earlier
- Read ROM from file as earlier code

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use STD.textio.all;

entity vending is
port(
    clk, reset, twenty, ten : in std_logic;
    ready, coin, dispense, ret : out std_logic);
end entity vending;

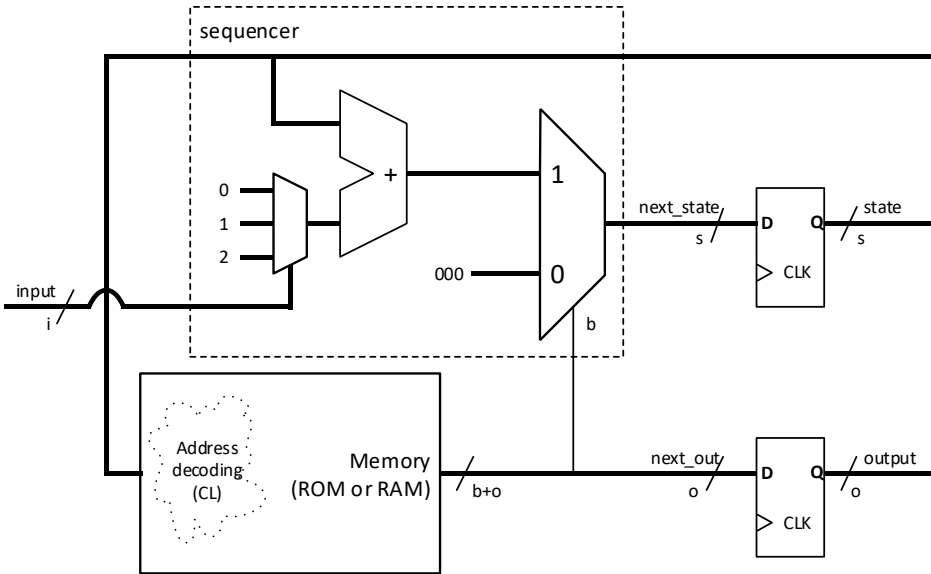
architecture microcode of vending is
constant data_width: natural := 5;
constant addr_width: natural := 3;
constant filename: string := "ROM_data_bits.txt";
type memory_array is array(2**addr_width-1 downto 0) of
    std_logic_vector(data_width-1 downto 0);

impure function initialize_ROM(file_name: string) return memory_array is
    file init_file: text open read_mode is file_name;
    variable current_line: line;
    variable result: memory_array;
begin
    for i in result'range loop
        readline(init_file, current_line);
        read(current_line, result(i));
    end loop;
    return result;
end function;

--initialize rom:
constant ROM_DATA: memory_array := initialize_ROM(filename);
signal address: std_logic_vector(addr_width-1 downto 0);
signal data: std_logic_vector(data_width-1 downto 0);

```

VHDL microcode example 2/2



```
--state assignment using std_logic (no "state_type"):
signal state, next_state : std_logic_vector(2 downto 0);
alias b : std_logic is data(4);
```

```
begin
  -- ROM data CL
  data <= ROM_DATA(to_integer(unsigned(address)));
  address <= state;
```

```
-- 1: register assignment:
process (clk, reset) is
begin
  if reset then
    ready <= '0';
    coin <= '0';
    dispense <= '0';
    ret <= '0';
    state <= (others => '0');
  elsif rising_edge(clk) then
    ready <= data(3);
    coin <= data(2);
    dispense <= data(1);
    ret <= data(0);
    state <= next_state;
  end if;
end process;
```

```
-- 2: combinational next_state logic (sequencer)
next_state <=
  (others => '0') when not b else
  std_logic_vector( unsigned(state) + 1) when ten else
  std_logic_vector( unsigned(state) + 2) when twenty else
  state;
end architecture microcode;
```

ROM (text file content)

00000
00000
00001
00010
10100
10100
10100
11000

- Address 7 is first line since we read in the 'range order ($2^{**}n-1$ **downto** 0).
 - To have address 0 first we should read in 'reverse_range

```
for i in result'range loop  
  readline(init_file, current_line);  
  read(current_line, result(i));  
end loop;
```

- Why do we have two lines with 0?
- What will happen if state is set to address 7 og 6..?

Memory	
Address (s)	Data (b+o)
000	1 1000
001	1 0100
010	1 0100
011	1 0100
100	0 0010
101	0 0001

Reducing delay

- Can we reduce output delay?

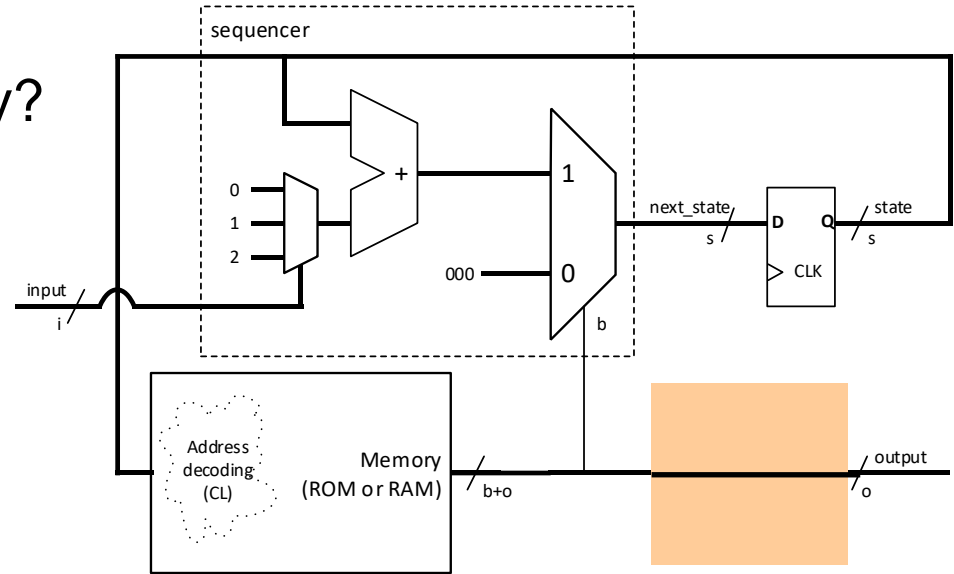
```
begin
  -- ROM data CL
  data <= ROM_DATA(to_integer(unsigned(address)));
  address <= state;

  -- output assignment based on state...
  ready    <= data(3);
  coin     <= data(2);
  dispense <= data(1);
  ret      <= data(0);

  -- 1: sequential state assignment:
  state <= (others => '0') when reset else next_state when rising_edge(clk);

  -- 2: combinatorial next_state logic
  next_state <=
    (others => '0') when not b else
    std_logic_vector( unsigned(state) + 1) when ten else
    std_logic_vector( unsigned(state) + 2) when twenty else
    state;

end architecture microcode;
```



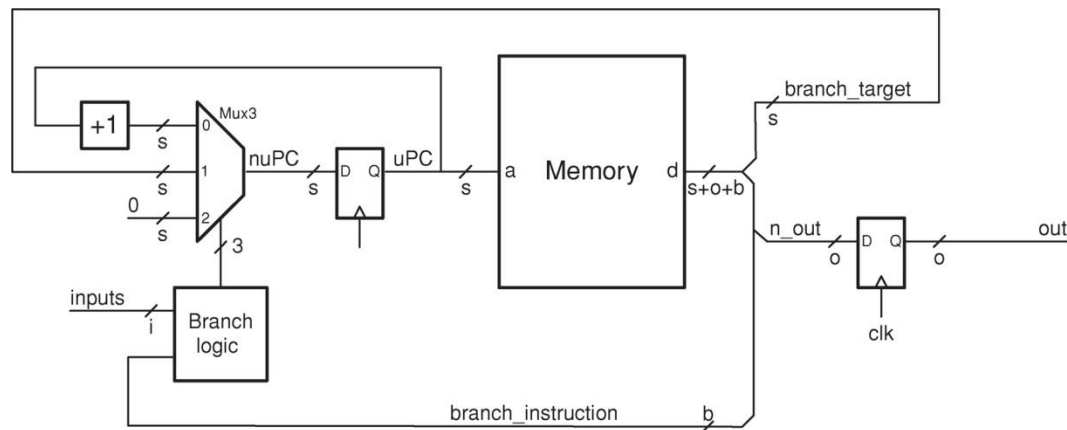
- What type of FSM is this?
- Will it work?

General Sequencer / Microsequencer

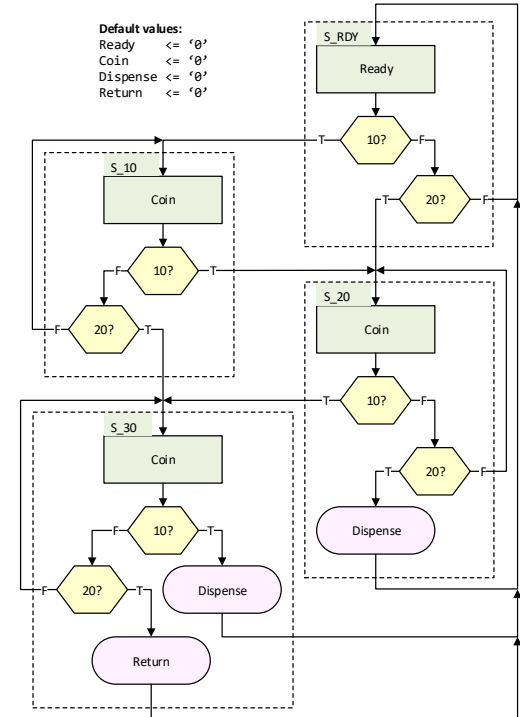
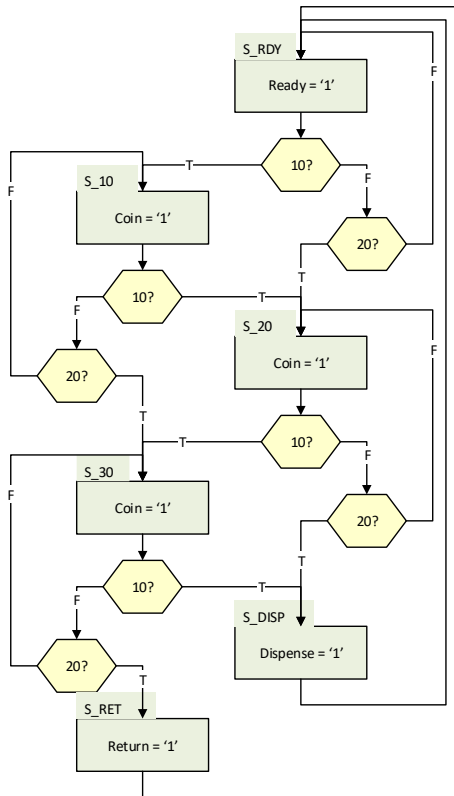
- A device that generates addresses
 - Typically a counter
 - + some logic for various types of jumping
 - Reduces the need to store subsequent addresses
 - *a sequencer does only make sense when there is some sort of order*
 - *It does not make sense if next state always can be any state (ie totally random)*

Microcoded processors

- A microcoded FSM with a sequencer can be seen as a microprocessor.
 - ROM stores instructions that are executed on each clock cycle.
 - uPC (Microprocessor Counter) is the current state.
- Branching is usually done with several bits, to enable different type of usage
- Input is the machine code we want to execute
- Processors have other functions and dedicated memory
 - ALU
 - Instruction memory
 - Data memory

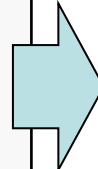


Going from Moore to Mealy (without sequencer)



State table conversion *(legal memory entries shown)*

Memory		State	Next state			Output			
Address (s+i)	Data (n_s+o)	State	No coin	10c	20c	Ready	Coin	Dispense	Return
000 00	000 1000	S_RDY	Self			1	0	0	0
000 01	001 1000			S_10					
000 10	010 1000				S_20				
001 00	001 0100	S_10	Self			0	1	0	0
001 01	010 0100			S_20					
001 10	011 0100				S_30				
010 00	010 0100	S_20	Self						
010 01	011 0100			S_30					
010 10	100 0100				S_DISP				
011 00	011 0100	S_30				0	0	0	0
011 00	100 0100			S_DISP					
011 00	101 0100				S_RET				
100 00	000 0010	S_DISP	S_RDY			0	0	1	0
100 01	000 0010			S_RDY					
100 10	000 0010				S_RDY				
101 00	000 0001	S_RET	S_RDY			0	0	0	1
101 01	000 0001			S_RDY					
101 10	000 0001				S_RDY				



Memory		State	Next state			Output			
Address (s+i)	Data (n_s+o)	State	No coin	10c	20c	Ready	Coin	Dispense	Return
00 00	00 1000	S_RDY	Self			1	0	0	0
00 01	01 1000			S_10					
00 10	10 1000				S_20				
01 00	01 0100	S_10	Self			0	1	0	0
01 01	10 0100			S_20					
01 10	11 0100				S_30				
10 00	10 0100	S_20	Self			0	1	0	0
10 01	11 0100			S_30					
10 10	00 0010				S_RDY				
11 00	11 0100	S_30	Self			0	1	0	0
11 01	00 0010			S_RDY					
11 10	00 0001				S_RDY				

- Going from 224 bit ROM to
 - 4 address bits and 6 output bits => 96 bit ROM (2^4) * 6)
- State should be msb in address to make comprehensible decoding
- **what about unused ROM entries (illegal combinations)?**: coming next slide

ROM data

- ROM data must come in correct sequence
 - here:
 - 3 legal input combinations per stored state
 - We must use multiple of 2^n ($=4$), otherwise we write in the wrong address
- Comments at line end = OK
 - Because we use **readline**

```
001000 S_RDY
011000 S_RDY -> S_10
101000 S_RDY -> S_20
000000 illegal state, no output, next state S_RDY
010100 S_10 -> S_10
100100 S_10 -> S_20
110100 S_10 -> S_30
000000 illegal state, no output, next state S_RDY
100100 S_20 -> S_20
110100 S_20 -> S_30
000010 S_20 -> S_RDY & dispense
000000 illegal state, no output, next state S_RDY
110100 S_30 -> S_30
000010 S_30 -> S_RDY & dispense
000001 S_30 -> S_RDY & retur
000000 illegal state, no output, next state S_RDY
```

VHDL microcoded mealy machine

- ROM size changed
 - 4 bit address gives 16 entries

- Reading in **reverse**'range

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use STD.textio.all;

entity vending is
port(
    clk, reset, twenty, ten : in std_logic;
    ready, coin, dispense, ret : out std_logic);
end entity vending;

architecture microcode_mealy of vending is
constant data_width: natural := 6;
constant addr_width: natural := 4;
constant filename: string := "ROM_mealy_data_bits.txt";
type memory_array is array(2**addr_width-1 downto 0) of
std_logic_vector(data_width-1 downto 0);

impure function initialize_ROM(file_name: string) return memory_array is
file init_file: text open read_mode is file_name;
variable current_line: line;
variable result: memory_array;
begin
for i in result'reverse_range loop
readline(init_file, current_line);
read(current_line, result(i));
end loop;
return result;
end function;
```

VHDL microcoded mealy machine

- State only 2 bits
- Address:
 - State is MSB in address (necessary)
 - Input gives rest of address

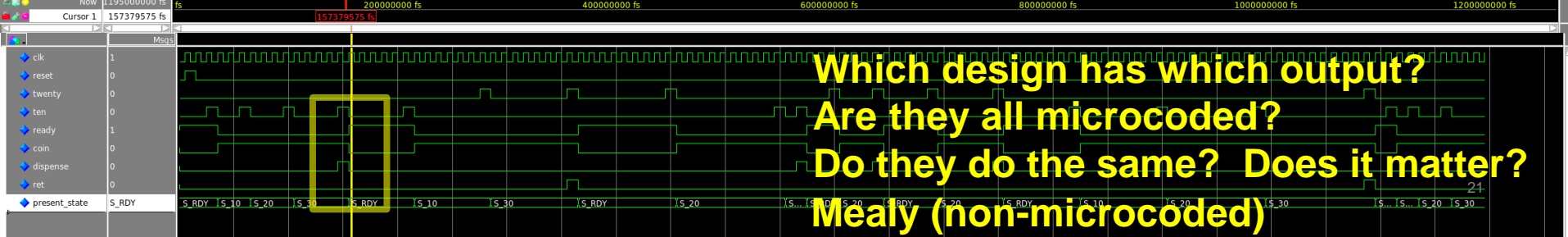
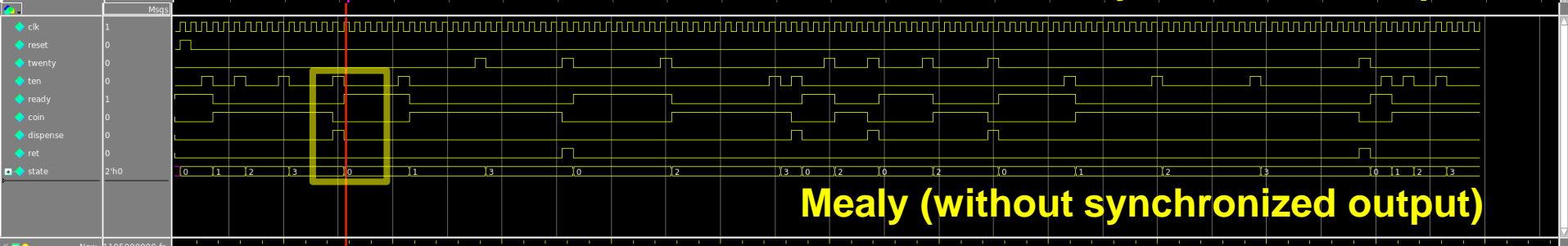
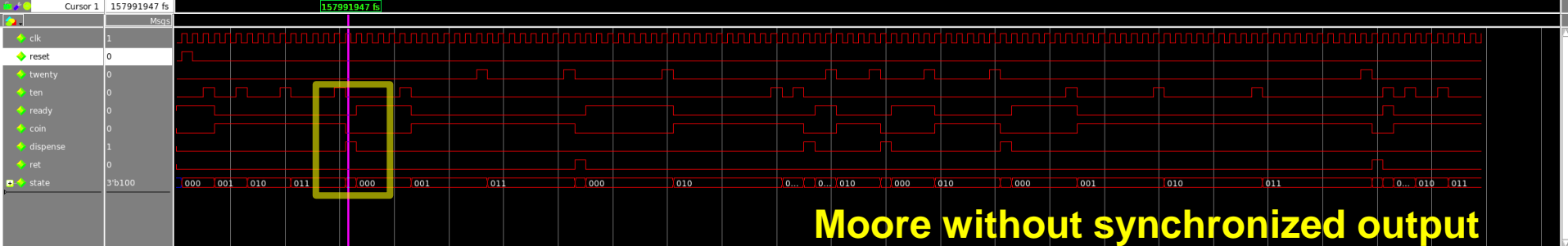
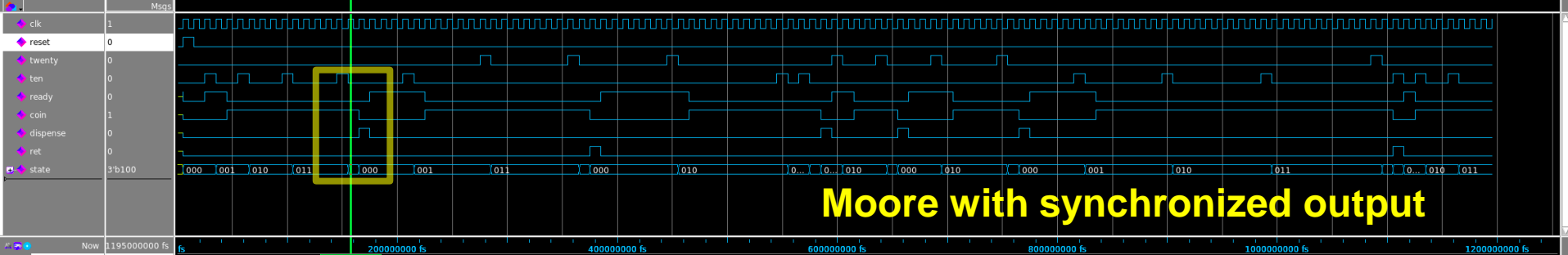
```
--initialize rom:
constant ROM_DATA: memory_array := initialize_ROM(filename);
signal address: std_logic_vector(addr_width-1 downto 0);
signal data: std_logic_vector(data_width-1 downto 0);

-- state register declaration
signal state : std_logic_vector(1 downto 0);
begin
-- ROM data CL
data <= ROM_DATA(to_integer(unsigned(address)));
address <= state & twenty & ten ; -- state is MSB

-- output assignment based on state...
ready <= data(3);
coin <= data(2);
dispense <= data(1);
ret <= data(0);

-- sequential state assignment:
state <= (others => '0') when reset else data(5 downto 4) when rising_edge(clk);

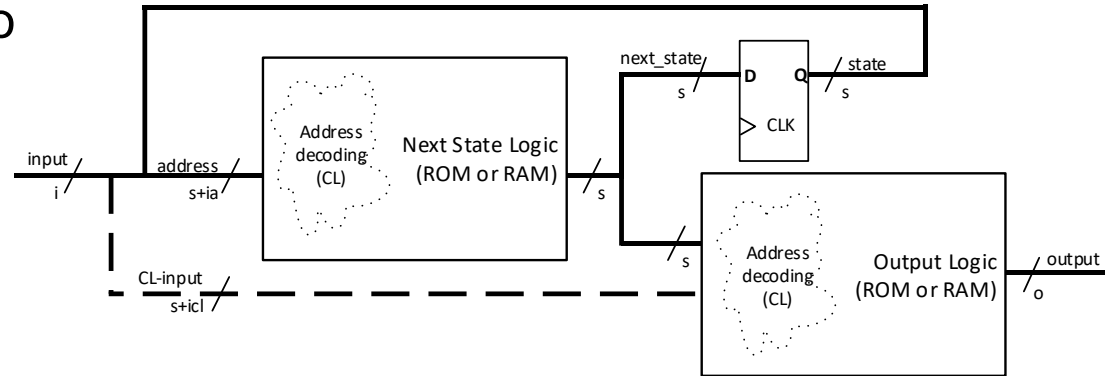
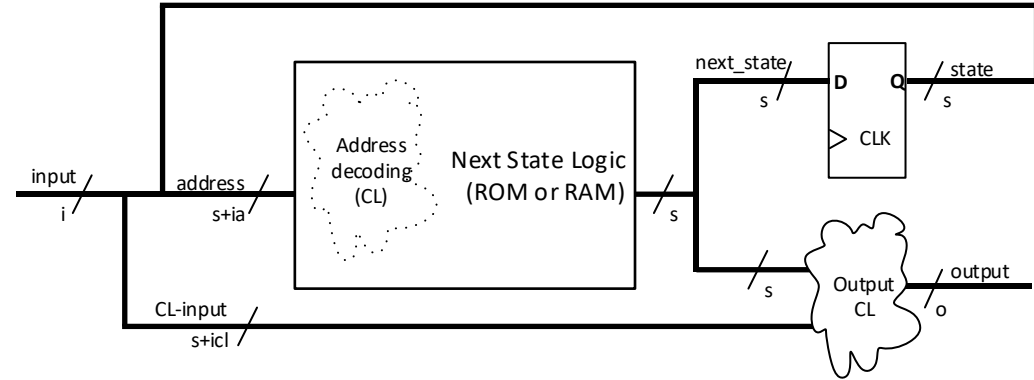
end architecture microcode_mealy;
```



Which design has which output?
 Are they all microcoded?
 Do they do the same? Does it matter?

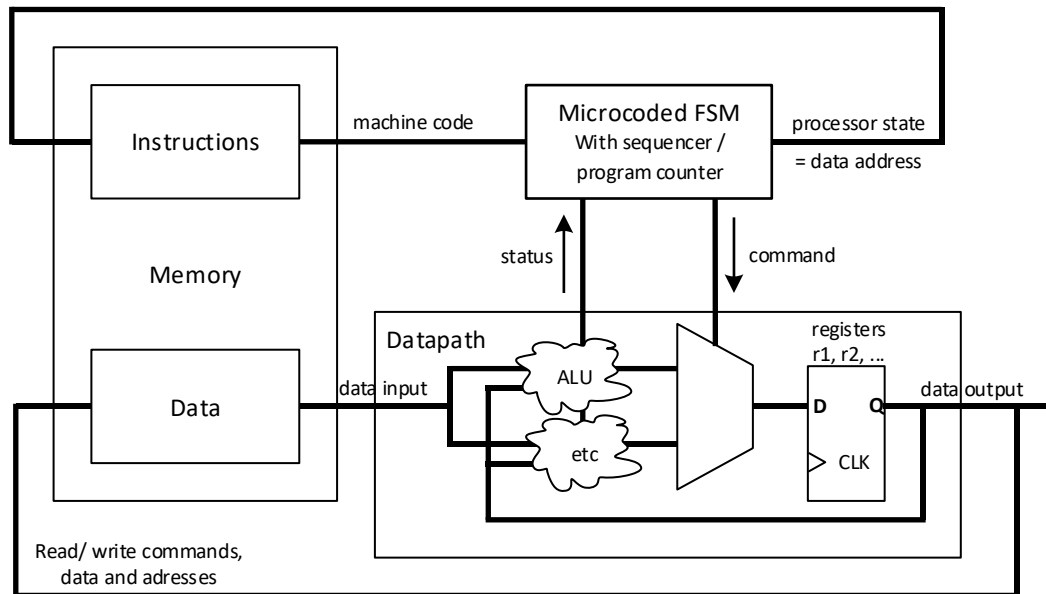
Microcode considerations..

- ROM size can be reduced by
 - Separating output CL
 - CL can be a separate ROM
 - Separate state CL and output CL
 - What does the synthesizer do with our FSMs?
 - Breaks it up into LUTs and flipflops
 - LUT = small ROM..



Microprocessors

- Microcoded state machines can and has been used to create processors.
 - Early x86 processors were entirely microcoded (8088, 8086, 80286, 80386).
 - Microcoded processors *can be patchable*.
 - => BIOS upgradeable, etc.



- ROM content dictates instruction set (machine code)
- Modern processors are normally not (fully) microcoded
 - Optimization and move towards RISC dictates hardwired circuitry for speed and power
 - Method can still be used –
 - for complex instructions, variable length instructions
 - To ensure updates can be implemented after shipping..
 - *One could argue this is what we actually when using LUT based FPGAs*

Summary of microcoded state machines

- All FSMs *can* be implemented using 1 ROM + state registers
 - The general solution suggests a mealy machine,
 - We get Moore having *the same output regardless of input* in each state
- Using 2 ROMs (separate state and output decoding)
 - May reduce memory usage
 - when none or only some input are used to determine output
- Sequencers, Branching- and Output logic
 - may reduce ROM size
 - adds structure outside the state ROM.
 - This is one way of implementing processors and instruction sets.
- Consider using microcode when...
 - *the state machine is (best) defined by a (large) table*
 - when changes to the state table likely will happen at some point in the future.

Suggested reading

- D&H 18
 - 18 p398-427

Next lesson

- Clock Domain Crossing “CDC”