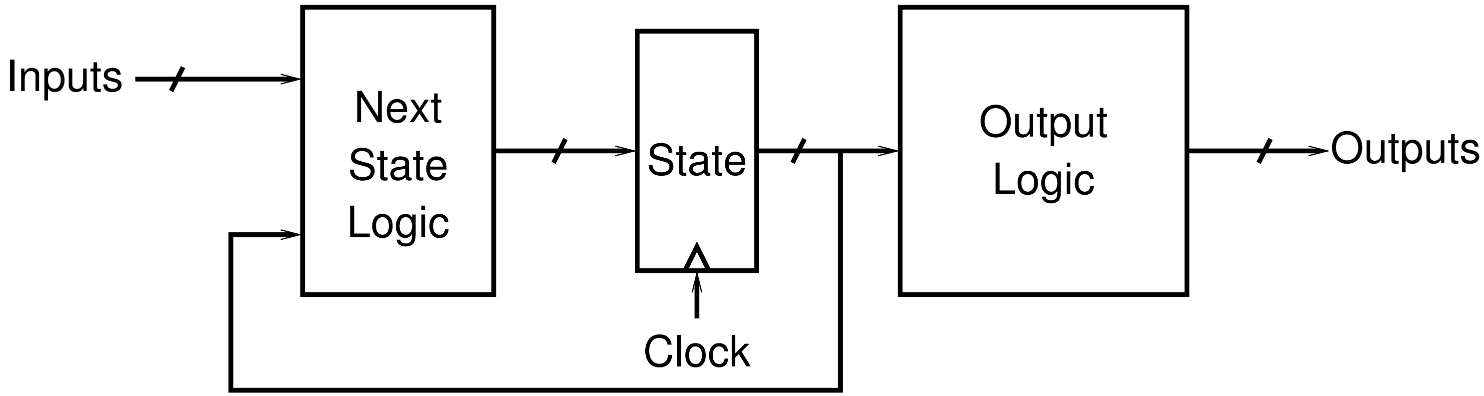
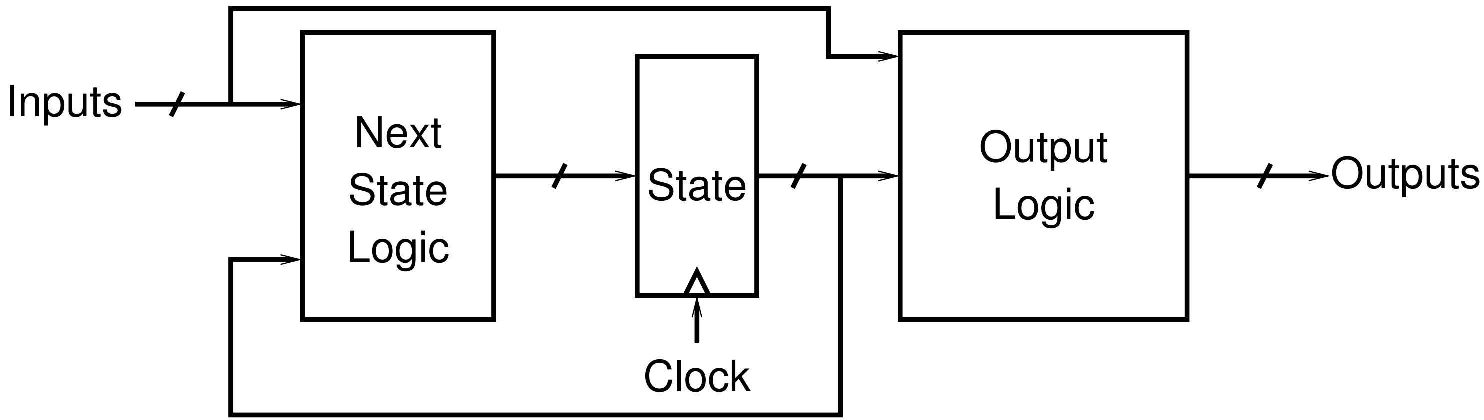
ASM

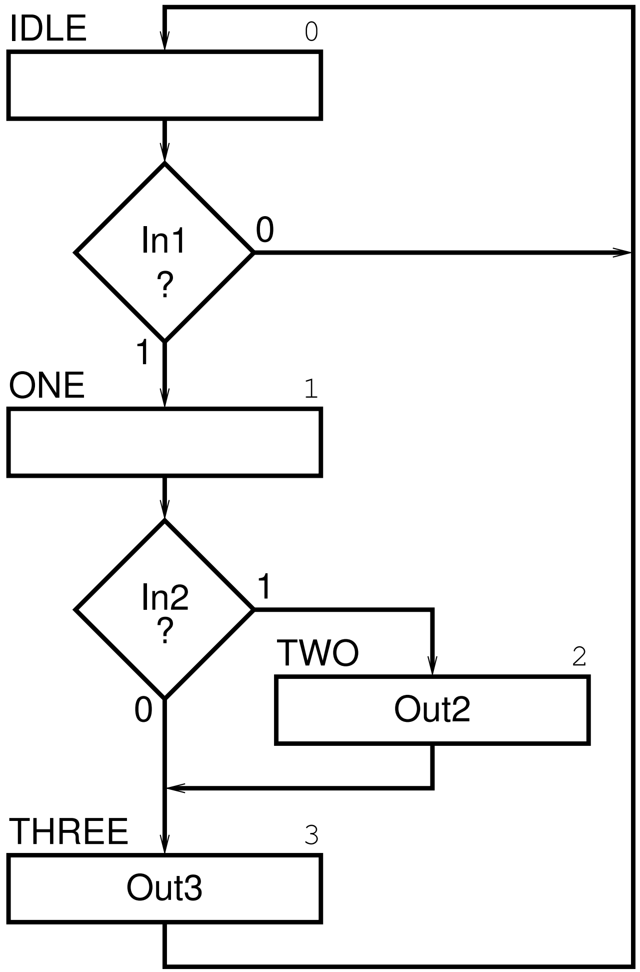
Moore State Machine



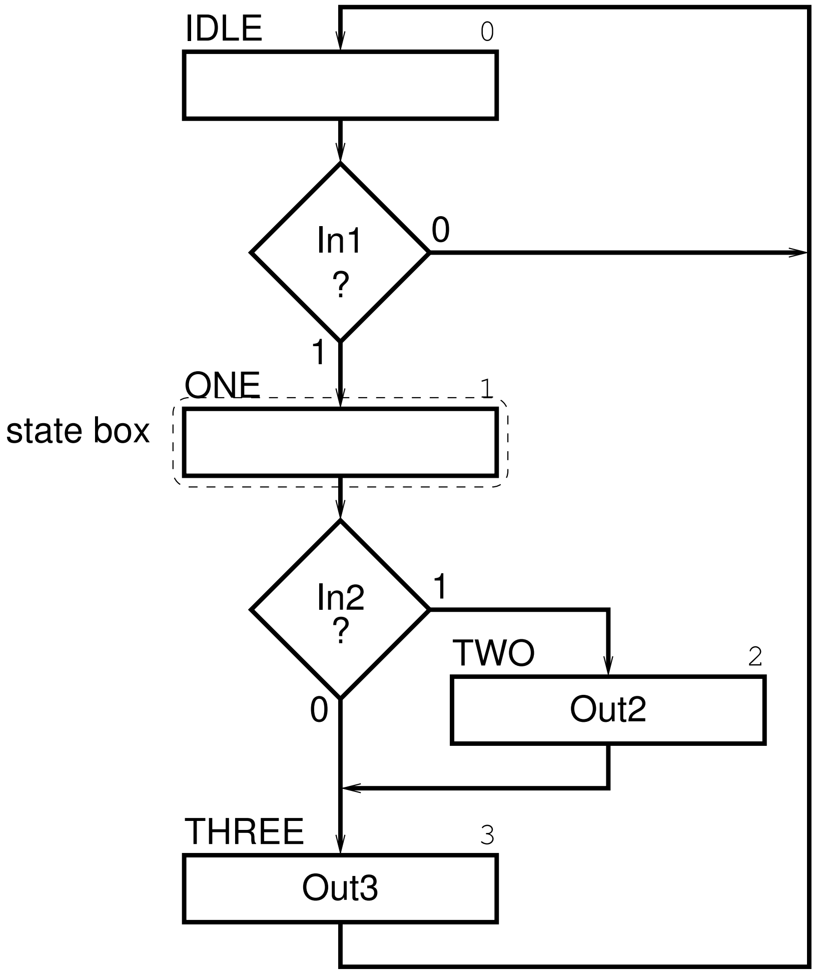
Mealy State Machine



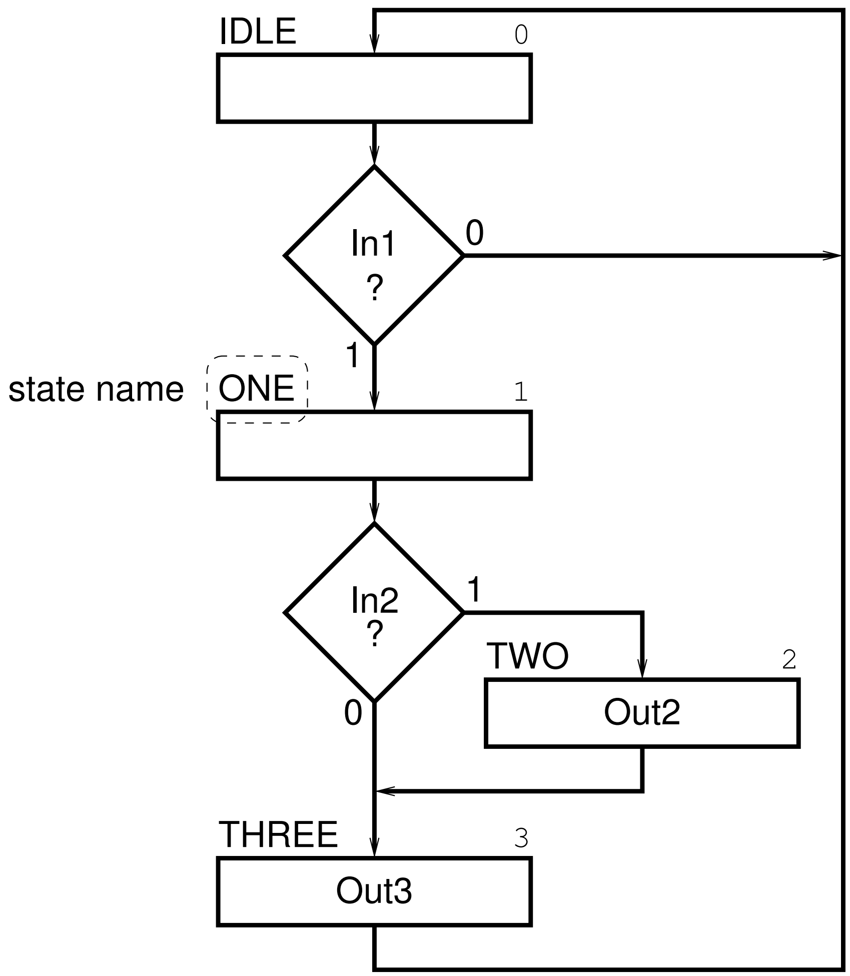
ASM1 – Moore Machine



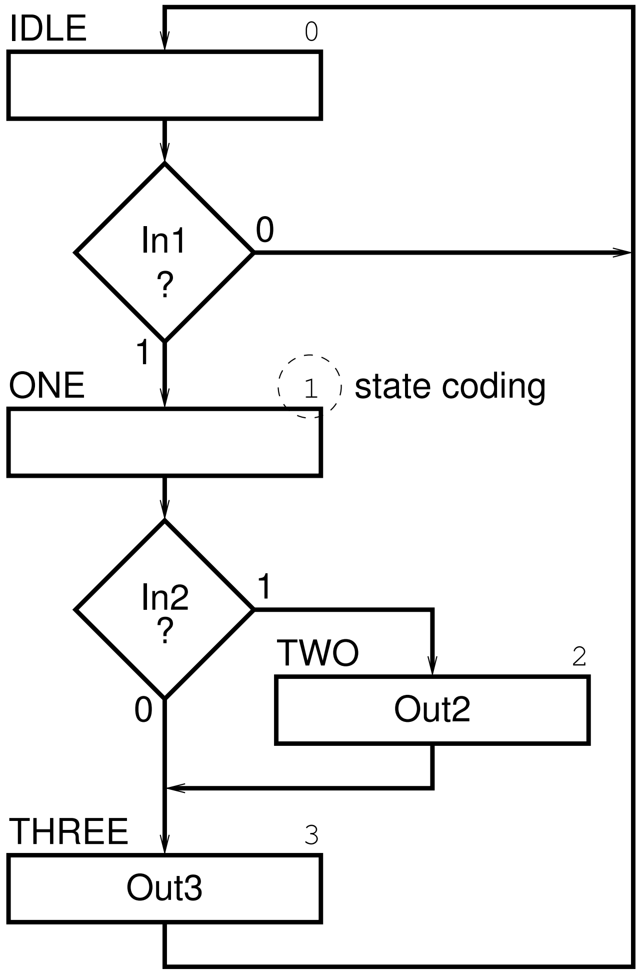
ASM1 – Moore Machine



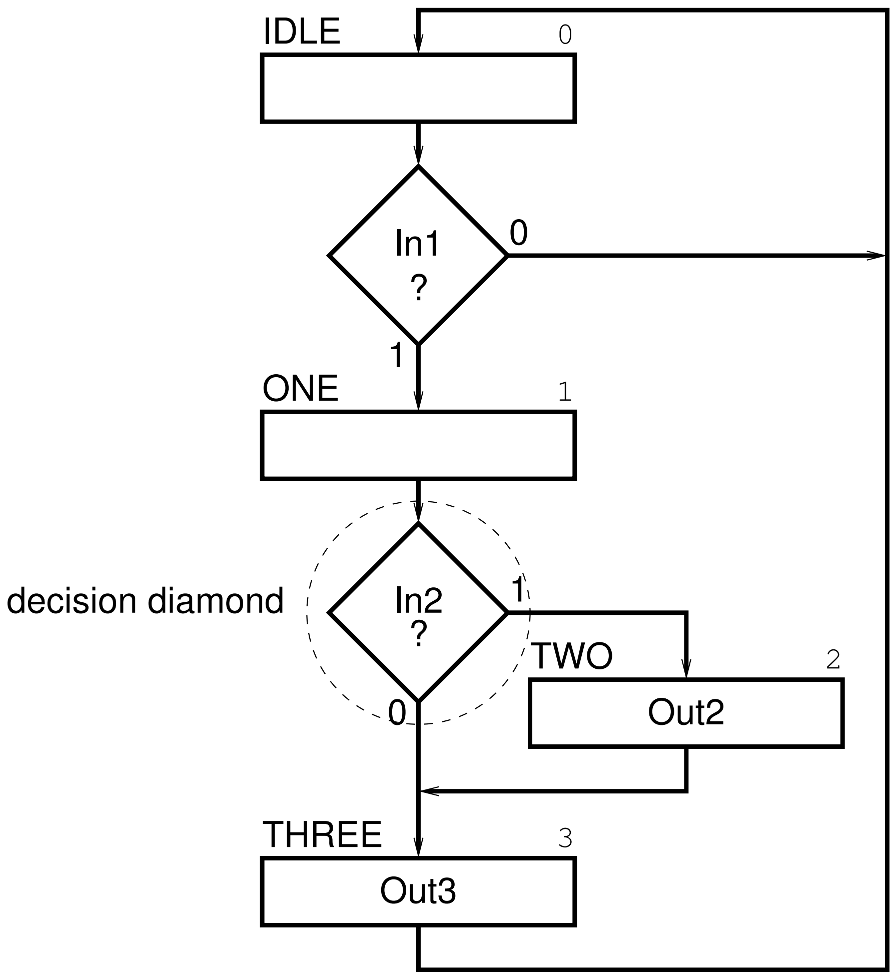
ASM1 – Moore Machine



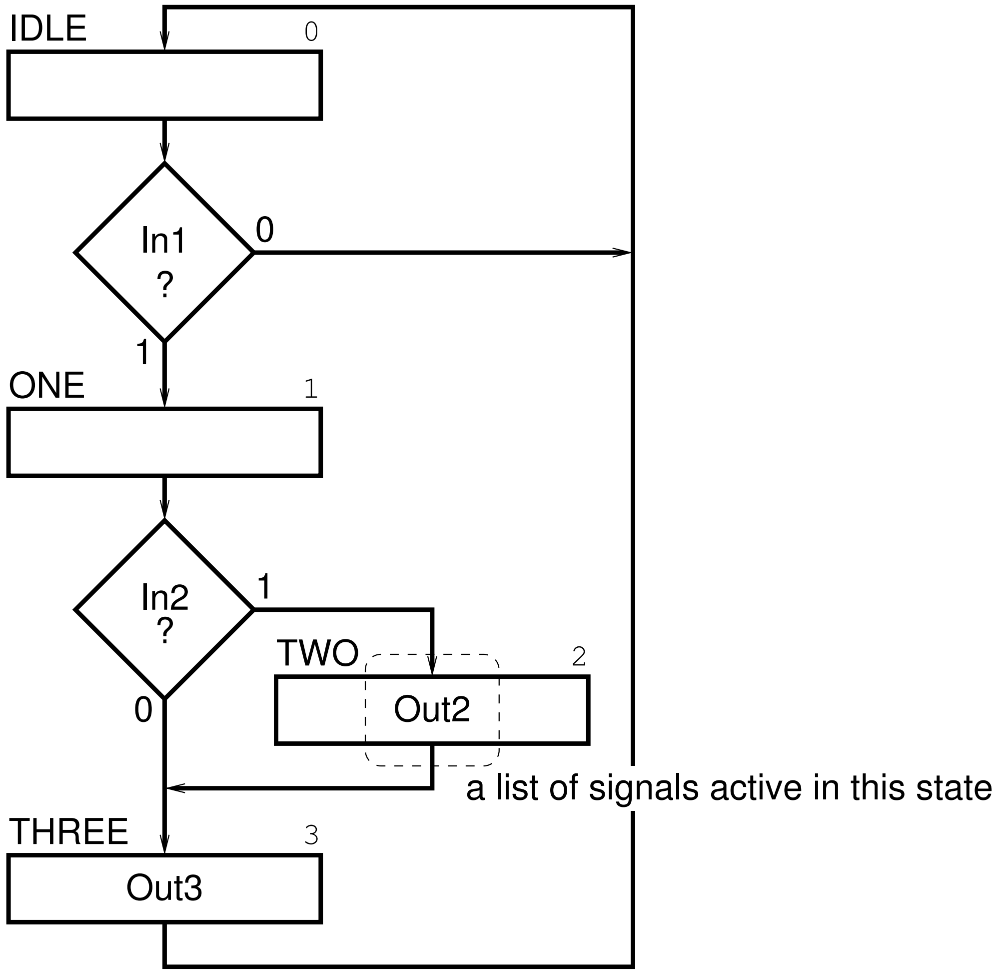
ASM1 – Moore Machine



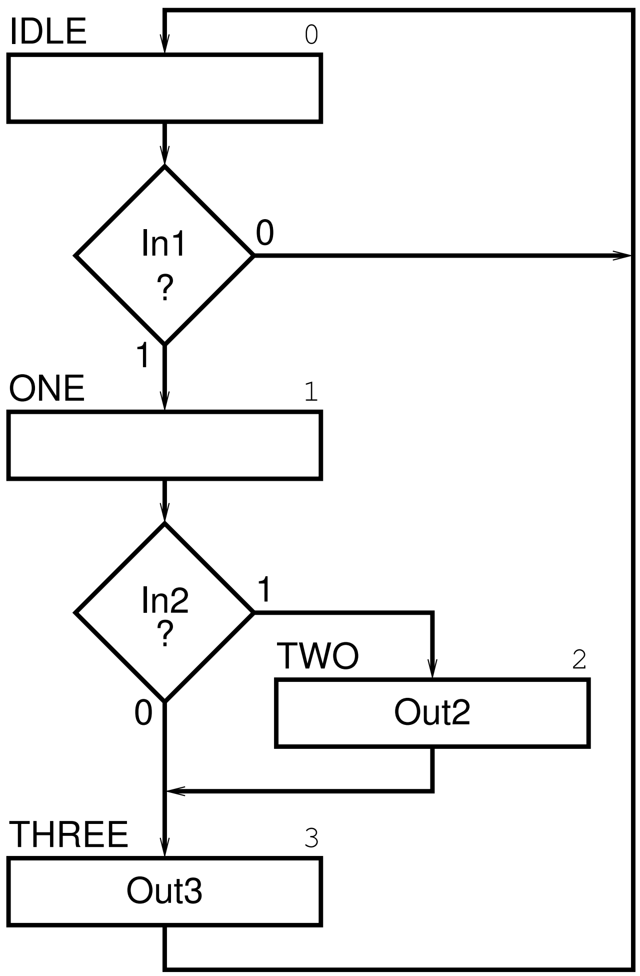
ASM1 – Moore Machine



ASM1 – Moore Machine



ASM1 – Moore Machine



ASM1[a]

```
// ----------- state declaration ----------
logic [1:0] state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)           state <= 0;
    else
      case (state)
        0:      if (In1)  state <= 1;
        1:      if (In2)  state <= 2;
                else      state <= 3;
        2:                state <= 3;
        3:                state <= 0;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  assign Out2 = (state == 2);
  assign Out3 = (state == 3);
```

ASM1^b – using state names and **enum**erated types[1]

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE=0, ONE=1, TWO=2, THREE=3 } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  assign Out2 = (state == TWO);
  assign Out3 = (state == THREE);
```

[1]a synthesis program may re-allocate the states
(the default state assignments shown here are optional)

ASM1^b – using state names and **enum**erated types[1]

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE  , ONE  , TWO  , THREE   } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  assign Out2 = (state == TWO);
  assign Out3 = (state == THREE);
```

[1]a synthesis program may re-allocate the states
(the default state assignments shown here are optional)

ASM1^b – using state names and **enum**erated types[2]

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE=0, ONE=1, TWO=2, THREE=3 } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  assign Out2 = (state == TWO);
  assign Out3 = (state == THREE);
```

[2] a synthesis program may re-allocate the states
(the default state assignments shown here are optional)

ASM1^c – using a procedural block to infer combinational logic

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE, ONE, TWO, THREE } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  always_comb
    begin
      Out2 = (state == TWO);
      Out3 = (state == THREE);
    end
```

ASM1^{d} – using **typedef**

```
// ----------- state declaration ----------
typedef enum logic [1:0] { IDLE, ONE, TWO, THREE } state_t;
state_t state;
// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  always_comb
    begin
      Out2 = (state == TWO);
      Out3 = (state == THREE);
    end
```

ASM1^e – using **if else** instead of **case**

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE, ONE, TWO, THREE } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      if ((state == IDLE) && In1)
        state <= ONE;
      else if ((state == ONE) && In2)
        state <= TWO;
      else if ((state == ONE) || (state == TWO))
        state <= THREE;
      else
        state <= IDLE;
// ------------- output logic -------------
logic Out2, Out3;
  always_comb
    begin
      Out2 = (state == TWO);
      Out3 = (state == THREE);
    end
```

ASM1^{f} – multiple assignment (default and override)

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE, ONE, TWO, THREE } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  always_comb
    begin
      Out2 = 0; Out3 = 0;              // <- default values assigned here

      if ( state == TWO )   Out2 = 1;  // <- only non-default values
      if ( state == THREE ) Out3 = 1;  //    assigned here
    end
```

ASM1^g – multiple assignment (default and override) with **case**

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE, ONE, TWO, THREE } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  always_comb
    begin
      Out2 = 0; Out3 = 0;              // <- default values assigned here
      case (state)  TWO   : Out2 = 1;  // <- only non-default values
                    THREE : Out3 = 1;  //    assigned here
      endcase
    end
```

ASM1^h – using a single procedural block *(This style is not advised for novices!)*[3]

```
// Caution: infers registered outputs for Out2 and Out3

enum logic [1:0] { IDLE, ONE, TWO, THREE } state;
logic Out2, Out3;

  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)            begin state <= IDLE;
                                  Out2  <= 0; Out3 <= 0; end
    else
      begin
        Out2 <= (state == ONE) && In2;
        Out3 <= (state == ONE) && !In2 || (state == TWO);
        case (state)
          IDLE :  if (In1)        state <= ONE;
          ONE  :  if (In2)        state <= TWO;
                  else            state <= THREE;
          TWO  :                  state <= THREE;
          THREE:                  state <= IDLE;
        endcase
      end
```

[3]here we predict in one state, the outputs that we need in the next state
– it is hard to see which output is associated with which state $\therefore$ error prone

ASM1^{i} – using a single procedural block *(This style is not advised for novices!)*[4]

```
// Caution: infers registered outputs for Out2 and Out3

enum logic [1:0] { IDLE, ONE, TWO, THREE } state;
logic Out2, Out3;

  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)            begin state <= IDLE;
                                  Out2  <= 0; Out3 <= 0; end
    else
      case (state)
        IDLE :  if (In1)    begin state <= ONE;
                                  Out2  <= 0; Out3 <= 0; end
        ONE  :  if (In2)    begin state <= TWO;
                                  Out2  <= 1; Out3 <= 0; end
                else        begin state <= THREE;
                                  Out2  <= 0; Out3 <= 1; end
        TWO  :              begin state <= THREE;
                                  Out2  <= 0; Out3 <= 1; end
        THREE:              begin state <= IDLE;
                                  Out2  <= 0; Out3 <= 0; end
      endcase
```

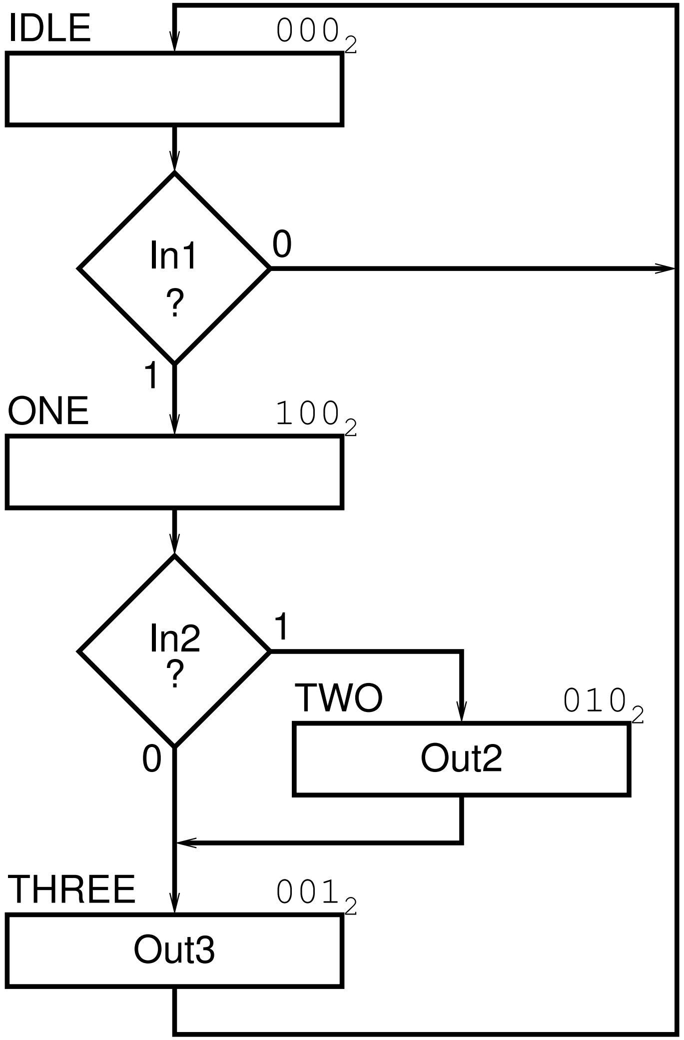
[4] here the outputs are clearly associated with the appropriate states
– includes duplicated sections of code $\therefore$ error prone

ASM1[j] – using a single procedural block *(This style is not advised for novices!)*[5]

```
// Caution: infers registered outputs for Out2 and Out3

enum logic [1:0] { IDLE, ONE, TWO, THREE } state;
logic Out2, Out3;

  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          begin state <= IDLE;
                                Out2  <= 0; Out3 <= 0; end
    else
       if ((state == IDLE) && In1)
                         begin state <= ONE;
                               Out2  <= 0; Out3 <= 0; end
      else if ((state == ONE) && In2)
                         begin state <= TWO;
                               Out2  <= 1; Out3 <= 0; end
      else if ((state == ONE) || (state == TWO))
                         begin state <= THREE;
                               Out2  <= 0; Out3 <= 1; end
      else
                         begin state <= IDLE;
                               Out2  <= 0; Out3 <= 0; end
```

[5]here **if else** structure is used to avoid duplication of state THREE code
– the resulting code is not easy to follow $\therefore$ error prone

ASM1* – Moore Machine (with optimized state coding)



ASM1^k – using a single procedural block (with optimized state coding)[6]

```
typedef enum logic [2:0]
  { IDLE=3'b000, ONE=3'b100, TWO=3'b010, THREE=3'b001 } state_t;
logic state, Out2, Out3;

  always_ff @(posedge Clock, negedge nReset)
    if (!nReset) {state,Out2,Out3} <= IDLE;
    else
      case ({state,Out2,Out3})
        IDLE :  if (In1)  {state,Out2,Out3} <= ONE;
                else      {state,Out2,Out3} <= IDLE;
        ONE  :  if (In2)  {state,Out2,Out3} <= TWO;
                else      {state,Out2,Out3} <= THREE;
        TWO  :            {state,Out2,Out3} <= THREE;
        THREE:            {state,Out2,Out3} <= IDLE;
      endcase
```

[6]here the outputs are considered to be part of the state

– **typedef** is used to define `state_t` but `state_t` is never used

 – only the state names are used

 – `state` is a single bit logic variable

– the resulting code is concise but is it easy to follow?

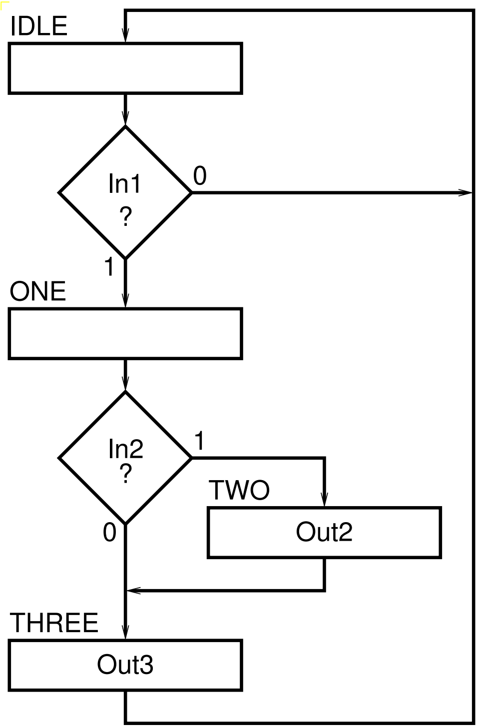
ASM - Single Procedural Block Approach?

Some ASMs lend themselves to implementation as a single procedural block ...

State + Next State Logic	Output Logic	
`always_ff`	**`assign`**	✓
`always_ff`	**`always_comb`**	✓
`always_ff`		✓

ASM - Single Procedural Block Approach?

Some ASMs lend themselves to implementation as a single procedural block ...

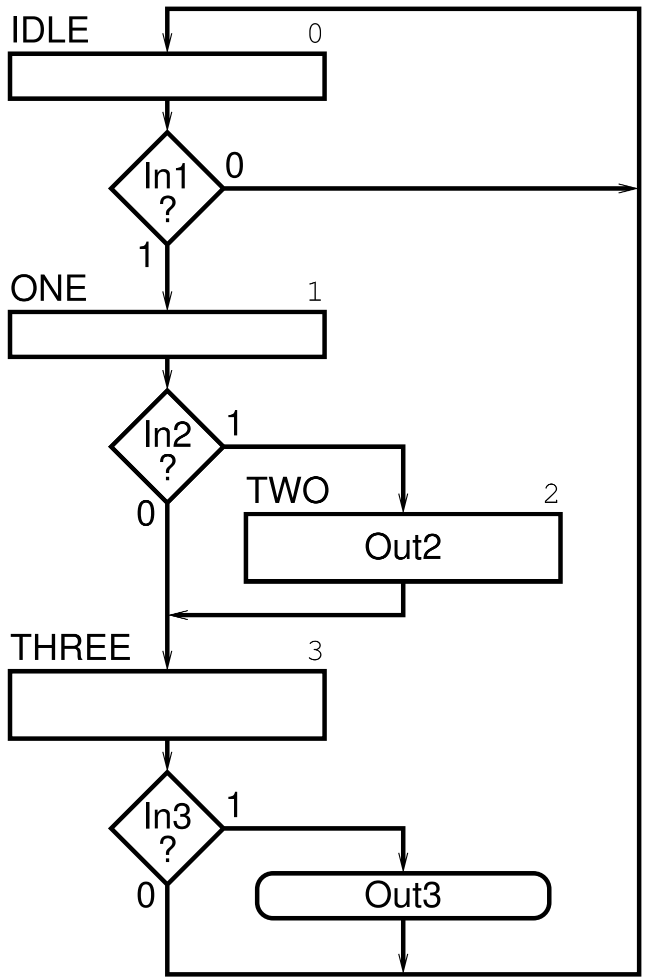


State + Next State Logic	Output Logic	
`always_ff`	`assign`	✓
`always_ff`	`always_comb`	✓
`always_ff`		✗

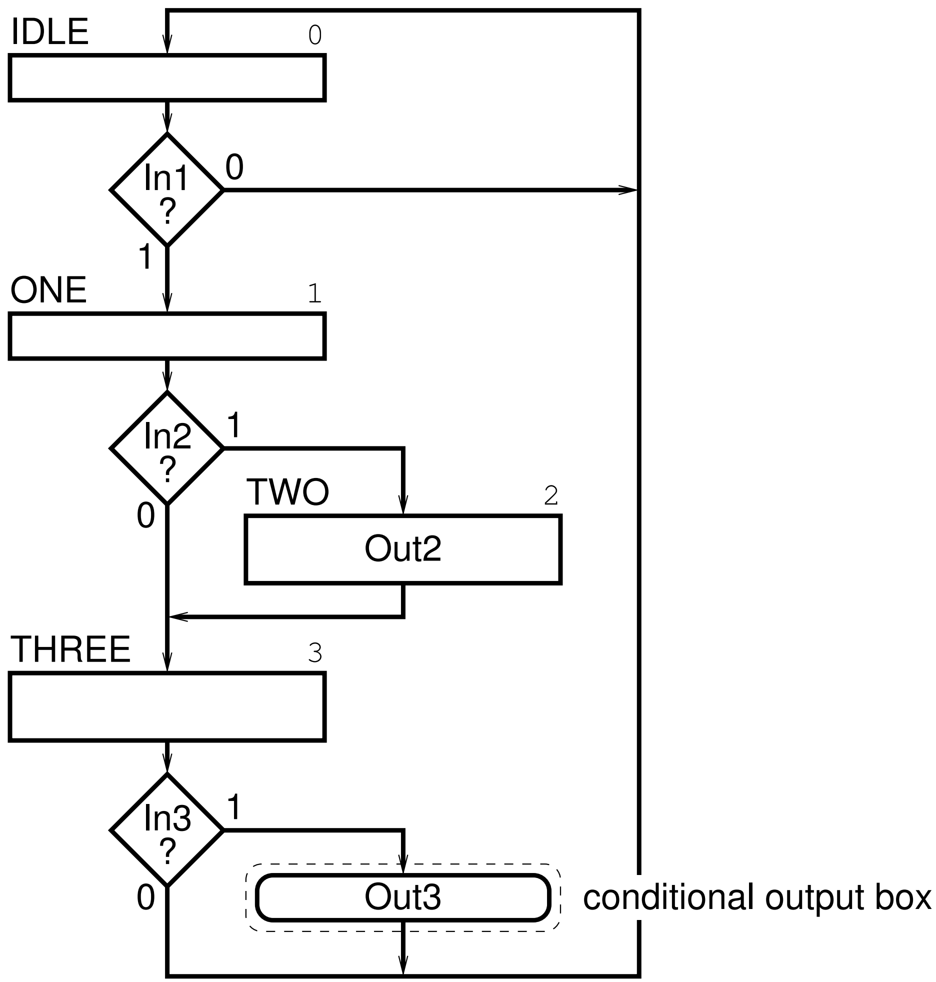
This is **NOT** one of them.

2015

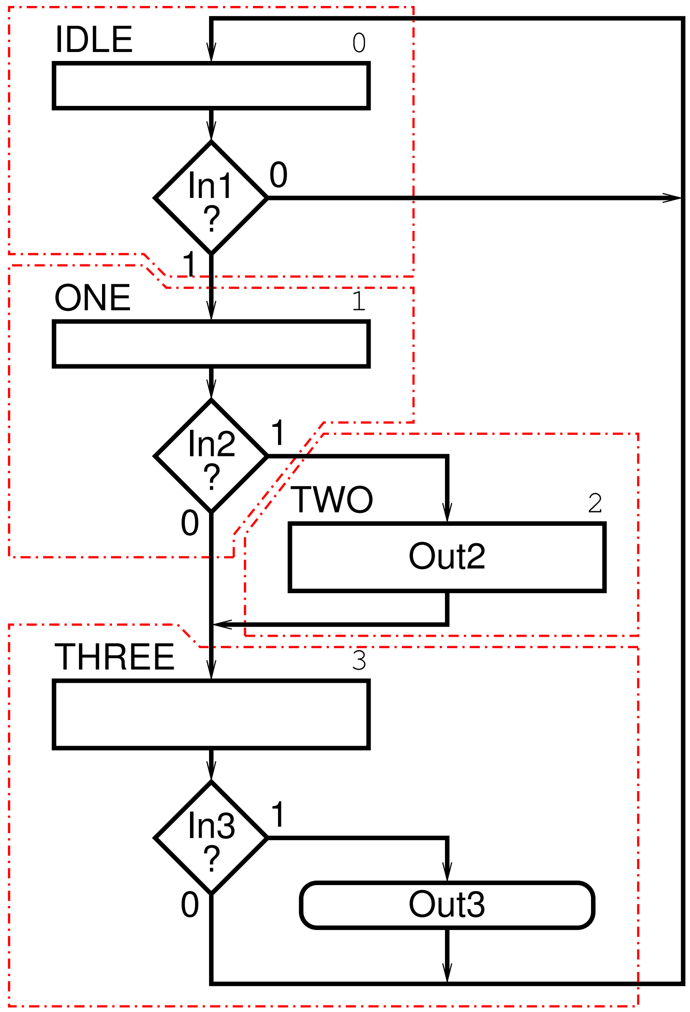
ASM2 – Mealy Machine



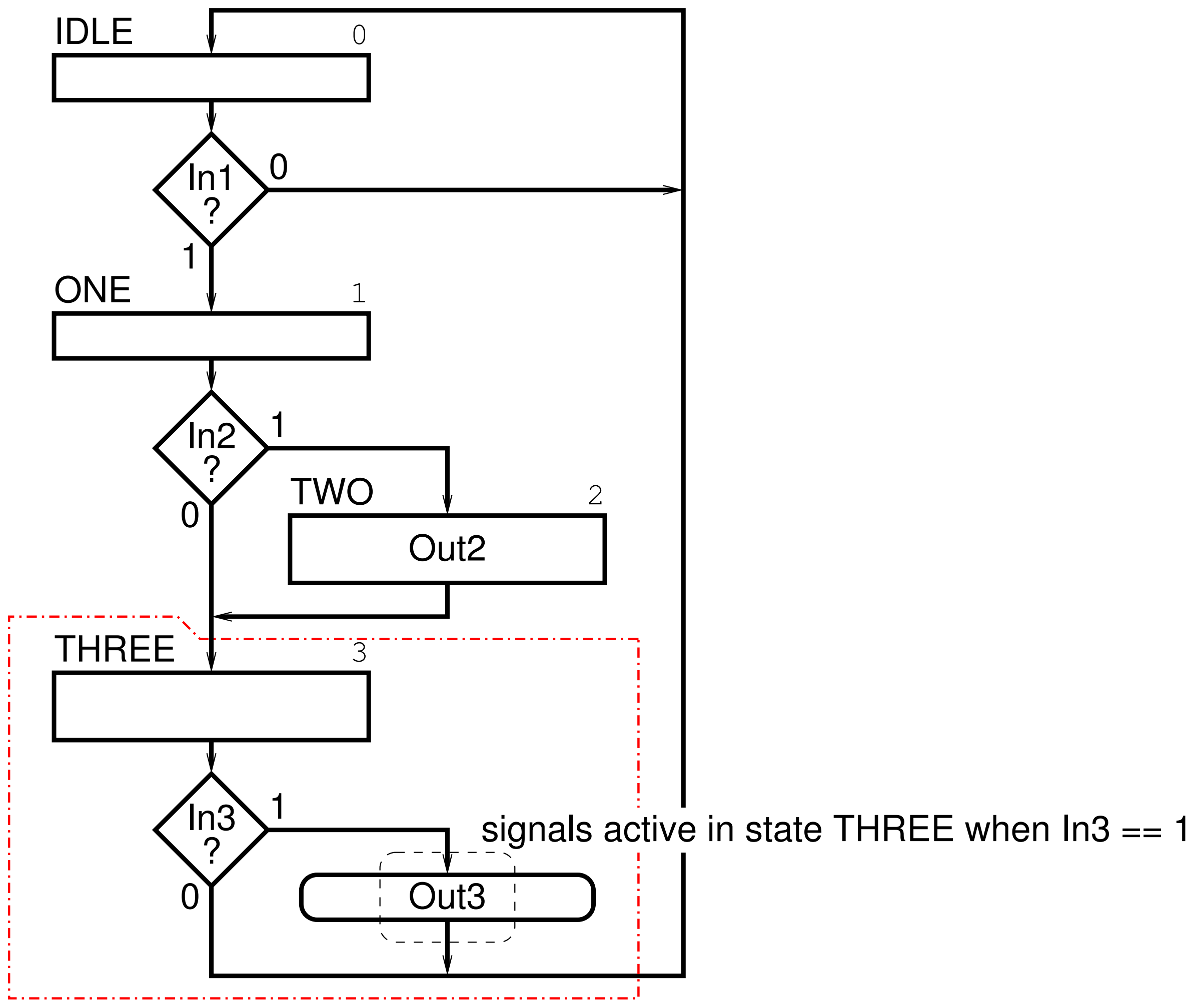
ASM2 – Mealy Machine



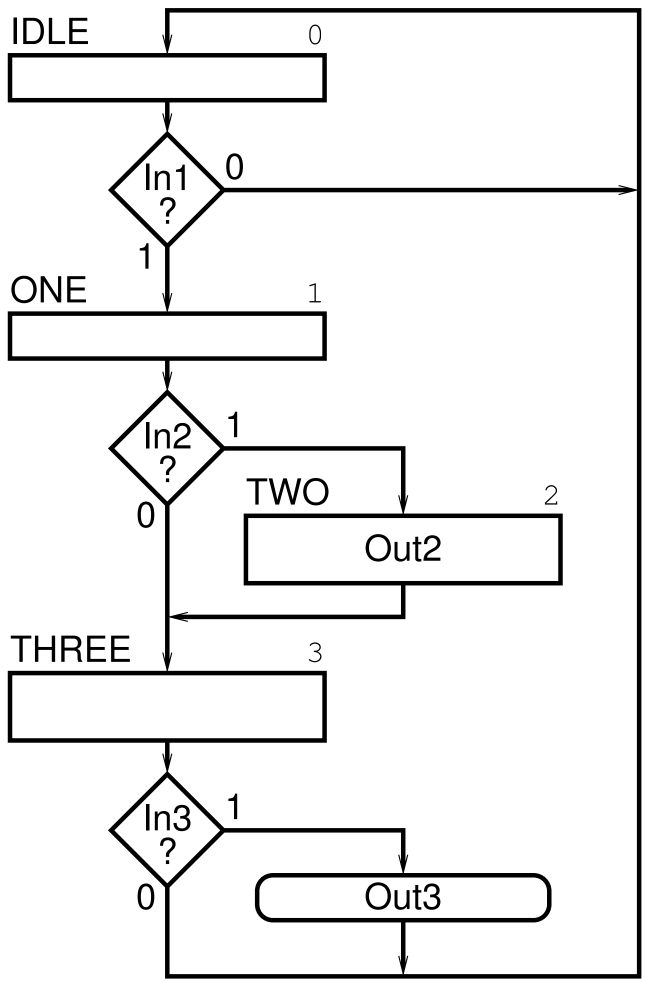
ASM2 – Mealy Machine



ASM2 – Mealy Machine



ASM2 – Mealy Machine



ASM2[a]

```
// ----------- state declaration ----------
logic [1:0] state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= 0;
    else
      case (state)
        0:      if (In1)  state <= 1;
        1:      if (In2)  state <= 2;
                else      state <= 3;
        2:                state <= 3;
        3:                state <= 0;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  assign Out2 = (state == 2);
  assign Out3 = (state == 3) && In3;
```

ASM2[b] – using state names

```
// ----------- state declaration ----------
enum logic [1:0] { IDLE=0, ONE=1, TWO=2, THREE=3 } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  assign Out2 = (state == TWO);
  assign Out3 = (state == THREE) && In3;
```

ASM2^c – using a procedural block to infer combinational logic

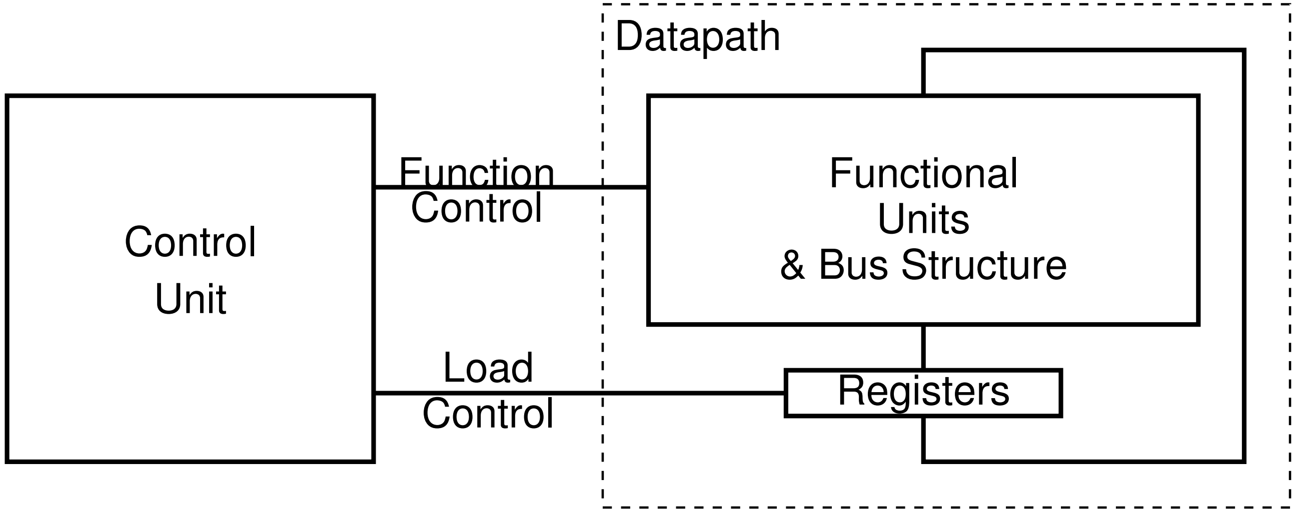
```
// ----------- state declaration ----------
enum logic [1:0] { IDLE, ONE, TWO, THREE } state;

// ------- state + next state logic -------
  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          state <= IDLE;
    else
      case (state)
        IDLE :  if (In1)  state <= ONE;
        ONE  :  if (In2)  state <= TWO;
                else      state <= THREE;
        TWO  :            state <= THREE;
        THREE:            state <= IDLE;
      endcase

// ------------- output logic -------------
logic Out2, Out3;
  always_comb
    begin
      Out2 = (state == TWO);
      Out3 = ((state == THREE) && In3);
    end
```

ASM2^h single procedural block not possible[7]

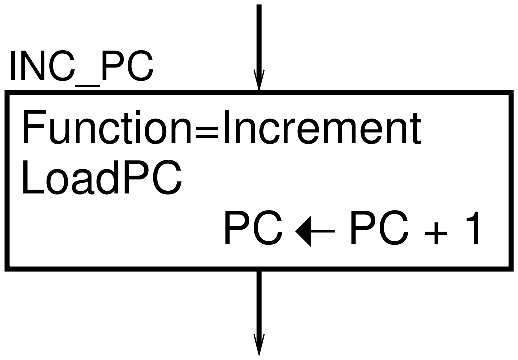
```
// ------ state block generates Out2 ------
enum logic [1:0] { IDLE, ONE, TWO, THREE } state;
logic Out2;

  always_ff @(posedge Clock, negedge nReset)
    if (!nReset)          begin state <= IDLE; Out2  <= 0; end
    else
      begin
        Out2 <= (state == ONE) && In2;
        case (state)
          IDLE :  if (In1)       state <= ONE;
          ONE  :  if (In2)       state <= TWO;
                  else           state <= THREE;
          TWO  :                 state <= THREE;
          THREE:                 state <= IDLE;
        endcase
      end

// ------ output logic generates Out3 -----
logic Out3;
  assign Out3 = (state == THREE) && In3;
```

[7] n.b. *single procedural block solution not possible for Mealy machine since we have conditional outputs.*

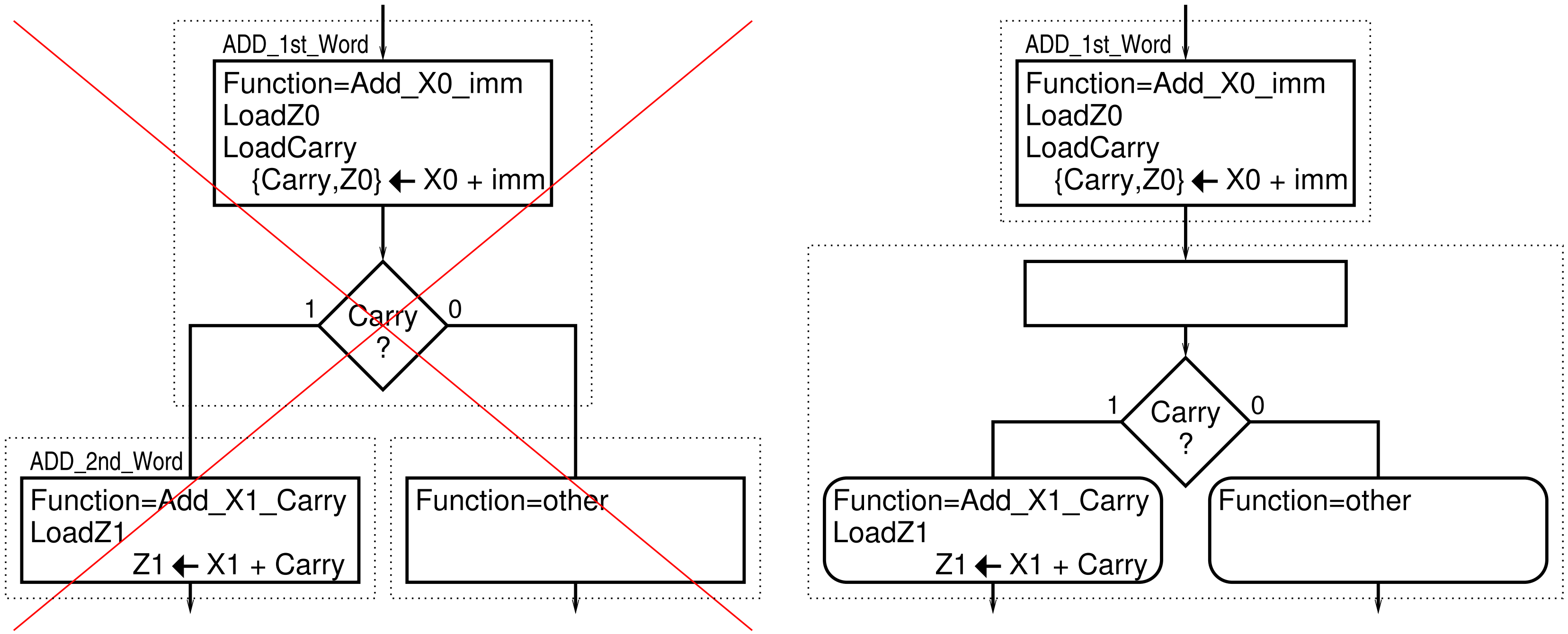
Datapath Control



For successful update we must simultaneously control function and register update:



Datapath Control



Since datapath registers are updated at the end of the clock cycle, we must wait until the next state before testing new values[8].

[8]in some cases it may be better to test values at the input to a register, e.g. if (nextCarry == 1)...