

# システム設計演習(前期分)

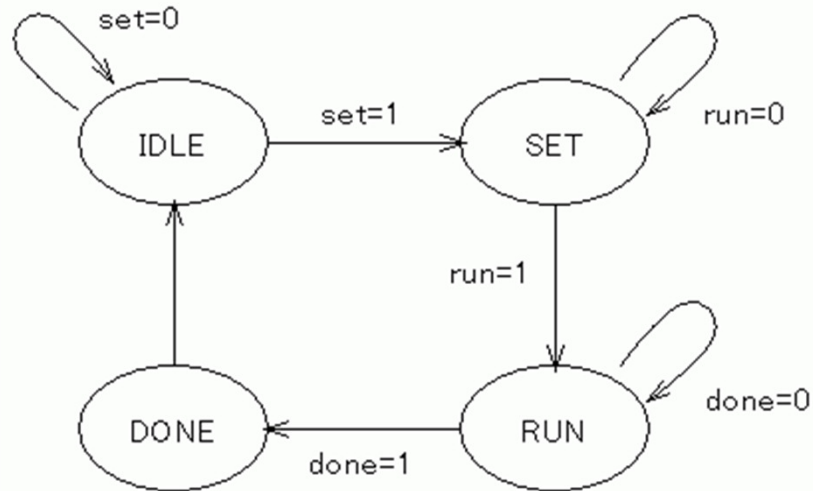
## 第7回

秋田純一

<http://j.mp/akita-class>

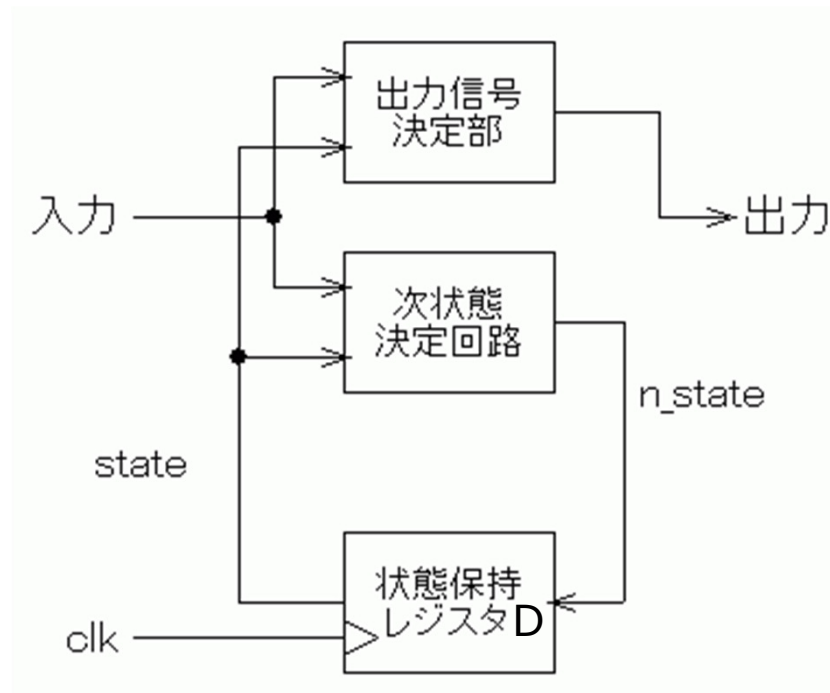
akita@ifdl.jp (@akita11)

# ステートマシンと状態遷移図(p.92～)



- ✓ 複数の「状態」(IDLE, SET, RUN, DONE)
- ✓ 状態の間はクロックの立ち上がりにあわせて”遷移”
  - ✓ 入力によって遷移先が変わることもある  
(この例ではIDLE→SETの遷移は入力set=1のとき)

# ステートマシンの一般構成



- ✓ 「状態」を記憶しておく  
“状態保持レジスタ”
  - ✓ 実体はD-FF
  - ✓ この出力＝“現状態”  
(state)
- ✓ 現状態と入力  
→ 次の遷移先の状態  
(n\_state)を決める
- ✓ clkの立ち上がりで  
n\_stateがstateになる  
＝「遷移」

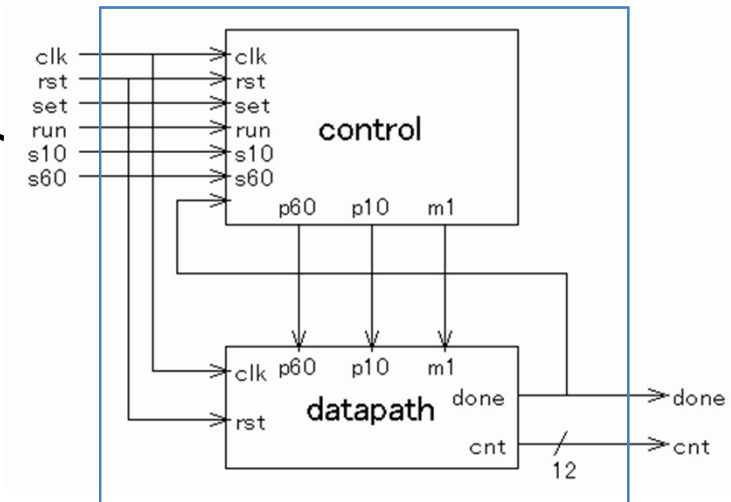
# 設計例: タイマ(p.94~)

## ☑ 入力

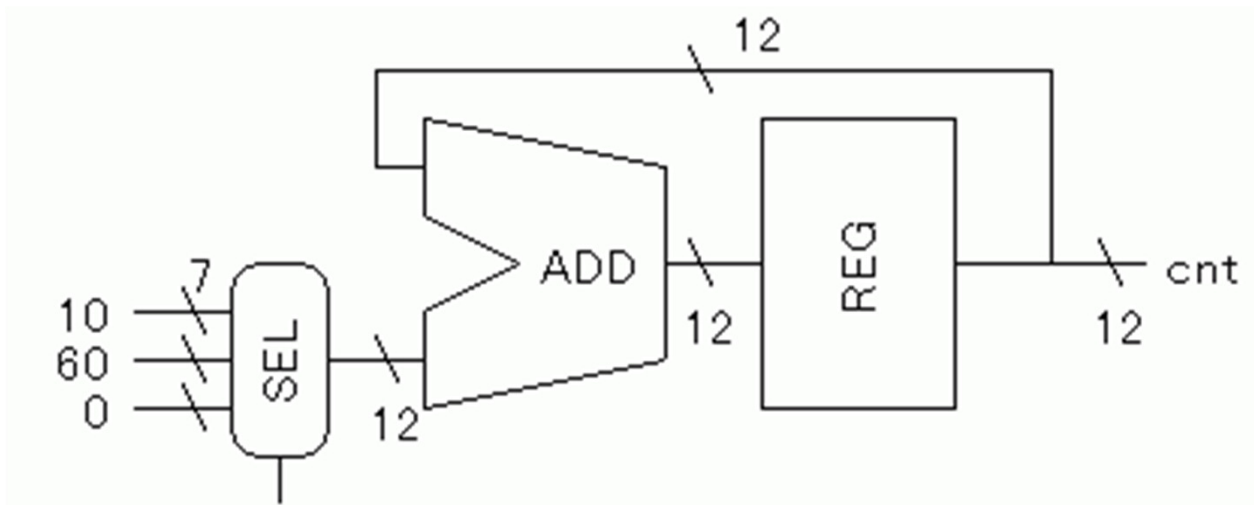
- ☑ clk: クロック。周波数=1Hz
- ☑ rst: リセット。rst=1のときに非同期リセット
- ☑ set: 1のとき、時間セット状態に移る
- ☑ run: 1のとき、カウントダウン状態に移る (タイマの起動)
- ☑ s10: 時間セット状態のとき、1ならば10秒加算する
- ☑ s60: 時間セット状態のとき、1ならば60秒加算する

## ☑ 出力

- ☑ done: 時間カウントが0になったときに、1クロック間だけ1となる出力
- ☑ cnt: 時間の出力(単位=秒)。12ビット。

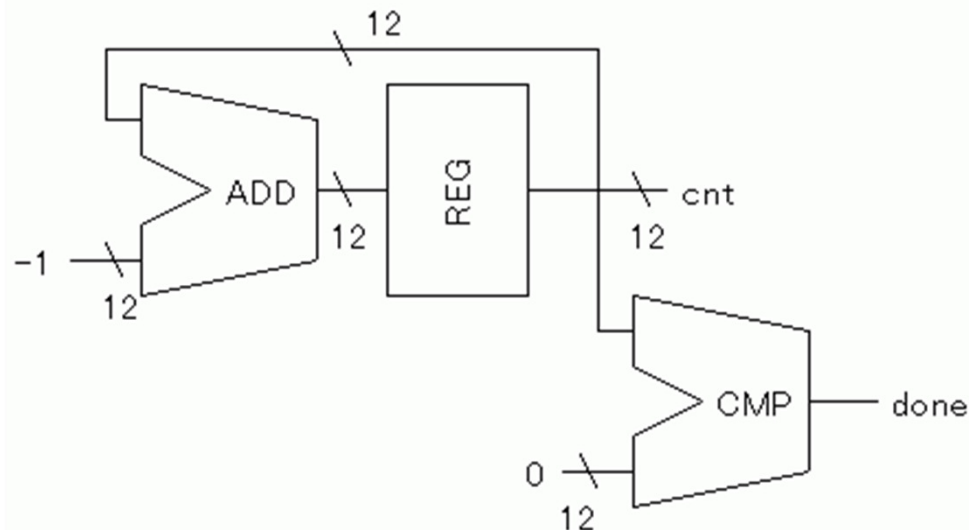


# データパス部の構成(1:時間設定)



- ☑ まず時間セットを行う状態(SET)での動作
  - ☑  $s_{10}=1 \rightarrow cnt$ を+10
  - ☑  $s_{60}=1 \rightarrow cnt$ を+60
- ☑ 現状態 = 現在のcntの値
  - ☑ ADD(加算回路)で、現在のcntと、「加算するべき値」を加算し、クロックの立ち上がりでcntに代入する
  - ☑ 「加算するべき値」=SELの出力  
: 10( $s_{10}=1$ のとき) / 60( $s_{60}=1$ のとき) / 0(その他=cntが変化しない)

## データパス部の構成(2:カウントダウン)

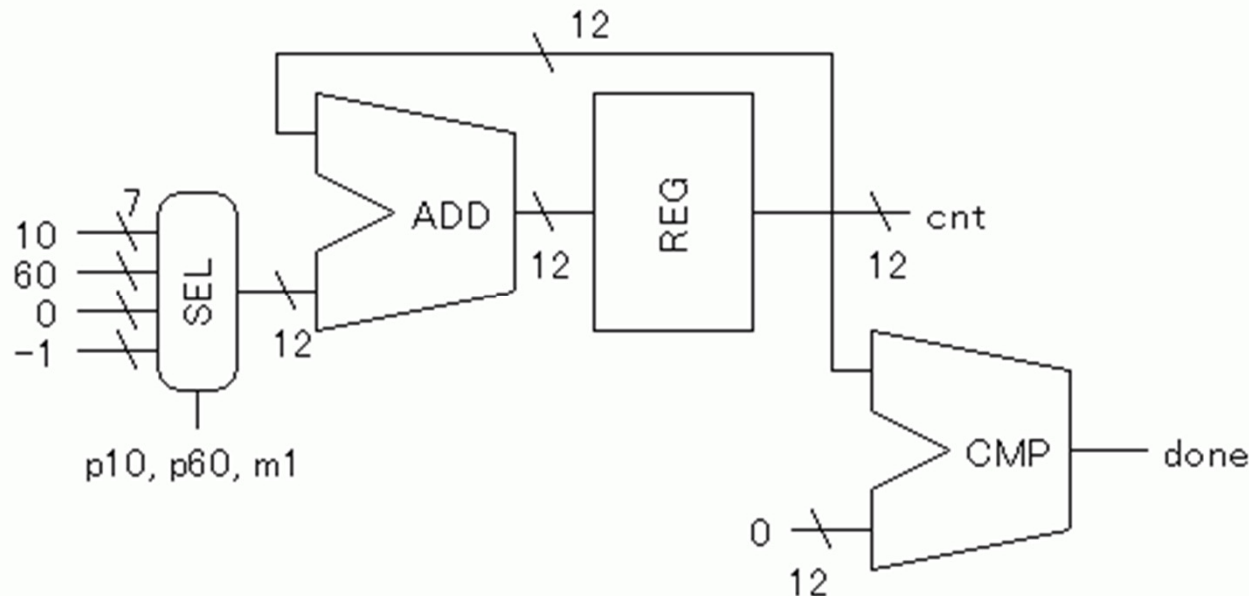


### ☑ カウントダウン中の状態(RUN)での動作

☑ 1秒ごと(=clkの立ち上がりごと)に cntが1ずつ減っていく  
(ADDの片方を"-1"とする)

☑ cnt=0となったとき=カウントダウン終了:出力done=1とする  
(コンパレータでcntと0を比較し、一致したらdone=1)

# データパス部の構成(3:全体)



- ✓ 「時間設定」と「カウントダウン」のデータパスを統合してみる
  - ✓ 両者の違いは、加算器ADDの片方の値+CMPの有無だけ
- ✓ 加算器ADDに与える値を、セレクタSELによって選択する
  - ✓ p10: p10=1のとき、cntに10を加える(s10=1に対応)
  - ✓ p60: p60=1のとき、cntに60を加える(s60=1に対応)
  - ✓ m1: m1=1のとき、cntを1減らす(カウントダウン動作時に対応)

# データパス部のVHDL記述の要点(p.99)

```
entity datapath is
port(
  clk, rst, p10, p60, m1: in std_logic;
  done: out std_logic;
  cnt: out std_logic_vector (11 downto 0));
end datapath;
```

```
architecture Behavioral of datapath is
signal reg12_out, reg12_in: std_logic_vector(11 downto 0);
signal sel_out: std_logic_vector(6 downto 0);
function sign_extend (a: std_logic_vector(6 downto 0))
  return std_logic_vector is
begin
  if (a(6) = '1') then return("11111" & a);
  else return("00000" & a);
  end if;
end sign_extend;
```

符号拡張の関数(7→12bit)  
(例)  
0000000→000000000000  
1000000→11111000000  
0000001→000000000001  
※最上位ビット=符号ビット  
(0=正、1=負:「2の補数」)

SELの記述

```
begin
process (p10, p60, m1) begin -- selector
  if (p10 = '1') then sel_out <= "0001010"; -- "10"
  elsif (p60 = '1') then sel_out <= "0111100"; -- "60"
  elsif (m1 = '1') then sel_out <= "1111111"; -- "-1"
  else sel_out <= "0000000";
  end if;
end process;
```

ADDの記述

```
process (reg12_out, sel_out) begin -- adder
  reg12_in <= reg12_out + sign_extend(sel_out);
end process;
```

REGの記述

```
process (clk, rst) begin -- register
  if (rst = '1') then reg12_out <= "000000000000";
  elsif (clk'event and clk = '1') then reg12_out <= reg12_in;
  end if;
end process;
```

CMPの記述

```
process (reg12_out) begin -- comparator
  if (reg12_out = "000000000000") then done <= '1';
  else done <= '0';
  end if;
end process;
cnt <= reg12_out;
end Behavioral;
```



# (補足)2の補数

- ✓ “0000” = 0 (10進数)
- ✓ “0001” = 1 (10進数)
- ✓ (-1) (10進数) は・・・？
  - ✓  $1 + (-1) = 0$  のはず
  - ✓ “0001” + “1111” = “0000”  
(※桁あふれの最上位ビットは無視する)
  - ✓ したがって “1111” = (-1) (10進数)
- ✓ 一般に「xを全ビット0/1反転し、+1した値」を「xの2の補数」と呼び、(-x)を表す
  - ✓ ※理由は考えてみましょう

# 制御部のVHDL記述の要点(p.103)

```
entity control is
  port(
    clk, rst, set, run, s10, s60, done: in std_logic;
    p10, p60, m1: out std_logic);
end control;
```

```
architecture Behavioral of control is
  type t_state is (IDLE_ST, SET_ST, RUN_ST, DONE_ST);
  signal state, next_state: t_state;
begin
```

```
  process (state, set, run, done) begin -- next state calc
    case state is
      when IDLE_ST =>
        if (set = '1') then next_state <= SET_ST;
        else next_state <= IDLE_ST;
        end if;
      when SET_ST =>
        if (run = '1') then next_state <= RUN_ST;
        else next_state <= SET_ST;
        end if;
      when RUN_ST =>
        if (done = '1') then next_state <= DONE_ST;
        else next_state <= RUN_ST;
        end if;
      when DONE_ST => next_state <= IDLE_ST;
    end case;
  end process;
```

次状態決定部

現状態変数

```
  process (clk, rst) begin -- state register
    if (rst = '1') then state <= IDLE_ST;
    elsif (clk'event and clk = '1') then state <= next_state;
    end if;
  end process;
```

```
  process (state, s10, s60, done) begin -- control signals
    p10 <= '0'; p60 <= '0'; m1 <= '0';
    case state is
      when IDLE_ST => null;
      when SET_ST =>
        if (s10 = '1') then p10 <= '1';
        elsif (s60 = '1') then p60 <= '1';
        end if;
      when RUN_ST =>
        if (done /= '1') then m1 <= '1';
        -- /= : Not Equal ※←教科書の記述は間違い
        end if;
      when DONE_ST => null;
    end case;
  end process;
end Behavioral;
```

出力の決定

# 制御部のステートマシンの構成

回路図

# トップ階層のVHDL記述の要点(p.106)

```
entity timer is
  port(
    clk, rst, set, run, s10, s60: in std_logic;
    done: out std_logic;
    cnt: out std_logic_vector(11 downto 0));
end timer;
```

```
architecture Behavioral of timer is
  component control
  port(
    clk, rst, set, run, s10, s60, done: in std_logic;
    p10, p60, m1: out std_logic);
  end component;
  component datapath
  port(
    clk, rst, p10, p60, m1: in std_logic;
    done: out std_logic;
    cnt: out std_logic_vector(11 downto 0));
  end component;
```

コンポーネント宣言  
(データパスと制御部)

```
signal p10, p60, m1, done_tmp: std_logic;
```

```
begin
  i0: control port map(clk, rst, set, run, s10, s60, done_tmp, p10, p60, m1);
  i1: datapath port map(clk, rst, p10, p60, m1, done_tmp, cnt);
```

データパスと制御部の  
呼び出し・接続

```
done <= done_tmp;
end Behavioral;
```

※インスタンス呼び出しのport mapは、2つの書き方がある

- port map (a=>A, b=>B); ←コンポーネント宣言のport内の信号名(a,b)と実際の信号(A,B)を対応づけ
- port map (A, B); ←コンポーネント宣言のport内の信号の「順」に実際の信号(A,B)を対応付け  
(※C言語などの関数の引数の書き方とにているのは後者)

# トップ階層の構成

回路図

# 構成を意識しないVHDL記述(p.109)

```
entity timer is
  port(
    clk, rst, set, run, s10, s60: in std_logic;
    done: out std_logic;
    cnt: out std_logic_vector(11 downto 0)
  );
end timer;

architecture Behavioral of timer is
  type t_state is (IDLE_ST, SET_ST, RUN_ST, DONE_ST);
  signal state: t_state;
  signal cnt_tmp: std_logic_vector(11 downto 0);
  signal sel_out: std_logic_vector(6 downto 0);
begin
  process (clk, rst) begin
    if (rst = '1') then
      cnt_tmp <= "000000000000";
      done <= '0';
      state <= IDLE_ST;
    elsif (clk'event and clk = '1') then
      case state is
        when IDLE_ST =>
          if (set = '1') then state <= SET_ST;
          else state <= IDLE_ST;
          end if;

```

現状態をwhenで書いて、  
入力に応じて次状態を代入  
(直感的にはわかりやすい)

```
        when SET_ST =>
          if (s10 = '1') then
            cnt_tmp <= cnt_tmp + "000000001010";
          elsif (s60 = '1') then
            cnt_tmp <= cnt_tmp + "000000111100";
          end if;
          if (run = '1') then state <= RUN_ST;
          else state <= SET_ST;
          end if;
        when RUN_ST =>
          cnt_tmp <= cnt_tmp - "000000000001";
          if (cnt_tmp = "000000000001") then
            done <= '1';
            state <= DONE_ST;
          else state <= RUN_ST;
          end if;
        when DONE_ST =>
          done <= '0';
          state <= IDLE_ST;
        end case;
      end if;
    end process;
    cnt <= cnt_tmp;
end Behavioral;
```

# 構成を意識しないVHDL記述の問題点

- ☑ 直感的でわかりやすいかも？
  - ☑ 各「現状態」に対して、入力に応じた遷移先を順に記述していただく
- ☑ ただし、このような「構成（データパス & 制御部）」を意識しないVHDL記述から論理合成で得られる回路は「質」が低くなる傾向がある
  - ☑ 回路規模が大きく、消費電力が大きくなりがち
  - ☑ データパスと制御部を分けて記述するほうが、より「質の高い」論理回路が得られるので好ましい
  - ☑ 「とりあえず動く回路」でよければ、どちらでもOK

# timer\_topの全体構成(メモ)



# timer\_topの入力・出力(メモ)