```
// Description: General Purpose FSM based controller
//
// Sens_a = sensor/control inputs, I(0) doubles as SPIdat
// SPIck_i = clock for loading FSM code in config register
// Clk_i = system clock
// Rst_ni = async assert, sync release reset. Resets all
//   but config register
// Q_ff = control outputs
//
`include "gpf_defines.inc"

module gpf(Sens_ai, SPIck_i, Clk_i, Rst_ni, Q_ff);
  input [`NIN:0] Sens_ai;
  input SPIck_i, Clk_i, Rst_ni;
  output reg [`NOUT:0] Q_ff;

  wire [`NIN:0] Sens_s;
  reg [`NST:0] State_ff;
  wire [`ILEN:0] Instr;
  wire Timeout;
  wire [`NOUT:0] Q_ns;
  wire [`NST:0] State_ns;
  wire Trig;
  wire [`NCNT:0] TimeSet;

  // 2 through 6 are the only valid op codes (after config reg has been
  // loaded)
  inst_out_of_range1: assert
      property (@(posedge Clk_i) Rst_ni |-> Instr[`ILEN:`ILEN-`OPLEN] > 1);
  inst_out_of_range2: assert
      property (@(posedge Clk_i) Rst_ni |-> Instr[`ILEN:`ILEN-`OPLEN] < 7);

  always @ (SPIck_i)
    begin
    fsm_not_reset: assert (Rst_ni == 0)
      else $error("FSM must be held in reset while config reg loads.");
    end

  sync U1 (.Async_ai(Sens_ai), .Clk_i(Clk_i), .Rst_ni(Rst_ni), .Sync_so(Sens_s));

  regcfg U2 (.Addr_i(State_ff), .SPIck_i(SPIck_i), .SPIdat_i(Sens_ai[0]),
.Instr_o(Instr));

  decode U3 (.Sens_ai(Sens_s), .Instr_i(Instr), .TimeOut_i(TimeOut), .NextOut_o(Q_ns),
            .State_i(State_ff), .State_nso(State_ns), .Trig_o(Trig), .Time_o(TimeSet) );

  timer U4 (.Trig_i(Trig), .Time_i(TimeSet), .Clk_i(Clk_i),
            .Rst_ni(Rst_ni), .TimeOut_o(TimeOut));

  always @ (posedge Clk_i, negedge Rst_ni)
    begin
    if (Rst_ni == 0) Q_ff <= `NOUT1'd0;
    else Q_ff <= Q_ns;
    end

  always @ (posedge Clk_i, negedge Rst_ni)
    begin
    if (Rst_ni == 0) State_ff <= `NST1'd0;
    else State_ff <= State_ns;
    end

endmodule
```

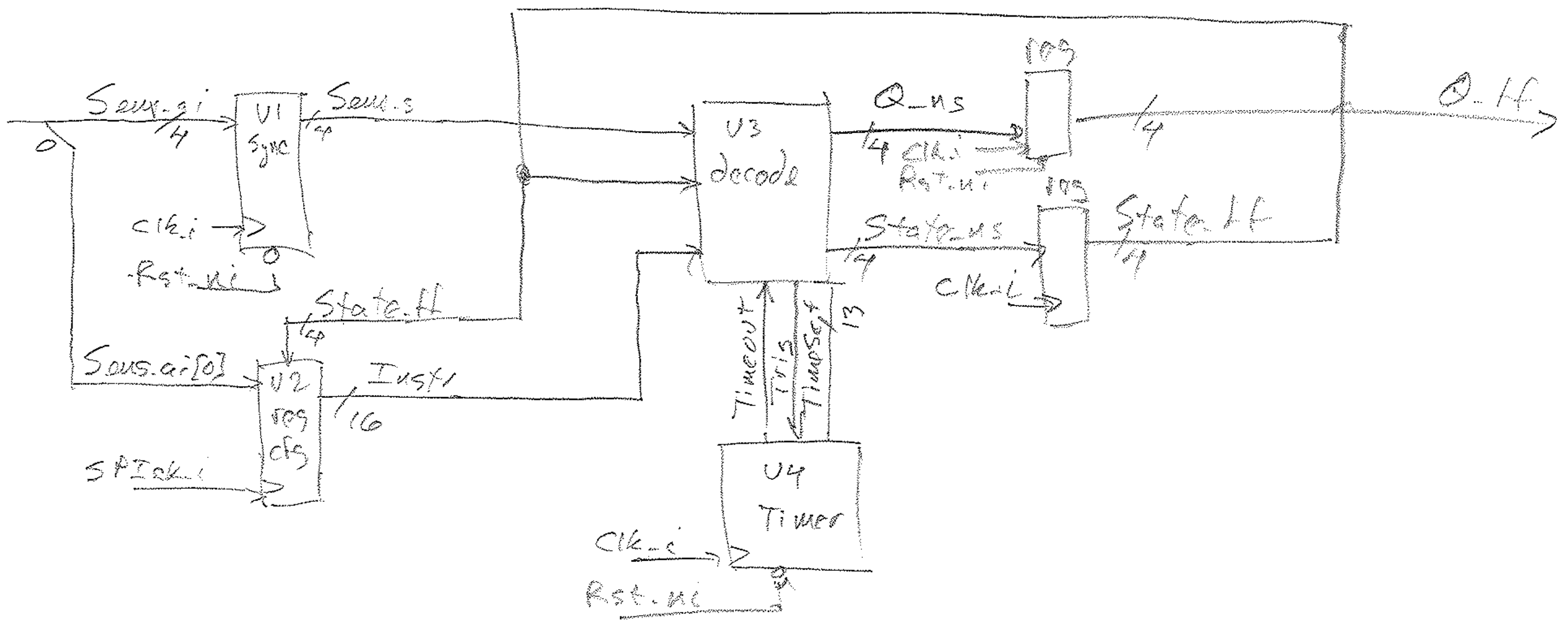
```
// Synchronizer

`include "gpf_defines.inc"

module sync(Async_ai,Clk_i,Rst_ni,Sync_so);
  input [`NIN:0] Async_ai;
  input Clk_i, Rst_ni;
  output [`NIN:0] Sync_so;
  reg [`NIN:0] Q1_ff,Q2_ff;

  always @ (posedge Clk_i, negedge Rst_ni)
    begin
    if (Rst_ni==0)
      begin
        Q1_ff <= `NIN1'd0;
        Q2_ff <= `NIN1'd0;
      end
    else
      begin
        Q1_ff <= Async_ai;
        Q2_ff <= Q1_ff;
       end
    end
  assign Sync_so = Q2_ff;
endmodule



// Description: Serially loadable register

`include "gpf_defines.inc"

module regcfg(Addr_i, SPIck_i, SPIdat_i, Instr_o);
  input [`NST:0] Addr_i;
  input SPIck_i, SPIdat_i;
  output [`ILEN:0] Instr_o;

  reg [`REGSZ:0] Reg_ff;
  integer ii;
  genvar gg;

  // load config register serially via SPI interface
  always @ (posedge SPIck_i)
    begin
    for (ii = 0; ii <`REGSZ; ii = ii+1)
      Reg_ff[ii] <= Reg_ff[ii+1];     //non-blocking assignments required in
sequential logic
    Reg_ff[`REGSZ] <= SPIdat_i;
    end

  generate
    for (gg=0; gg<`ILEN1; gg=gg+1)
      begin : extract_slice_of_Reg_ff
      assign Instr_o[gg] = Reg_ff[Addr_i*`ILEN1+gg];
      end
  endgenerate

endmodule
```

```
// Description: Instruction decode

`include "gpf_defines.inc"

module decode(Sens_ai, Instr_i, TimeOut_i, State_i, State_nso, NextOut_o, Trig_o,
Time_o);
  input [`NIN:0]    Sens_ai;
  input [`ILEN:0]   Instr_i;
  input             TimeOut_i;
  input [`NST:0]    State_i;

  output [`NST:0]   State_nso;  reg    [`NST:0]   State_nso;
  output [`NOUT:0]  NextOut_o; reg    [`NOUT:0]  NextOut_o;
  output            Trig_o;    reg               Trig_o;
  output [`NCNT:0]  Time_o;    reg    [`NCNT:0]  Time_o;

  reg [`NIN+1:0] AllInputs,InMask,InMasked;
  reg dInDetect;

  always @ *
    begin
    NextOut_o = Instr_i[`OUTPUTS];
    State_nso = Instr_i[`NEXTST];
    Trig_o = 0;
    Time_o = 0;
    AllInputs = {Sens_ai,TimeOut_i};
    InMask = Instr_i[`INMASK];
    InMasked = InMask & AllInputs;
    dInDetect = |InMasked;

    case  (Instr_i[`OPCODE])
     `BAND : // branch if all inputs 1, else go to instr++
       begin
       if (~(InMasked == InMask))
         State_nso = State_i + 1;
       end
     `BOR :  // branch if any input 1, else instr++
       begin
       if (~dInDetect)
         State_nso = State_i + 1;
       end
     `SETT : // set timer count, start count down
       begin
       Time_o = Instr_i[`DURAT];
       State_nso = State_i + 1;
       Trig_o = 1;
       end
     `WAND : // spin wait until all inputs 1
       begin
       if (~(InMasked == InMask))
         State_nso = State_i;
       end
       // else hold at current state
     `WOR : // spin wait until some input is 1
       begin
       if (~dInDetect)
         State_nso = State_i;
       end
       // else hold at current state
    endcase
    end
endmodule
```

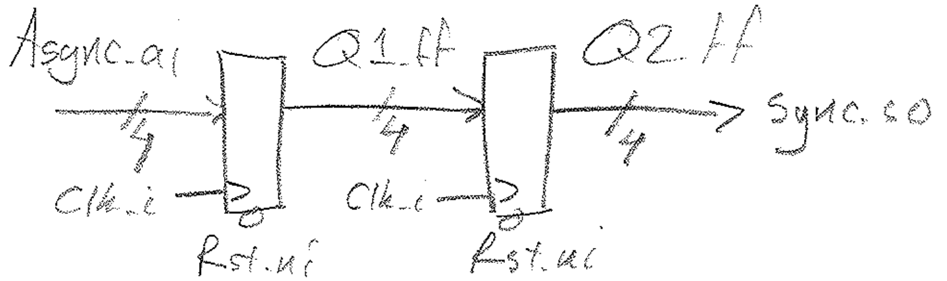
```
// Description: Variable duration count down timer
//
`include "gpf_defines.inc"


module timer(Trig_i, Time_i, Clk_i, Rst_ni, TimeOut_o);
  input Trig_i;
  input [`NCNT:0] Time_i;
  input Clk_i, Rst_ni;
  output TimeOut_o;

  reg [`NCNT:0] Count_ff;

  always @ (posedge Clk_i, negedge Rst_ni)
    begin
    if (Rst_ni==0)
      Count_ff <= `NCNT1'd0;
    else // clock edge branch
      if (Trig_i==1) Count_ff <= Time_i;
      else if (Count_ff > 0) Count_ff <= Count_ff - 1;
      else Count_ff <= `NCNT1'd0;
    end

  assign TimeOut_o = (Count_ff==1) ? 1 : 0;

endmodule
```

```
`define NIN 3   // number of inputs -1
`define NIN1 4   // number of inputs
`define NINT 5   // number of inputs including timer
`define NOUT 3  // number of outputs -1
`define NOUT1 4  // number of outputs
`define NCNT 12  // number of counter bits -1
`define NCNT1 13  // number of counter bits
`define OPLEN 2 // number of opcode bits -1
`define OPLEN1 3 // number of opcode bits
`define NST 3   // number of state bits -1
`define NST1 4   // number of state bits
`define MAXST 15 // maximum state code
`define MAXST1 16 // maximum number of states
`define ILEN 15  // length of max instruction -1
`define ILEN1 16  // length of max instruction
`define REGSZ 255  // register size = # states x instruction size - 1
`define REGSZ1 256  // register size = # states x instruction size
`define BYTE 7 // byte length - 1
`define MAX_STRING 1000 // max string length to read from file
`define EOF -1  //End of file character from getc

// Instruction formats
//
// opcode (OPLEN+1), next state(NST+1), input mask(NIN1+1),  outputs(NOUT+1)
// opcode (OPLEN+1), duration(ILEN + 1 - (OPLEN + 1))
// 3,4,5,4
//
// branch on and of intputs band
//   opcode next_state input_mask (including timer)
`define BAND 2
// branch on or of inputs   bor
`define BOR  3
// set timer                sett
//   opcode duration
`define SETT 4
// wait on and of inputs    wand
`define WAND 5
// wait on or of inputs     wor
`define WOR 6

//Note: BAND nextst, input mask = 00000 always takes the branch
// WAND with input mask = 00000 will always increment the state
// BOR with input mask = 00000 will never take the branch
// WOR with input mask = 00000 will lock you in the current state

// Instruction fields
`define OPCODE 15:13
`define NEXTST 12:9
`define INMASK 8:4
`define OUTPUTS 3:0
`define DURAT 12:0
```

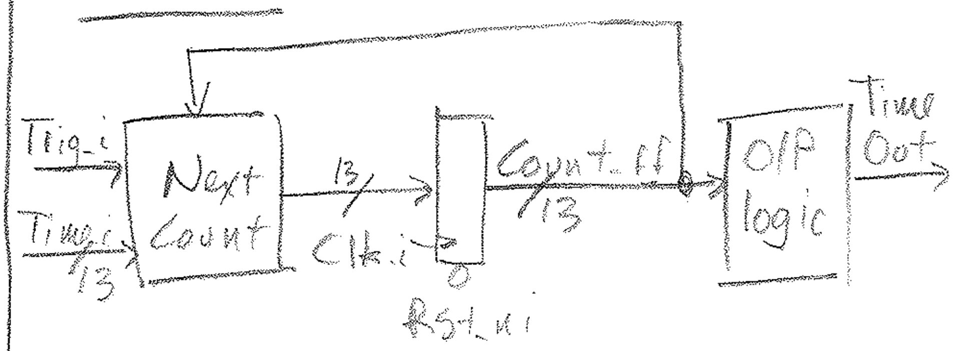
gpf


Sens_ai
4
0
U1
Sync
4
Sens_s
clk_i
0
Rst_ni
State_ff
4
Sens_ai[0]
U2
reg
cfg
SPIclk_i
Instr
16
U3
decode
Q_ns
4
clk_i
Rst_ni
reg
4
Q_ff
State_ns
4
reg
State_ff
4
clk_i
Timeout
Trig
Timeset
13
U4
Timer
Clk_i
Rst_ni

sync

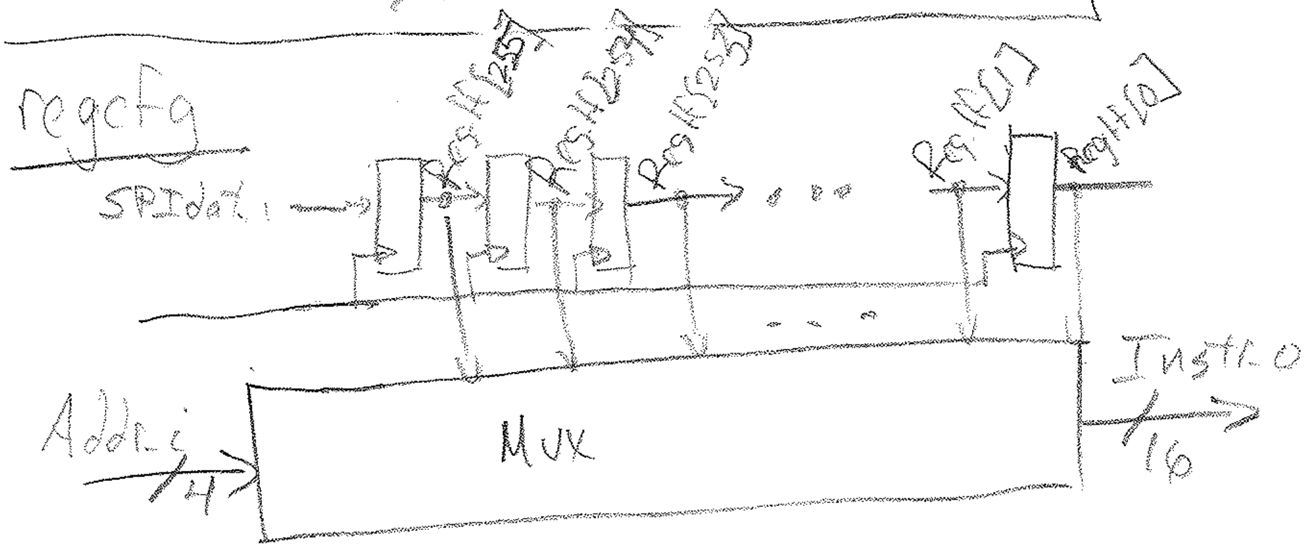


timer

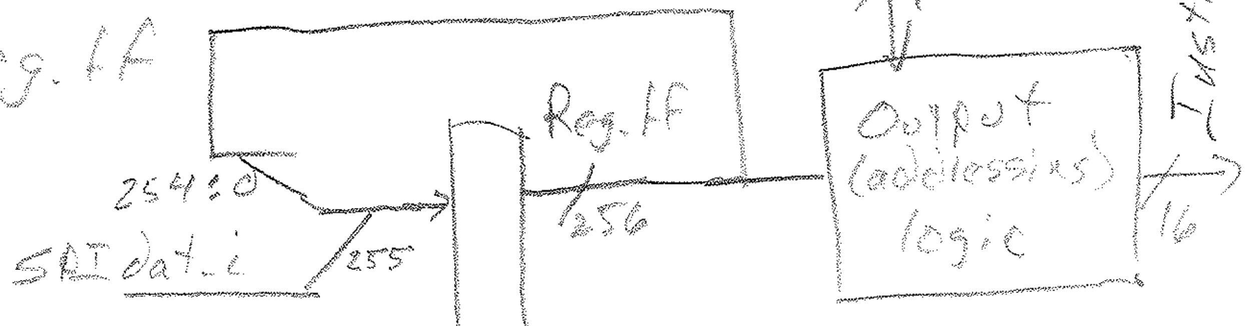


No diagram required for decode

regfg



Another way to draw Reg.ff
^
easier


Addr_i
4
254:0
SPIdat_i
255
Reg.ff
256
Output (addressing) logic
Instr_o
16