TSTE12 Design of Digital Systems Lecture 7

Kent Palmkvist

LINKÖPING UNIVERSITY

Agenda

- Practical issues
- Highlevel simulation models
- Algorithm level design
  - Larger models
  - Time multiplexing
- RTL level models
  - Control units
- Gate level models

TSTE12 Deadlines Y,D,ED

- Initial version design sketch and project plan tuesday 17 September
- Weekly meetings should start
  - Internal weekly meeting with transcript sent to supervisor
- Lab 2 soft deadline Wednesday 24 September at 21.00
  - Lab 2 results will be checked after project end

TSTE12 Deadlines MELE, erasmus

- First project meeting completed
- Tuesday 17 September: First version of requirement specification
- Wednesday 18 September 21.00: Lab 1 deadline
  - Pass required to be allowed continued project participation

Handin (homework), Individual!

- 1st handin published today Monday 16 September
  - Deadline Monday 23 September 23:30
- Use only plan text editor (emacs, vi, modelsim or similar) for code entry.
- Solve tasks INDIVIDUALLY
- Submit answers using Lisam assignment function
  - 4 different submissions for code, one for each code task
  - 1 submission for all theory question answers
- Use a special terminal window when working with handins

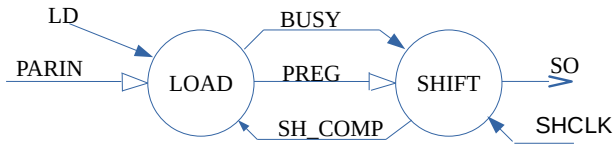
  module load TSTE12 ; TSTE12handin

Example: Paralell to serial converter

- English text description
  - The 8-bit parallel word (PARIN) is loaded into the converter when the control signal LD makes a zero to one transition At this time the status signal BUSY is set high. The data is shifted out serially at a rate controlled by the input shift clock SHCLK. Shifting occurs at the rise of the clock. BUSY remains high until shifting is complete. While BUSY is high, no further loads will be accepted.
- Note some sentences are shared between functions
- Two processes: LOAD and SHIFT

Parallel to serial converter, cont.

- LOAD: (a) 8-bit parallel word (PARIN) load when LD makes a zero to one transition. Set BUSY high. (b) BUSY remains high until shift complete. No new loads while BUSY high
- SHIFT: (a) Data shifted out controlled by rising edge of SHCLK. (b) BUSY remain high until shift complete




```
entity PAR_TO_SER is
  port(LD,SHCLK: in BIT;
       PARIN: in BIT_VECTOR(0 to 7);
       BUSY: inout BIT := '0';
       SO: out BIT);
end PAR_TO_SER;
```


PMG version

- Corresponding code based on processes
- PMG defines interface of each process + signals between the processes
- Code start by defining processes and comments about activities

```
architecture TWO_PROC of PAR_TO_SER is
  signal SH_COMP: BIT :='0';
  signal PREG: BIT_VECTOR(0 to 7);
begin

  LOAD:process(LD,SH_COMP)
  begin
    ---- Activities:
      ----1)Register Load
      ----2)Busy Set
      ----3)Busy Reset
  end process LOAD;

  SHIFT:process(BUSY,SHCLK)
    variable COUNT: INTEGER;
    variable OREG: BIT_VECTOR(0 to 7);
  begin
    ----Activities:
      ----1)Shift Initialize
      ----2)Shift
      ----3)Shift Complete
  end process SHIFT;

end TWO_PROC;
```

PMG -> Code

- Each process has a check for an event, and then a part that execute the data operations

```
SHIFT:process(BUSY,SHCLK)
  variable COUNT: INTEGER;
  variable OREG: BIT_VECTOR(0 to 7);
begin
  ----Activities:
  if BUSY'EVENT and
     BUSY = '1' then
    ----1)Shift Initialize
    COUNT := 7;
      OREG := PREG;
    SH_COMP <= '0';
  end if;
  if SHCLK'EVENT and
     SHCLK= '1'and
     BUSY='1' then
    ----2)Shift
    SO<=OREG(COUNT);
    COUNT := COUNT - 1;
    ----3)Shift Complete
    if COUNT < 0 then
      SH_COMP <= '1';
    end if;
  end if;
end process SHIFT;
```

```
LOAD:process(LD,SH_COMP)
begin
  ---- Activities:
  if LD'EVENT and LD='1'
     and BUSY='0' then
    ----1)Register Load
    PREG <=  PARIN;
    ----2)Busy Set
    BUSY <= '1';
  end if;
    if SH_COMP'EVENT
       and SH_COMP='1' then
      ----3)Busy Reset
      BUSY <= '0';
    end if;
end process LOAD;
```


Timing example

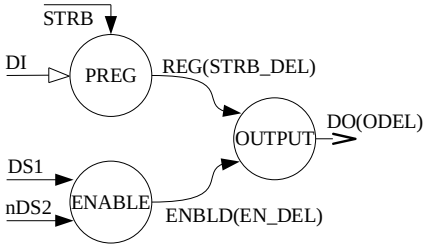
- New model: Buffered register
  - Loaded on rise of the strobe (STRB)
- English description:
  - The register is loaded on the rise of the strobe (STRB), and assuming that the output buffers are enabled, the output of the buffers will change t_{SD} nanoseconds later. The enable condition for the register buffer is the AND of the DS1 and invers of DS2 inputs. Any change in the enable condition will cause the outputs to change t_{ED} nanoseconds later.

Timing example, cont.

- Three processes: PREG, ENABLE, OUTPUT
- Add delay on wires

  t_{SD} = STRB_DEL + ODEL

  t_{ED} = EN_DEL + ODEL



Timing example, cont.

```
entity BUFF_REG is
  generic(
    STRB_DEL,EN_DEL,ODEL: TIME);
  port(
    DI:  in BIT_VECTOR(1 to 8);
    STRB: in BIT;
    DS1:  in BIT;
    NDS2: in BIT;
    DO: out BIT_VECTOR(1 to 8));
end BUFF_REG;


architecture THREE_PROC of BUFF_REG is
    signal REG: BIT_VECTOR(1 to 8);
    signal ENBLD: BIT;
  begin
```

```
  PREG: process(STRB)
    begin
      if (STRB = '1') then
        REG <=DI after STRB_DEL;
      end if;
  end process PREG;

  ENABLE: process(DS1,NDS2)
    begin
      ENBLD <= DS1 and not NDS2 after EN_DEL;
  end process ENABLE;

  OUTPUT: process(REG,ENBLD)
    begin
      if (ENBLD = '1') then
        DO <= REG after ODEL;
      else
        DO <= "11111111" after ODEL;
      end if;
  end process OUTPUT;
end THREE_PROC;
```

Process complexity trade-off

- Number of signals
  - Many signals => slow simulation
- Large processes
  - Complex behavior may not match specification
- Ease of mapping to hardware
  - More processes may simplify mapping

Checking timing

- Additional requirements
  - DI stable SUT ns before STRB rise
  - DI stable HT ns after STRB rise
  - STRB minimum high duration MPW ns
- Implement checks using assert statements

```
assert not (not STRB'stable and (STRB = '1')
          and not DI'stable(SUT))
    report “Setup Time Failure”;
```

Timing Check placement

- Tests in architecture must be copied between architectures
  - May introduce errors
  - If changed, many architectures must be changed
- Solution: Place checks in the entity
  - Check always executed, independent of selected architecture

Timing check example

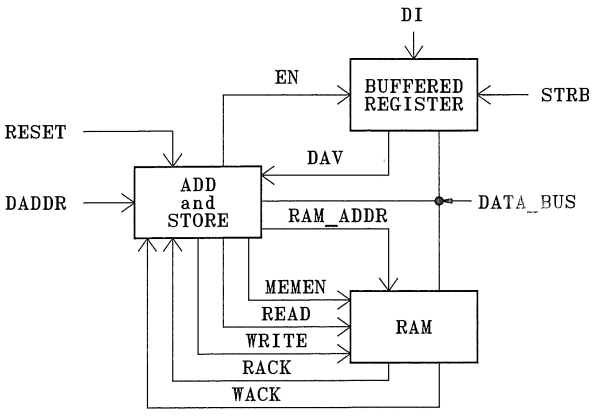
```
Entity BUFF_REG is
    Generic (STRB_DEL, EN_DEL, ODEL,SUT,HT,MPW: TIME);
    Port (DI: in bit_vector(1 to 8);
          STRB : in bit ; DS1 : in bit;
          NDS2 : in bit;
          DO : out bit_vector(1 to 8));
  Begin
    Assert STRB'stable or (STRB = '0') or DI'stable(SUT)
         Report "Setup time Failure";

    Assert STRB'delayed(HUT)'stable or
           (STRB'delayed(HT) = '0') or DI'STABLE(HT)
         Report "Hold Time Failure";

    Assert STRB'stable or (STRB = '1') or
           STRB'delayed'stable(MPW)
         Report "Minimum pulse width failure";

End BUFF_REG;
```

Algorithm level design

- Focus on functions at high abstraction level
  - Subsystems
  - Algorithms to use
- Ignore timing, datapaths etc.

Bigger system example

- 3 units, Add and Store, Buffered Register, RAM
- Predefined sequence
  - Store value in buffer
  - move buffer value to add and store unit
  - Read memory content and add to input value
  - Write back result to memory



Bigger system example, cont.

- DATA_BUS driven by all modules
  - Requires a resolution function
  - Preset to ZZZ to get a useful value (X will always give X)
  - Note this is done in the entity! Reason: inout => Driver on the entity

```
entity RAM is
  generic(RDEL,DISDEL,ACK_DEL,ACK_PW: TIME);
   port(DATA: inout BUS1(7 downto 0):="ZZZZZZZZ";
        ADDR: in MVL4_VECTOR(4 downto 0);
        RD,WRITE,CS: in MVL4;
        RACK,WACK: out MVL4);
end RAM;
```


Simple RAM model

- Use of MVL4 => use drive and sense functions

```
architecture SIMPLE of RAM   is
begin
 MEM: process (CS,RD,WRITE)
  type MEMORY is array(0 to 31) of MVL4_VECTOR(7 downto 0);
  variable MEM: MEMORY:= (others => (others  => '0'));
 begin
   if CS = '1' then
     if RD = '1' then
       DATA <= DRIVE(MEM(INTVAL(ADDR))) after RDEL;
       RACK <= '1' after ACK_DEL,
               '0' after ACK_DEL + ACK_PW;
     elsif WRITE = '1' then
       MEM (INTVAL(ADDR)):= SENSE(DATA,'1');
       WACK <= '1' after ACK_DEL, '0' after ACK_DEL+ACK_PW;
     end if;
   else
     DATA <= "ZZZZZZZZ"  after DISDEL;
   end if;
 end process MEM;
end SIMPLE;
```

Bigger example, cont.

- The RAM-model uses an aggregate to initialize all elements to zero
- ADD and Store is a form of a state machine
  - Go through a sequence step by step
  - Execute some function in each step
  - Each step ends in a wait
- Divide system into datapath and control
- Clock generation as earlier (loop with run)

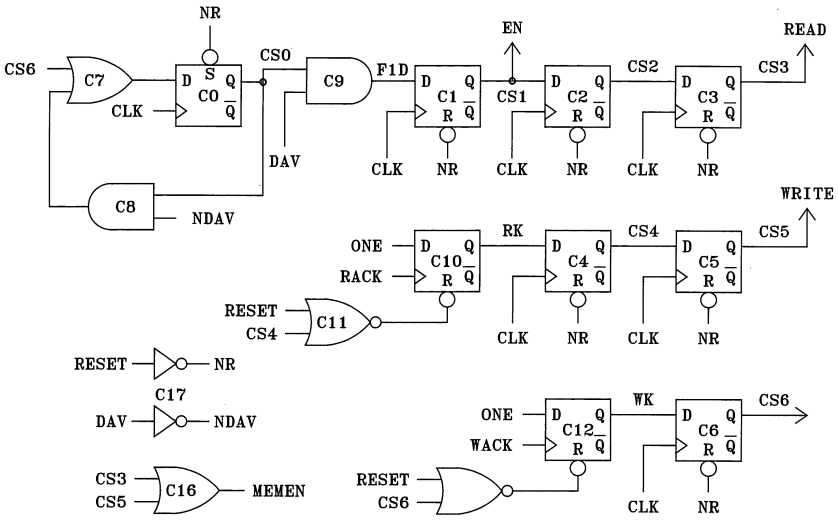
Bigger example, cont.

```
CON: process
  variable DATA_REG:
            MVL4_VECTOR(7 downto 0);
 begin
   if RESET = '1' then           --CS0
     DATA <= "ZZZZZZZZ"after DIS_DEL;
   end if;
   wait on DAV until DAV = '1';
----------------------------------------
   EN <= '1' after CON_DEL;      --CS1
   wait for CLK_PER;
----------------------------------------
   EN <= '0' after CON_DEL;
   DATA_REG := SENSE(DATA,'1');  --CS2
   wait for CLK_PER;
----------------------------------------
   MADDR <=  DADDR after MA_DEL;
   MEMEN <= '1' after CON_DEL;   --CS3
   READ <= '1' after CON_DEL;
   wait on RACK until RACK ='1';
----------------------------------------
   DATA_REG :=
      ADD8(SENSE(DATA,'1'),DATA_REG);
   READ <= '0'after CON_DEL;
   MEMEN <= '0'after CON_DEL;    --CS4
   wait for CLK_PER;
----------------------------------------
   DATA <= DRIVE(DATA_REG) after DO_DEL;
   WRITE <= '1'after CON_DEL;
   MEMEN <= '1'after CON_DEL;    --CS5
   wait on WACK until WACK ='1';
----------------------------------------
   WRITE <= '0'after CON_DEL;
   MEMEN <= '0'after CON_DEL;    --CS6
   DATA <= "ZZZZZZZZ" after DIS_DEL;
   wait for CLK_PER;
end process CON;
```

Control state machine

- Hardware aspects on the control machine
  - Wait can not be used in synthesis
  - Use a manual direct translation technique
- One-hot encoding
  - Simple and straight forward
  - Suitable for FPGA implementation
  - Low complexity decoding of state

One-hot encoded controller



Control step classes

- Automatic increase
  - From step c_i to c_{i+1} after some time
- Handshake
  - Wait for DAV, CS1 => EN = 1, Buffer resets DAV when EN = 1
- Asynchronous stepping
  - CS3 to CS4: Wait for external RACK edge, RACK may be shorter than 1 clock period!

Hardware vs Behavioral model

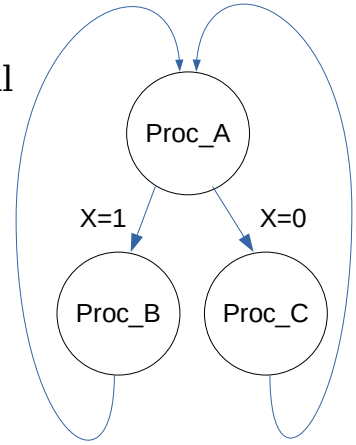
- Important to have same behavior of hardware and VHDL model
- Reset behavior is different
  - The model only checks for reset in CS0
  - Hardware checked reset everywhere
  - Different behavior between model and HW! Bad.
  - Add reset check in every control step

Why single clock domainn

- Reset problems
  - Even single clock domain should synchronize asynchronous reset inputs
  - Must guarantee that whole circuit releases from reset at the same time
- Communication problems
  - Possible race between data and clock
  - Metastability

Control branching

- Want a control machine able to handle conditional execution of sequences
  - Similar to hierarchical finite state machine (FSM)
  - VHDL cannot do jumps, only breaking loops
- Working implementation approach
  - Sequences in individual processes
  - Check at end of process which process to start next
- Output signals from state machine
  - Require Resolution function as assignment done in multiple processes (need to turn off non-active processes)



Each processes should control signal S

Control branching using multiple processes

```
architecture TWO of WAIT_STEPS is

 signal TRIGGERB,TRIGGERBA,
        TRIGGERC,TRIGGERCA: DOT1 := '0';
 signal SINT: RINTEGER register;

begin

A: process
  begin
    SINT <= null;
   wait on RUN,TRIGGERBA,TRIGGERCA until RUN = '1';
    SINT <= 0;                    ---Step 0
   wait for CLK_DEL;
    SINT <= 1;                    ---Step 1
   wait for CLK_DEL;
    SINT <= null;
    if X = '1' then
      TRIGGERB <= not(TRIGGERB);
    else
      TRIGGERC <= not(TRIGGERC);
    end if;
  end process A;

B: process
  begin
    SINT <= null;
   wait on TRIGGERB;
    SINT <= 2;                    ---Step 2
   wait for CLK_DEL;
    SINT <= 3;                    ---Step 3
   wait for CLK_DEL;
    SINT <= null;
    TRIGGERBA <= not(TRIGGERBA);
  end process B;

C: process
  begin
    SINT <= null;
   wait on TRIGGERC;
    SINT <= 4;                    ---Step 4
   wait for CLK_DEL;
    SINT <= 5;                    ---Step 5
   wait for CLK_DEL;
    SINT <= null;
    TRIGGERCA <= not(TRIGGERCA);
  end process C;
  S <= SINT;

end TWO;
```


Time multiplexing

- Problem: Multiple processes driving one signal
  - Multiple drivers (one for each process)
  - Want to enable separate drivers at non-overlapping intervals
  - Assigned signal value should keep the value even after driver enable removed (memory function)
  - Use signal type containing a resolution function
- Remember: This is NOT for synthesis

Time Multiplexing, non-working

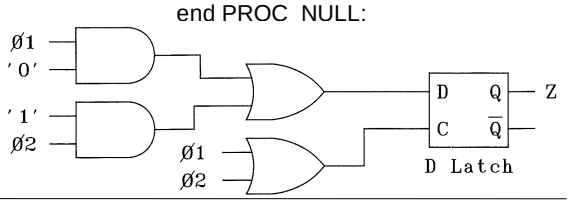
- Two-phase clock
  - One pulse each alternating
- Resolved output signal Z
  - Allows multiple assignment
- Problem
  - Z = ’X’ when PH_TWO assign ’1’
  - Assignment from PH_ONE will not turn off
  - Each driver always outputs last assigned value

```
entity TIME_MUX is
  generic(DEL1,DEL2: TIME);
  port(PHASE_ONE,PHASE_TWO: in MVL4;
       Z: out DOTX := '0');
end TIME_MUX;

architecture PROCESS_IF_0 of TIME_MUX is
begin

  PH_ONE:process(PHASE_ONE)
  begin
    if PHASE_ONE = ’1’ then
      Z <= '0' after DEL1;
    end if;
  end process;

  PH_TWO:process(PHASE_TWO)='1')
  begin
    if PHASE_TWO = ’1’ then
      Z <= '1' after DEL2;
    end if;
  end PH_TWO;

end PROCESS_IF_0;
```


Time multiplexing, cont.

- Ordinary processes
  - Assignment of null disables driver
  - Keyword register in signal declaration
    - defines what happen when all driver are null

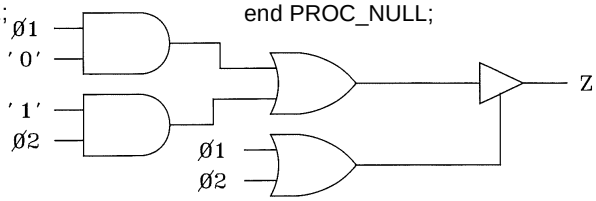
```
architecture PROC_NULL of TIME_MUX is
  signal ZINT: DOTX register;
begin

  PH_ONE: process(PHASE_ONE)
  begin
    if PHASE_ONE = '1' then
      ZINT <= '0' after DEL1;
    else
      ZINT <= null;
    end if;
  end process PH_ONE;

  PH_TWO: process(PHASE_TWO)
  begin
    if PHASE_TWO = '1' then
      ZINT <= '1' after DEL2;
    else
      ZINT <= null;
    end if;
  end process PH_TWO;

  Z <= ZINT;

end PROC  NULL:
```



Time multiplexing, cont.

- Use bus instead of register
  - Default resolution function used if not driven
    - Resolution function recieves an empty input
  - MVL4 resolution function start with ’Z’ value!

```
architecture PROC_NULL of TIME_MUX is
  signal ZINT: DOTX bus;
begin

  PH_ONE: process(PHASE_ONE)
  begin
    if PHASE_ONE = '1' then
      ZINT <= '0' after DEL1;
    else
      ZINT <= null;
    end if;
  end process PH_ONE;

  PH_TWO: process(PHASE_TWO)
  begin
    if PHASE_TWO = '1' then
      ZINT <= '1' after DEL2;
    else
      ZINT <= null;
    end if;
  end process PH_TWO;

  Z <= ZINT;

end PROC_NULL;
```



Time multiplexing, cont.

- Without the use of Register/Bus.
- Separate signals, check 'QUIET to find active assignment

```
entity TIME_MUX is
  generic(DEL1,DEL2: TIME);
  port(PHASE_ONE,PHASE_TWO: in
MVL4;
       Z: buffer MVL4);
end TIME_MUX;

architecture QUIET_MUX of TIME_MUX is
  signal  PH1,PH2,Z1,Z2: MVL4;
begin

  PH_ONE: process(PHASE_ONE)
  begin
    if PHASE_ONE = '1' then
      Z1 <= '0' after DEL1;
    end if;
  end process PH_ONE;

  PH_TWO: process(PHASE_TWO)
  begin
    if PHASE_TWO = '1' then
      Z2 <= '1' after DEL2;
    end if;
  end process PH_TWO;

  Z <= Z1 when not Z1'quiet else
       Z2 when not Z2'quiet else
       Z;

end QUIET_MUX;
```

Register transfer level (RTL)

- At this level can the following aspects be analysed
  - Compare timing between different units at register level
    - Delay in subfunctions, etc.
  - Resource allocation
    - Number of buses, registers, processing elements etc.
  - Scheduling (when to perform an operation)
  - Control structure (e.g., microcoded control units)
  - Bus design

Difference between behavior and dataflow descriptions

- Behavioral model
  - System with two registers and an adder
  - Behavior description does not indicate how operations are performed
  - Command selects operation
  - Only signals that corresponds to saved data

```
entity REG_SYS is
  port(C: in BIT;
       COM: in BIT_VECTOR(0 to 1);
       INP: in BIT_VECTOR(0 to 7));
end REG_SYS;

architecture ALG of REG_SYS is
  signal R1,R2: BIT_VECTOR(0 to 7);
begin

  process(C)
  begin
    if C='1' then
      case COM is
        when "00" => R1 <= INP;
        when "01" => R2 <= INP;
        when "10" => R1 <= ADD8(R1,R2);
        when "11" =>
           R1 <= ADD8(R1,INC8(not(R2)));
      end case;
    end if;
  end process;

end ALG;
```

Behavioral vs Dataflow, cont.

- Dataflow model
  - More signals (many for communication)
  - Operations are registers, multiplexes, or arithmetic/logic operations
  - Global decoding using signals D00 to D11
  - Corresponds to a data flow graph

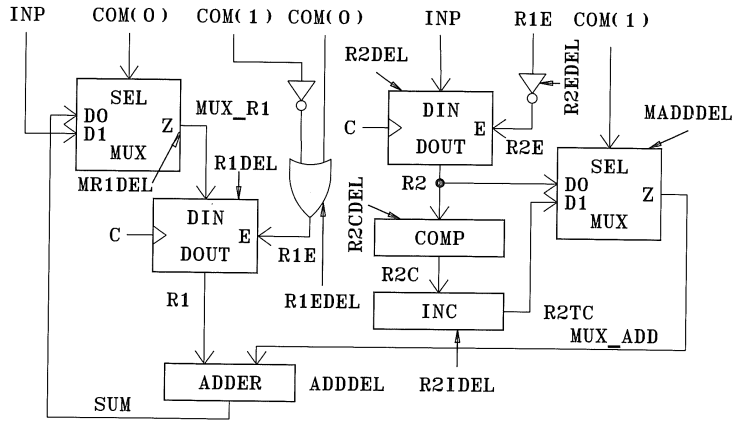
Behavioral vs Dataflow, cont.

```
architecture DF1 of REG_SYS is
  signal MUX_R1,R1,R2,R2C,R2TC,MUX_ADD,SUM:
         BIT_VECTOR(0 to 7);
  signal D00,D01,D10,D11,R1E: BIT;

begin
  D00 <= not COM(0) and not COM(1);
  D01 <= not COM(0) and COM(1);    ---Command Decoder
  D10 <= COM(0) and not COM(1);
  D11 <= COM(0) and COM(1);
  MUX_R1  <= SUM when D00 = '0' else INP; --Reg 1 Mux
  R1E <= D00 or D10 or D11;

R1_REG: process(C)    -- Register 1
  begin
    if (C='1') and C'EVENT) then
      if (R1E = '1') then
        R1 <= MUX_R1;
      end if;
    end if;
  end process;
```

```
R2_REG: process(C)       --Register 2
  begin
    if (C='1') and C'EVENT) then
      if (D01 = '1') then
        R2 <= INP;
      end if;
    end if;
  end process;

  R2C <= not R2;                          ---Complement
  R2TC <= INC8(R2C);                      ---Increment
  MUX_ADD <= R2TC when D11 = '1' else R2; ---Adder Mux
  SUM <= ADD8(R1,MUX_ADD);                ---Adder

end DF1;
```

Described Dataflow implementation

- One-to-one mapping



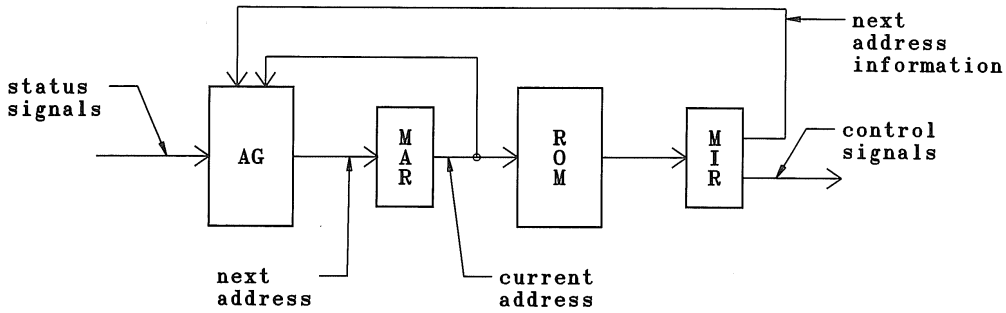

```
  MUX_R1  <= SUM when D00 = '0'
                 else INP;
  R1E <= D00 or D10 or D11;
R1_REG: process(C) begin
  if (C='1') and C'EVENT and
      (R1E='1') then
    R1 <= MUX_R1;
  end if; end process;
R2_REG: process(C) begin
  if (C='1') and C'EVENT and
     (D01='1') then
    R2 <= INP;
  end if; end process;
  R2C <= not R2;
  R2TC <= INC8(R2C);
  MUX_ADD <= R2TC when D11 = '1'
             else R2;
  SUM <= ADD8(R1,MUX_ADD);
```


Control units

- Hard wired
  - Moore (output only dependent on state)
  - Mealy (output dependent on state and input)
  - Fast
  - Custom designed
- Microcoded
  - Cheap
  - Standardized (easy to reuse)

Microcoded control unit

- General structure
  - AG = Adress generator
  - MAR = Memory Adress Register
  - MIR = Memory Instruction register

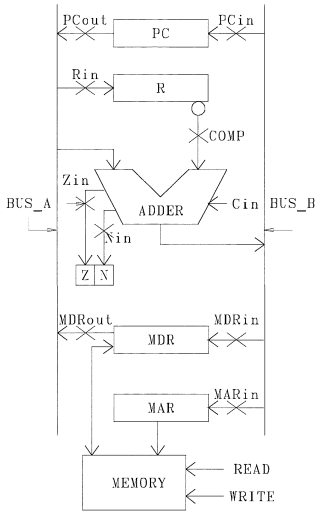


Microcoded control unit

- Advantages
  - Easy to create a generic design
  - Only ROM contents needs to be replaced
  - Easy to change existing design
  - Short design time (low design cost)
  - May use compiler to create ROM contents
- Drawbacks
  - Slower in many cases (ROM must be read)
    - Only Moore type of controllers
  - Small controllers are more expensive due to extra register and ROM
  - Must be designed for worst case regarding required features

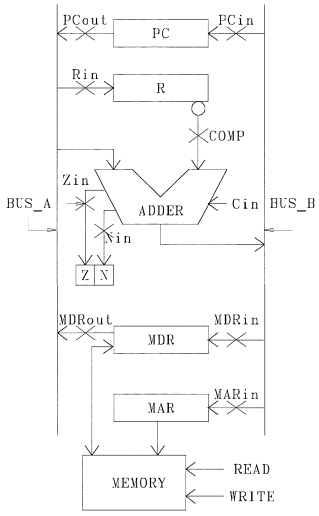
Microcoded control unit, example

- Controller for an extremly small RISC processor
  - 4 register (PC, R, MDR, MIR)
  - 1 subtraction unit
  - Some multiplexers and busses
  - Use the same add unit both for instruction operation and PC update
  - Cost: 9 clock cycles per instruction
- Only one instruction: subtract with branch on negative result
  - 3 byte instruction
    - 1st operand address, 2nd operand address, branch address



Controller for structure

- One instruction execution
  0. PCout,Zin,MARin,READ,ZEND
  1. MDRout,MARin,READ
  2. MDRout,Rin
  3. Pcout,Cin,PCin,MARin,READ
  4. MDRout,MARin,READ
  5. MDRout,COMP,Cin,Nin,MDRin,WRITE
  6. PCout,Cin,PCin,MARin,READ
  7. PCout,Cin,PCin,NNEND
  8. MDRout,Pcin
- 2 loops, 0-7 or 0-8



Control with two jumps, microcoded

- All control steps described in a ROM table
- Easy to understand
- Easy to redesign

```
-------- The microinstructions ROM  ----------------------
ROM: process(C)
  type SQ_ARRAY is array(0 to 8,0 to 8) of BIT;
  constant MEM : SQ_ARRAY:=
-- 0       1    2    3     4     5    6    7     8    COLUMN
MDR_OUT,MAR_IN,N_IN,R_IN,PC_IN,ZEND,C_IN,WRITE,NNEND,micins
(('0',    '1', '0', '0',  '0',  '1', '0', '0',  '0'), --0
 ('1',    '1', '0', '0',  '0',  '0', '0', '0',  '0'), --1
 ('1',    '0', '0', '1',  '0',  '0', '0', '0',  '0'), --2
 ('0',    '1', '0', '0',  '1',  '0', '1', '0',  '0'), --3
 ('1',    '1', '0', '0',  '0',  '0', '0', '0',  '0'), --4
 ('1',    '0', '1', '0',  '0',  '0', '1', '1',  '0'), --5
 ('0',    '1', '0', '0',  '1',  '0', '1', '0',  '0'), --6
 ('0',    '0', '0', '0',  '1',  '0', '1', '0',  '1'), --7
 ('1',    '0', '0', '0',  '1',  '0', '0', '0',  '0'));--8
begin
  MDR_OUT  <=   MEM(INTVAL(C),0) after ROM_DEL;
  MAR_IN   <=   MEM(INTVAL(C),1) after ROM_DEL;
  N_IN     <=   MEM(INTVAL(C),2) after ROM_DEL;
  R_IN     <=   MEM(INTVAL(C),3) after ROM_DEL;
  PC_IN    <=   MEM(INTVAL(C),4) after ROM_DEL;
  ZEND     <=   MEM(INTVAL(C),5) after ROM_DEL;
  C_IN     <=   MEM(INTVAL(C),6) after ROM_DEL;
  WRITE    <=   MEM(INTVAL(C),7) after ROM_DEL;
  NNEND    <=   MEM(INTVAL(C),8) after ROM_DEL;
end process ROM;
```


URISC controller, Mealy

- Inclear sequence
- Hard to modify
- Faster

```
--Hard Wired Control Unit
--Decoder
--First Stage Decoding
ST0 <= not C(2) and not C(1) and not C(0) after AND_DEL;
ST1 <= not C(2) and not C(1) and C(0) after AND_DEL;
ST2 <= not C(2) and C(1) and not C(0) after AND_DEL;
ST3 <= not C(2) and C(1) and  C(0) after AND_DEL;
ST4 <= C(2) and not C(1) and not C(0) after AND_DEL;
ST5 <= C(2) and not C(1) and C(0) after AND_DEL;
ST6 <= C(2) and C(1) and not C(0) after AND_DEL;
ST7 <= C(2) and C(1) and C(0) after AND_DEL;
--Second Stage Decoding
ST07 <= ST0 or ST7 after OR_DEL;
ST25 <= ST2 or ST5 after OR_DEL;
ST36 <= ST3 or ST6 after OR_DEL;
ST57 <= ST5 or ST7 after OR_DEL;
ST78 <= ST7 or C(3) after OR_DEL;
--Control Signals
PC_OUT <= (ST07 or ST36) and not C(3)
          after (OR_DEL + AND_DEL);
C_IN <= ST36 or ST57 after OR_DEL;
PC_IN <= ST36 or ST78 after OR_DEL;
MAR_IN <= not(ST25 or ST78) after (OR_DEL + INV_DEL);
MDR_OUT <=not PC_OUT after INV_DEL;
READ <= MAR_IN; COMP <= ST5; N_IN <= ST5; MDR_IN <= ST5;
WRITE <= ST5; R_IN <= ST2; ZIN <= ST0; ZEND <=ST0;
NNEND <= ST7;
```

More on microcoded controllers

- Lecture 11 will cover more details on microcoded controller structures
  - Introduces also lab 3
- Lab 3 includes an example of a microcoded controller structure
  - Controller used to control a user interface and a datapath
  - Y and D program students have seen this approach in computer technology courses
    - Used there for creating machine instruction implementations

Gate level simulation

- All designs will eventually reach the gate level
- Need accuracy to allow check of timing requirements
  - Setup time on flip-flops
  - Clock signals
  - Races, hazards
  - Glitch example (inverter + and with rising edge input)
- Models must be efficient
  - Large number of gates
  - Slow simulation due to accuracy
    - Still much faster than spice simulation

How accurate can a gate model be?

- Example: 2 input OR-gate

```
Entity OR2 IS
  Port (I1, I2 : in bit; O : out bit);
END OR2;
Architecture DELTA_DEL of OR2 IS
BEGIN
 O <= I1 OR I2;
END DELTA_DEL;
Architecture FIXED_DEL OF OR2 IS
BEGIN
  O <= I1 OR I2 after 3 ns;
END FIXED_DEL;
```

```
ENTITY OR2G IS
  Generic (DEL: TIME)M
  Port (I1, I2 : in bit; O : out
    bit);
END OR2G;
Architecture GNR_DEL of
    OR2G IS
BEGIN
  O <= I1 OR I2 after DEL;
END GNR_DEL;
```


Model accuracy

- Models are better and better, but not good enough
  - Multiple timing models required
  - typical delay, max, min
- Want single model, only changing one constant
  - Timing_CONTROL
  - Set one constant to define type of timing (min, max, typical)

Code example

```
use work.TIMING_CONTROL.all;
entity OR2_TV is
  generic(TMIN,TMAX,TTYP: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2_TV;

architecture VAR_T of OR2_TV is
begin
  O <= I1 or I2 after T_CHOICE(TIMING_SEL,
              TMIN,TMAX,TTYP);
end VAR_T;
```

```
package TIMING_CONTROL is
  type TIMING is (MIN,MAX,TYP,DELTA);
  constant TIMING_SEL: TIMING := TYP;
  function T_CHOICE(TIMING_SEL: TIMING;
                    TMIN,TMAX,TTYP: TIME)
    return TIME;
end TIMING_CONTROL;

package body TIMING_CONTROL is
  function T_CHOICE(TIMING_SEL: TIMING;
                    TMIN,TMAX,TTYP: TIME)
    return TIME is
  begin
    case TIMING_SEL is
      when DELTA => return 0 ns;
      when TYP => return TTYP;
      when MAX => return TMAX;
      when MIN => return TMIN;
    end case;
  end T_CHOICE;
end TIMING_CONTROL;
```


Additional timing details

- Timing is asymmetric
  - Different rise and fall times
  - Needs modeling

```
entity OR2GV is
  generic(TPLH,TPHL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2GV;

architecture VAR_DEL of OR2GV is
begin
  process(I1,I2)
    variable OR_NEW,OR_OLD:BIT;
  begin
    OR_NEW := I1 or I2;
    if OR_NEW = '1' and OR_OLD = '0' then
      O <= OR_NEW after TPLH;
    elsif OR_NEW = '0' and OR_OLD = '1' then
      O <= OR_NEW after TPHL;
    end if;
    OR_OLD := OR_NEW;
  end process;
end VAR_DEL;
```

Load dependency

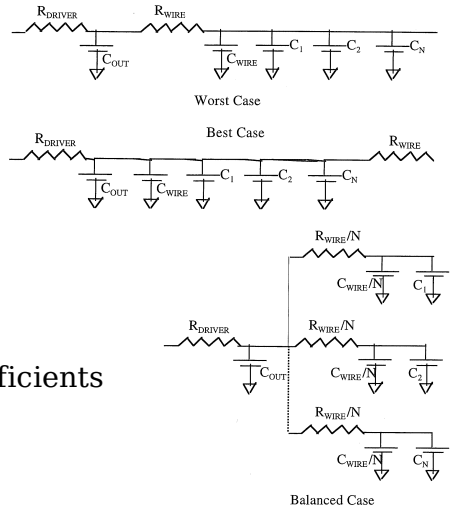
- Every attached gate input slows the output speed
  - Large fan-out
  - Load is gate dependent
    - Number of transistor gates connected
    - Size of transistors on input gate
- Each connection corresponds to a small delay
  - Model each individual input wire delay
  - Gate delay included in output wire delay
- Not good enough still
  - Delay depends on edge slope, temperature, etc.

Common model used in synopsis library compiler

- $D_{TOTAL} = D_I + D_S + D_T + D_C$
- D_I = Intrinsic delay inherent in gate and independent of where/how it is used
- D_S = Slope delay caused by ramp time of the input signal
- D_T Transition delay caused by loading of the output pin (approx R_{driver} (C_{wire}+C_{pin}))
- D_C Connect media delay to an input pin (wire delay).

Different max and min

- Wire delay (D_C) more complicated
  - Worst case
  - Best case
  - Balanced
- Technology library
  - Large amount of information
  - Usually described as tables
  - Sometimes described as polynomial coefficients



Back annotation

- The process of abstraction
  - adding more details to a high level model by analyzing a lower abstraction level model
  - Example: Layout information used to generate timing information in a gate netlist
- Standardized way: SDF
  - Add timing info from layout to gate level
  - Useful for general timing requirements and properties)
  - Delays module path, device, interconnect, and port

SDF file format

- Timing checks: setup, hold, recovery, removal, skew, width, period, and no change
- Timing constraints: path, skew, period, sum, and diff
- Each trippel defines min, typical, and max delay
  - One for positive edge
  - One for negative edge

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE top/b/c)
  (DELAY
    (ABSOLUTE
      (IOPATH (posedge clk) q (2:3:4) (5:6:7))
      (PORT clr (2:3:4) (5:6:7))
    )
  )
  (TIMINGCHECK
    (SETUPHOLD d (posedge clk) (3:4:5) (-1:-1:-1))
    (WIDTH clk (4.4:7.5:11.3))
  )
)
```


SDF File format, cont.

- Design/instance-specific or type/library-specific data
- Timing environment:
- intended operating timing environment
- Scaling, environmental, and technology parameters
- Incremental delay builds on the previous models timing by adding/subtracting timing information
- Absolute replaces timing information

Gate models of increasing complexity

- Creating accurate library models is time consuming
- Delay, timechecks etc. can be done in many different ways
- A standard has evolved that defines what parameters to use
  - Simplifies back annotation
  - Allows for accelerated models (hard-coded)

VITAL models of gates

- Three parts: Input delay, Functional and Path delay

```
library IEEE;
use IEEE.VITAL_Primitives.all;
library LIBVUOF;
use LIBVUOF.VTABLES.all;
architecture VITAL of ONAND is
  attribute VITAL_LEVEL1 of VITAL : architecture is TRUE;
  SIGNAL A_ipd  : STD_ULOGIC := 'X';
  SIGNAL B_ipd  : STD_ULOGIC := 'X';
  SIGNAL C_ipd  : STD_ULOGIC := 'X';
  SIGNAL D_ipd  : STD_ULOGIC := 'X';
begin
--  INPUT PATH DELAYS
  WireDelay : block
  begin
    VitalWireDelay (A_ipd, A, tipd_A);
    VitalWireDelay (B_ipd, B, tipd_B);
    VitalWireDelay (C_ipd, C, tipd_C);
    VitalWireDelay (D_ipd, D, tipd_D);
  end block;
```

```
--  BEHAVIOR SECTION
  VITALBehavior : process (A_ipd, B_ipd, C_ipd, D_ipd)
   -- functionality results
    VARIABLE Results: STD_LOGIC_VECTOR(1 to 1):=(others => 'X');
    ALIAS Y_zd : STD_LOGIC is Results(1);
   -- output glitch detection variables
    VARIABLE Y_GlitchData : VitalGlitchDataType;
  begin
   --  Functionality Section
    Y_zd := (NOT ((D_ipd) AND ((B_ipd) OR (A_ipd) OR C_ipd))));
   --  Path Delay Section
    VitalPathDelay01 (
      OutSignal => Y,
      GlitchData => Y_GlitchData,
      OutSignalName => "Y",
      OutTemp => Y_zd,
      Paths => (0 => (A_ipd'last_event, tpd_A_Y, TRUE),
                1 => (B_ipd'last_event, tpd_B_Y, TRUE),
                2 => (C_ipd'last_event, tpd_C_Y, TRUE),
                3 => (D_ipd'last_event, tpd_D_Y, TRUE)),
      Mode => OnDetect,
      Xon => Xon,
      MsgOn => MsgOn,
      MsgSeverity => WARNING);
  end process;
end VITAL;
```

Detection of timing errors

- Input path delay: Transport delay dependent on previous value and wire delay
- Functional part. Boolean expression or lookup tables for fast simulation
- Path delay: output delay, glitch handling
- Models often includes error detection
  - Short spikes, short setup/hold timing etc.
  - Unacceptable values (Z or X)
  - Unacceptable input combinations (both set and reset active on SR flipflop)

LINKÖPING UNIVERSITY