Register Transfer Methodology (RTL)

We typically use **algorithms** to accomplish complex tasks

Although it is common to execute algorithms on a GPU, a hardware implementation is sometimes needed because of power and performance constraints

RT methodology is a design process that describes system operation by a sequence of data transfers and manipulations among **registers**

This methodology supports sequential execution semantics used by microprocessors to execute a program

Consider an algorithm that computes the sum of 4 numbers, divides by 8 and rounds the result to the nearest integer

```
size = 4;
sum = 0;
for i in (0 to size-1) do
    { sum = sum + a(i); }
```

Register Transfer Methodology (RTL)

```
q = sum/8;
r = sum rem 8;
if (r > 3)
   { q = q + 1; }
outp = q;
```

Characteristics of an algorithm:

- Algorithms use **variables**, memory locations with a symbolic addresses, to store intermediate results
- Algorithms are executed sequentially and the order of the steps is important

One approach is to convert **sequential execution** into a **structural data flow**, where the sequence is embedded in the 'flow of data'

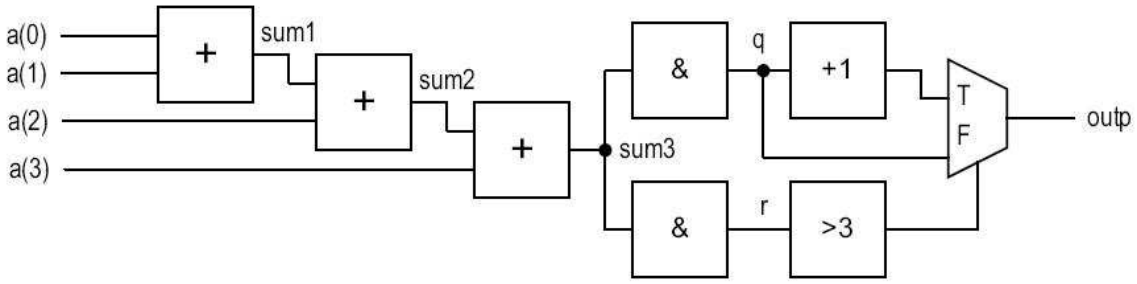
This is accomplished by mapping an algorithm into a system of *cascading hardware blocks*, where each block represents a statement in the algorithm

Register Transfer Methodology (RTL)

For example, the previous algorithm can be **unrolled** into a data flow diagram

```
sum <= 0;
sum0 <= a(0);
sum1 <= sum0 + a(1);
sum2 <= sum1 + a(2);
sum3 <= sum2 + a(3);
q <= "000" & sum3(8 downto 3);
r <= "00000" & sum3(2 downto 0);
outp <= q + 1 when (r > 3) else q;
```

Note that this is very different from the algorithm -- the circuit is strictly combinational with NO memory elements



The *structural data flow* model can only be applied to small tasks and is not flexible

Register Transfer Methodology (RTL)

Register Transfer Methodology introduces hardware that *matches* the variable and sequential execution model

- Registers are used to store intermediate data (model symbolic variables)
- A control path (FSM) is used to specify the order of register operations
- A data path is added to implement the operations (FSMD)

The basic action in RT methodology is the *register transfer operation*:

$$r_{\text{dest}} \leftarrow f(r_{\text{src1}}, r_{\text{src2}}, \ldots, r_{\text{src3}})$$

The function *f* uses the contents of the source registers, plus external inputs in some cases

Difference between an algorithm and an RT register is the implicit embedding of *clk*

- At the rising edge of the clock, the outputs of registers r_{src1}, r_{src2} become available
- The outputs drive the inputs of a combinational circuit that represents *f( )*
- At the **next rising edge** of the clock, the result is stored into r_{dest}

FSMD

The function *f()* can be any expression that is representable by a combinational circuit

$r \leftarrow 1$
$r \leftarrow r$
$r0 \leftarrow r1$
$n \leftarrow n - 1$
$y \leftarrow a \oplus b \oplus c \oplus d$
$s \leftarrow a^2 + b^2$

Note that we will continue to use the notation *_reg* and *_next* for the current output and next input of a register

The notation

$r_1 \leftarrow r_1 + r_2$

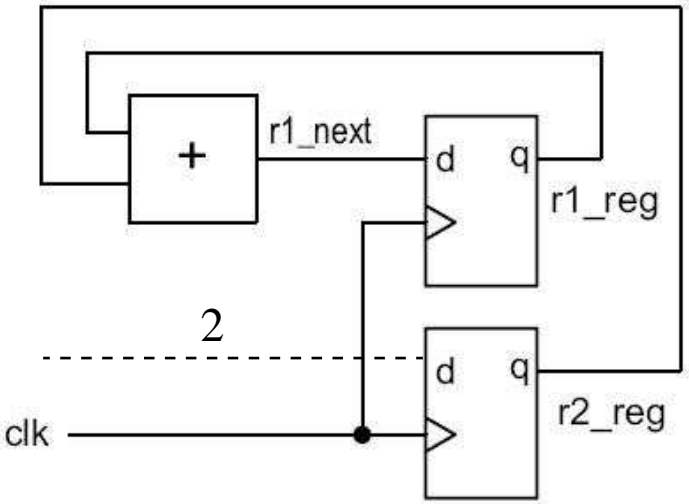
is translated as

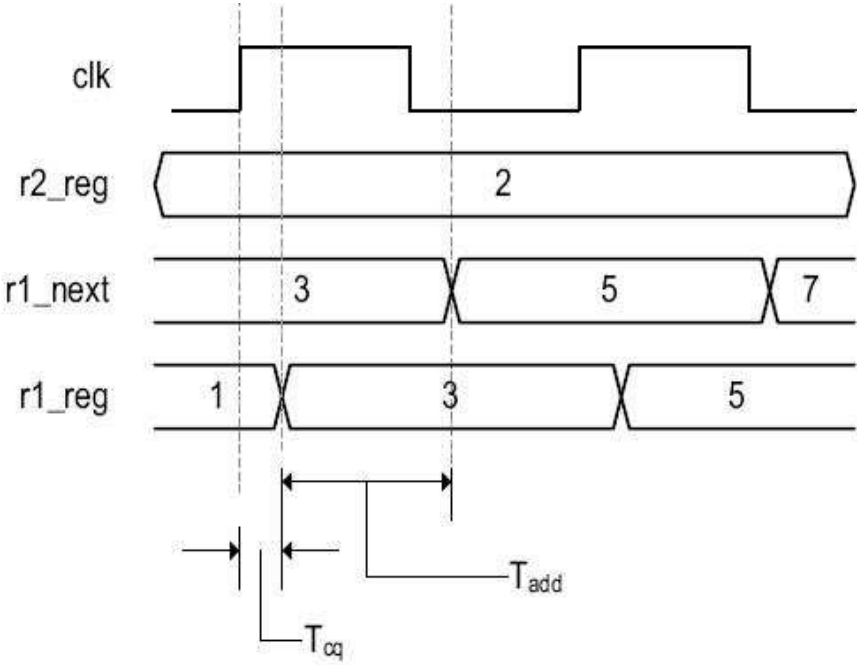
```
r1_next <= r1_reg + r2_reg;
r1_reg <= r1_next; -- on the next rising edge of clk
                   -- this inside the FF process block
```

FSMD

Be sure to study this carefully because it is heavily used in digital systems

$$r \leftarrow r1 + r2$$





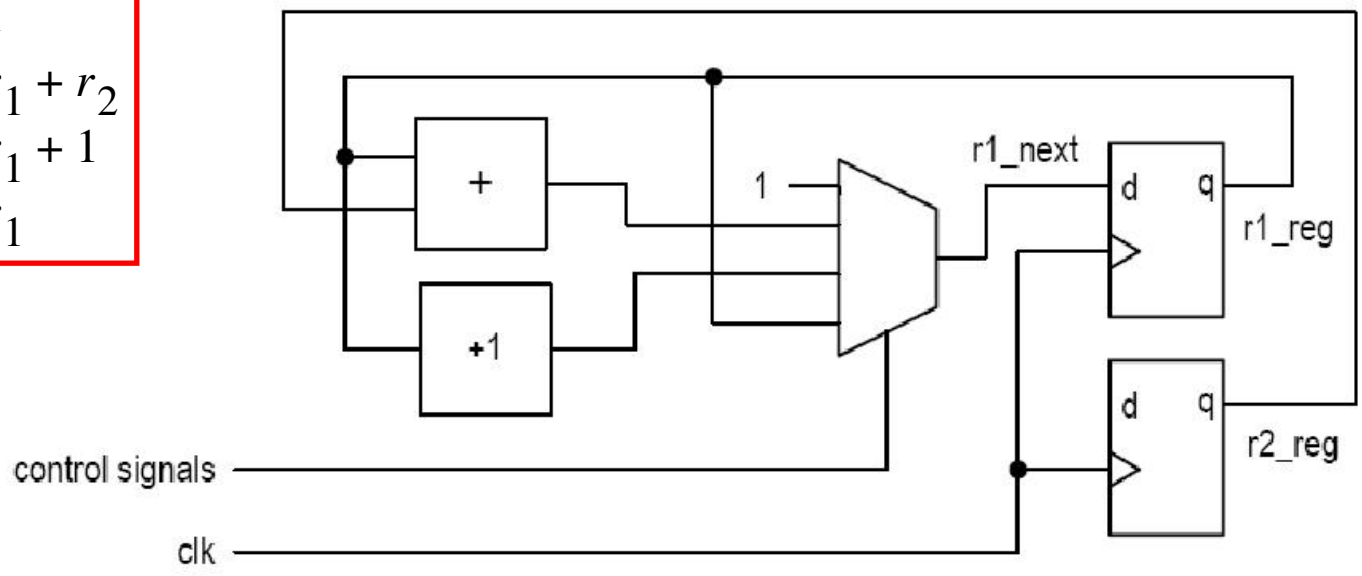
Multiple RT operations

An algorithm consists of many steps and a *destination* register my be loaded with different values over time

FSMD

Consider the following sequence of operations

$r_1 \leftarrow 1$
$r_1 \leftarrow r_1 + r_2$
$r_1 \leftarrow r_1 + 1$
$r_1 \leftarrow r_1$

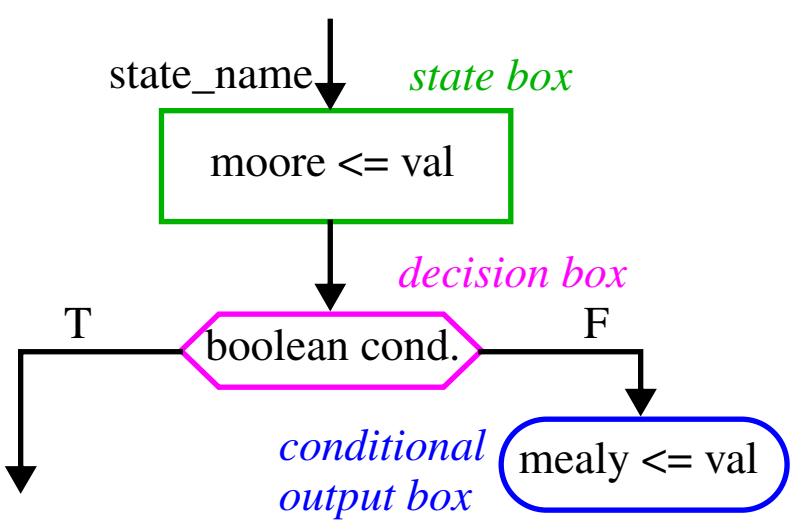


Since r_1 is the destination of multiple operations, we need a MUX to route the proper value to its input

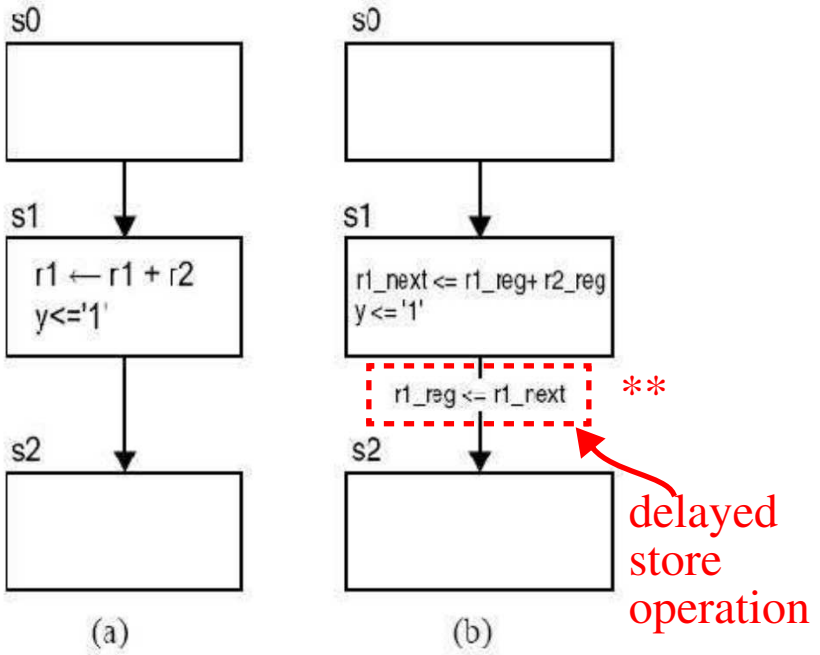
An FSM is used to drive the *control signals* so that the sequence of operations are carried out in the order given

FSMD and ASMD

An extended ASM chart known as **ASMD** (ASM with datapath) chart can be used to represent the FSMD



ASMD



State transitions and register updates occur at the same time on the rising edge of the clk

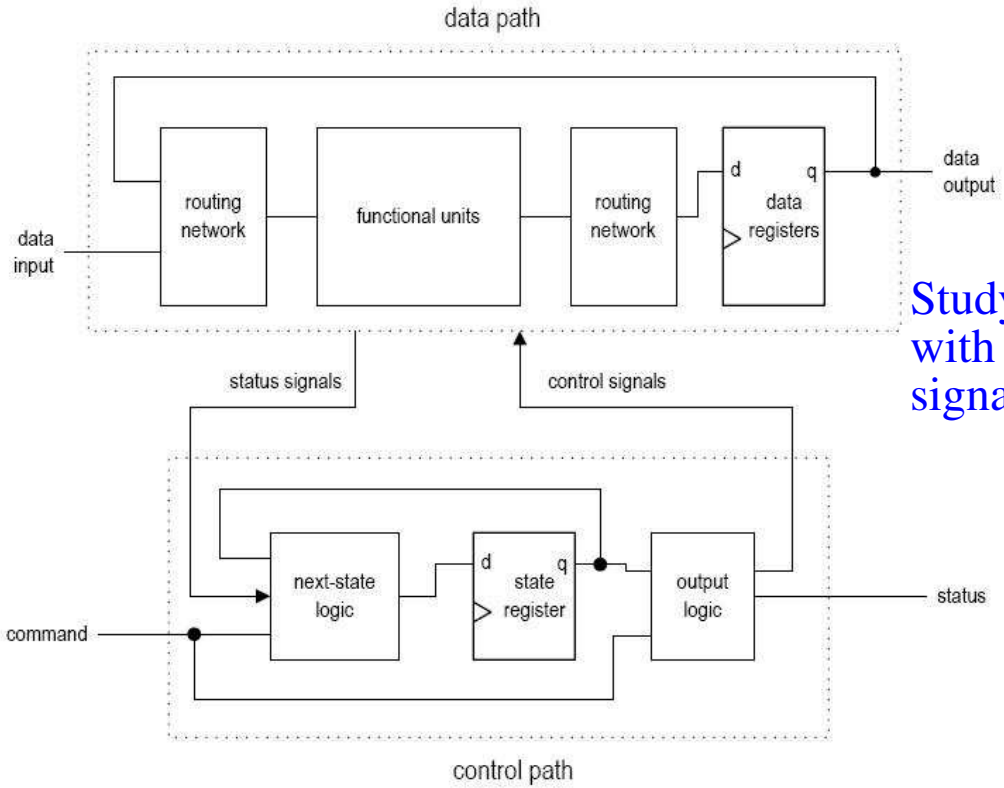
DELAYED STORE: The new value of r_1 is only available when the FSM enters the s_2 state

FSMD

NOTE: When a register is NOT being updated with a new value, it is assumed that it maintains its current value, i.e.,

$r_1 \leftarrow r_1$ These actions are NOT shown in the ASMD/state chart

Conceptual block diagram of an FSMD



Data Path

Regular sequential circuit

Study and become familiar with the input/output signals of both modules

Control Path

Random sequential circuit

FSMD Design Examples

Consider a repetitive addition multiplier

Basic algorithm: 7*5 = 7+7+7+7+7

```
if (a_in=0 or b_in=0) then
    { r = 0; }
else
    {
    a = a_in;
    n = b_in;
    r = 0;
    while (n != 0)
        {
        r = r + a;
        n = n - 1;
        }
    }
    return(r);
```

FSMD Design Examples

This code is a better match to an ASMD because ASMD does not have a **loop** construct

```
    if (a_in = 0 or b_in = 0) then
       { r = 0; }
    else
       {
       a = a_in;
       n = b_in;
       r = 0;
 op:   r = r + a;
       n = n - 1;
       if (n = 0) then
          { goto stop; }
       else
          { goto op; }
       }
stop: return(r);
```

FSMD Design Examples

To implement this in hardware, we must first define the I/O signals

- *a_in*, *b_in*: 8-bit unsigned input
- *clk*, *reset*: 1-bit input
- *start*: 1-bit command input
- *r*: 16-bit unsigned output
- *ready*: 1-bit status output -- asserted when unit has completed and is ready again

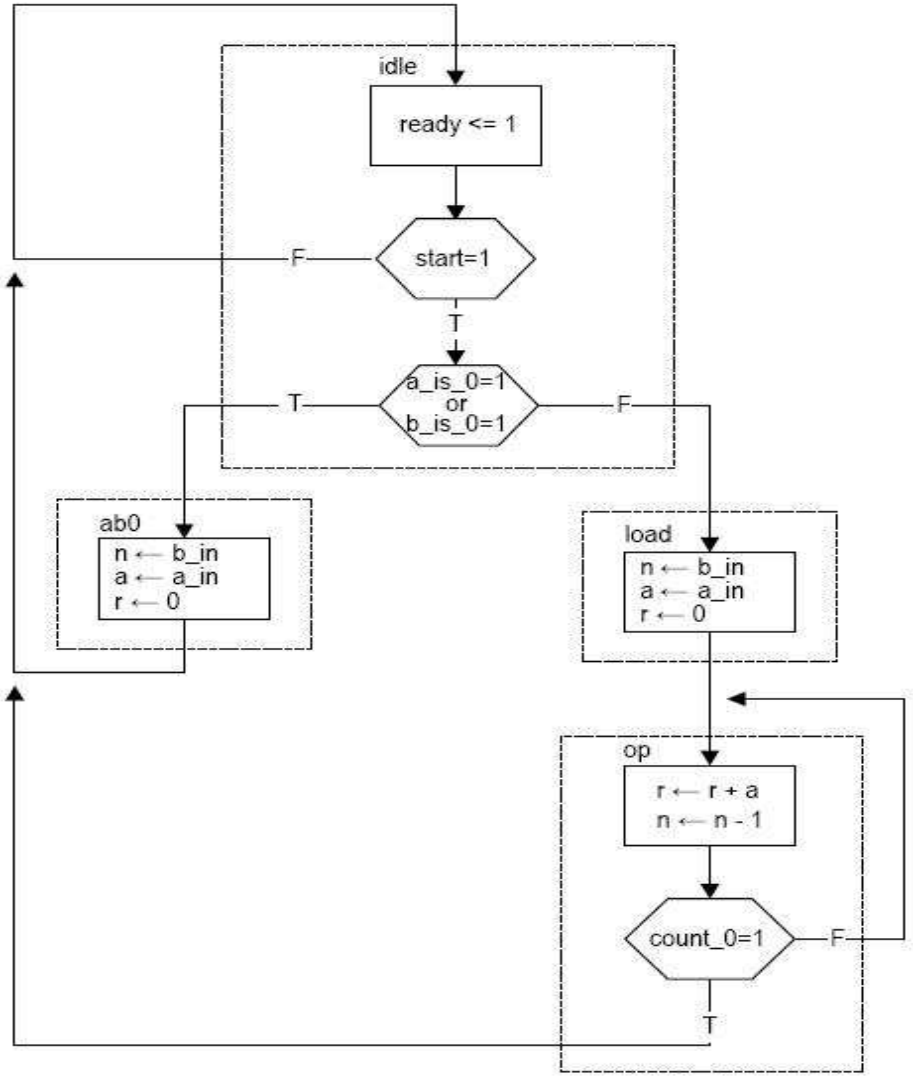
The *start* and *ready* signals are added to support sequential operation

When this unit is embedded in a larger design, and the main system wants to perform multiplication

- It checks *ready*
- If '1', it places inputs on *a_in* and *b_in* and asserts the *start* signal

FSMD Design Examples

The ASMD uses *a*, *n* and *r* data registers to emulate the three variables



FSMD Design Examples

With the ASMD chart available, we can refine the original block diagram

We first divide the system into a *data path* and a *control path*

For the control path, the input signals are *start*, *a_is_0*, *b_is_0* and *count_0* -- the first is an external signal, the latter three are status signals from the data path

These signals constitute the inputs to the FSM and are used in the *decision boxes*

The output of the control path are *ready* and control signals that specify the RT operations of the data path

In this example, we use the state register as the output control signals

Visualizing the data path can be accomplished by doing the following:

- List all RT operations
- Group RT operation according to the destination register
- Add combinational circuit/mux
- Add status circuits

FSMD Design Examples

For example

- RT operation with the *r* register

  $r \leftarrow r$ (in the idle state)
  $r \leftarrow 0$ (in the load and ab0 states)
  $r \leftarrow r + a$ (in the op state)

- RT operations with the *n* register

  $n \leftarrow n$ (in the idle state)
  $n \leftarrow$ b_in (in the load and ab0 state)
  $n \leftarrow n - 1$ (in the op state)

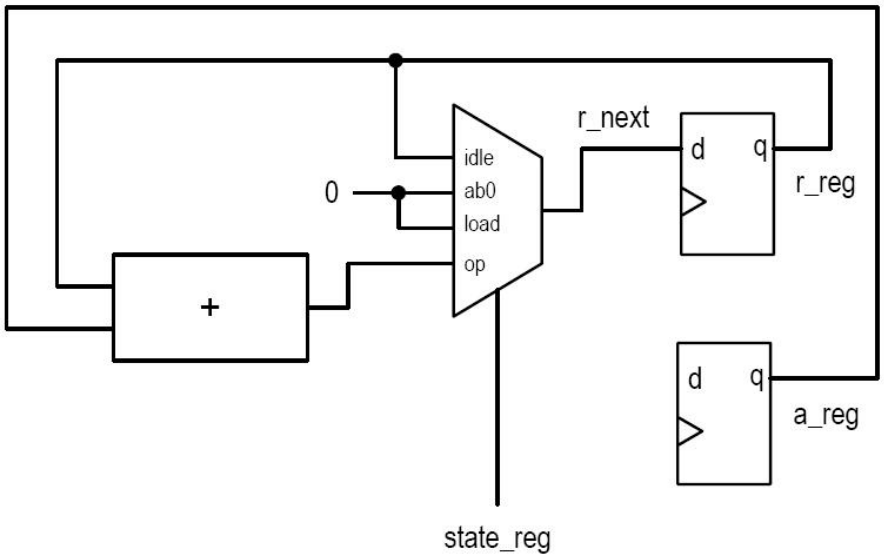
- RT operations with the *a* register

  $a \leftarrow a$ (in the idle and op states)
  $a \leftarrow$ a_in (in the load and ab0 states)

Note that the **default** operations MUST be included to build the proper data path

FSMD Design Examples

Let's consider the circuit associated with the *r* register



The three possible sources, 0, *r* and *r*+*a* are selected using a MUX

The select signals are labeled symbolically with the state names

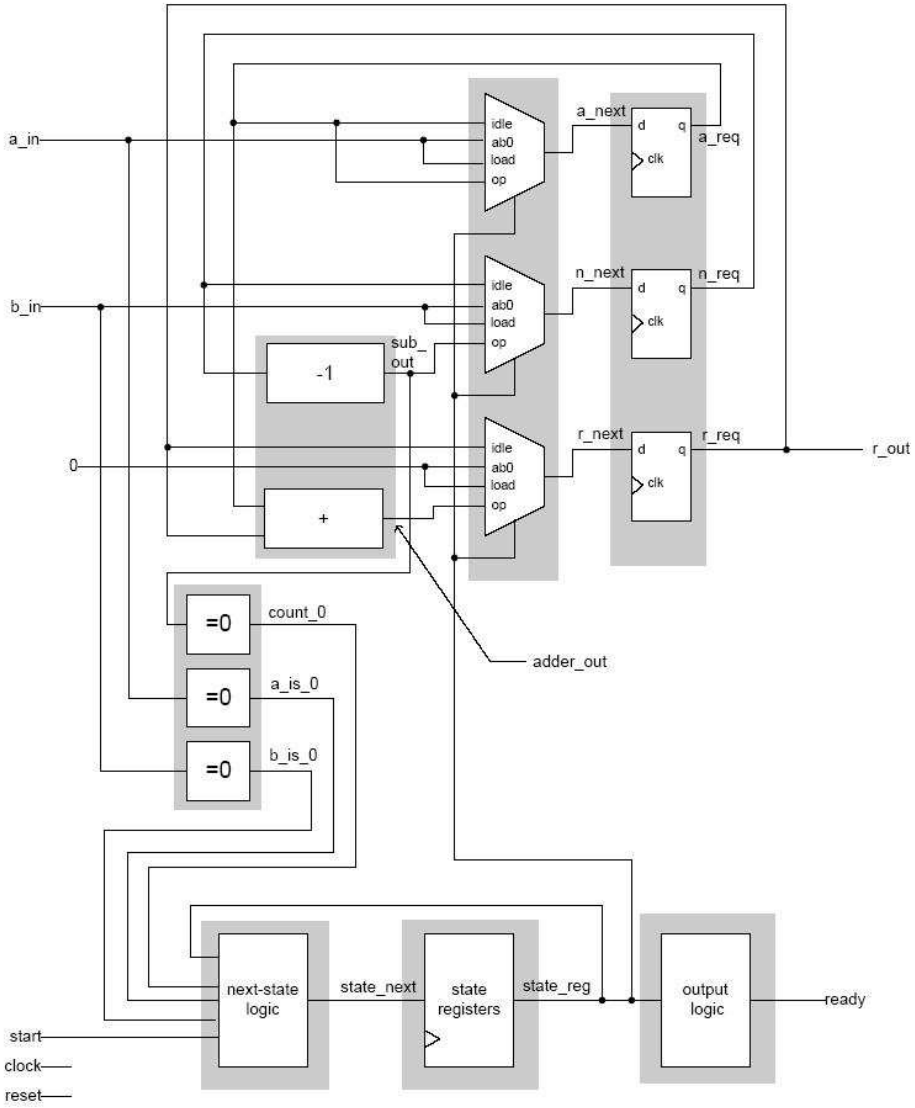
The routing is consistent with what is given on the previous slide

We can repeat this process for the other two registers and combine them

The status signals are implemented using three comparators

FSMD Design Examples

The entire (un-optimized) control and data path



FSMD Design Examples

```
architecture two_seg_arch of seq_mult is
   constant WIDTH: integer := 8;
   type state_type is (idle, ab0, load, op);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
   signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
   begin

   -- state and data register
   process(clk, reset)
      begin
      if (reset = '1') then
         state_reg <= idle;
         a_reg <= (others => '0');
         n_reg <= (others => '0');
         r_reg <= (others => '0');
```

Two Segment VHDL Descriptions of FSMDs

```
        elsif (clk'event and clk = '1') then
            state_reg <= state_next;
            a_reg <= a_next;
            n_reg <= n_next;
            r_reg <= r_next;
        end if;
    end process;

    -- combinational circuit
    process(start, state_reg, a_reg, n_reg, r_reg, a_in,
        b_in, n_next)
        begin
        state_next <= state_reg;
        a_next <= a_reg;
        n_next <= n_reg;
        r_next <= r_reg;
        ready <='0';
```

Two Segment VHDL Descriptions of FSMDs

```
case state_reg is
   when idle =>
      if (start = '1') then
         if (a_in = "00000000" or
               b_in = "00000000") then
            state_next <= ab0;
         else
            state_next <= load;
         end if;
      end if;
      ready <= '1';

   when ab0 =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(b_in);
      r_next <= (others => '0');
      state_next <= idle;
```

Two Segment VHDL Descriptions of FSMDs

```
            when load =>
               a_next <= unsigned(a_in);
               n_next <= unsigned(b_in);
               r_next <= (others => '0');
               state_next <= op;

            when op =>
               n_next <= n_reg - 1;
               r_next <= ("00000000" & a_reg) + r_reg;
               if (n_next = "00000000") then
                  state_next <= idle;
               end if;
         end case;
      end process;

      r <= std_logic_vector(r_reg);
   end two_seg_arch;
```