Register Transfer Methodology: Practice

In this lecture, we will look at several examples of RT methodology applied to a variety of applications.

Examples include control of a *clockless* device, hardware acceleration of a sequential algorithm, and control- and data-oriented applications:

- Design Example: One-Shot Pulse Generator
- Design Example: GCD
- Design Example: UART
- Design Example: SRAM Interface Controller
- Design Example: Square Root Approximation Circuit

**One-Shot Pulse Generator**

Used to illustrate differences between a *regular sequential circuit*, an *FSM* and *RT methodology*.

A one-shot pulse generator generates a single, fixed-width pulse (5 clk cycles wide) when triggered.
Register Transfer Methodology: Practice
We divided sequential circuits into three types:

- Regular sequential circuit => regular next-state logic
  For example, a mod-10 counter

\[
\text{r\_next} \leftarrow (\text{others} \Rightarrow '0') \text{ when } \text{r\_reg} = (\text{TEN} - 1) \text{ else } \text{r\_reg} + 1;
\]
Register Transfer Methodology: Practice

- FSM => random next-state logic

For example, edge-detection circuit

-- next-state logic

```vhdl
process (state_reg, strobe)
begin
  case state_reg is
  when zero =>
    if (strobe = '1') then
      state_next <= edge;
    else
      state_next <= zero;
    end if;
  others =>
    ...
end case;
```

Register Transfer Methodology: Practice

- FSMD (RT methodology) => both types, most flexible and capable
  For example, a multiplier
One-Shot Pulse Generator

A one-shot pulse generator has 2 input signals, go (trigger pulse) and stop and one output signal, pulse

The pulse signal is asserted when go is asserted for one clk cycle (if go is asserted again within 5 clk cycles, it is ignored)

If stop is asserted during the 5 clk cycle period, pulse is set back to ’0’ immediately

This circuit contains a regular part (a counter) and a random part (idle or pulse)

FSM implementation
One-Shot Pulse Generator

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pulse_5clk is
  port(
    clk, reset: in std_logic;
    go, stop: in std_logic;
    pulse: out std_logic
  );
end pulse_5clk;

architecture fsm_arch of pulse_5clk is
  type fsm_state_type is
    (idle, delay1, delay2, delay3, delay4, delay5);
  signal state_reg, state_next: fsm_state_type;
begin
```
One-Shot Pulse Generator

-- state register

process(clk, reset)
begin
if (reset = '1') then
state_reg <= idle;
elif (clk'event and clk='1') then
state_reg <= state_next;
end if;
end process;

-- next-state logic & output logic

process(state_reg, go, stop)
begin
pulse <= '0';
case state_reg is
when idle =>
if (go = '1') then
state_next <= delay1;
end if;
end case;
end process;
One-Shot Pulse Generator

else
    state_next <= idle;
end if;

when delay1 =>
    if (stop = '1') then
        state_next <= idle;
    else
        state_next <= delay2;
    end if;
pulse <= '1';

when delay2 =>
    if (stop = '1') then
        state_next <= idle;
    else
        state_next <= delay3;
    end if;
pulse <= '1';
One-Shot Pulse Generator

when delay3 =>
    if (stop = '1') then
        state_next <= idle;
    else
        state_next <= delay4;
    end if;
    pulse <= '1';
when delay4 =>
    if (stop = '1') then
        state_next <= idle;
    else
        state_next <= delay5;
    end if;
    pulse <= '1';
when delay5 =>
    state_next <= idle;
    pulse <= '1';
One-Shot Pulse Generator

\[\text{end case;}\]
\[\text{end process;}\]
\[\text{end fsm\_arch;}\]

Regular sequential circuit implementation

It can be considered a mod-5 counter with a special control circuit to enable/disable the counting (a flag FF is used for this)

\text{architecture regular\_seq\_arch of pulse\_5clk is}
\text{constant P\_WIDTH: natural := 5;}
\text{signal c\_reg, c\_next: unsigned(3 downto 0);}
\text{signal flag\_reg, flag\_next: std\_logic;}
\text{begin}

\text{-- register}
\text{process(clk, reset)}
\text{begin}
\text{if (reset = '1')} \text{then}
\text{c\_reg <= (others=>'}0'\text{);}
One-Shot Pulse Generator

flag_reg <= '0';

elsif (clk'event and clk = '1') then
    c_reg <= c_next;
    flag_reg <= flag_next;

end if;

end process;

-- next-state logic
process(c_reg, flag_reg, go, stop)
begin
    c_next <= c_reg;
    flag_next <= flag_reg;

    if (flag_reg = '0') and (go = '1') then
        flag_next <= '1';
        c_next <= (others=>'0');
    elsif (flag_reg = '1') and 
        ((c_reg = P_WIDTH-1) or (stop = '1')) then
        flag_next <= '0';

end process;
One-Shot Pulse Generator

    elsif (flag_reg = '1') then
        c_next <= c_reg + 1;
    end if;
end process;

-- output logic
pulse <= '1' when flag_reg='1' else '0';
end regular_seq_arch;

Although this implements the functionality, it is ’clumsy’ and cluttered
The flag FF functions as some sort of state register that keeps track of the current condition of the circuit

The RT methodology is the clearest
It uses two states indicating whether the counter is active or not

In the delay state, the counter is incremented if stop is ’0’ and count has not reached 5
One-Shot Pulse Generator

```vhdl
architecture fsmd_arch of pulse_5clk is
    constant P_WIDTH: natural := 5;
    type fsmd_state_type is (idle, delay);
    signal state_reg, state_next: fsmd_state_type;
    signal c_reg, c_next: unsigned(3 downto 0);
begin
    -- state and data registers
    process(clk, reset)
    begin
        if (reset = '1') then
            state_reg <= idle;
            c_reg <= (others => '0');
        end if;
        state_reg <= state_next;
        c_reg <= c_next;
    end process;
end fsmd_arch;
```
One-Shot Pulse Generator

```vhdl
elsif (clk'event and clk = '1') then
    state_reg <= state_next;
    c_reg <= c_next;
end if;
end process;

-- next-state logic & data path functional units/routing
process(state_reg, go, stop, c_reg)
begin
    pulse <= '0';
    c_next <= c_reg;
    case state_reg is
        when idle =>
            if (go = '1') then
                state_next <= delay;
            else
                state_next <= idle;
            end if;
end process;
```
One-Shot Pulse Generator

\[
c_{\text{next}} <= (\text{others=>}'0');
\]

\[
\text{when delay} => \\
\quad \text{if (stop = '1') then} \\
\qquad \text{state}_{\text{next}} <= \text{idle}; \\
\quad \text{else} \\
\qquad \text{if (c\_reg = P\_WIDTH-1) then} \\
\qquad\qquad \text{state}_{\text{next}} <= \text{idle}; \\
\qquad \text{else} \\
\qquad\qquad \text{state}_{\text{next}} <= \text{delay}; \\
\qquad\qquad \text{c}_{\text{next}} <= \text{c\_reg + 1}; \\
\qquad \text{end if}; \\
\quad \text{end if}; \\
\quad \text{pulse} <= '1';
\]

\[
\text{end case;}
\]

\[
\text{end process;}
\]

end fsmd_arch;
Programmable One-Shot Pulse Generator

To further illustrate the capability of the one-shot generator, consider a version that is programmable:

- The desired width can be programmed between 1 and 7
- The circuit enters the programming mode when both the go and stop signals are asserted
- The desired width shifted in via the go signal in the next three clock cycles

Although possible to derive this using an FSM or a regular sequential circuit, it requires a great deal of effort

See text for VHDL code and SRAM controller implementation
Greatest Common Divisor

Returns the greatest common divisor of 2 positive nums, \( \gcd(1, 10) = 1 \), \( \gcd(12, 9) = 3 \)

It is possible to compute GCD without division as follows:

\[
\gcd(a, b) = \begin{cases} 
  a & \text{if } a = b \\
  \gcd(a - b, b) & \text{if } a > b \\
  \gcd(a, b - a) & \text{if } a < b 
\end{cases}
\]

Pseudocode

\[
\begin{align*}
  a &= a_{\text{in}}; \\
  b &= b_{\text{in}}; \\
  \text{while } (a \neq b) \\
  \{ \\
  &\text{if } (b > a) \text{ then} \\
  &\quad a = a - b; \\
  &\text{else} \\
  &\quad b = b - a; \\
  \} \\
  r &= a;
\end{align*}
\]
Greatest Common Divisor

Modified pseudo algorithm with *goto* to better match ASMD

\begin{verbatim}
a = a_in;
b = b_in;
sw: if (a = b) then
goto st;
else
  if (b > a) then
    a = b;
b = a;
  end if;
a = a - b;
goto sw;
end if;
st: r = a;
\end{verbatim}
Greatest Common Divisor

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gcd is
  port(
    clk, reset: in std_logic;
    start: in std_logic;
    a_in, b_in: in std_logic_vector(7 downto 0);
    ready: out std_logic;
    r: out std_logic_vector(7 downto 0)
  );
end gcd ;

architecture slow_arch of gcd is
  type state_type is (idle, swap, sub);
  signal state_reg, state_next: state_type;
```
Greatest Common Divisor

```vhdl
signal a_reg, a_next, b_reg, b_next:
    unsigned(7 downto 0);

begin

-- state & data registers
process(clk, reset)
begin

    if (reset = '1') then
        state_reg <= idle;
        a_reg <= (others=>'0');
        b_reg <= (others=>'0');
    elsif (clk'event and clk = '1') then
        state_reg <= state_next;
        a_reg <= a_next;
        b_reg <= b_next;
    end if;

end process;
```

Greatest Common Divisor

-- next-state logic & data path functional units/routing

process (state_reg, a_reg, b_reg, start, a_in, b_in)
begin
    a_next <= a_reg;
    b_next <= b_reg;
    case state_reg is
    when idle =>
        if (start = '1') then
            a_next <= unsigned(a_in);
            b_next <= unsigned(b_in);
            state_next <= swap;
        else
            state_next <= idle;
        end if;
    when swap =>
        if (a_reg = b_reg) then
            state_next <= idle;
        end if;
end case;
end begin;
Greatest Common Divisor

```vhdl
else
    if (a_reg < b_reg) then
        a_next <= b_reg;
        b_next <= a_reg;
    end if;
    state_next <= sub;
end if;
when sub =>
    a_next <= a_reg - b_reg;
    state_next <= swap;
end case;
end process;

-- output
ready <= '1' when state_reg = idle else '0';
r <= std_logic_vector(a_reg);
end slow_arch;
```
Greatest Common Divisor

The worst case scenario is with gcd(1,2^8-1), which requires 2^8 - 1 iterations of the loop.

For a circuit with an N-bit input, run time is bound by O(2^N).

One method to speed this up is to look at the LSB to determine if the inputs are even or odd.

\[
gcd(a, b) = \begin{cases} 
    a & \text{if } a = b \\
    2 \gcd\left(\frac{a}{2}, \frac{b}{2}\right) & \text{if } a \neq b \text{ and } a, b \text{ even} \\
    \gcd\left(a, \frac{b}{2}\right) & \text{if } a \neq b \text{ and } a \text{ odd, } b \text{ even} \\
    \gcd\left(\frac{a}{2}, b\right) & \text{if } a \neq b \text{ and } a \text{ even, } b \text{ odd} \\
    \gcd(a - b, b) & \text{if } a > b \text{ and } a, b \text{ odd} \\
    \gcd(a, b - a) & \text{if } a < b \text{ and } a, b \text{ odd} 
\end{cases}
\]

Divide-by-2 is easily implemented in hardware.

What is the best method to handle 2*gcd(a/2, b/2)?

The recursive relationship suggests this may happen more than once, e.g., 12, 36 -> 2*gcd(6,18) -> 4*gcd(3,9)
Greatest Common Divisor

Best way is to count the number of times (using register $n$) that this occurs and multiply at the end by $2^n$ to get the final result.

In swap state, check LSBs of $a$ and $b$
- If $a(0) = 0$, then shift right
- Also, $n$ is incremented if both are even

If $a$ and $b$ are odd, they are compared and swapped (if necessary) - then enter sub state

An extra state, res, is added to ’restore’ the final GCD value
- Here, $a$ is shifted left repeatedly (mult. by 2) until $n$ becomes 0

Text gives VHDL code
Greatest Common Divisor

How much have we improved performance by?

Assume the width of input operands is $N$ bits

The algorithm gradually reduces the values in $a_{\text{reg}}$ and $b_{\text{reg}}$ until they are equal

In the worst case, there are $2N$ bits to process

If one value is even, the LSB is shifted out and the number of bits is reduced by 1

If both values are odd, a subtraction is performed and the difference is even -- reducing the number of bits in the next iteration by 1.

Therefore, under the most pessimistic scenario, the $2N$ bits can be processed in $2 \times 2N$ iterations

Thus, we have reduced the complexity from $O(n^2)$ to $O(n)$

Text covers further improvements that uses extra hardware to replace bit-by-bit ops
Greatest Common Divisor

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gcd is
  port(
    clk, reset: in std_logic;
    start: in std_logic;
    a_in, b_in: in std_logic_vector(7 downto 0);
    ready: out std_logic;
    r: out std_logic_vector(7 downto 0)
  );
end gcd ;

architecture fast_arch of gcd is
  type state_type is (idle, swap, sub, res);
  signal state_reg, state_next: state_type;
  signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
```

Greatest Common Divisor

```vhdl
signal n_reg, n_next: unsigned(2 downto 0);
begin

-- state & data registers
process(clk, reset)
begin
  if (reset = '1') then
    state_reg <= idle;
    a_reg <= (others => '0');
    b_reg <= (others => '0');
    n_reg <= (others => '0');
  elsif (clk'event and clk='1') then
    state_reg <= state_next;
    a_reg <= a_next;
    b_reg <= b_next;
    n_reg <= n_next;
  end if;
end process;
```
Greatest Common Divisor

-- next-state logic & data path functional units/routing

process(state_reg, a_reg, b_reg, n_reg, start, a_in, b_in, n_next)
begin
a_next <= a_reg;
b_next <= b_reg;
n_next <= n_reg;
case state_reg is
when idle =>
  if (start = '1') then
    a_next <= unsigned(a_in);
b_next <= unsigned(b_in);
n_next <= (others => '0');
  state_next <= swap;
  else
    state_next <= idle;
  end if;
end case;
end begin;
end process;
Greatest Common Divisor

when swap =>
    if (a_reg = b_reg) then
        if (n_reg = 0) then
            state_next <= idle;
        else
            state_next <= res;
        end if;
    else
        if (a_reg(0) = '0') then -- a even
            a_next <= '0' & a_reg(7 downto 1);
        end if;
        if (b_reg(0) = '0') then -- both ev.
            b_next <= '0' & b_reg(7 downto 1);
            n_next <= n_reg + 1;
        end if;
        state_next <= swap;
    else -- a odd
    end if;
Greatest Common Divisor

if (b_reg(0) = '0') then  -- b even
    b_next <= '0' & b_reg(7 downto 1);
    state_next <= swap;
else  -- both a_reg and b_reg odd
    if (a_reg < b_reg) then
        a_next <= b_reg;
        b_next <= a_reg;
    end if;
    state_next <= sub;
end if;
end if;

when sub =>
    a_next <= a_reg - b_reg;
    state_next <= swap;
Greatest Common Divisor

    when res =>
        a_next <= a_reg(6 downto 0) & '0';
        n_next <= n_reg - 1;
        if (n_next = 0) then
            state_next <= idle;
        else
            state_next <= res;
        end if;
    end case;
end process;

--output
ready <= '1' when state_reg = idle else '0';
r <= std_logic_vector(a_reg);
end fast_arch;

(See text for UART receiver example)
Square Root Approximation Circuit

UART is an example of a control-oriented application -- here we look at an example of a data-oriented application (computation-intensive)

Although data-oriented applications can be implemented using combinational resources, in practice, there are limits and sharing must be used.

For the square root approx. circuit, we use simple adder-type components to obtain an approximate value for:

\[ \sqrt{a^2 + b^2} \approx \max(((x - 0.125x) + 0.5y), x) \]

where \( x = \max(|a|, |b|) \) and \( y = \min(|a|, |b|) \)

Here, \( a \) and \( b \) are signed integers.

The constants and operations, 0.125*\( x \) and 0.5*\( y \) corresponds to shift right by 3 bits and 1 bit, respectively.

Therefore, we don’t need a multiplication circuit.
Square Root Approximation Circuit

Pseudocode:

\[
\begin{align*}
    a &= a_{\text{in}}; \\
    b &= b_{\text{in}}; \\
    t1 &= \text{abs}(a); \\
    t2 &= \text{abs}(b); \\
    x &= \max(t1, t2); \\
    y &= \min(t1, t2); \\
    t3 &= x \times 0.125; \\
    t4 &= y \times 0.5; \\
    t5 &= x - t3; \\
    t6 &= t4 + t5; \\
    t7 &= \max(t6, x); \\
    r &= t7;
\end{align*}
\]

Here, we intentionally avoided the reuse of the same variable name on the left-hand statements to assist with conversion to VHDL.

This can be translated directly as a data-flow implementation (no control structure).
Square Root Approximation Circuit

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sqrt is
  port(
    a_in, b_in: in std_logic_vector(7 downto 0);
    r: out std_logic_vector(8 downto 0)
  );
end sqrt;

architecture comb_arch of sqrt is
  constant WIDTH: natural := 8;
signal a, b, x, y: signed(WIDTH downto 0);
signal t1, t2, t3, t4, t5, t6, t7:
  signed(WIDTH downto 0);
begin
Square Root Approximation Circuit

\[
\begin{align*}
    a &= \text{signed}(a_{\text{in}}(\text{WIDTH}-1) \ & a_{\text{in}}); \\
    b &= \text{signed}(b_{\text{in}}(\text{WIDTH}-1) \ & b_{\text{in}}); \\
    t1 &= a \text{ when } a > 0 \text{ else 0} - a; \\
    t2 &= b \text{ when } b > 0 \text{ else 0} - b; \\
    x &= t1 \text{ when } t1 - t2 > 0 \text{ else } t2; \\
    y &= t2 \text{ when } t1 - t2 > 0 \text{ else } t1; \\
    t3 &= \text{"000"} \ & x(\text{WIDTH downto 3}); \\
    t4 &= \text{"0"} \ & y(\text{WIDTH downto 1}); \\
    t5 &= x - t3; \\
    t6 &= t4 + t5; \\
    t7 &= t6 \text{ when } t6 - x > 0 \text{ else } x; \\
    r &= \text{std\_logic\_vector}(t7); \\
\end{align*}
\]

end comb_arch;
Square Root Approximation Circuit

This implementation requires one adder and six subtractors

These operations are **not** mutually exclusive and therefore sharing is **NOT** possible

The code contains only concurrent signal assignment statements

The order is not important

Sequence of execution is embedded in the flow of data

To examine the dependency and movement of data, we use a **data flow graph**

- **Nodes** (circle) represent an operation
- **Arcs** represent input and output variables

The data flow graph illustrates that the algorithm has only a *limited degree of parallelism* b/c at most two operations can be executed concurrently (see next slide)

The seven arithmetic components of the previous VHDL code can**NOT** significantly increase performance, and therefore, these resources are wasted

RT methodology can share resources and is a better alternative in this case
Square Root Approximation Circuit

Tasks in converting a dataflow graph to an ASMD chart

- Scheduling: when a function (circle) can start execution
- Binding: which functional unit is assigned to perform the operation

An important design constraint is the number of functional units assigned to perform the operation

Allocate minimal number to reduce circuit size
Allocate maximum number to exploit FULL parallelism
Find a mid-point that trade-offs size and performance

Finding an optimal schedule involves sophisticated algorithms and is difficult
Square Root Approximation Circuit

In square root algorithm, all operations can be performed by a modified addition unit.

Also, no function unit is needed for shifting (should not be scheduled)

Scheduling with 2 functional units

Note that *0.125 and *0.5 are removed

The dataflow graph is divided into 5 time intervals (states)

Left graph gives one option which binds two ops on left to one unit and 5 ops on right to second

Right graph gives an alternative
Square Root Approximation Circuit

This schedule uses one functional unit and requires TWO extra time intervals to complete the operation.

Once scheduling and binding are complete, the data-flow graph can be transformed into an ASMD chart.

Each time interval represents a **state** in the chart.

Also, a **register** is needed when a signal is passed through a state boundary.

The variables of the dataflow graph are mapped into the registers of the ASMD chart.

The ASMD chart (next slide) shows two operations are performed in the \( s1 \) and \( s2 \) states. 

*start* and *ready* and state *idle* are added to interface circuit with external system.
Square Root Approximation Circuit

Optimizations can be performed to reduce number of registers and simplify the routing

Instead of creating a new reg. for each variable, we can reuse if its value is no longer needed

This corresponds to *renaming* the variables in the dataflow graph

Here, we can use three registers to do the whole dataflow graph

Here, we can use *three* registers
- Use \( r1 \) to replace \( a, t1 \) and \( y \)
- Use \( r2 \) to replace \( b, t2 \) and \( x \)
- Use \( r3 \) to replace \( t5, t6 \) and \( t7 \)
Square Root Approximation Circuit

Revised ASMD chart

Registers renaming shown earlier as registers in parenthesis
Square Root Approximation Circuit

To ensure proper sharing, the two functional units are isolated from one another and coded in two segments

One unit for \textit{abs} and \textit{min}

One unit for \textit{abs}, \textit{min}, - and +

\begin{verbatim}
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sqrt is
    port(
        clk, reset: in std_logic;
        start: in std_logic;
        a_in, b_in: in std_logic_vector(7 downto 0);
        ready: out std_logic;
        r: out std_logic_vector(8 downto 0)
    );
end sqrt;
\end{verbatim}
Square Root Approximation Circuit

```vhdl
architecture seq_arch of sqrt is
  constant WIDTH: integer := 8;
  type state_type is (idle, s1, s2, s3, s4, s5);
  signal state_reg, state_next: state_type;
  signal r1_reg, r2_reg, r3_reg:
    signed(WIDTH downto 0);
  signal r1_next, r2_next, r3_next:
    signed(WIDTH downto 0);
  signal sub_op0, sub_op1, diff, au1_out:
    signed(WIDTH downto 0);
  signal add_op0, add_op1, sum, au2_out:
    signed(WIDTH downto 0);
  signal add_carry: integer;
begin
```
Square Root Approximation Circuit

-- state & data registers

process(clk, reset)
begin
if (reset = '1') then
    state_reg <= idle;
    r1_reg <= (others => '0');
    r2_reg <= (others => '0');
    r3_reg <= (others => '0');
elsif (clk'event and clk = '1') then
    state_reg <= state_next;
    r1_reg <= r1_next;
    r2_reg <= r2_next;
    r3_reg <= r3_next;
end if;
end process;
Square Root Approximation Circuit

-- next-state logic and data path routing

process (start, state_reg, r1_reg, r2_reg, r3_reg, a_in, b_in, au1_out, au2_out)

begin
r1_next <= r1_reg;
r2_next <= r2_reg;
r3_next <= r3_reg;
ready <= '0';
case state_reg is
when idle =>
if (start = '1') then
r1_next <= signed(a_in(WIDTH-1) & a_in);
r2_next <= signed(b_in(WIDTH-1) & b_in);
state_next <= s1;
else
state_next <= idle;
end if;
end if;
ready <= '1';
Square Root Approximation Circuit

\texttt{when s1 =>}
\begin{align*}
&\texttt{r1\_next} \leftarrow \texttt{au1\_out}; \quad \texttt{t1}=|a| \\
&\texttt{r2\_next} \leftarrow \texttt{au2\_out}; \quad \texttt{t2}=|b| \\
&\texttt{state\_next} \leftarrow s2;
\end{align*}

\texttt{when s2 =>}
\begin{align*}
&\texttt{r1\_next} \leftarrow \texttt{au1\_out}; \quad \texttt{y}=\min(t1,t2) \\
&\texttt{r2\_next} \leftarrow \texttt{au2\_out}; \quad \texttt{x}=\max(t1,t2) \\
&\texttt{state\_next} \leftarrow s3;
\end{align*}

\texttt{when s3 =>}
\begin{align*}
&\texttt{r3\_next} \leftarrow \texttt{au2\_out}; \quad \texttt{t5}=x-0.125x \\
&\texttt{state\_next} \leftarrow s4;
\end{align*}

\texttt{when s4 =>}
\begin{align*}
&\texttt{r3\_next} \leftarrow \texttt{au2\_out}; \quad \texttt{t6}=0.5y+t5 \\
&\texttt{state\_next} \leftarrow s5;
\end{align*}
Square Root Approximation Circuit

when s5 =>
    r3_next <= au2_out; -- t7=max(t6,x)
    state_next <= idle;
end case;
end process;

-- arithmetic unit 1
-- subtractor
    diff <= sub_op0 - sub_op1;

-- input routing
begin
    case state_reg is
        when s1 => -- 0-a
            sub_op0 <= (others=>'0');
            sub_op1 <= r1_reg; -- a
Square Root Approximation Circuit

```vhdl
when others =>  -- s2: t2-t1
    sub_op0 <= r2_reg;  -- t2
    sub_op1 <= r1_reg;  -- t1

end case;
end process;

-- output routing
process(state_reg, r1_reg, r2_reg, diff)
begin
  case state_reg is
  when s1 => -- |a|
    if (diff(WIDTH) = '0') then  -- (0-a)>0
      au1_out <= diff;  -- -a
    else
      au1_out <= r1_reg;  -- a
    end if;
  end case;
end process;
```

Square Root Approximation Circuit

```vhdl
when others => -- s2: min(a,b)
    if (diff(WIDTH) = '0') then --(t2-t1)>0
        au1_out <= r1_reg; -- t1
    else
        au1_out <= r2_reg; -- t2
    end if;
end case;
end process;

-- arithmetic unit 2
-- adder
sum <= add_op0 + add_op1 + add_carry;

-- input routing
process(state_reg, r1_reg, r2_reg, r3_reg)
begin
```
Square Root Approximation Circuit

case state_reg is

when s1 => -- 0-b
    add_op0 <= (others=>'0'); --0
    add_op1 <= not r2_reg; -- not b
    add_carry <= 1;

when s2 => -- t1-t2
    add_op0 <= r1_reg; --t1
    add_op1 <= not r2_reg; --not t2
    add_carry <= 1;

when s3 => -- -- x-0.125x
    add_op0 <= r2_reg; --x
    add_op1 <= not("000" & r2_reg(WIDTH downto 3));
    add_carry <= 1;
Square Root Approximation Circuit

```
when s4 => -- 0.5*y + t5
    add_op0 <= "0" & r1_reg(WIDTH downto 1);
    add_op1 <= r3_reg;
    add_carry <= 0;

when others => -- t6 - x
    add_op0 <= r3_reg; --t1
    add_op1 <= not r2_reg; --not x
    add_carry <= 1;
end case;
end process;

-- output routing
process(state_reg, r1_reg, r2_reg, r3_reg, sum)
begin
    case state_reg is
    when s1 => -- |b|
```
Square Root Approximation Circuit

if (sum(WIDTH) = '0') then  -- (0-b)>0
  au2_out <= sum;  -- -b
else
  au2_out <= r2_reg;  -- b
end if;

when s2 =>
  if (sum(WIDTH) = '0') then
    au2_out <= r1_reg;
  else
    au2_out <= r2_reg;
  end if;
when s3 | s4 =>  -- +,-
  au2_out <= sum;
Square Root Approximation Circuit

when others => -- s5
    if (sum(WIDTH) = '0') then
        au2_out <= r3_reg;
    else
        au2_out <= r2_reg;
    end if;
end case;
end process;

-- output
    r <= std_logic_vector(r3_reg);
end seq_arch;
High Level Synthesis

Deriving an optimal RT design for data-oriented applications is not a simple task

This task is known as high-level synthesis, also called behavioral synthesis (misleadingly)

Synthesis starts with a set of constraints and an abstract VHDL description similar to the algorithm’s pseudocode

High-level synthesis software converts the initial description into an FSMD and automatically derives code for the control and data path

The transformation is basically modeled by the last two VHDL code fragments given above

Objective is to find an optimal schedule and binding to minimize the required hardware resources, to maximize performance or to obtain the best trade-off for a given constraint

Mainly used for computation intensive applications, e.g., DSP