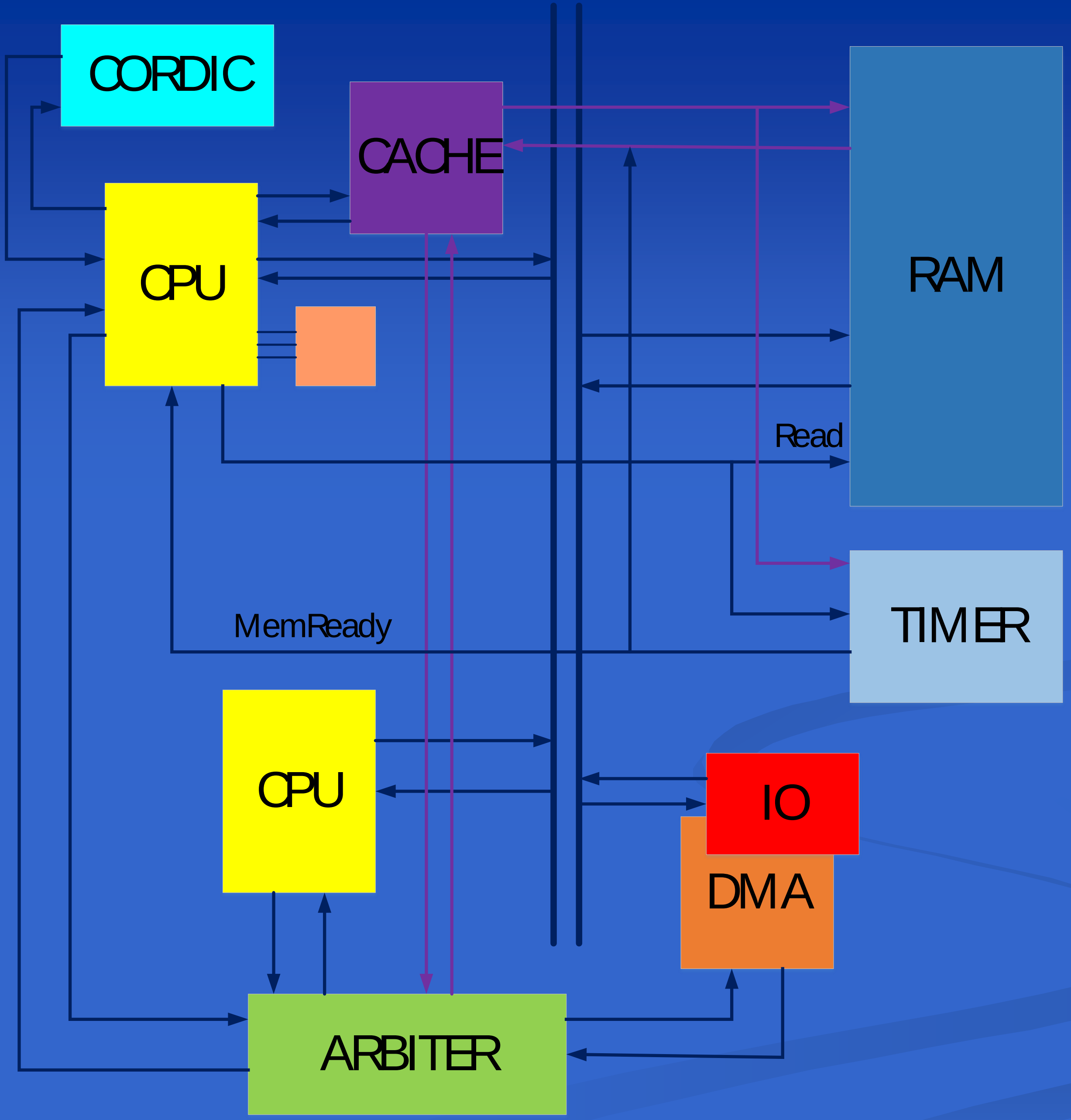
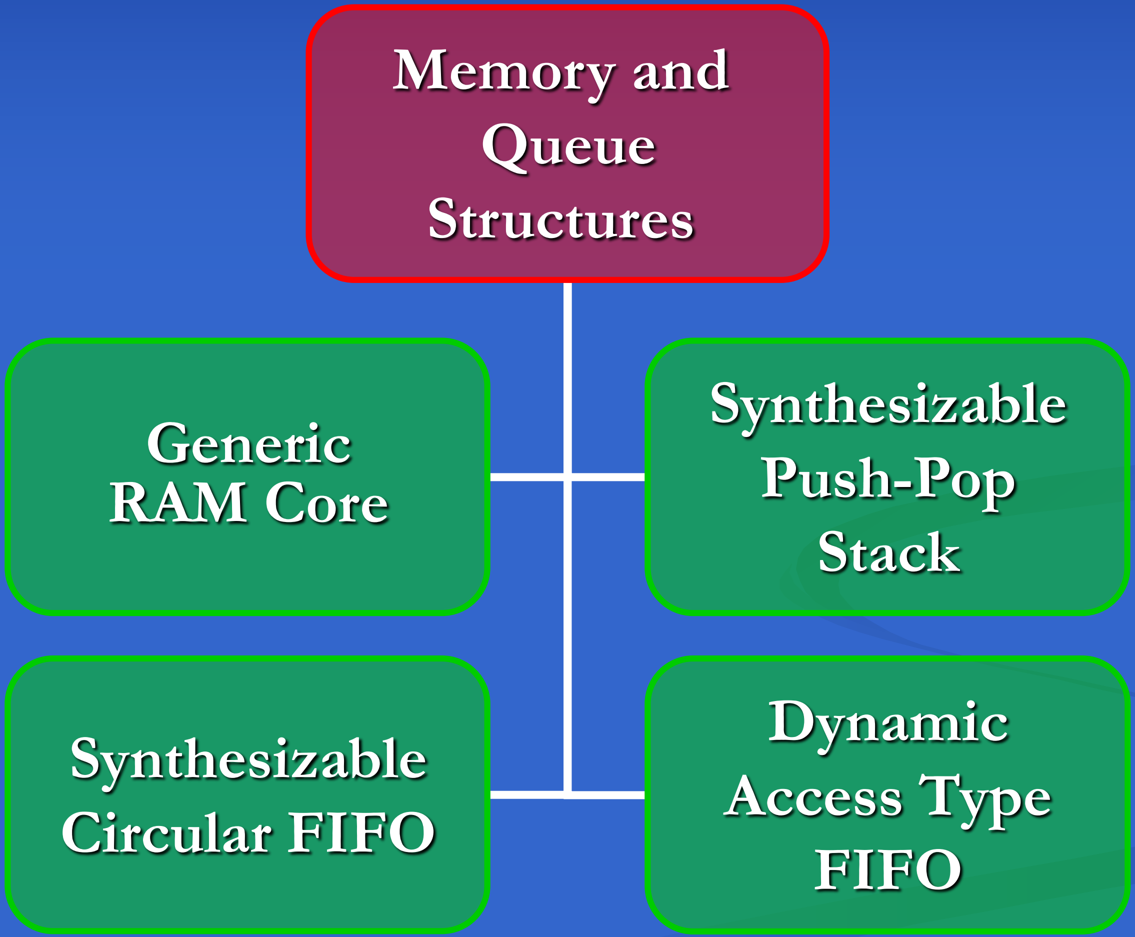
Chapter 8
Hardware Cores and Models

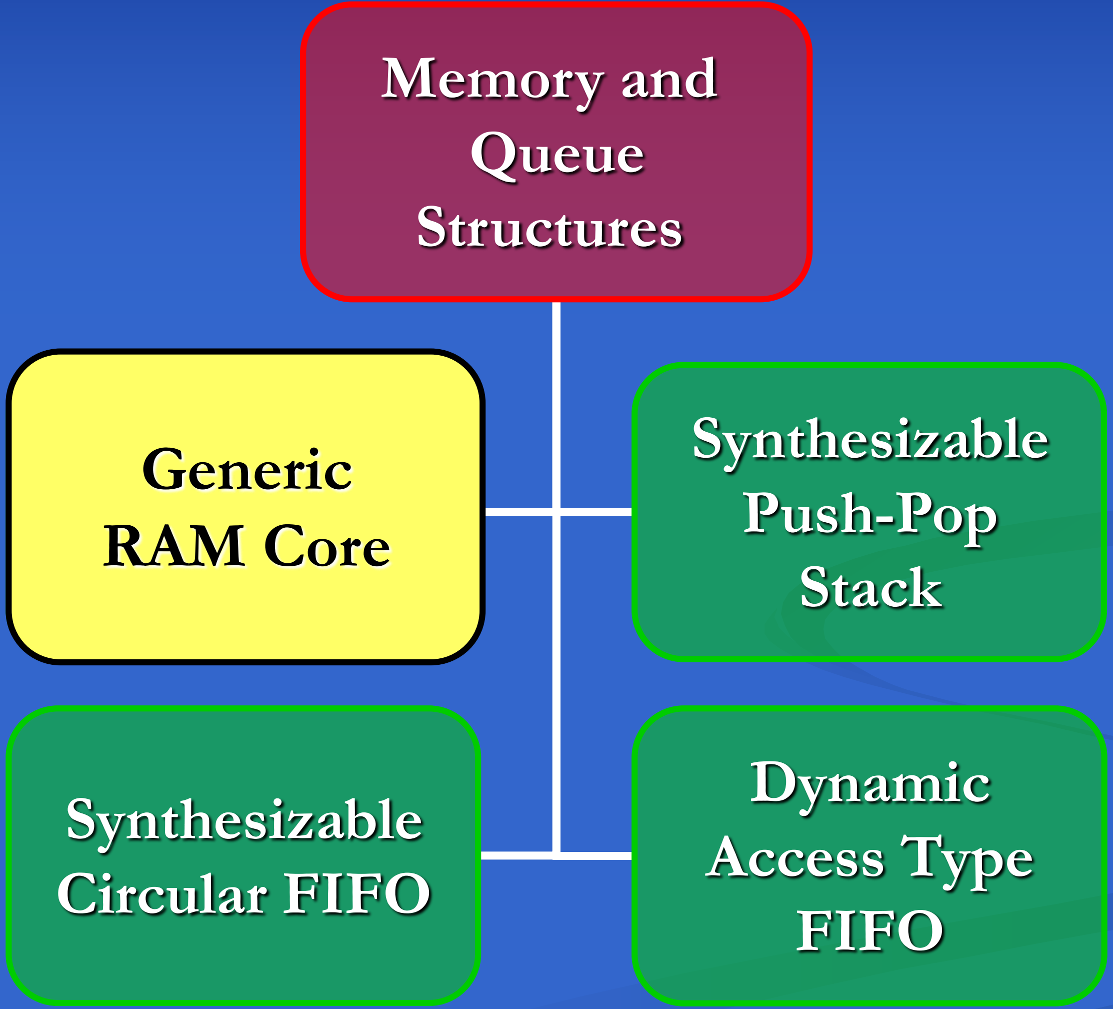
Hardware Cores and Models


CORDIC
CACHE
CPU
RAM
Read
TIMER
MemReady
CPU
IO
DMA
ARBITER

Memory and Queue Structures


Memory and Queue Structures
Generic RAM Core
Synthesizable Push-Pop Stack
Synthesizable Circular FIFO
Dynamic Access Type FIFO

Generic RAM Core


Memory and Queue Structures
Generic RAM Core
Synthesizable Push-Pop Stack
Synthesizable Circular FIFO
Dynamic Access Type FIFO

Generic RAM Core

```
PROCEDURE init_mem (VARIABLE memory: OUT mem;
                    CONSTANT datafile: STRING) IS
   FILE stddata : TEXT;
   VARIABLE l : LINE;
   VARIABLE data : std_logic_vector(memory'RANGE(2));
BEGIN
   FILE_OPEN (stddata, datafile, READ_MODE);
   FOR i IN memory'RANGE(1) LOOP
      READLINE (stddata, l); READ (l, data);
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         memory (i,j) := data(j);
      END LOOP;
   END LOOP;
END PROCEDURE init_mem;
```

- TEXTIO Based Memory init and dump Procedure

Generic RAM Core

```
PROCEDURE dump_mem (VARIABLE memory: IN mem;
                    CONSTANT datafile: STRING) IS
   FILE stddata : TEXT;
   VARIABLE stdvalue : std_logic;
   VARIABLE l : LINE;
BEGIN
   FILE_OPEN (stddata, datafile, WRITE_MODE);
   FOR i IN memory'RANGE(1) LOOP
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         stdvalue := memory (i, j);
         WRITE (l, stdvalue);
      END LOOP;
      WRITELINE (stddata, l);
    END LOOP;
END PROCEDURE dump_mem;
```

- TEXTIO Based Memory init and dump Procedure (Continued)

Generic RAM Core

```
USE IEEE.std_logic_TEXTIO.ALL;
ENTITY std_logic_ram IS
   PORT (address, datain : IN std_logic_vector;
         dataout : OUT std_logic_vector;
         cs, rwbar : IN std_logic; opr : IN BOOLEAN);
END ENTITY std_logic_ram;
--
ARCHITECTURE behavioral OF std_logic_ram IS
   TYPE mem IS ARRAY (NATURAL RANGE <>,
                      NATURAL RANGE <>) of std_logic;
BEGIN
   . . . . . . . . . . . . . . .
   . . . . . . . . . . . . . . .
END ARCHITECTURE;
```

- Std_logic Unconstrained Memory

Generic RAM Core

```
PROCESS
      CONSTANT memsize : INTEGER := 2**address'LENGTH;
      VARIABLE memory : mem (0 TO memsize-1,
                                datain'RANGE);
   BEGIN
      id: IF opr'EVENT THEN
         IF opr=TRUE THEN init_mem (memory, "memdata.dat");
         ELSE dump_mem (memory, "memdump.dat"); END IF;
      END IF;
      wr: IF cs = '1' THEN
         IF rwbar = '0' THEN -- Writing
            FOR i IN dataout'RANGE LOOP
               memory(conv_integer(address),i):=datain (i);
            END LOOP;
         . . . . . . . . . . . . . .
         . . . . . . . . . . . . . .
```

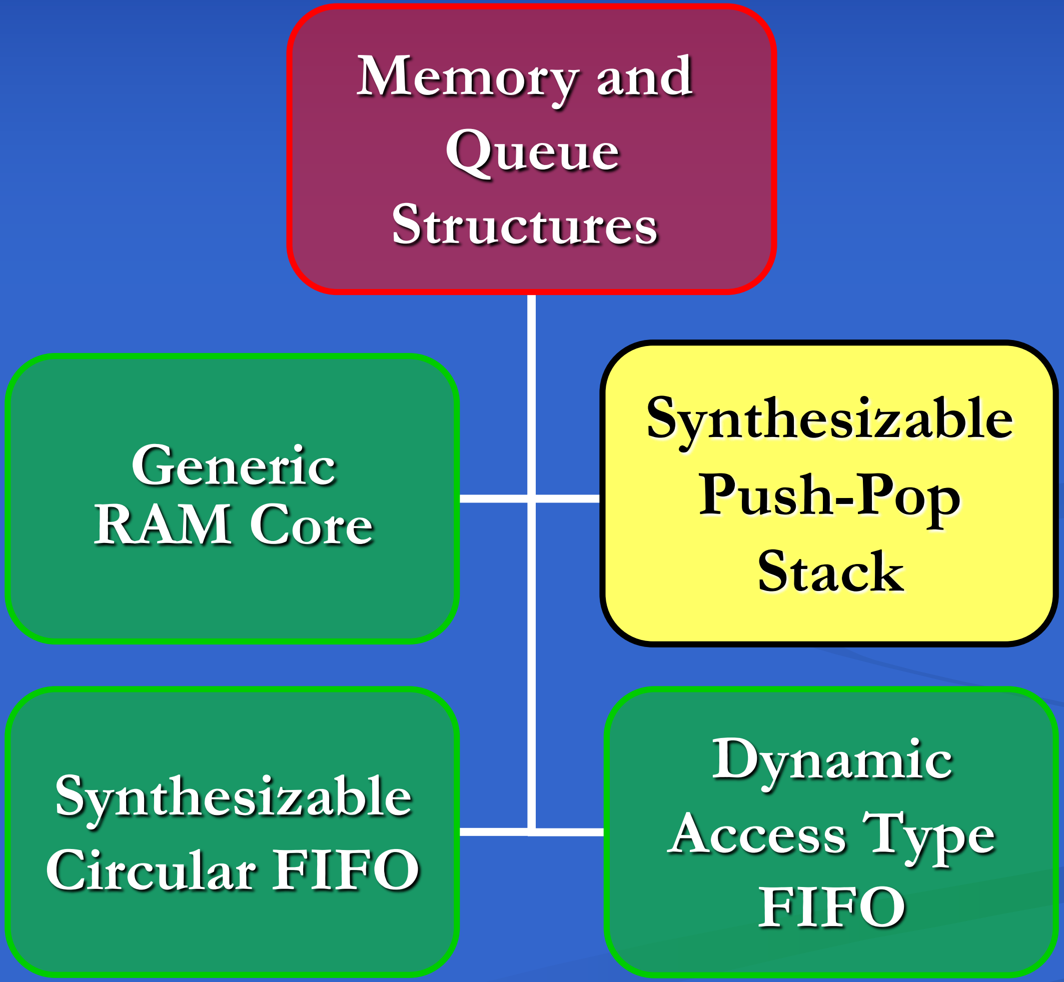
- Std_logic Unconstrained Memory (Continued)

Generic RAM Core

```
PROCESS
      CONSTANT memsize : INTEGER := 2**address'LENGTH;
      VARIABLE memory : mem (0 TO memsize-1,
                                   datain'RANGE);
   BEGIN
            . . . . . . . . . . . . . .
            ELSE                      -- Reading
               FOR i IN datain'RANGE LOOP
                  dataout(i)<=memory(conv_integer(address),i);
               END LOOP;
            END IF;
      END IF;
      WAIT ON cs, rwbar, address, datain, opr;
   END PROCESS;
```

- Std_logic Unconstrained Memory (Continued)

Synthesizable Push-Pop Stack


Memory and Queue Structures
Generic RAM Core
Synthesizable Push-Pop Stack
Synthesizable Circular FIFO
Dynamic Access Type FIFO

Synthesizable Push-Pop Stack

```
ENTITY stack IS
   GENERIC ( max: std_logic_vector := "101111");
   PORT (STin : IN std_logic_vector;
         clk, push, pop : IN std_logic;
         opr : IN BOOLEAN;
         STout : OUT std_logic_vector;
         empty, full : OUT std_logic);
END ENTITY stack;
--
```

- Stack Controller Outline

Synthesizable Push-Pop Stack

```
ARCHITECTURE behavioral OF stack IS
   SIGNAL ramin, ramout : std_logic_vector (STin'RANGE);
   SIGNAL ramaddr, pntr : std_logic_vector (max'RANGE)
                        := (OTHERS => '0');
   SIGNAL cs, rwbar, full_temp : std_logic:= '0';
   SIGNAL empty_temp : std_logic:= '1';

BEGIN
   -- UPDATING PNTR
   -- POP/PUSH
   -- INSTANTIATE MEMORY
   -- HANDLING EMPTY AND FULL
END ARCHITECTURE behavioral;
```

- Stack Controller Outline (Continued)

Synthesizable Push-Pop Stack

```
-- UPDATING PNTR
Update_pntr: PROCESS (clk)
BEGIN
   IF (clk = '1' AND clk'EVENT) THEN
      IF pop = '1' THEN
         IF empty_temp /= '1' THEN
            pntr <= pntr - 1;
         END IF;
      ELSIF push = '1' THEN
         IF full_temp /= '1' THEN
            pntr <= pntr + 1;
         END IF;
      END IF;
   END IF;
END PROCESS;
```

- Stack Pointer Update

Synthesizable Push-Pop Stack

```
-- POP/PUSH
pop_push: PROCESS (pop, push ,STin, ramout, pntr)
BEGIN
   ramaddr <= (OTHERS => '0');
   cs <= '0';
   rwbar <= '1';
   ramin <= (OTHERS => '0');
   STout <= (STin'RANGE => '0');
   IF (pop = '1' AND empty_temp = '0') THEN
      ramaddr <= pntr - 1;
      cs <= '1';
      rwbar <= '1';
      STout <= ramout;
   . . .
```

- *Pop_push* Process

Synthesizable Push-Pop Stack

```
-- POP/PUSH
pop_push: PROCESS (pop, push ,STin, ramout, pntr)
BEGIN
   . . . . . . . . . . . . . .
   . . . . . . . . . . . . . .
   ELSIF (push = '1' AND full_temp = '0') THEN
      ramaddr <= pntr;
      cs <= '1';
      rwbar <= '0';
      ramin <= STin;
   END IF;
END PROCESS pop_push;
```

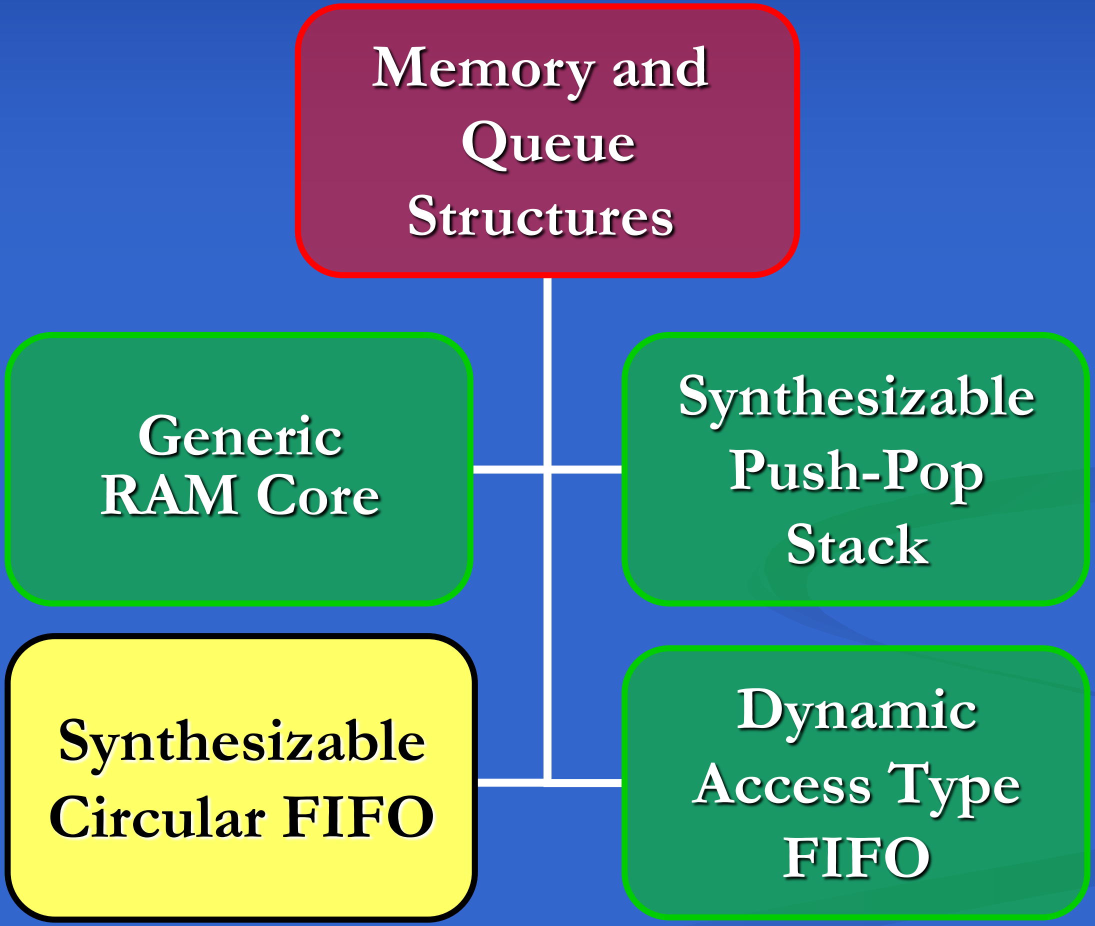
- *Pop_push* Process (Continued)

Synthesizable Push-Pop Stack

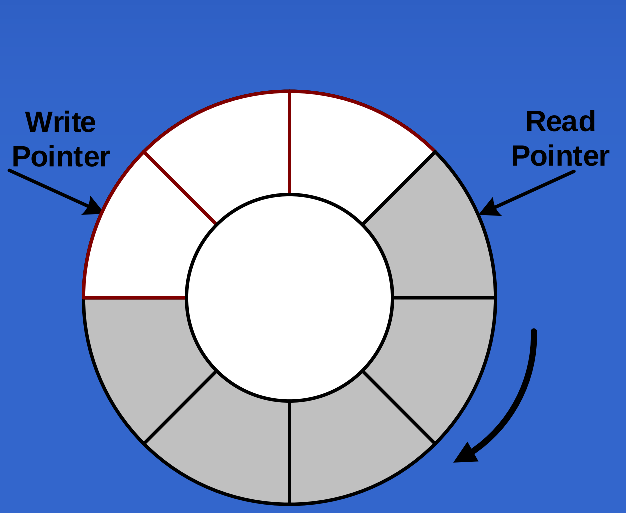
```
-- INSTANTIATE MEMORY
   UU1: ENTITY WORK.std_logic_ram (behavioral)
   PORT MAP (ramaddr, ramin, ramout, cs, rwbar, opr);

-- HANDLING EMPTY AND FULL
   empty_temp <= '1' WHEN (pntr = (pntr'RANGE => '0')) ELSE '0';
   full_temp <= '1' WHEN (pntr = max) ELSE '0';

   empty <= empty_temp;
   full <= full_temp;
```

- RAM Instantiation and *empty* and *full* Flags

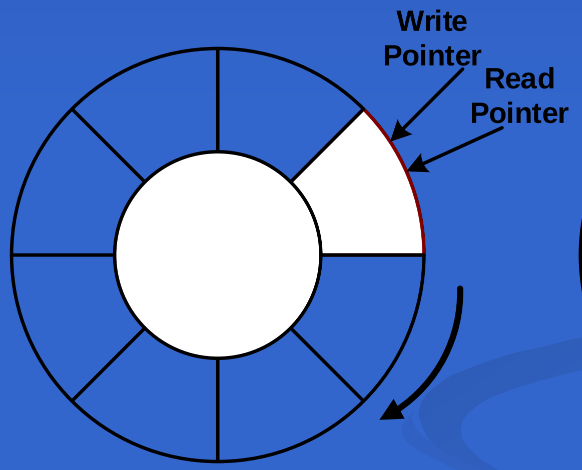
Synthesizable Circular FIFO


Memory and Queue Structures
Generic RAM Core
Synthesizable Push-Pop Stack
Synthesizable Circular FIFO
Dynamic Access Type FIFO

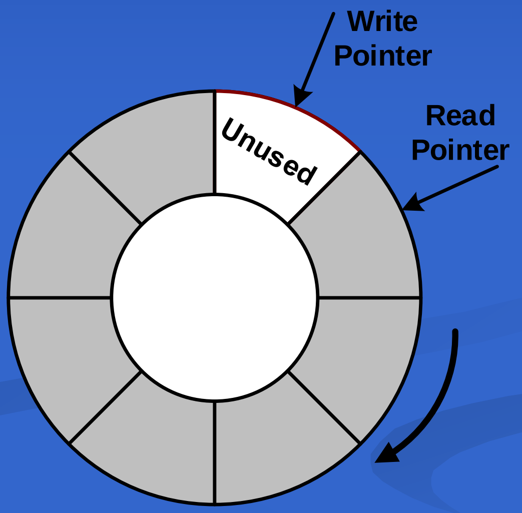
Synthesizable Circular FIFO



Can read or write



Empty



Full

- Circular FIFO

Synthesizable Circular FIFO

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
ENTITY fifo_unconst IS
   GENERIC (fifo_size : std_logic_vector := "1000");
   PORT (data_in : IN std_logic_vector;
         clk : IN std_logic;
         rst, rd, wr : IN std_logic;
         empty, full : OUT std_logic;
         data_out : OUT std_logic_vector);
END ENTITY ;
--
```

- FIFO VHDL Code Outline

Synthesizable Circular FIFO

```
ARCHITECTURE procedural OF fifo_unconst IS

   CONSTANT fsz : INTEGER := conv_integer (fifo_size);
   CONSTANT asz : INTEGER := fifo_size'LENGTH - 1;
   CONSTANT wsz : INTEGER := data_in'LENGTH; --word_size;
   TYPE memory IS ARRAY (NATURAL RANGE <>) OF std_logic_vector (wsz-1 DOWNTO 0);
   SIGNAL fifo_ram : memory (0 TO fsz-1);
   SIGNAL rd_ptr, wr_ptr:std_logic_vector(asz-1 DOWNTO 0) := (OTHERS => '0');
   SIGNAL full_temp, empty_temp : std_logic;
BEGIN

END ARCHITECTURE;
```

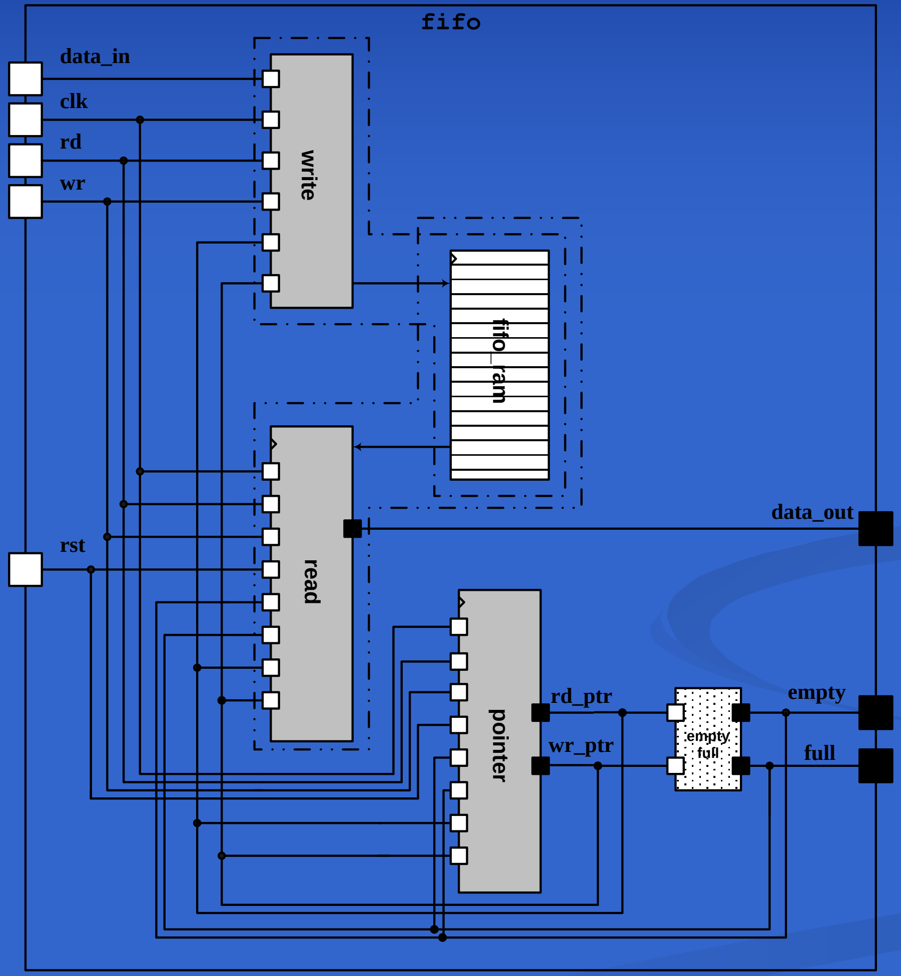
- **FIFO VHDL Code Outline (Continued)**

Synthesizable Circular FIFO

```
ARCHITECTURE procedural OF fifo_unconst IS
   . . . . . . . . . . . . . .
   . . . . . . . . . . . . . .
BEGIN
-- WRITE
-- READ
-- POINTER
   empty_temp <= '1' WHEN ( rd_ptr=wr_ptr) ELSE '0';
   full_temp <= '1' WHEN ( rd_ptr=wr_ptr + 1) ELSE '0';
   empty <= empty_temp;
   full <= full_temp;
END ARCHITECTURE;
```

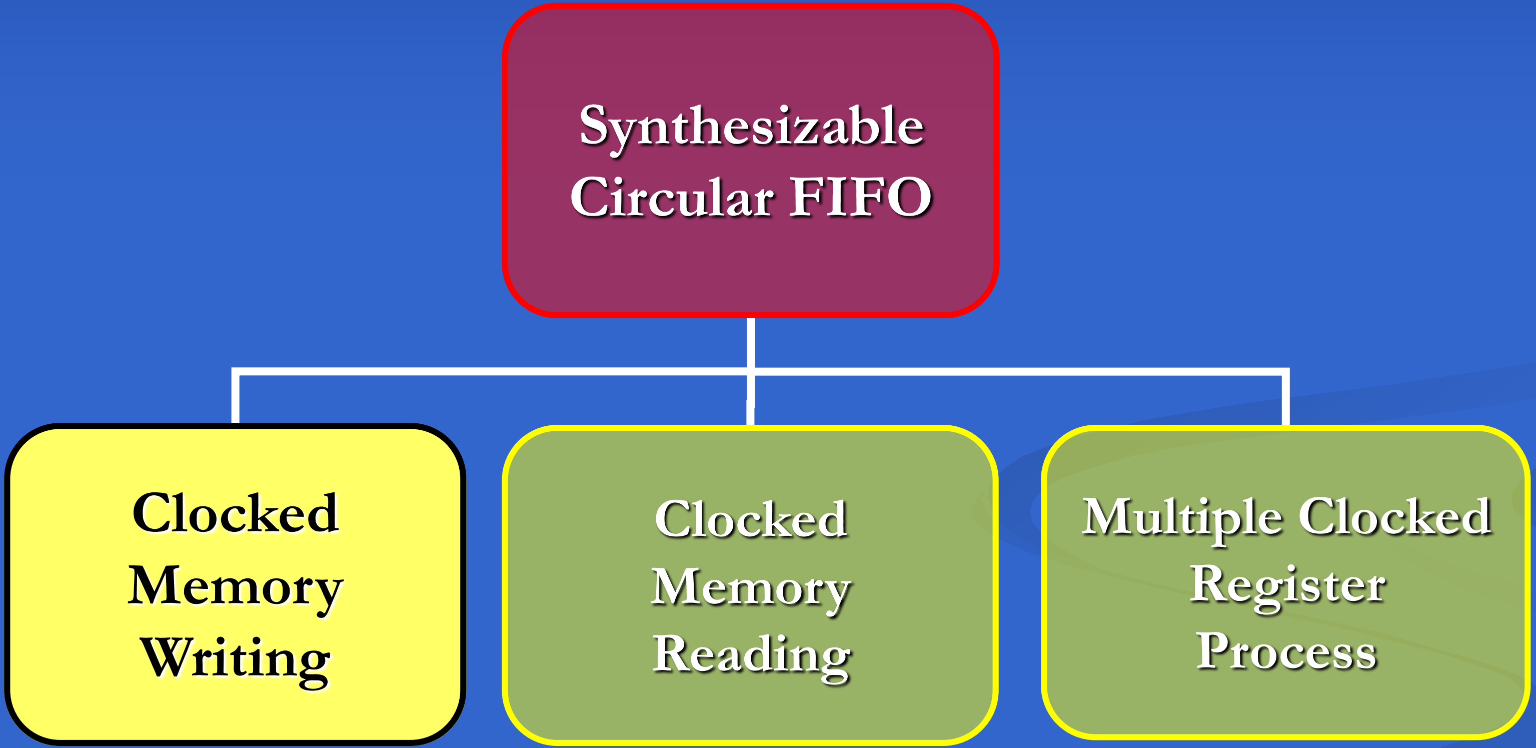
- **FIFO VHDL Code Outline (Continued)**

Synthesizable Circular FIFO



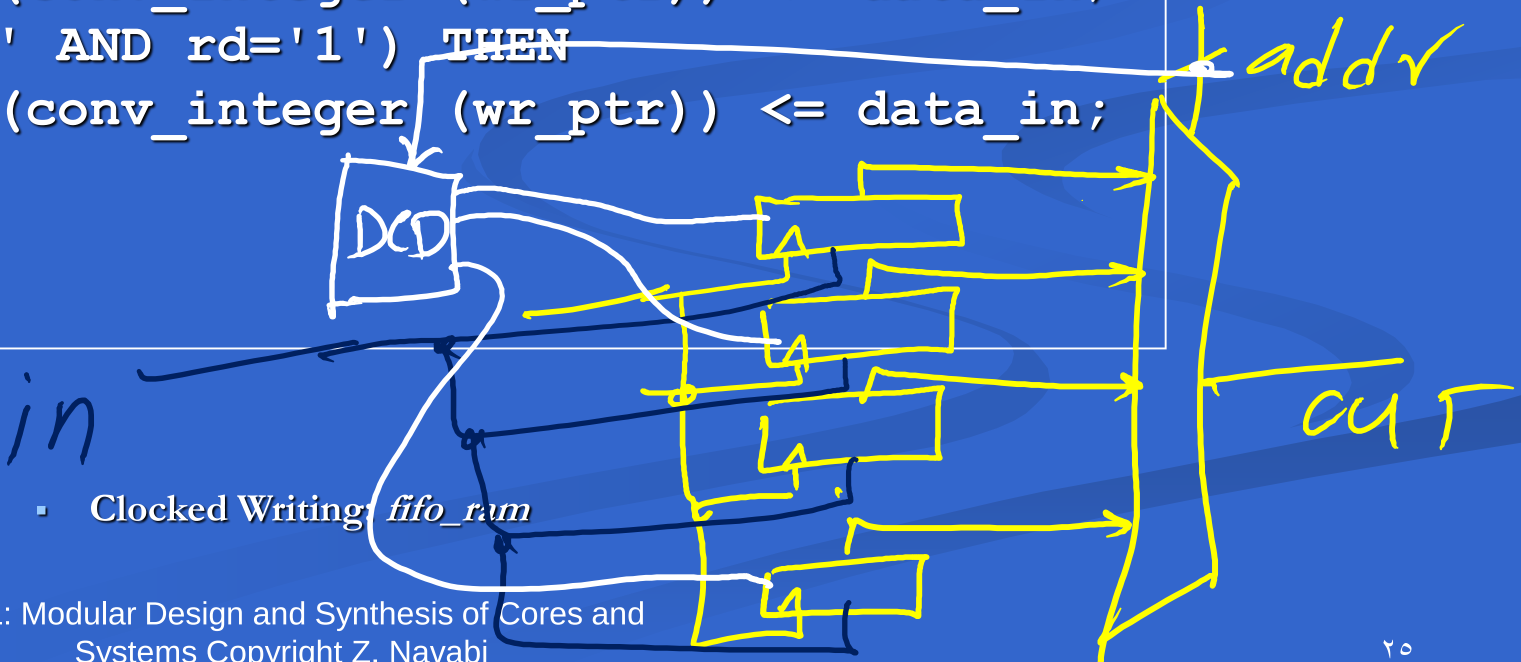
- FIFO Block Diagram

Synthesizable Circular FIFO


Synthesizable Circular FIFO
Clocked Memory Writing
Clocked Memory Reading
Multiple Clocked Register Process

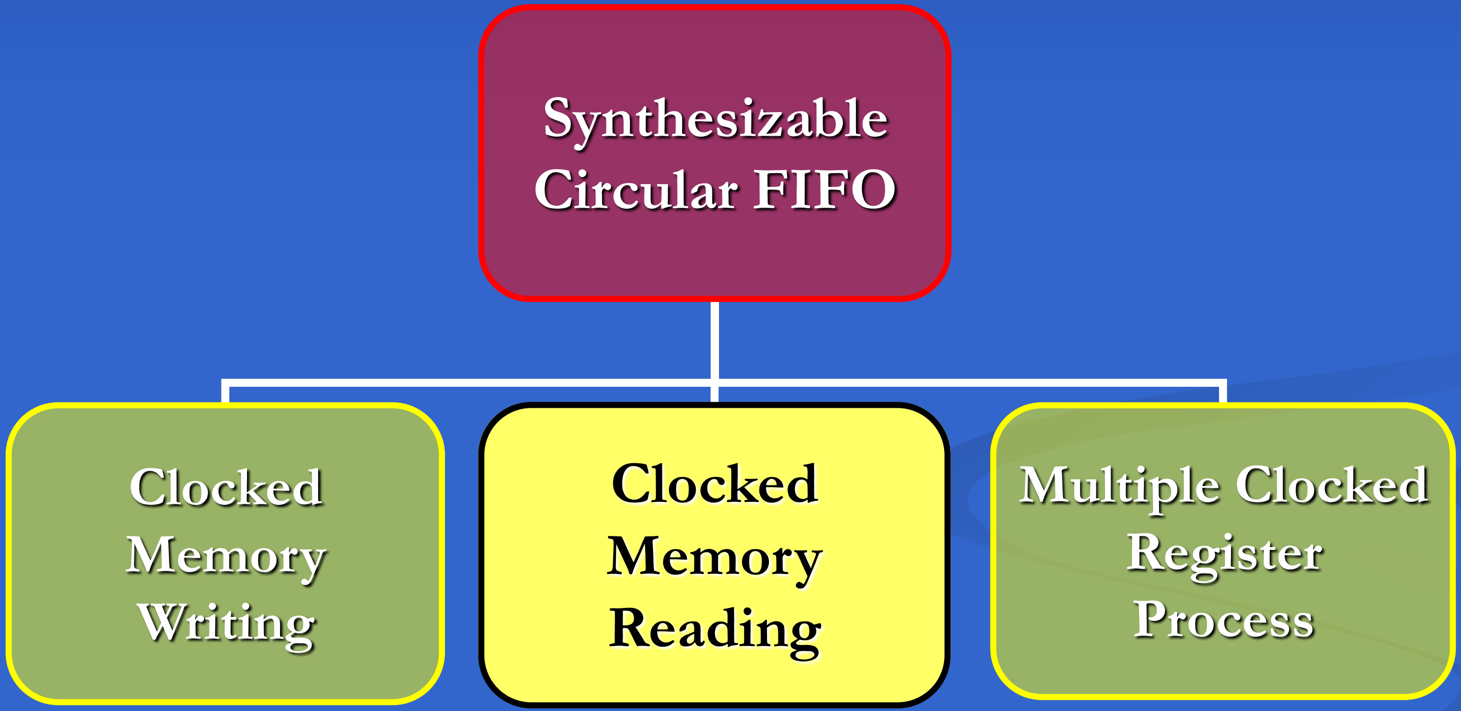
Clocked Memory Writing

```
write : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF (wr='1' AND full_temp='0') THEN
         fifo_ram (conv_integer (wr_ptr)) <= data_in;
      ELSIF (wr='1' AND rd='1') THEN
         fifo_ram (conv_integer (wr_ptr)) <= data_in;
      END IF;
   END IF;
END PROCESS;
```

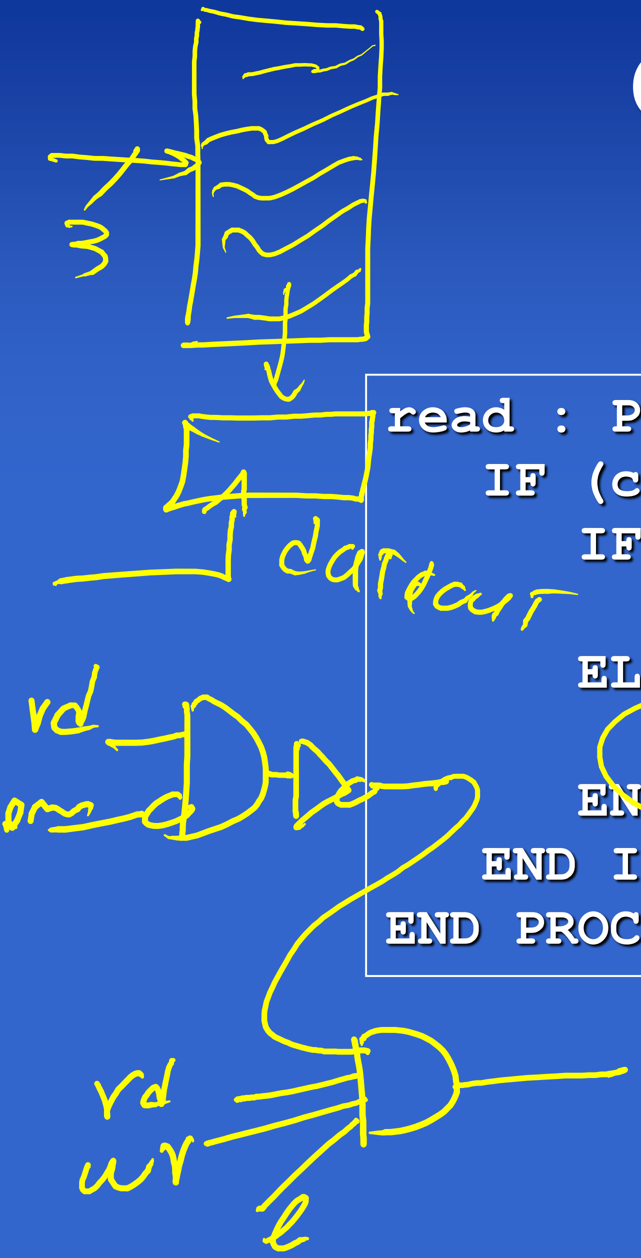


- Clocked Writing: *fifo_ram*

Clocked Memory Reading


Synthesizable Circular FIFO
Clocked Memory Writing
Clocked Memory Reading
Multiple Clocked Register Process

Clocked Memory Reading



```
read : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF (rd='1' AND empty_temp='0') THEN
         data_out <= fifo_ram (conv_integer (rd_ptr));
      ELSIF (rd='1' AND wr='1' AND empty_temp='1') THEN
         data_out <= fifo_ram (conv_integer (rd_ptr));
      END IF;
   END IF;
END PROCESS;
```

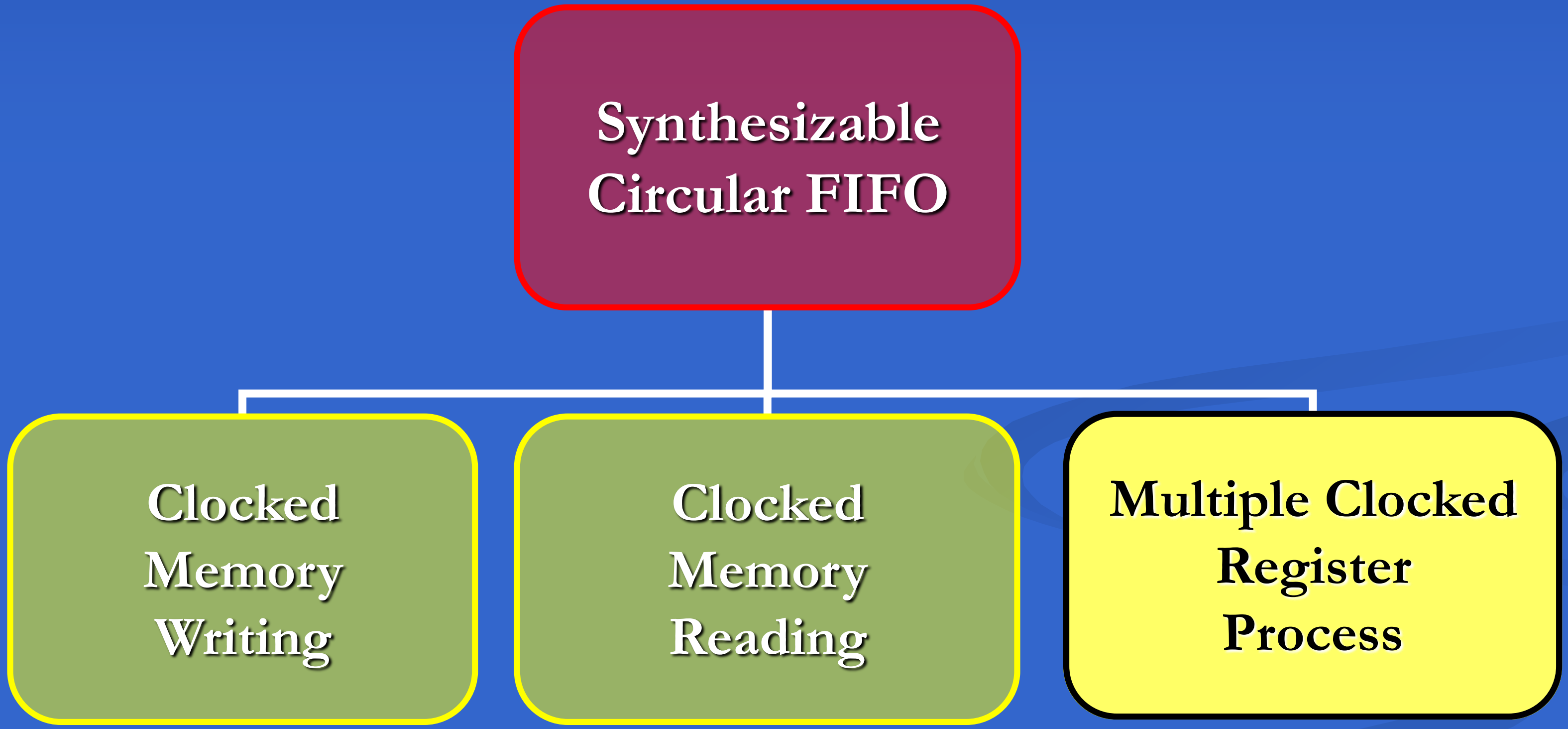
$\overline{(r \cdot \bar{e})} \cdot r \cdot w \cdot e$

$(\bar{r} + e) \cdot r \cdot w \cdot e$

$= r \cdot w \cdot e$

- Clocked Reading: *fifo_ram*

Multiple Clocked Register Process


Synthesizable Circular FIFO
Clocked Memory Writing
Clocked Memory Reading
Multiple Clocked Register Process

Multiple Clocked Register Process

```
pointer : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF rst='1' THEN
         wr_ptr <= (OTHERS => '0');
         rd_ptr <= (OTHERS => '0');
      ELSE
         . . . . . . . . . . . . .
         . . . . . . . . . . . . .
         . . . . . . . . . . . . .

      END IF;
   END IF;
END PROCESS;
```

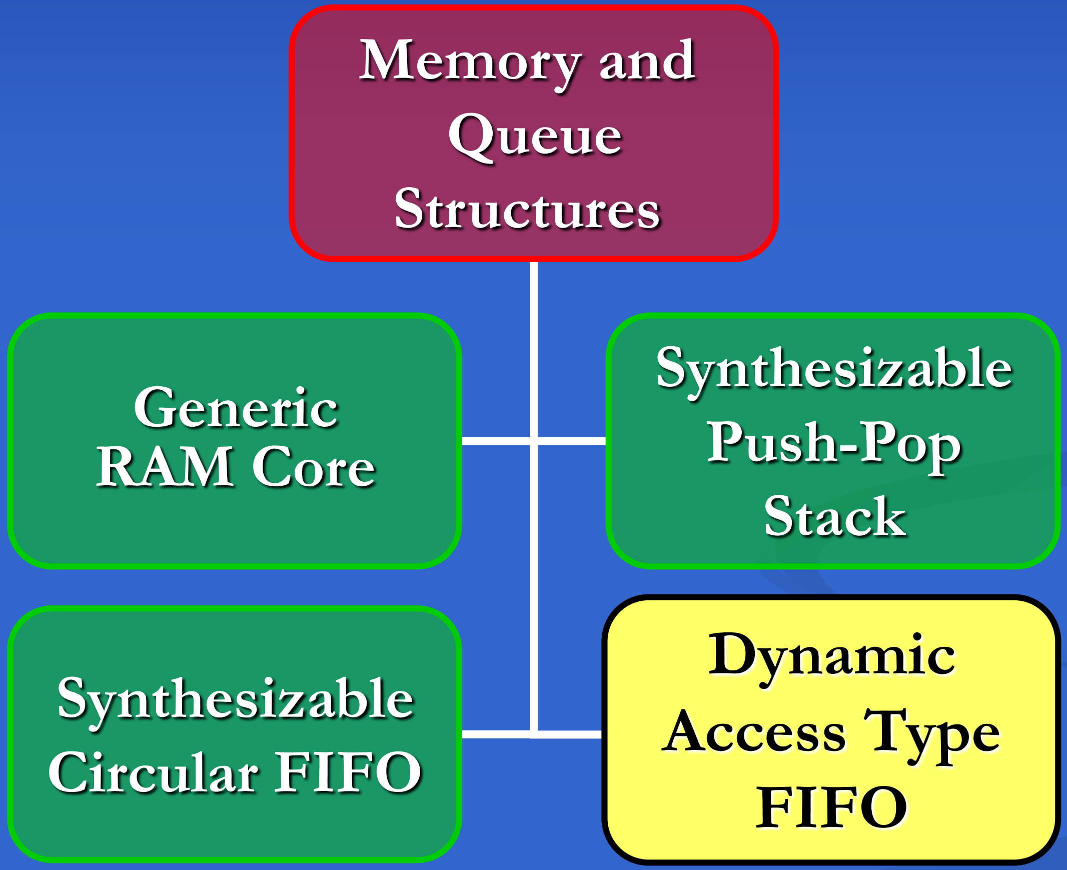
- Updating FIFO Pointers

Multiple Clocked Register Process

```
      . . . . . . . . . . . . . . . .
      ELSE
         IF (wr='1' AND full_temp='0') OR (wr='1' AND rd='1') THEN
            wr_ptr <= wr_ptr+1;
         ELSE
            wr_ptr <= wr_ptr;
         END IF;
         IF (rd='1' AND empty_temp='0') OR (wr='1' AND rd='1') THEN
            rd_ptr <= rd_ptr+1;
         ELSE
            rd_ptr <= rd_ptr;
         END IF;
      END IF;
```

- Updating FIFO Pointers (Continued)

Dynamic Access Type FIFO


Memory and Queue Structures
Generic RAM Core
Synthesizable Push-Pop Stack
Synthesizable Circular FIFO
Dynamic Access Type FIFO

Dynamic FIFO Structure

```
TYPE fifo_element;
TYPE pointer IS ACCESS fifo_element;
TYPE fifo_element IS RECORD
      data : std_logic_vector (7 DOWNTO 0);
      link : pointer;
END RECORD;
SHARED VARIABLE head, tail : pointer :=
NULL;
```

- Dynamic FIFO Structure

Different Hardware Levels

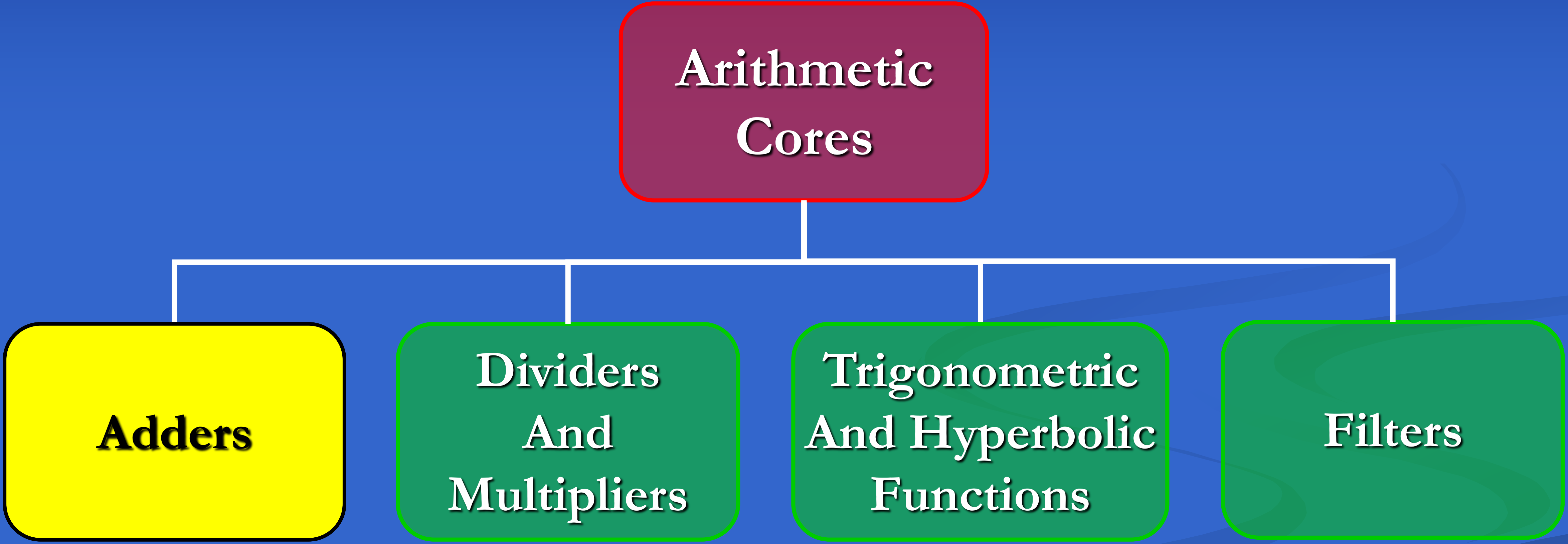
- Arithmetic Cores
- Wrappers
- Interfaces
- CPUs

- All applications may not fit this categorization

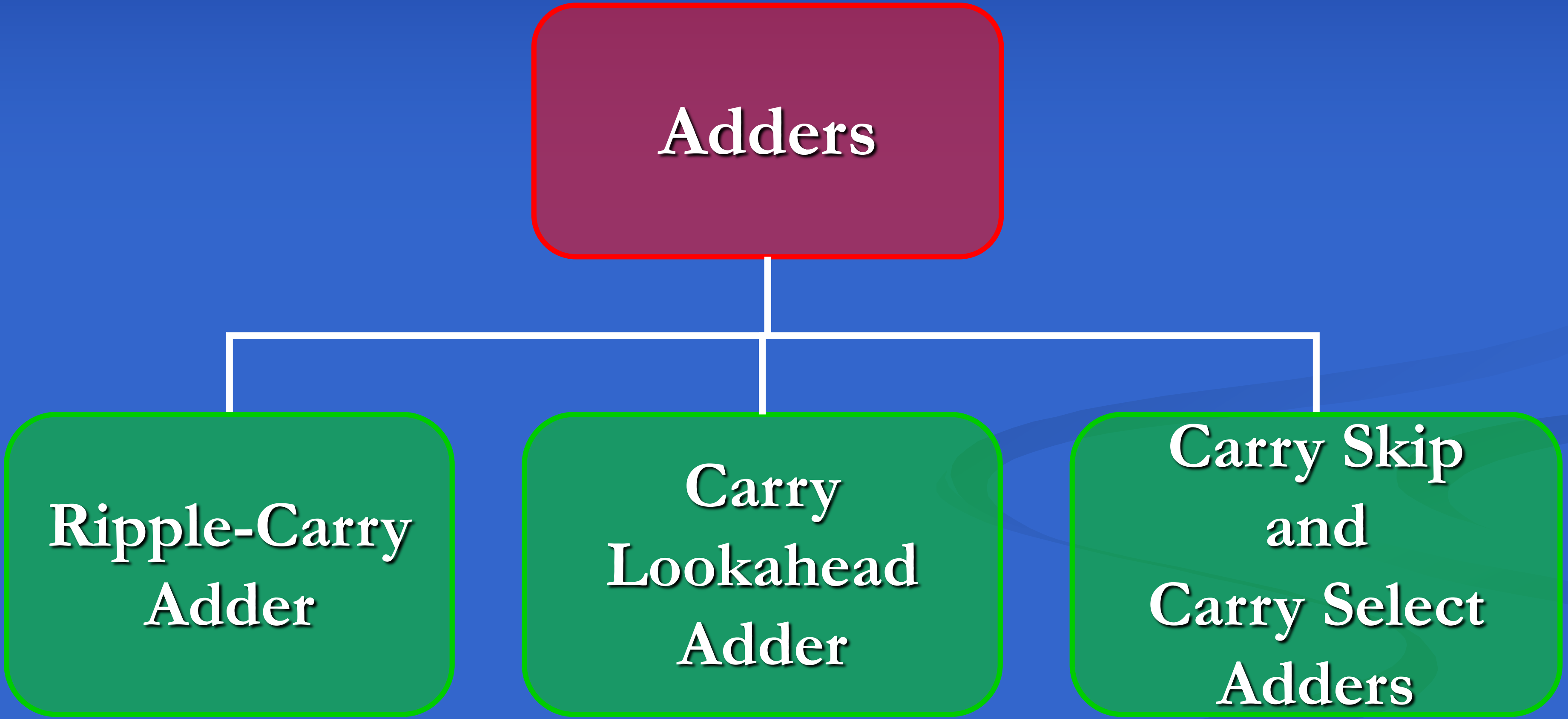
Arithmetic Cores

- Completely independent from connection handling
- Can be used as embedded cores in embedded designs
  - Carry Lookahead Adder
  - Sequential Multiplier
  - Booth Multiplier
  - Sinh of x
  - FIR filter

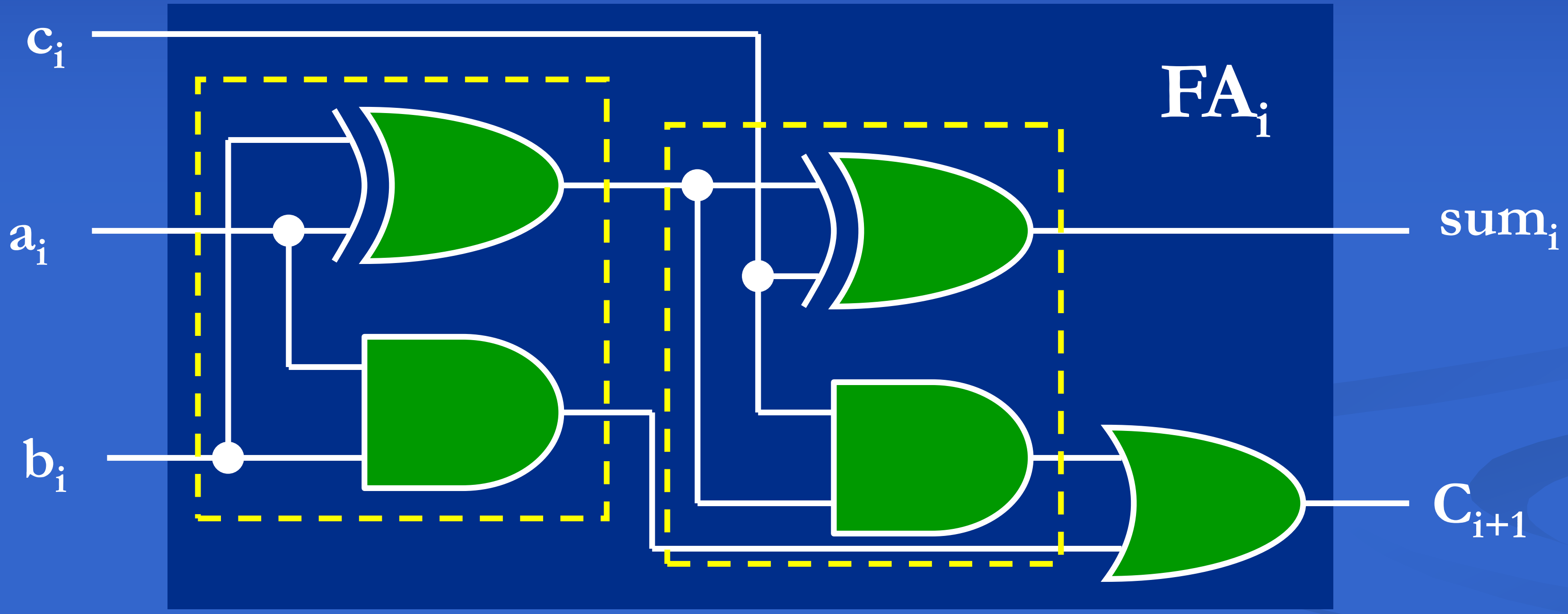
Arithmetic Cores



Adders


Adders
Ripple-Carry Adder
Carry Lookahead Adder
Carry Skip and Carry Select Adders

One-bit Full-Adder Circuit



$$sum_i = a_i \text{ xor } b_i \text{ xor } c_i$$

$$c_{i+1} = a_i\, b_i + b_i\, c_i + a_i\, c_i$$

Ripple-Carry Adder

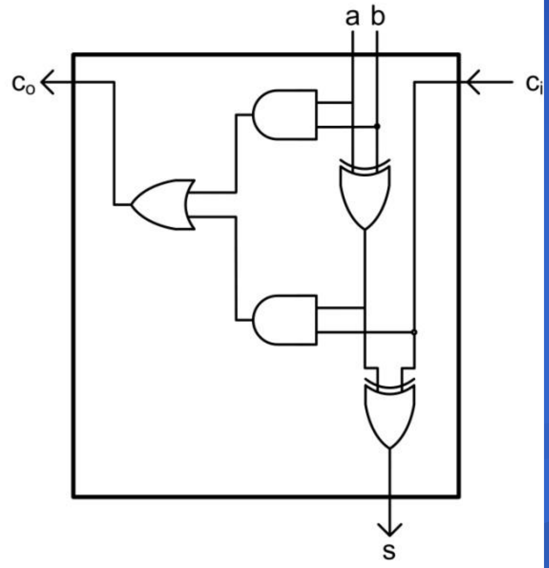
a	b	c_i	c_0	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

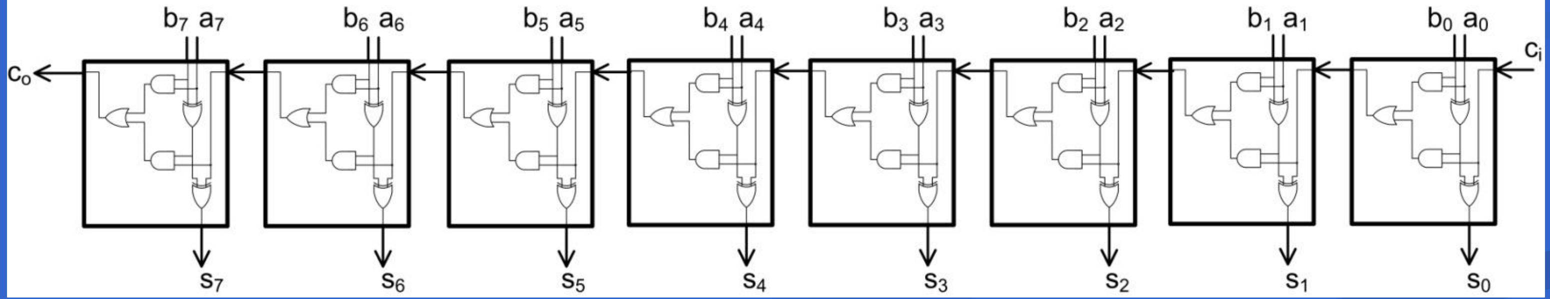
c_i \ a b	00	01	11	10
0			1	
1		1	1	1

$c_o = ab + ac_i + bc_i$

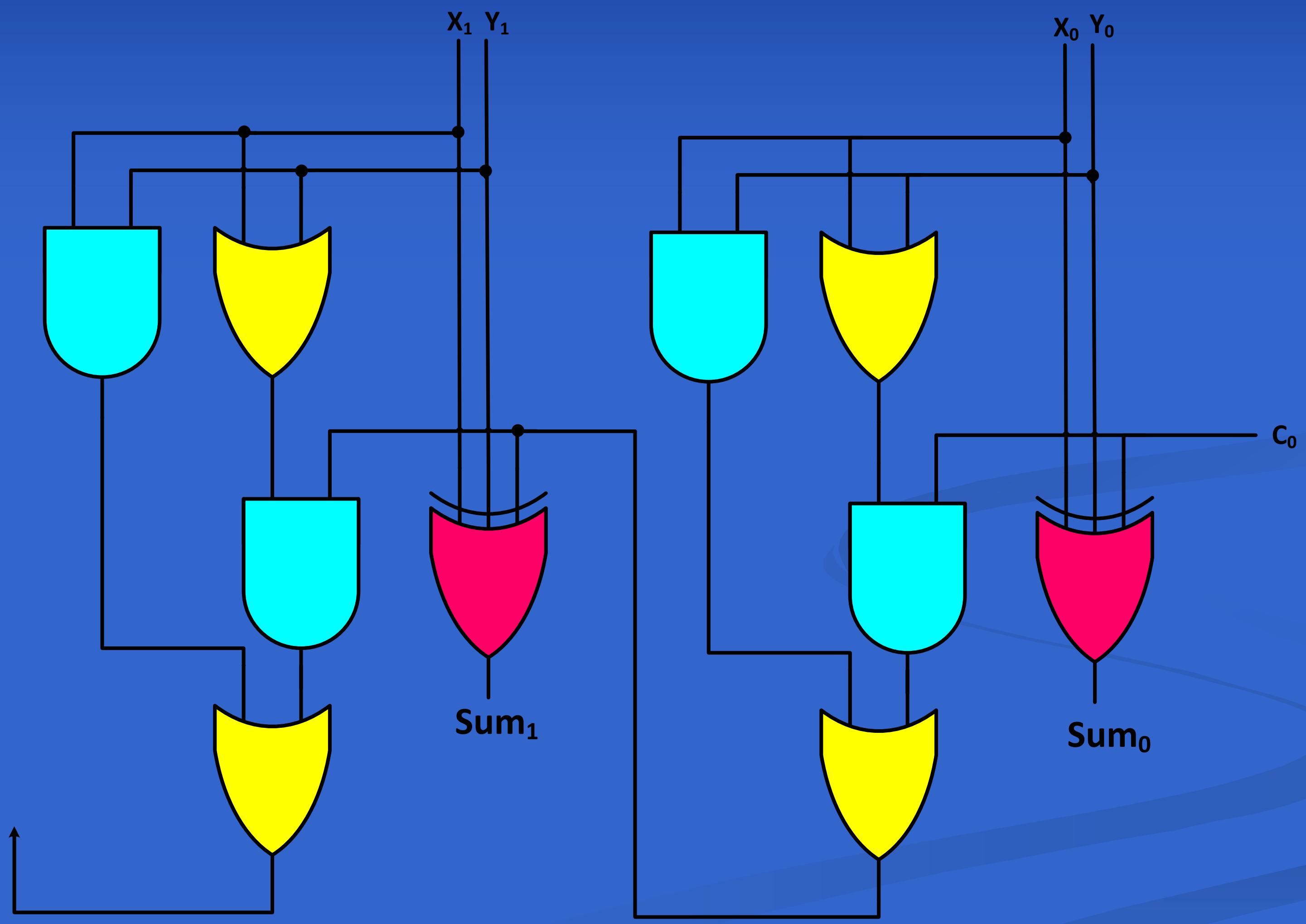
c_i \ a b	00	01	11	10
0		1		1
1	1		1	

$s = a \oplus b \oplus c_i$

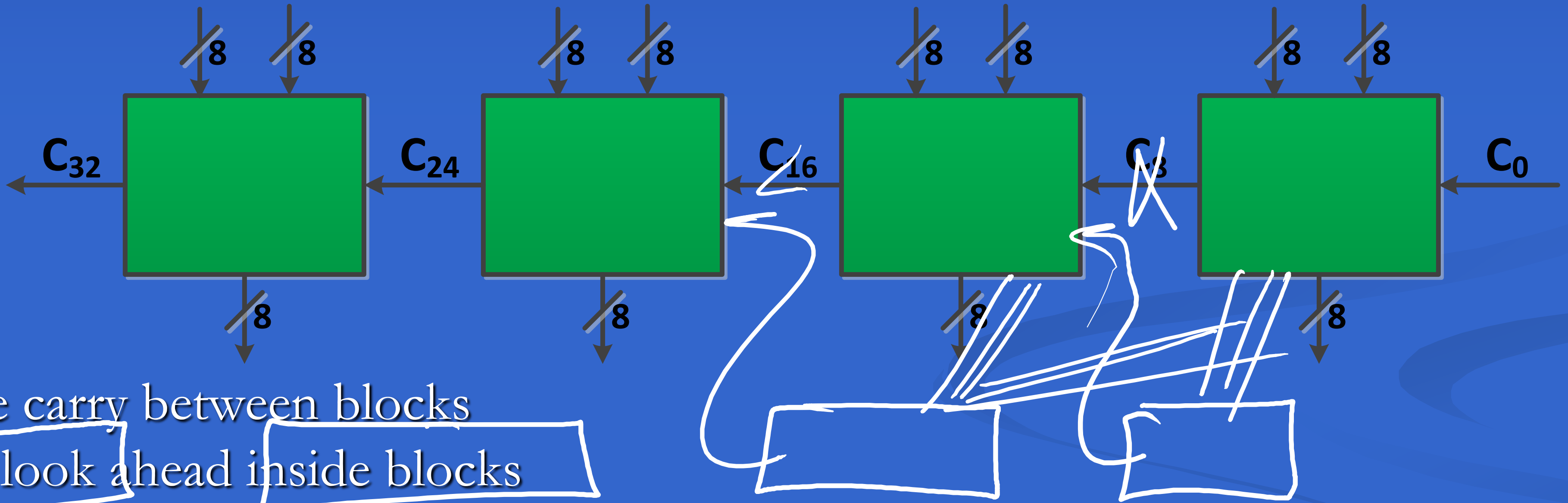




Carry Lookahead Adders

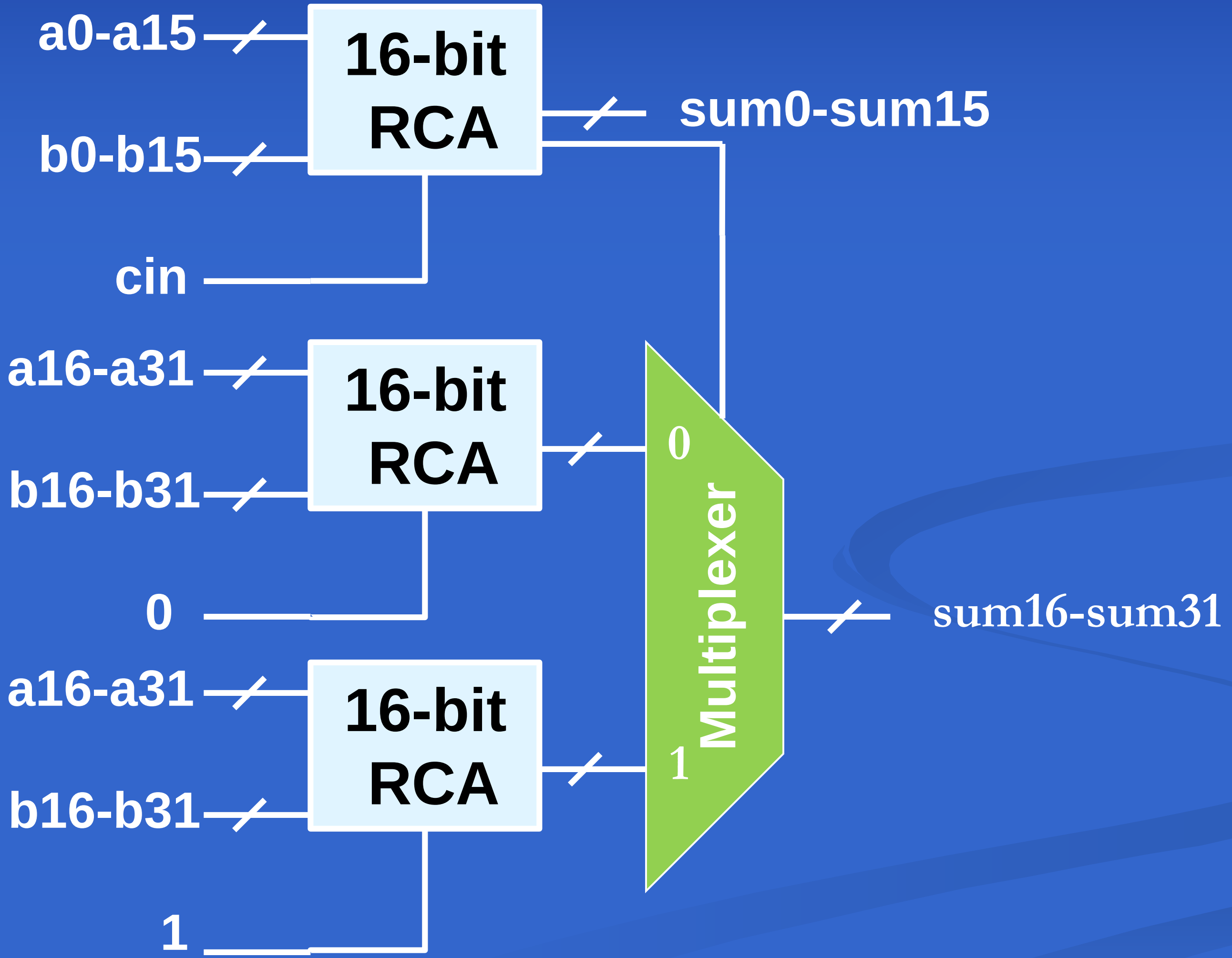


Group Propagate (PG) and Group Generate (GG) for an 8-bit CLA

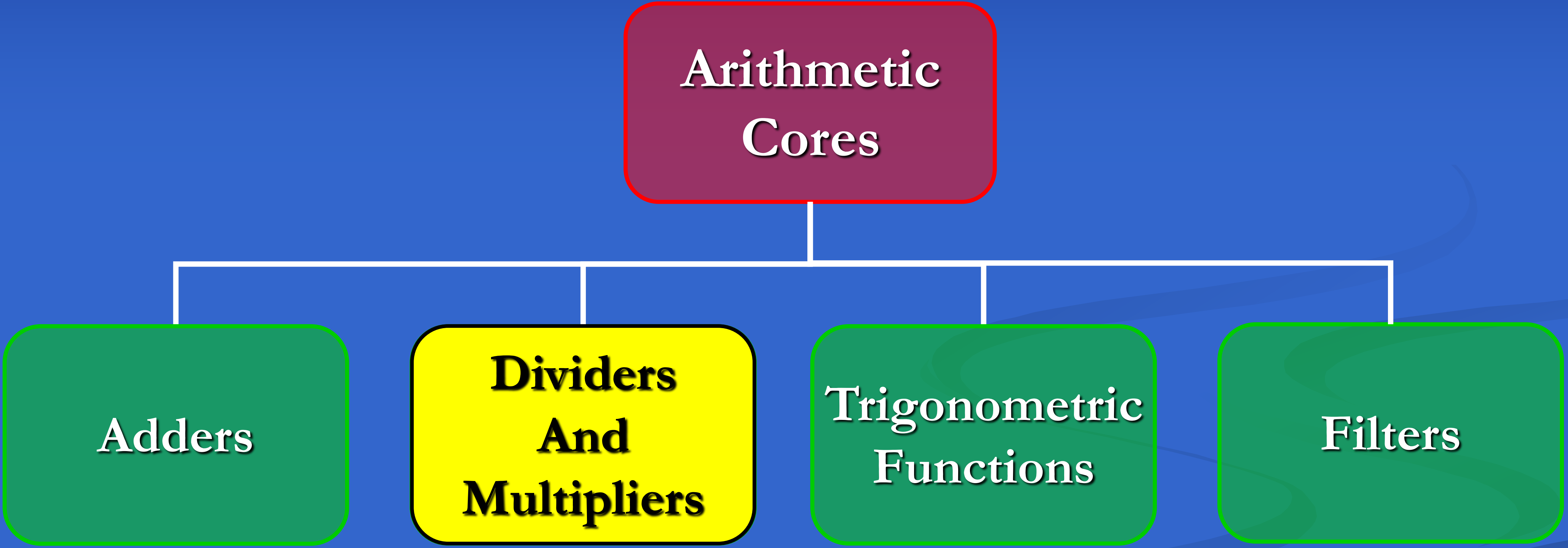


- Ripple carry between blocks
- Carry look ahead inside blocks

Carry-Select Adder

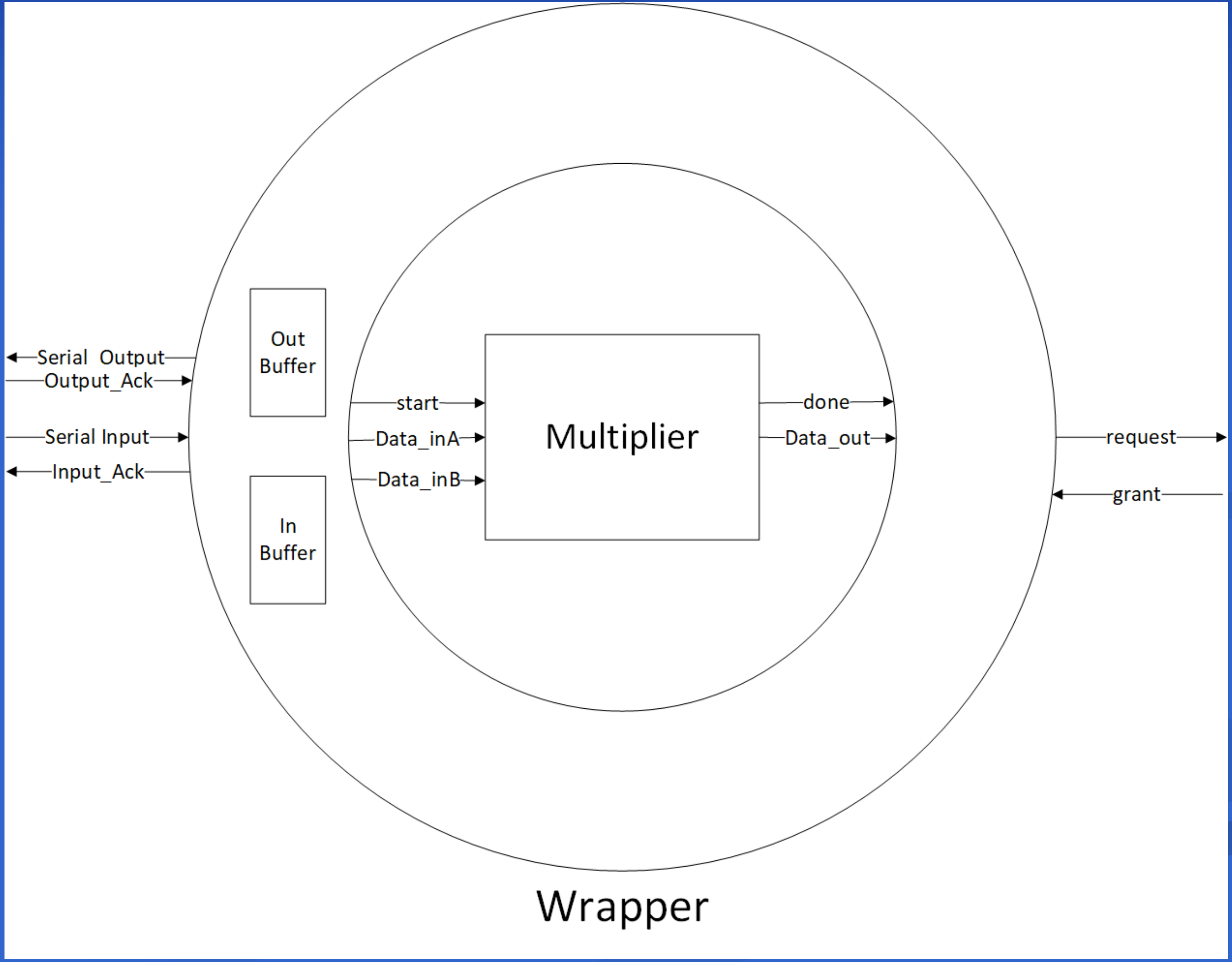


Arithmetic Cores



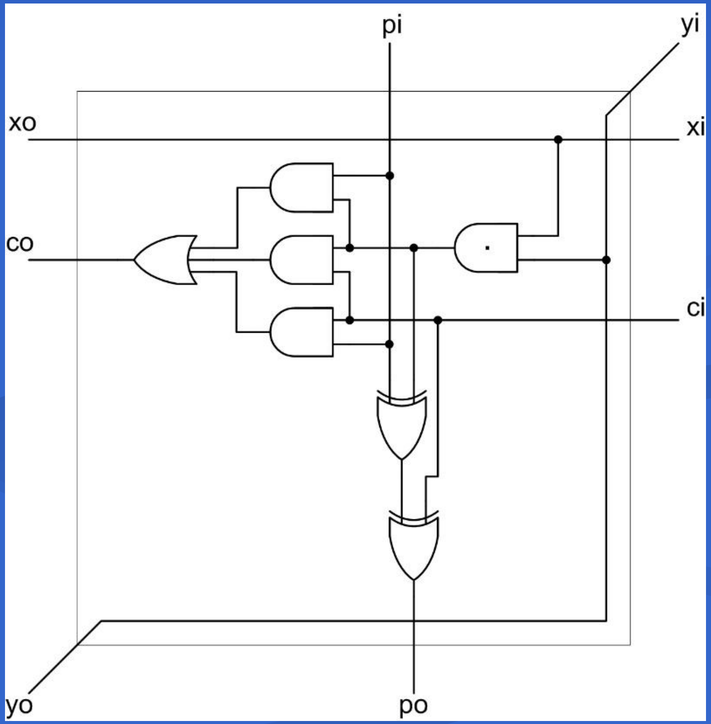
Multiplication

- Sequential Multiplier
- Array Multiplier
- Booth Multiplier



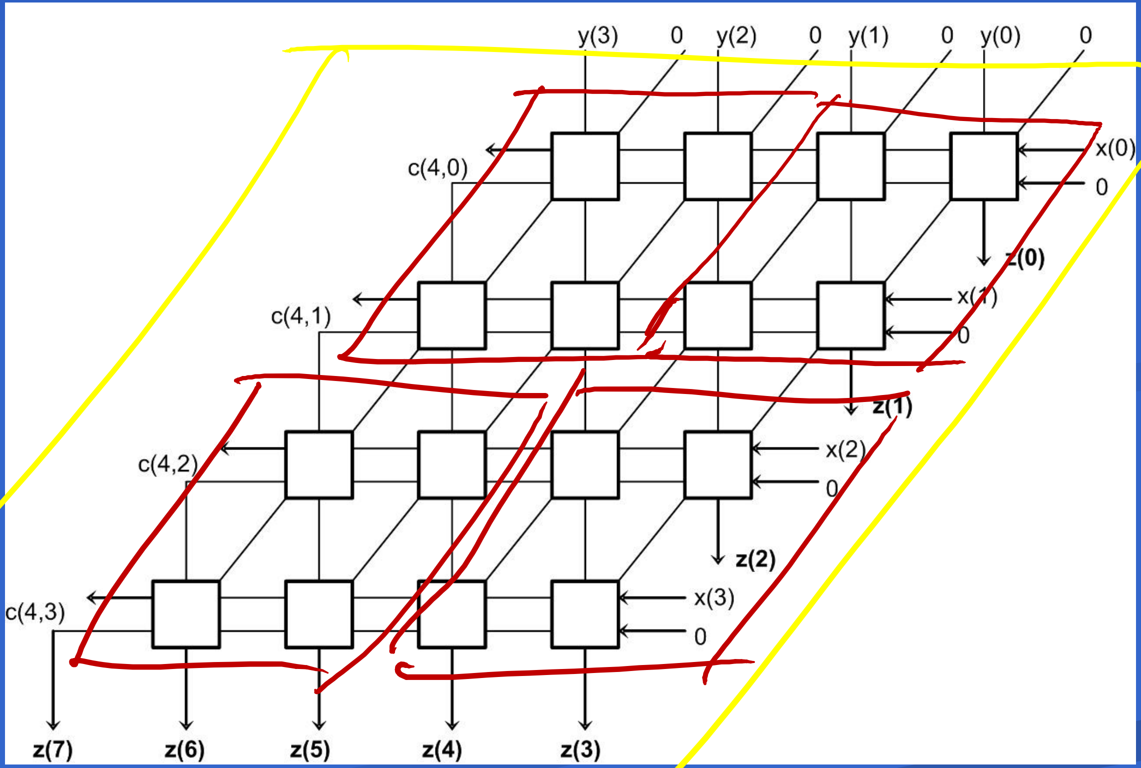
Array Multiplier

- Figure circuit multiplies its *xi* and *yi* inputs using the AND gate that is marked with a dot
- Adds this result with its input partial product *pi*, using its carry input *ci*.
- This cell generates a partial product *po*, a carry output *co*, and passes *xi* and *yi* inputs on to its outputs (*xo* and *yo*).

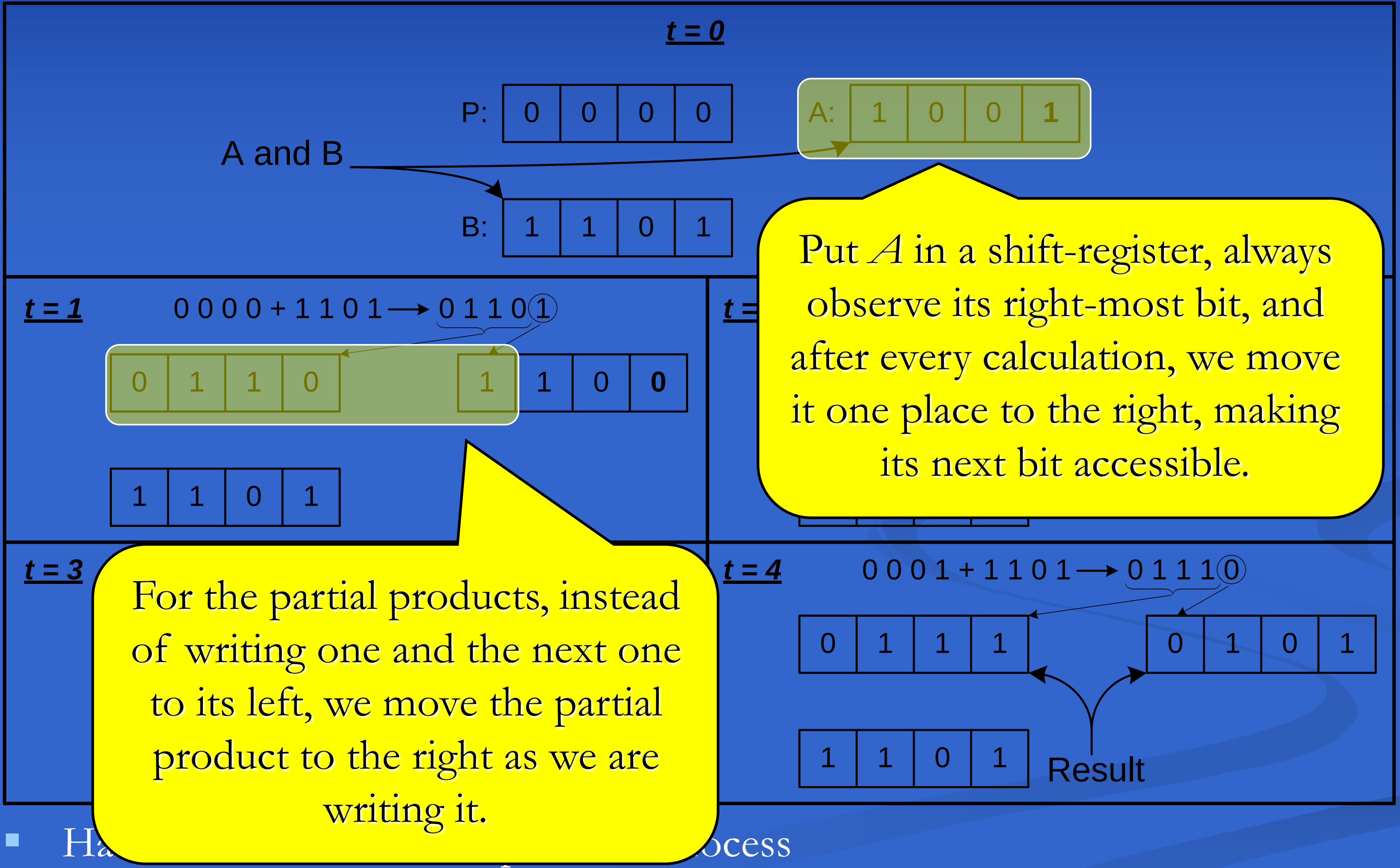


Array Multiplier

- 4×4 array multiplier that uses 16 of the multiplier cells.
- A 32-bit multiplier requires 1024 such cells.

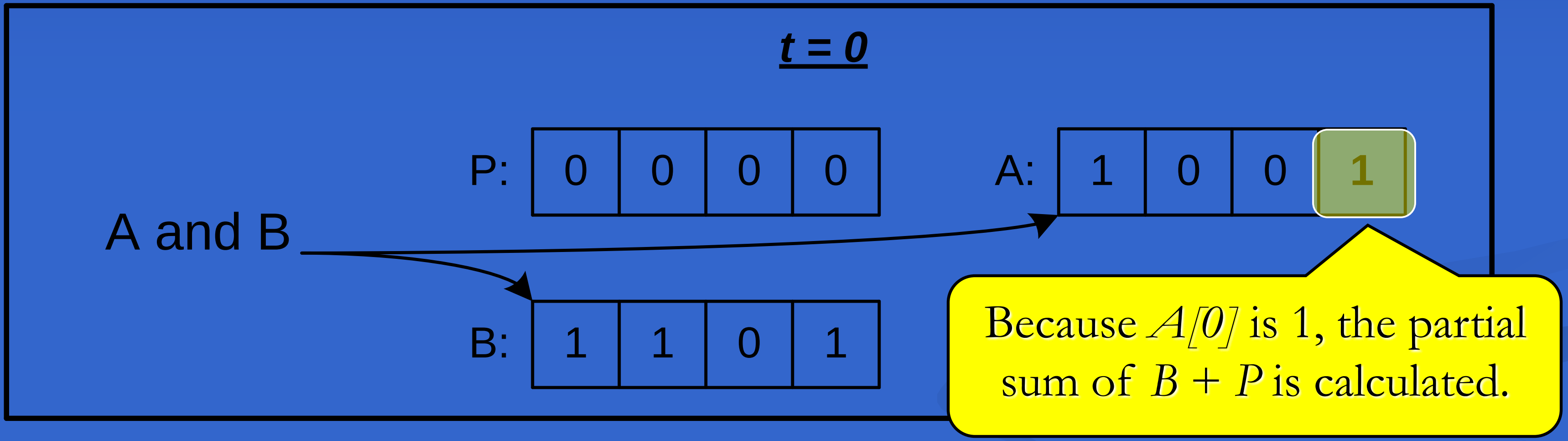


Shift-and-add Multiplication Process


t = 0
P: 0 0 0 0
A: 1 0 0 1
A and B
B: 1 1 0 1
t = 1
0 0 0 0 + 1 1 0 1 → 0 1 1 0 1
0 1 1 0 1 1 0 0
1 1 0 1
t = 3
t = 4
0 0 0 1 + 1 1 0 1 → 0 1 1 1 0
0 1 1 1 0 1 0 1
1 1 0 1
Result
Put *A* in a shift-register, always observe its right-most bit, and after every calculation, we move it one place to the right, making its next bit accessible.
For the partial products, instead of writing one and the next one to its left, we move the partial product to the right as we are writing it.

- Ha ... ocess

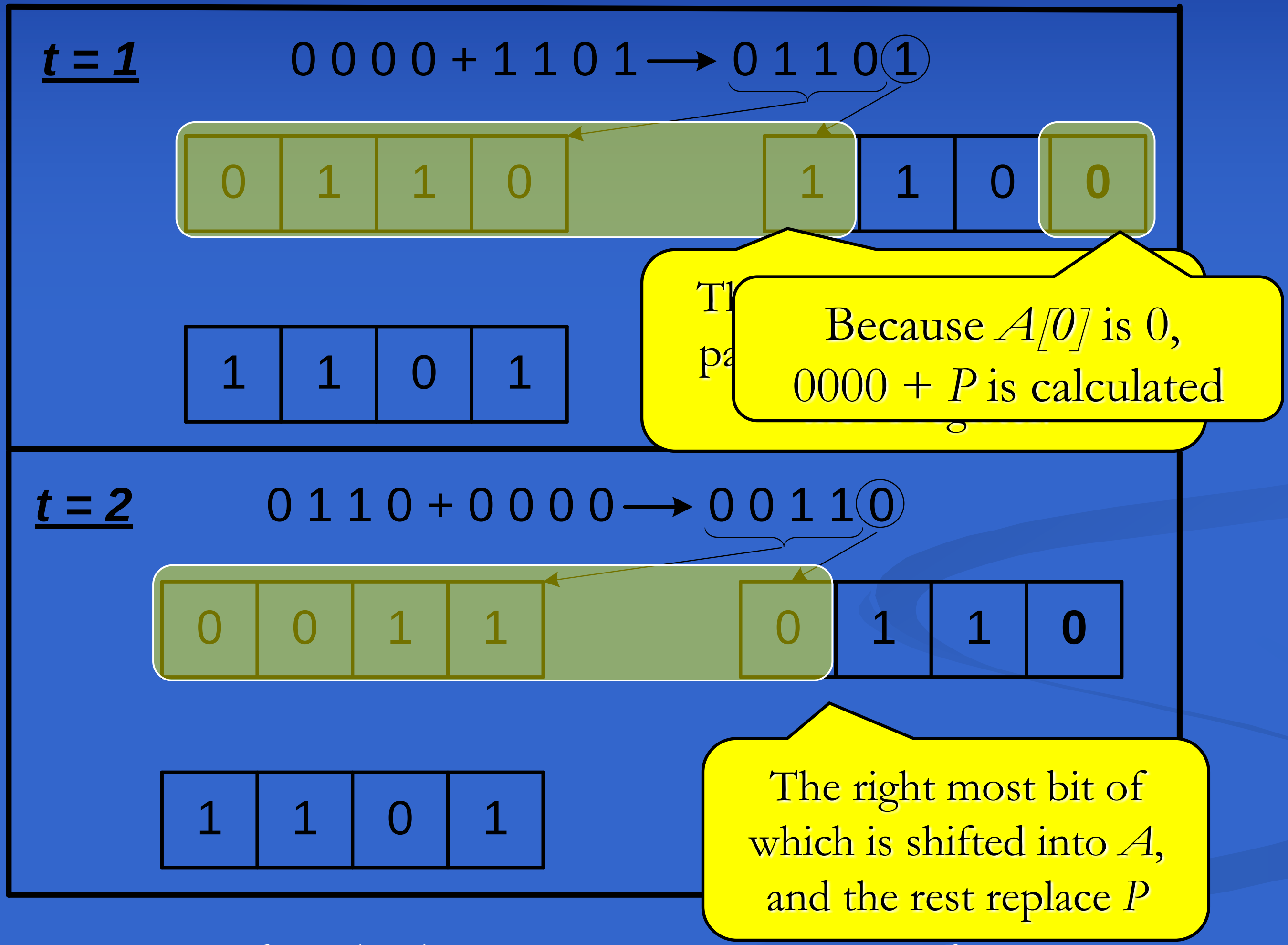
Shift-and-add Multiplication Process



Because *A[0]* is 1, the partial sum of *B* + *P* is calculated.

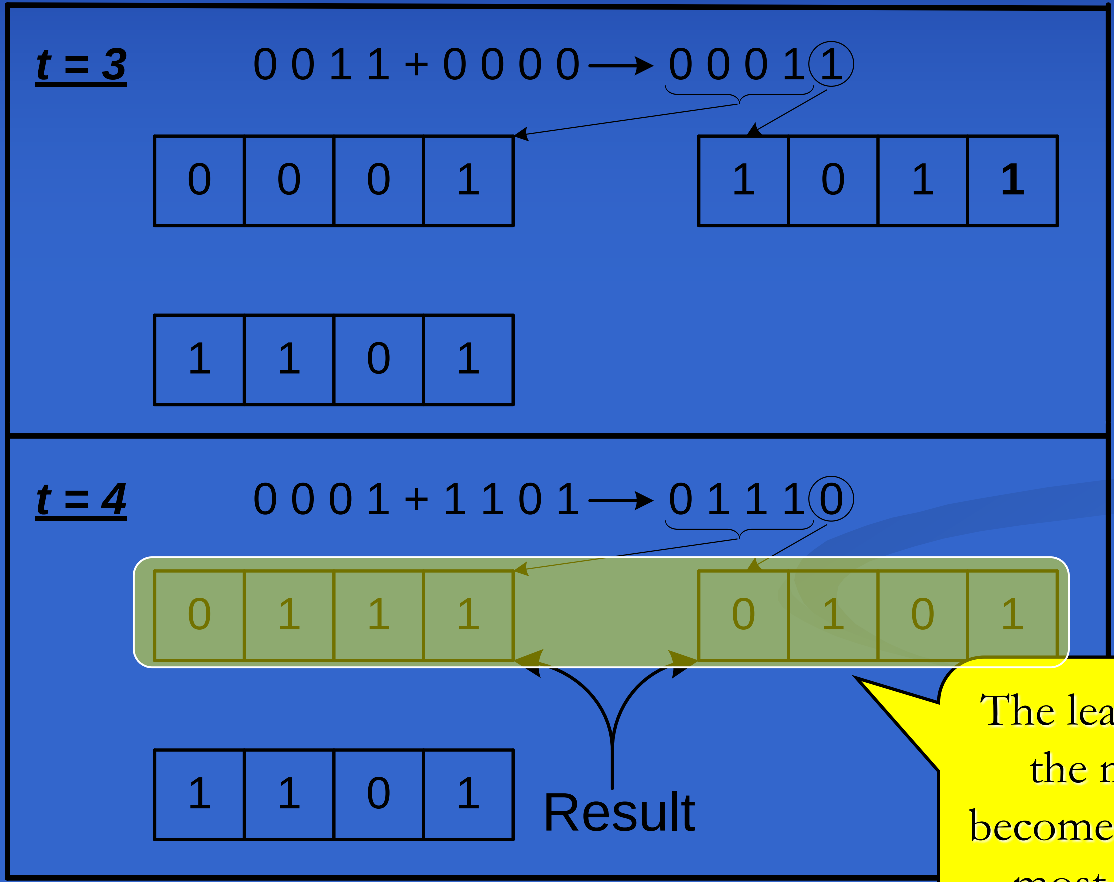
- Hardware Oriented Multiplication Process (Continued)

Shift-and-add Multiplication Process



- Hardware Oriented Multiplication Process (Continued)

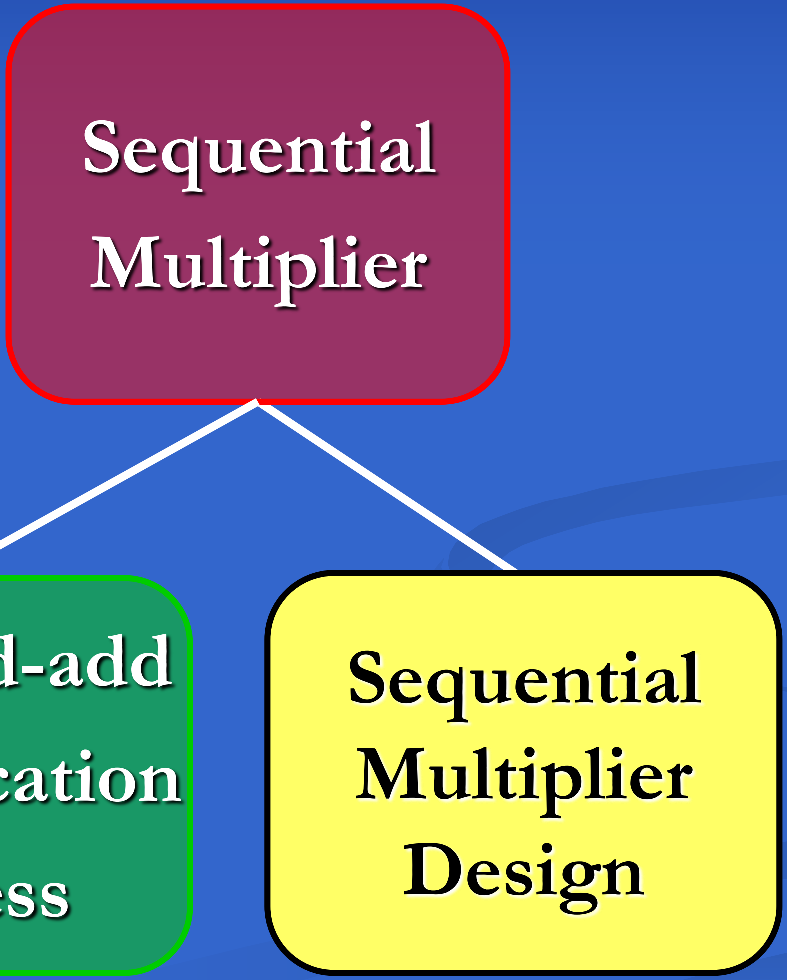
Shift-and-add Multiplication Process



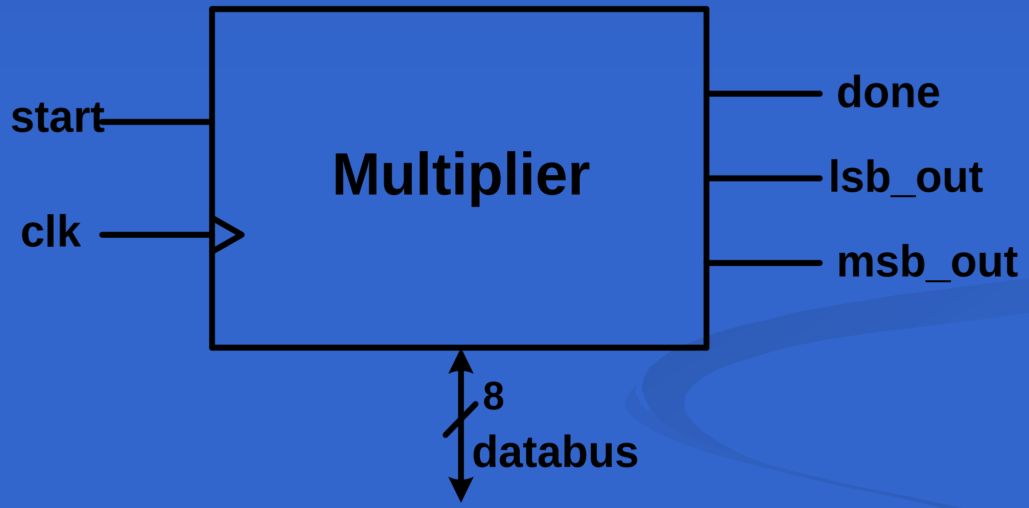
The least significant 4 bits of the multiplication result become available in *A* and the most-significant bits in *P*.

- Hardware Oriented Multiplication Process (Continued)

Sequential Multiplier Design

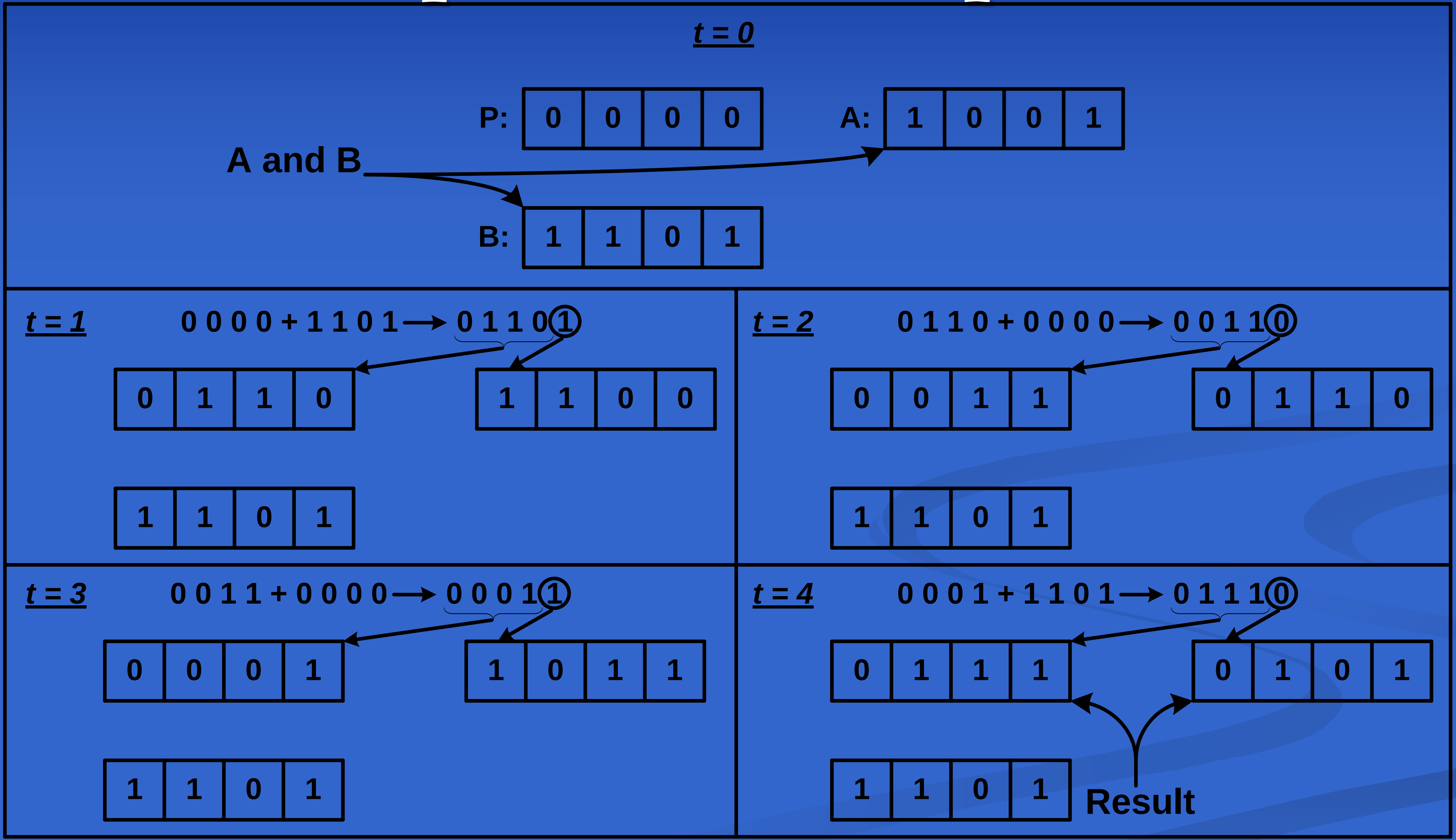

Sequential Multiplier
Shift-and-add Multiplication Process
Sequential Multiplier Design

Sequential Multiplier



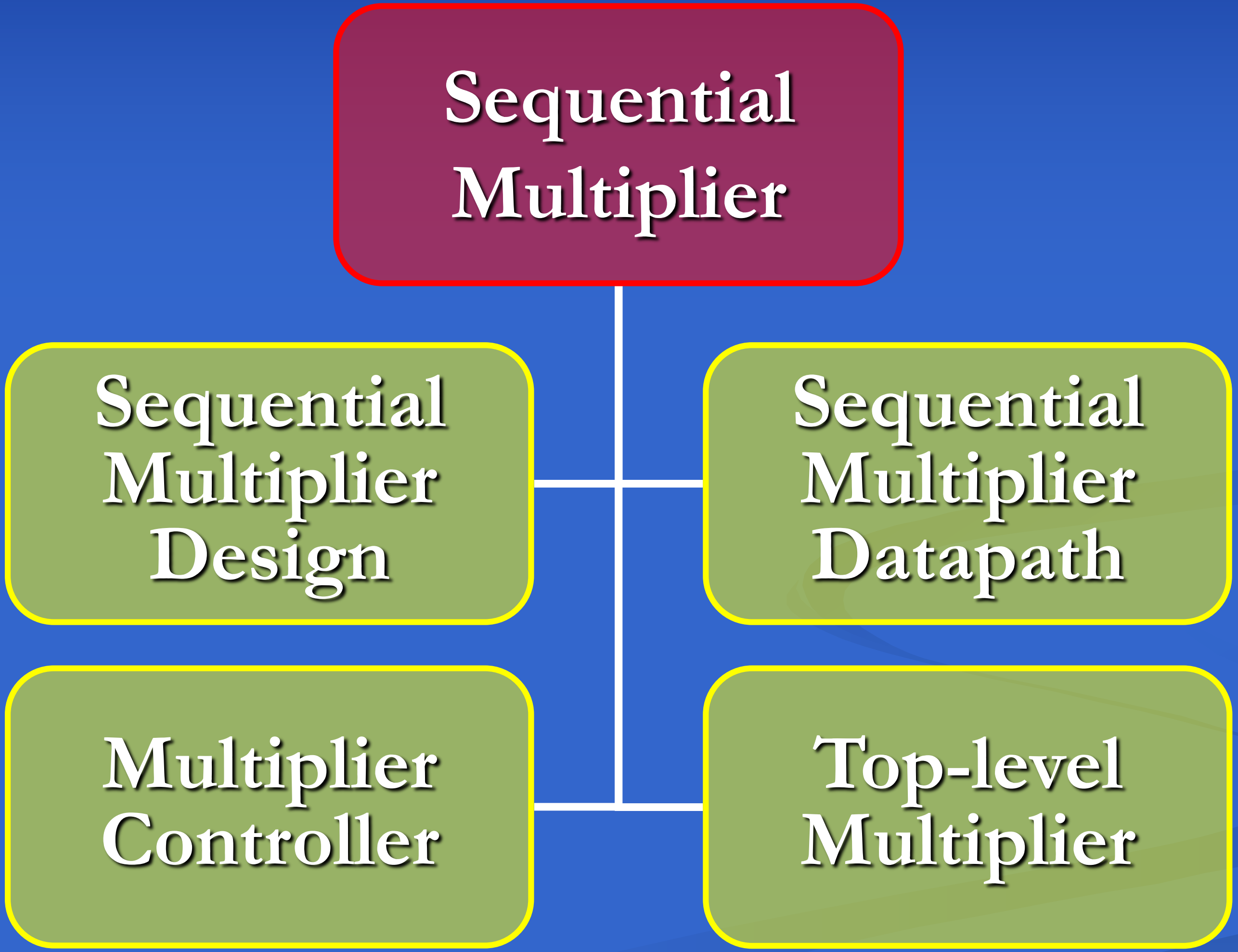
- Multiplier Block Diagram

Sequential Multiplier

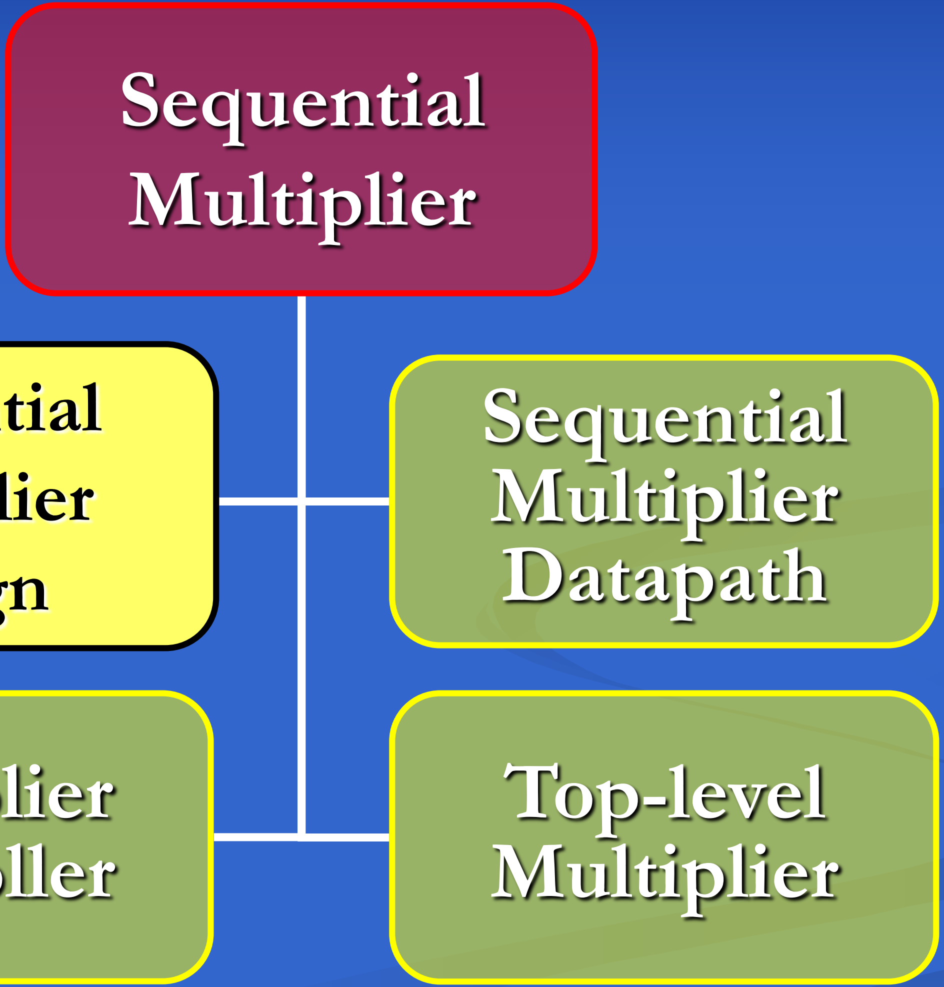


- Hardware Oriented Multiplication Process

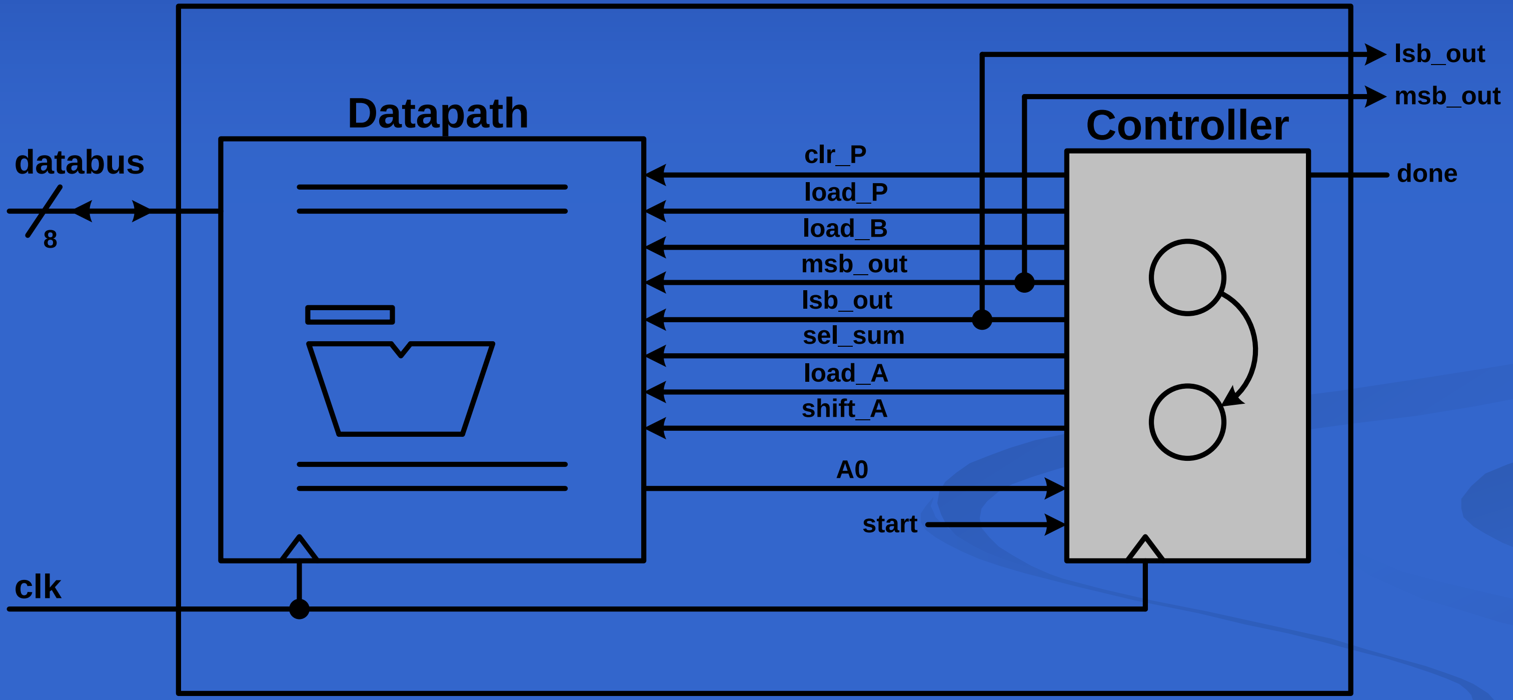
Sequential Multiplier



Sequential Multiplier Design

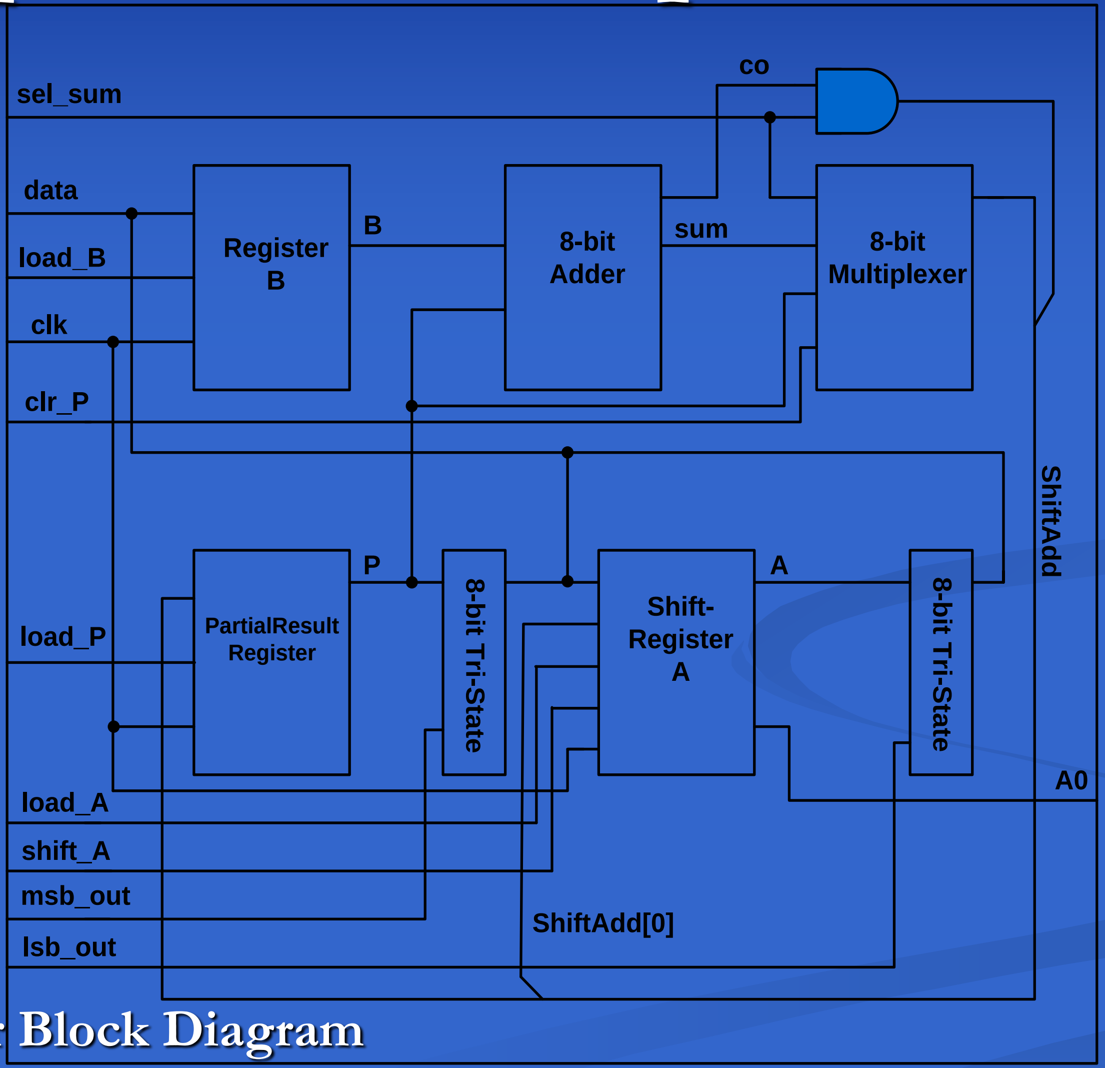


Sequential Multiplier Design



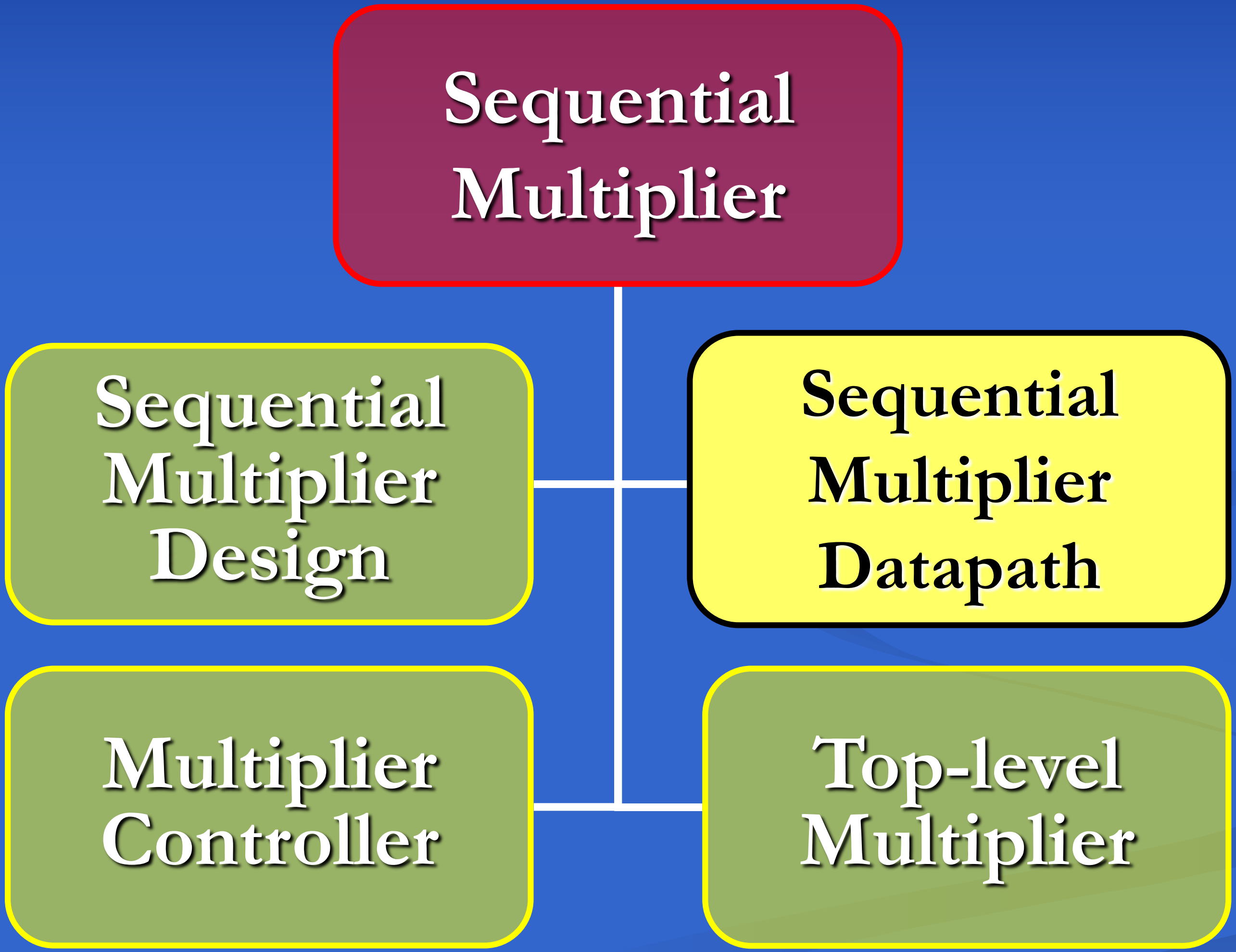
- Datapath and Controller

Sequential Multiplier Design



- Multiplier Block Diagram

Sequential Multiplier Datapath



Sequential Multiplier Datapath

```
ENTITY datapath IS
   PORT (clk, clr_P, load_P, load_B : IN std_logic;
         msb_out, lsb_out, sel_sum : IN std_logic;
         load_A, shift_A : IN std_logic;
       data : INOUT std_logic_vector (7 DOWNTO 0);
       A0 : OUT std_logic);
END ENTITY;
--
ARCHITECTURE procedural OF datapath IS
   SIGNAL sum, ShiftAdd : std_logic_vector (7 DOWNTO 0);
   SIGNAL A, B, P : std_logic_vector (7 DOWNTO 0);
   SIGNAL co : std_logic;
   SIGNAL op : std_logic_vector (1 DOWNTO 0);
   SIGNAL result : std_logic_vector (8 DOWNTO 0);
   . . . . . . . . . . . . .
END ARCHITECTURE procedural;
```

- Shift-and-add Multiplier Datapath

Sequential Multiplier Datapath

```
PROCESS (clk) BEGIN
   IF(clk = '0' AND clk'EVENT) THEN
      IF (load_B = '1') THEN B <= data;
      END IF;
   END IF;
END PROCESS;
--
PROCESS (clk) BEGIN
  IF(clk = '0' AND clk'EVENT) THEN
     IF (load_P = '1') THEN
       P <= (co AND sel_sum) & ShiftAdd (7 DOWNTO 1);
     END IF;
  END IF;
END PROCESS;
--
```

- Shift-and-add Multiplier Datapath (Continued)

Sequential Multiplier Datapath

```
  PROCESS (clk) BEGIN
      IF(clk = '0' AND clk'EVENT) THEN
         CASE op IS
            WHEN "01" => A <= ShiftAdd(0) &
                              A(7 DOWNTO 1);
            WHEN "10" => A <= data;
            WHEN OTHERS => A <= A;
         END CASE;
      END IF;
   END PROCESS;
   . . . . . . . . . . . . . . . . .
```

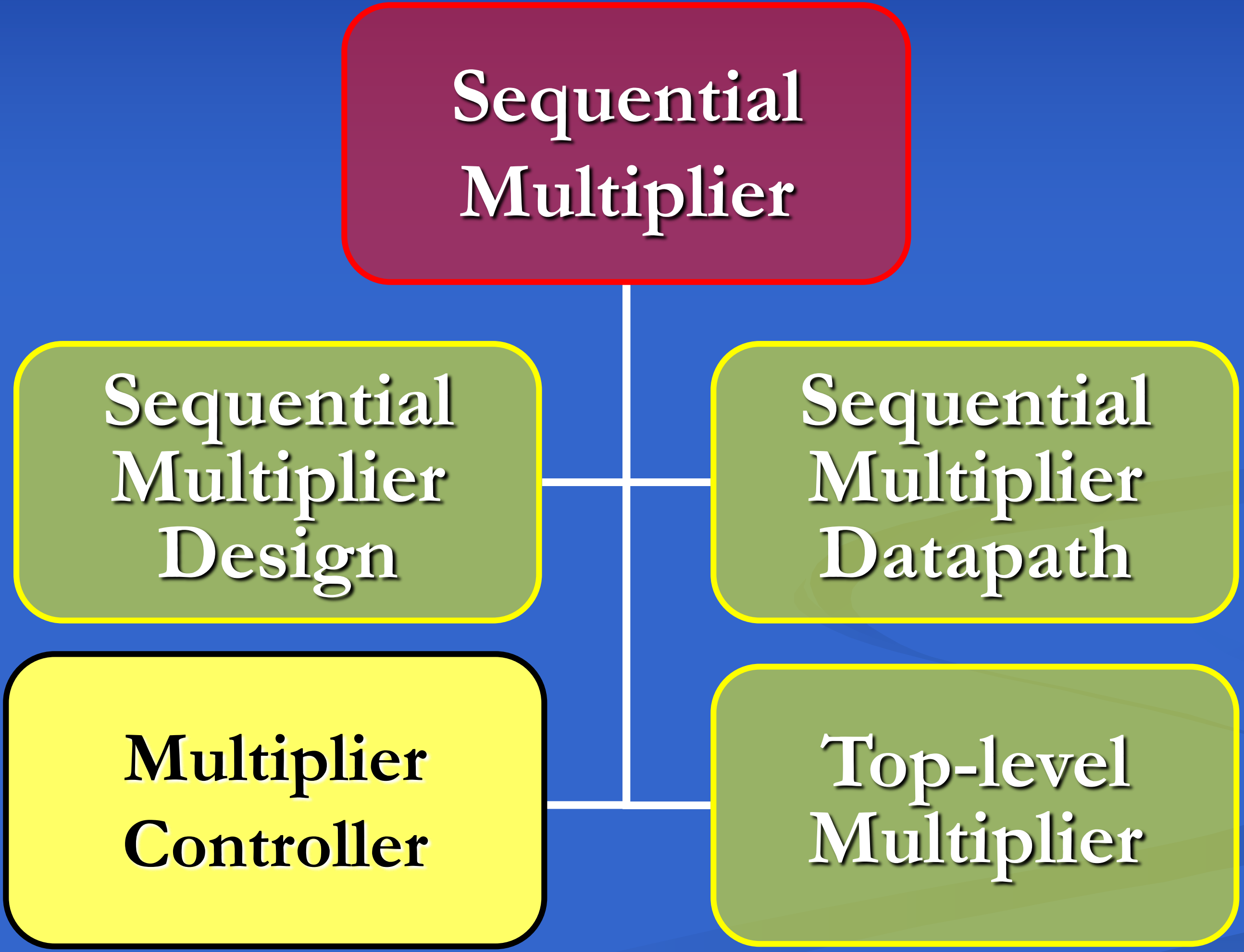
- Shift-and-add Multiplier Datapath (Continued)

Sequential Multiplier Datapath

```
    . . . . . . . . . . . . . .
    result <= ('0'&P) + ('0'&B);
    co <= result(8);
    sum <= result(7 DOWNTO 0);

    A0 <= A(0);
    ShiftAdd <= (OTHERS => '0') WHEN clr_P = '1' ELSE
                 P WHEN sel_sum = '0' ELSE sum;
    data <= A WHEN lsb_out = '1' ELSE (OTHERS => 'Z');
    data <= P WHEN msb_out = '1' ELSE (OTHERS => 'Z');
    op <= load_A & shift_A;
END ARCHITECTURE procedural;
```

- Shift-and-add Multiplier Datapath (Continued)

Multiplier Controller



Multiplier Controller

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
ENTITY controller IS
   PORT (clk, start, A0 : IN std_logic;
         clr_P, load_P, load_B : OUT std_logic;
         msb_out, lsb_out, sel_sum : OUT std_logic;
         load_A, Shift_A, done : OUT std_logic);
END ENTITY;
--
```

- Multiplier Controller

Multiplier Controller

```
ARCHITECTURE procedural OF controller IS
   TYPE state IS (idle, init,
                  m1, m2, m3, m4, m5, m6, m7, m8,
                  rslt1, rslt2);
   SIGNAL current : state;
BEGIN
   sequential: PROCESS (clk) BEGIN
      IF (clk = '0' AND clk'EVENT) THEN
         CASE current IS
            WHEN idle =>
               IF start = '0' THEN current <= idle;
               ELSE
                  current <= init;
               END IF;
            . . . . . . . . . . . . . .
   END PROCESS; --
```

- Multiplier Controller (Continued)

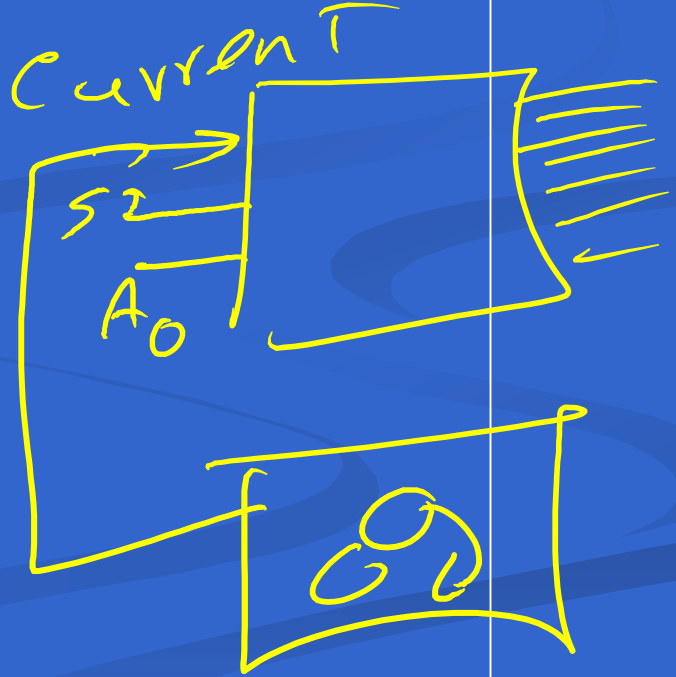
Multiplier Controller

```
    sequential: PROCESS (clk) BEGIN
        IF (clk = '0' AND clk'EVENT) THEN
            CASE current IS
                . . . . . . . . . . . . . . .
                WHEN init =>
                    current <= m1;
                WHEN m1 | m2 | m3 | m4 | m5 | m6 | m7 | m8
                             =>
                    current <= state'SUCC(current);
                WHEN rslt1 =>
                    current <= rslt2;
                WHEN rslt2 =>
                    current <= idle;
                WHEN OTHERS =>
                    current <= idle;
            END CASE;
```

- Multiplier Controller

Multiplier Controller

```
sequential: PROCESS (clk) BEGIN
   . . . . . . . . . . . . . . .
END PROCESS; --
combinational: PROCESS (current, start, A0) BEGIN
      clr_P <= '0'; load_P <= '0';
      load_B <= '0';
      msb_out <= '0'; lsb_out <= '0';
      sel_sum <= '0'; load_A <= '0';
      Shift_A <= '0'; done <= '0';
      CASE current IS
         WHEN idle =>
            IF start = '0' THEN
               done <= '1';
            ELSE
               load_A <= '1';
               clr_P<= '1';
               load_P <= '1';
            END IF;
```



Multiplier Controller

```
combinational: PROCESS (current, start, A0) BEGIN
    CASE current IS
        . . . . . . . . . . . . . . . .
      WHEN init =>
         load_B <= '1';
      WHEN m1 | m2 | m3 | m4 | m5 | m6 | m7 | m8
              =>
         Shift_A <= '1';
         load_P <= '1';
         IF (A0 = '1') THEN
            sel_sum <= '1';
         END IF;
      WHEN rslt1 =>
         lsb_out <= '1';
      WHEN rslt2 =>
         msb_out <= '1';
```

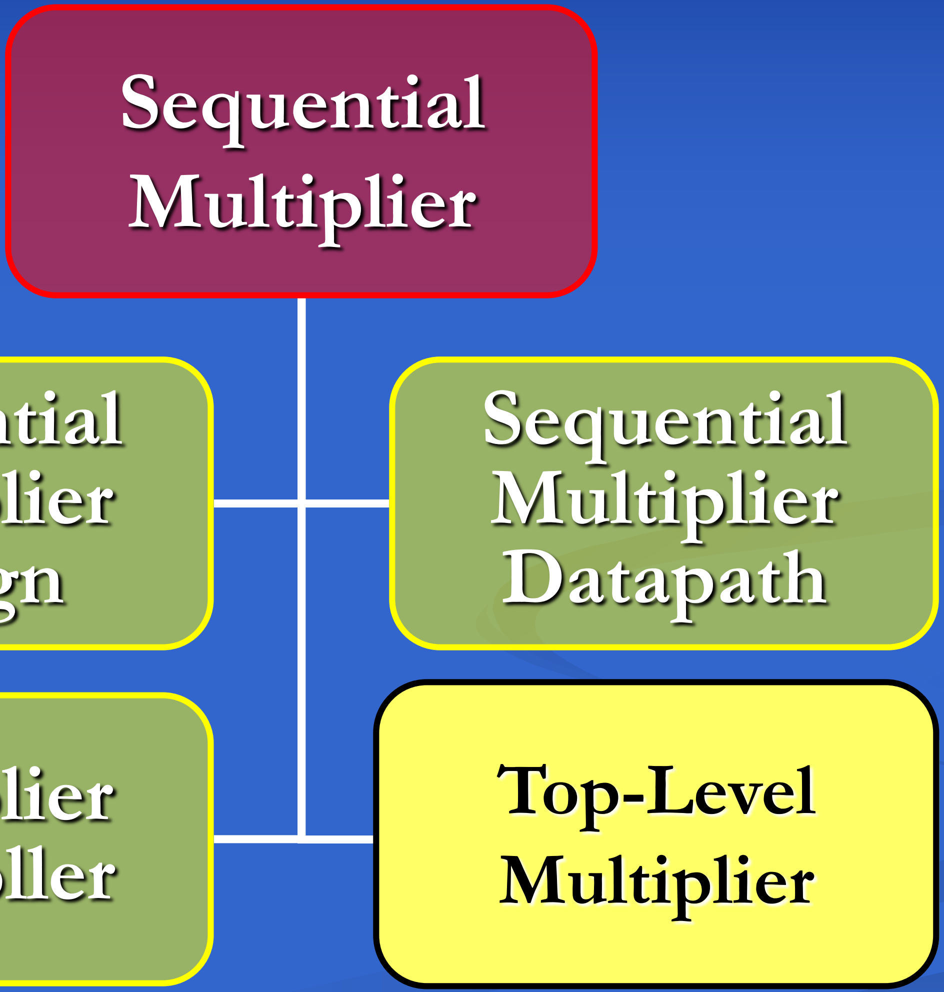
- Multiplier Controller

Multiplier Controller

```
      combinational: PROCESS (current, start, A0) BEGIN
            CASE current IS
                  . . . . . . . . . . . . . . . .
                WHEN rslt2 =>
                     msb_out <= '1';
                 WHEN OTHERS =>
                     clr_P <= '0'; load_P <= '0';
                     load_B <= '0'; msb_out <= '0';
                     lsb_out <= '0'; sel_sum <= '0';
                     load_A <= '0'; Shift_A <= '0';
                     done <= '0';
             END CASE;
     END PROCESS;
END ARCHITECTURE procedural;
```

- Multiplier Controller

Top-level Multiplier



Top-level Multiplier

```
ENTITY Multiplier IS
   PORT (clk, start : IN std_logic;
         databus : INOUT std_logic_vector (7 DOWNTO 0);
       lsb_out, msb_out, done : OUT std_logic);
END ENTITY;
--
ARCHITECTURE structural OF Multiplier IS
   SIGNAL clr_P, load_P, load_B, msb_out_t, A0 : std_logic;
   SIGNAL lsb_out_t, sel_sum, load_A, Shift_A : std_logic;
BEGIN
   . . . . . . . . . . . . .
   . . . . . . . . . . . . .
END ARCHITECTURE structural;
```

- Top-level Multiplier Module

Top-level Multiplier

```
ARCHITECTURE structural OF Multiplier IS
BEGIN
   dpu : ENTITY WORK.datapath(procedural)
         PORT MAP (clk, clr_P, load_P, load_B,
                   msb_out_t, lsb_out_t, sel_sum,
                   load_A, Shift_A, databus, A0 );
   cu : ENTITY WORK.controller(procedural)
        PORT MAP (clk, start, A0, clr_P, load_P, load_B,
                  msb_out_t, lsb_out_t, sel_sum,
                  load_A, Shift_A, done );
   msb_out <= msb_out_t;
   lsb_out <= lsb_out_t;
END ARCHITECTURE structural;
```

- Top-level Multiplier Module (Continued)

Booth Multiplier

- Booth algorithm is for signed number multiplication.
- The algorithm is similar to the sequential multiplication shift-and-add algorithm, except that two bits, instead of only one bit, will be considered for making shift, add, and subtract decisions.
- An extra bit (initially 0) is added to the right of A, and decisions for adding B to the partial product (P+B) and shifting, subtracting B from the partial product (P-B and shifting, or just shifting the partial product will be based on the right-most two bits of the extended A.

Booth Multiplier- Example

A×B

B=01101101

A=10110110

A is a negative number.

A: 101101100 : $+0 \times 2^0 =$ 000

101101100 : $-1 \times 2^1 =$ -002

101101100 : $-0 \times 2^2 =$ 000

101101100 : $+1 \times 2^3 =$ +008

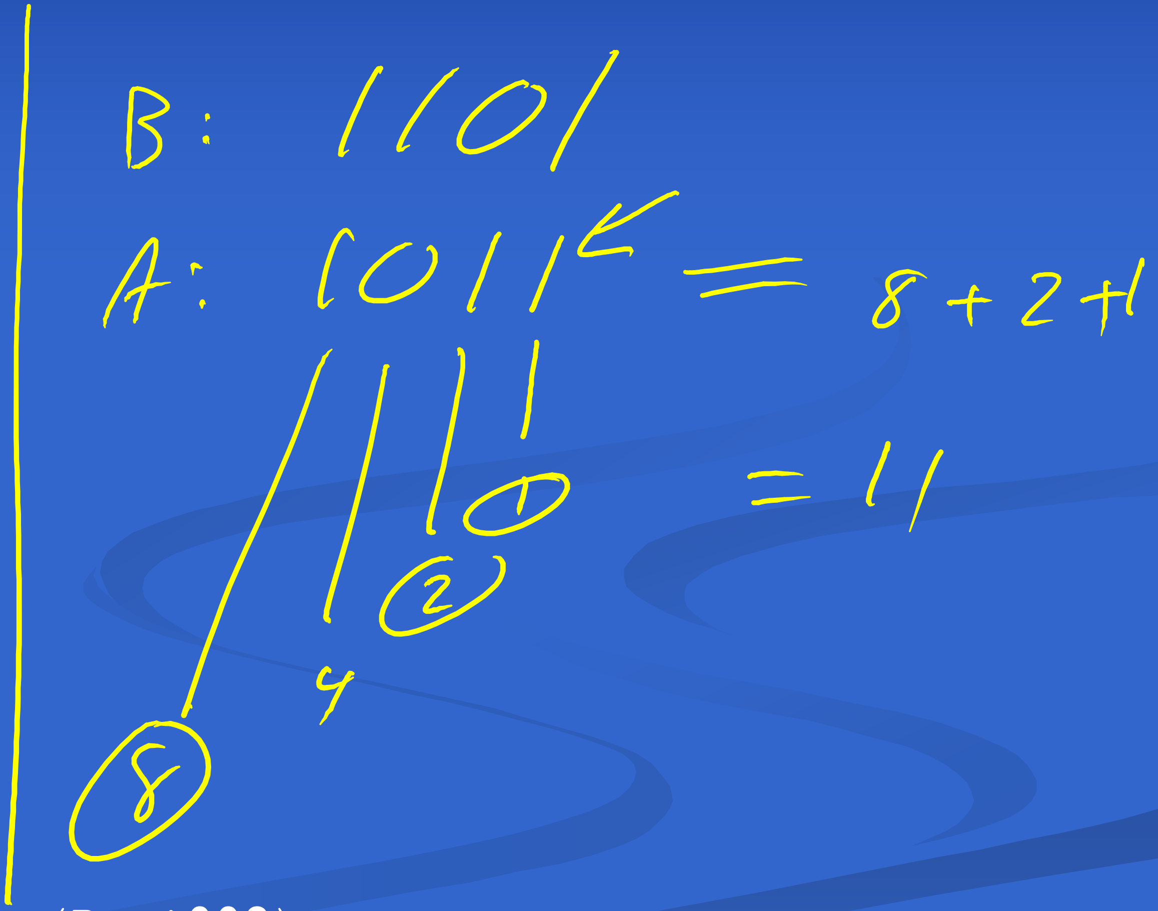
101101100 : $-1 \times 2^4 =$ -016

101101100 : $-0 \times 2^5 =$ 000

101101100 : $+1 \times 2^6 =$ +064

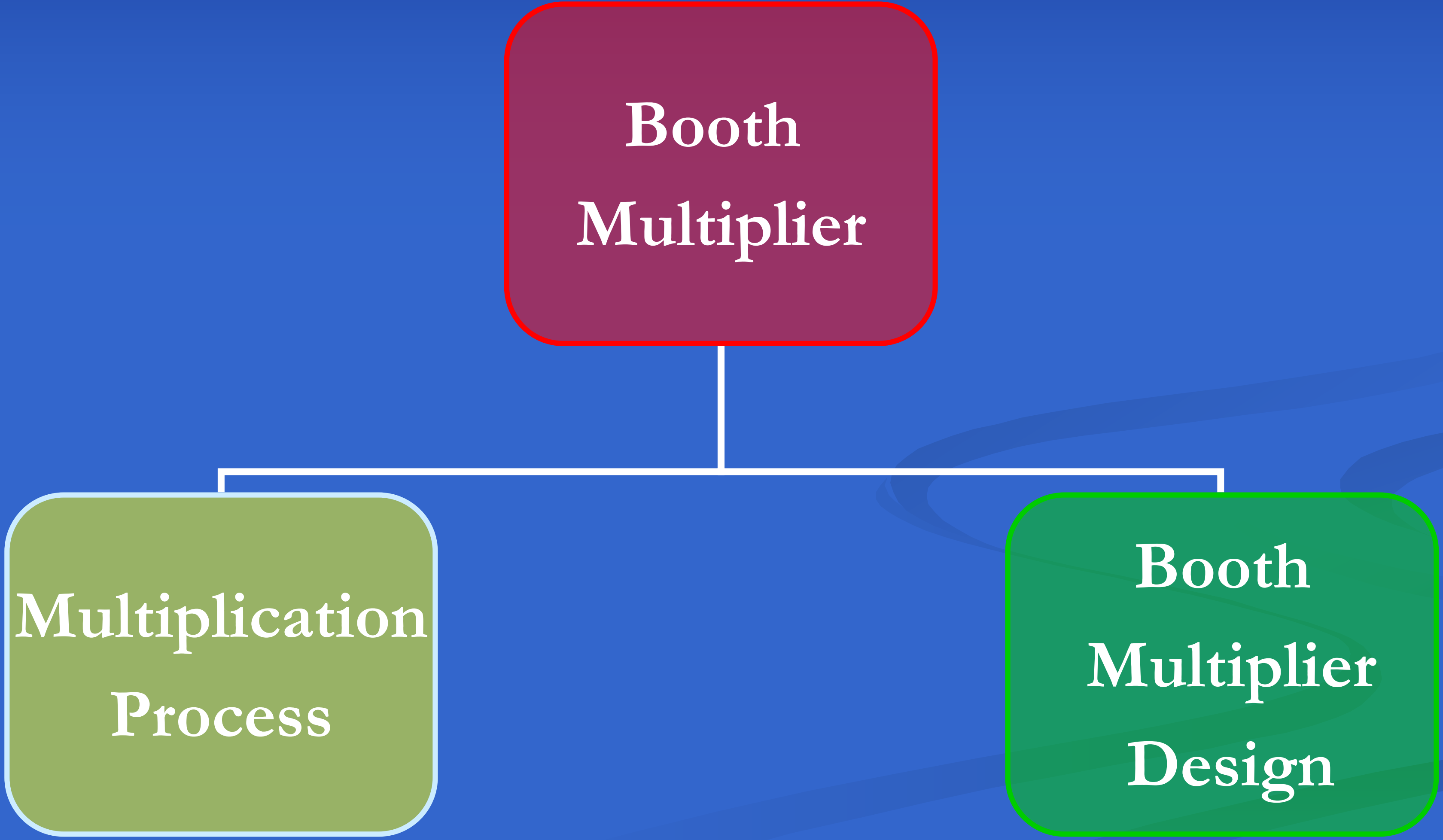
101101100 : $-1 \times 2^7 =$ -128

-074



A×B= (B× +000) + (B× -002) + (B× -000) + (B× +008) +
(B×-016) + (B× -000) + (B× +064) + (B× -128)

Booth Multiplier



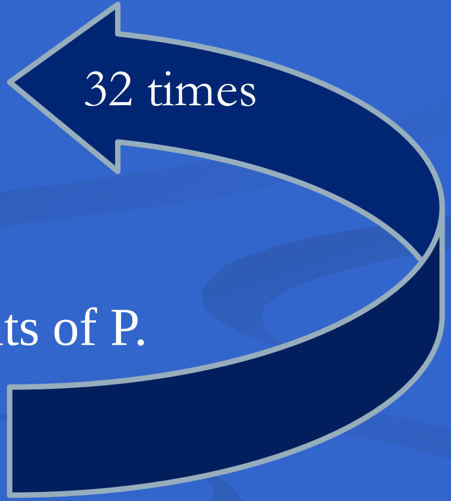
Multiplication Process

- Initialization:
  - Two registers Mcand containing first operand and P containing the result
    Mcand is 32-bit and P is 65-bit register
  - Mcand = A, P = {32'b0, B, 1'b0}

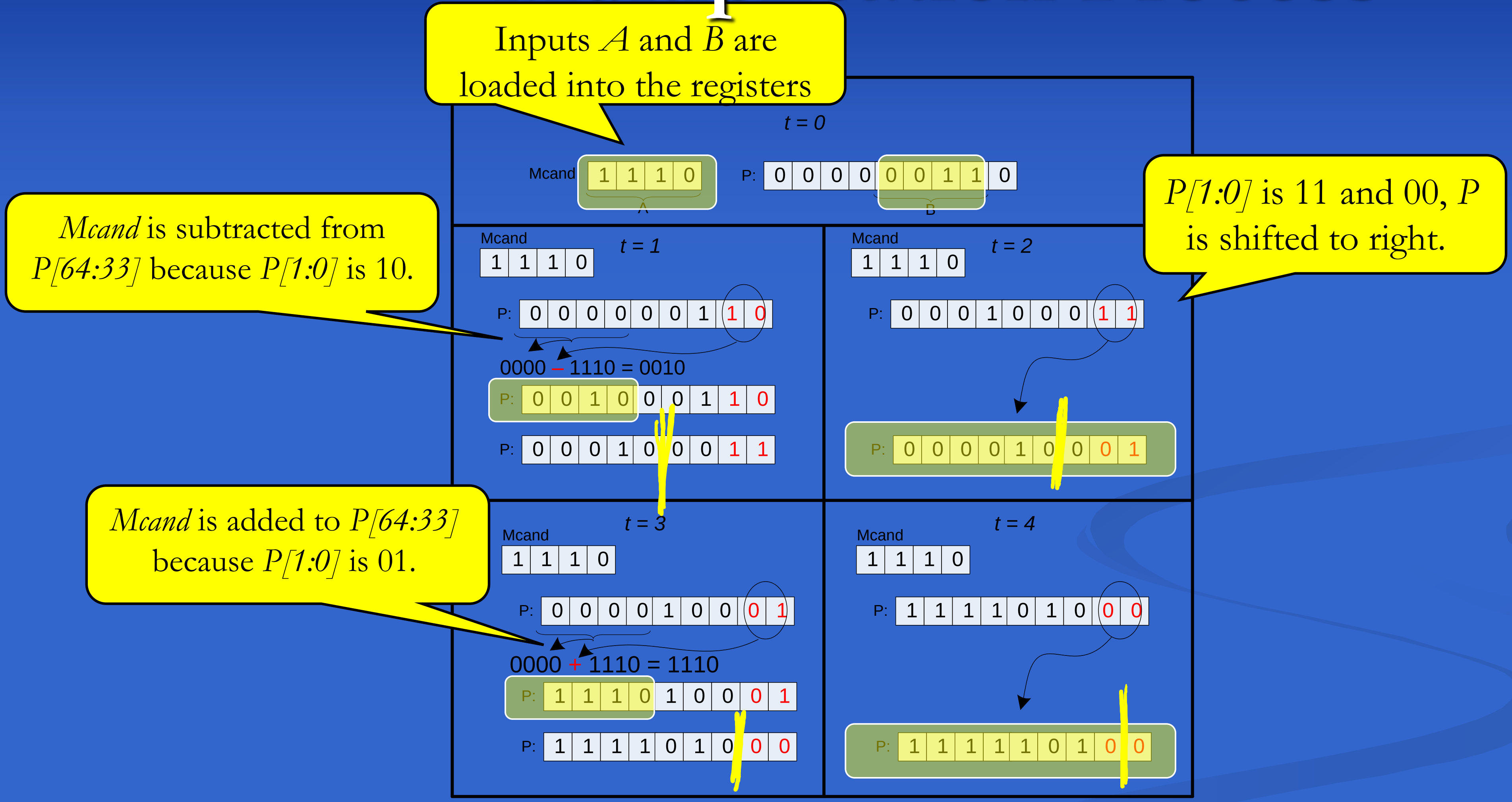
- step1: Check the two lowest bits of P
  - 11 or 00: go to step 3.
  - 01 or 10: go to step 2
- step2:
  - LSBs of P: 01 => Mcand is added to the most 32 bits of P.
  - LSBs of P: 10 => Mcand is subtracted from the most 32 bits of P.

- step3: Shift P one place to the right

- End of Multiplication: P[64:1] contains the result.

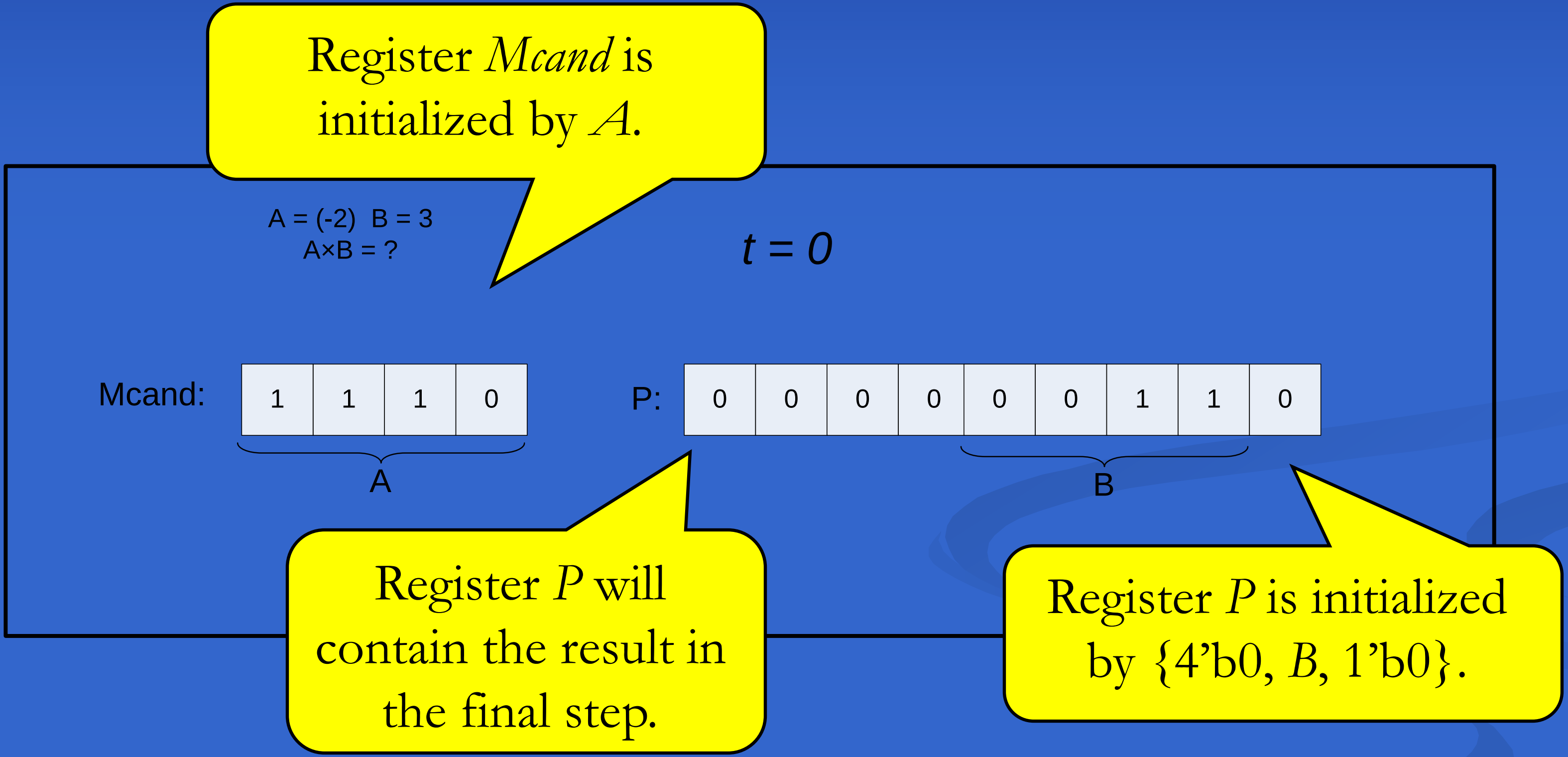


Multiplication Process



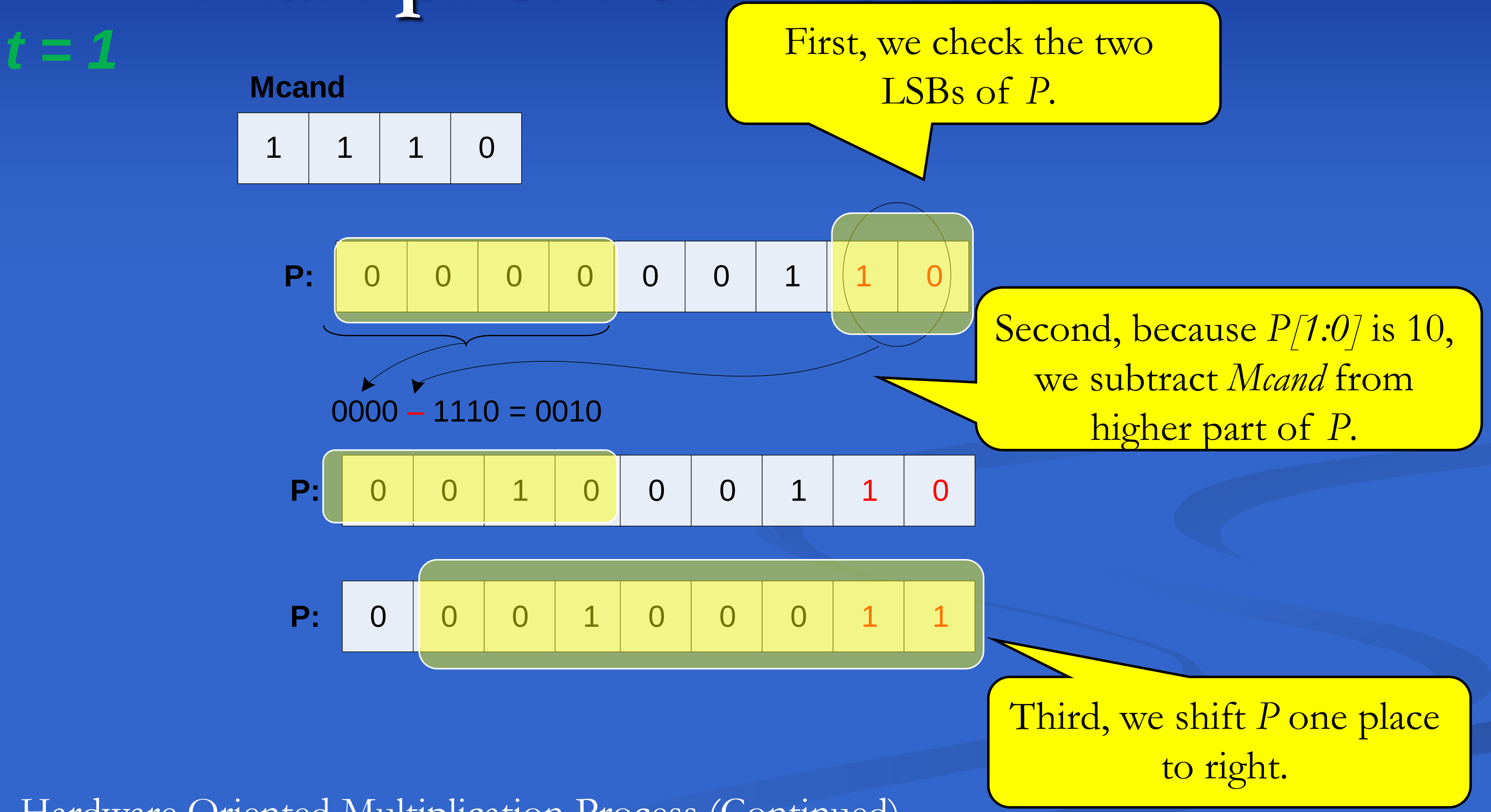
- Hardware Oriented Multiplication Process (continued)

Multiplication Process


Register *Mcand* is initialized by *A*.
A = (-2) B = 3
A×B = ?
t = 0
Mcand:
1 1 1 0
A
P:
0 0 0 0 0 0 1 1 0
B
Register *P* will contain the result in the final step.
Register *P* is initialized by {4'b0, *B*, 1'b0}.

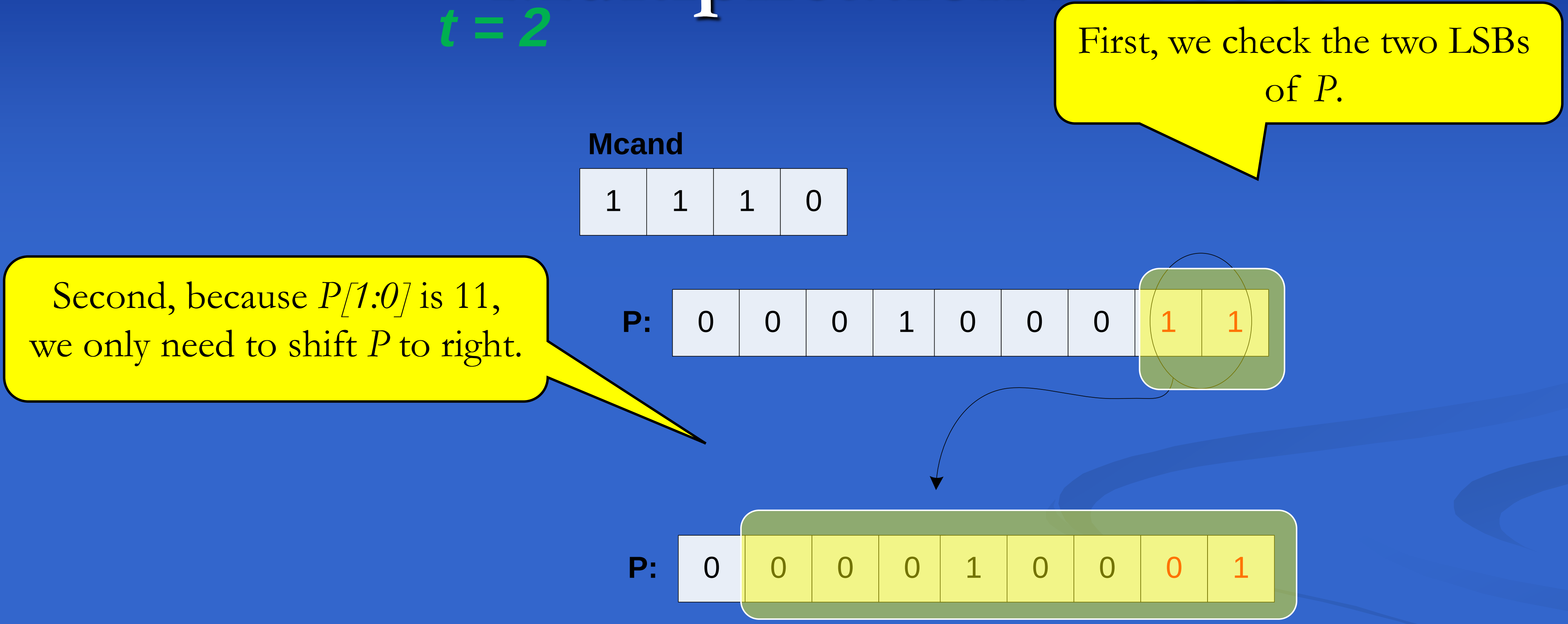
- Hardware Oriented Multiplication Process (Continued)

Multiplication Process


t = 1
Mcand
1 1 1 0
P: 0 0 0 0 0 0 1 1 0
First, we check the two LSBs of P.
0000 – 1110 = 0010
Second, because P[1:0] is 10, we subtract Mcand from higher part of P.
P: 0 0 1 0 0 0 1 1 0
P: 0 0 0 1 0 0 0 1 1
Third, we shift P one place to right.

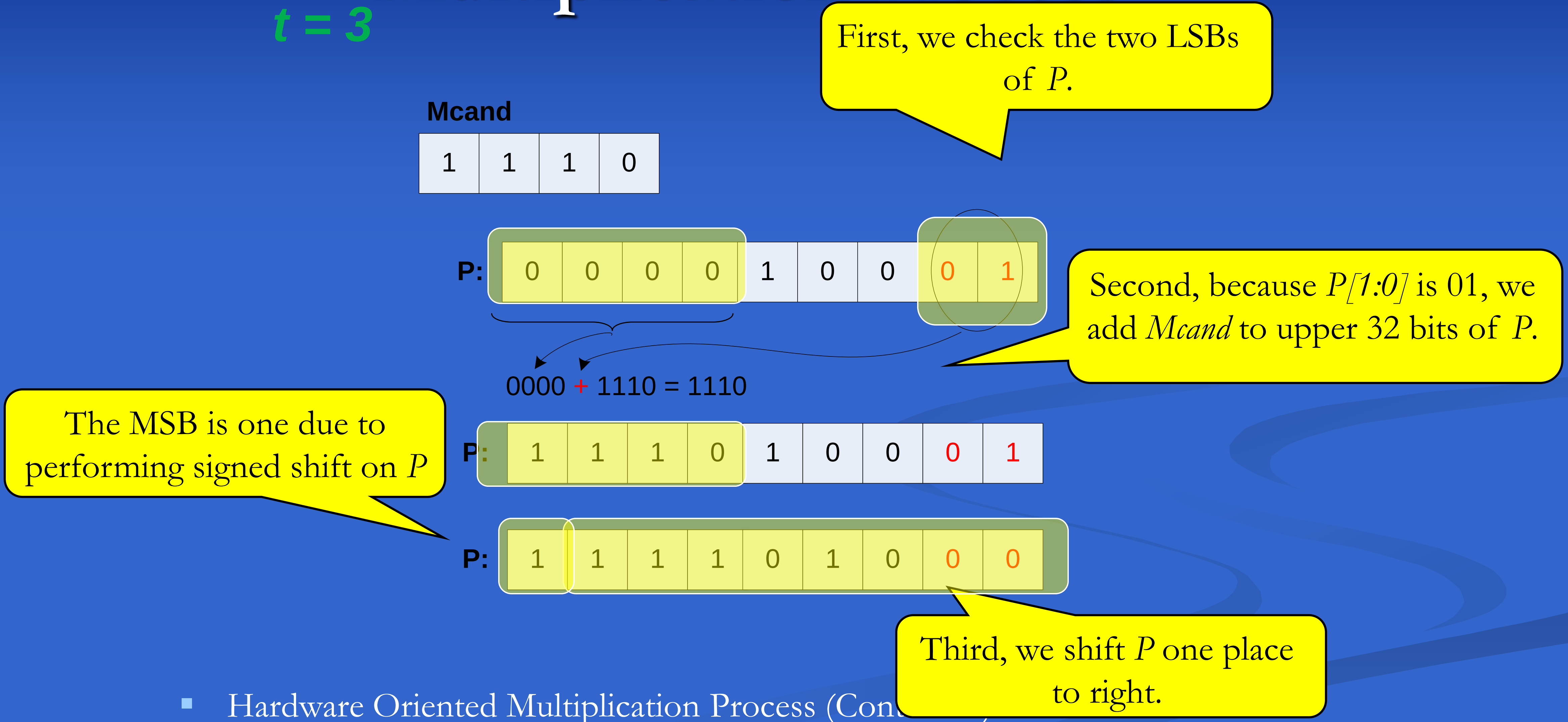
- Hardware Oriented Multiplication Process (Continued)

Multiplication Process


t = 2
First, we check the two LSBs of *P*.
Mcand
1 1 1 0
Second, because *P[1:0]* is 11, we only need to shift *P* to right.
P: 0 0 0 1 0 0 0 1 1
P: 0 0 0 0 1 0 0 0 1

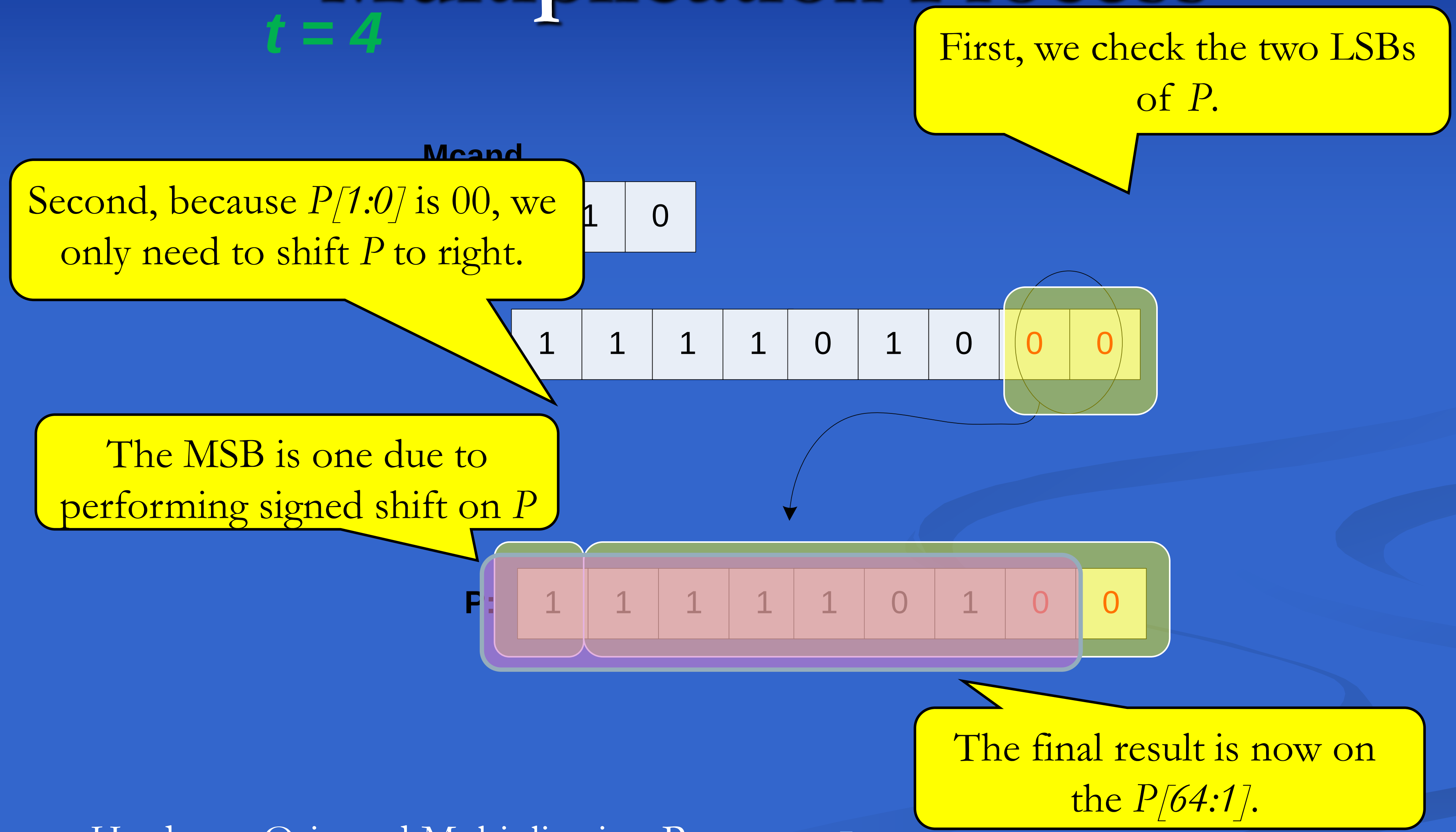
- Hardware Oriented Multiplication Process (Continued)

Multiplication Process


t = 3
First, we check the two LSBs of P.
Mcand
1 1 1 0
P: 0 0 0 0 1 0 0 0 1
Second, because P[1:0] is 01, we add Mcand to upper 32 bits of P.
0000 + 1110 = 1110
The MSB is one due to performing signed shift on P
P: 1 1 1 0 1 0 0 0 1
P: 1 1 1 1 0 1 0 0 0
Third, we shift P one place to right.

- Hardware Oriented Multiplication Process (Con

Multiplication Process



First, we check the two LSBs of *P*.

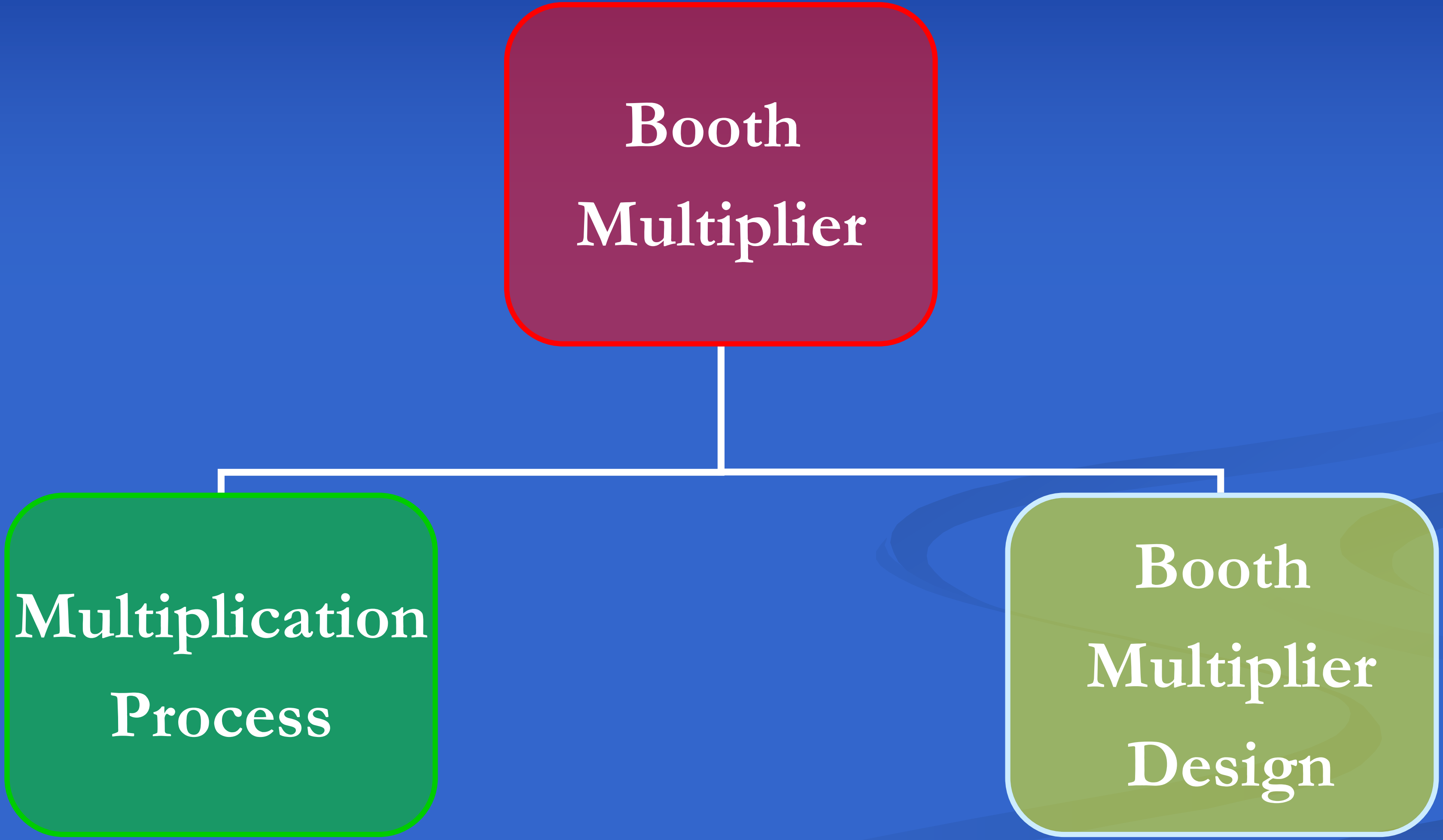
Second, because *P[1:0]* is 00, we only need to shift *P* to right.

The MSB is one due to performing signed shift on *P*

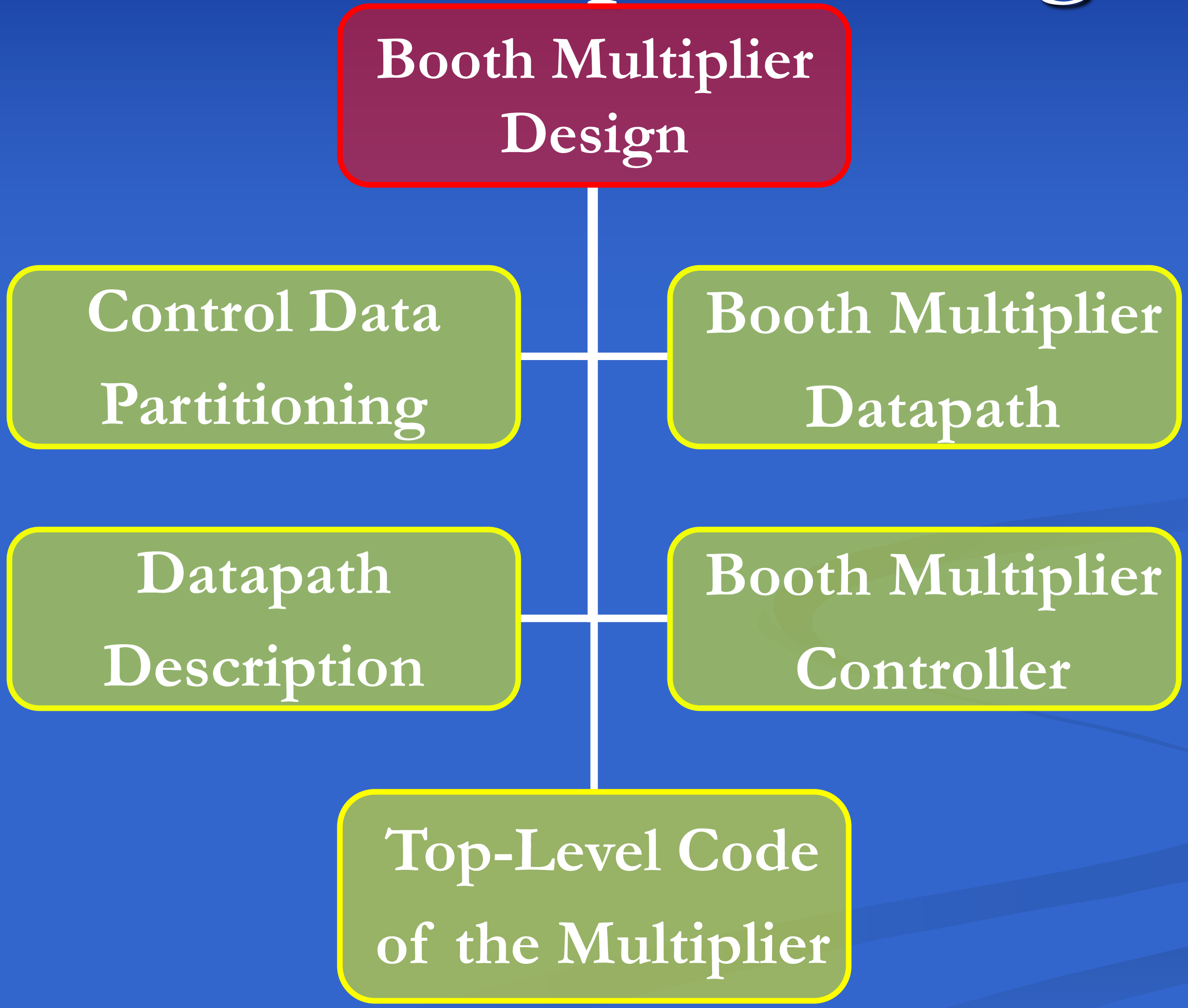
The final result is now on the *P[64:1]*.

Hardware Oriented Multiplication Process

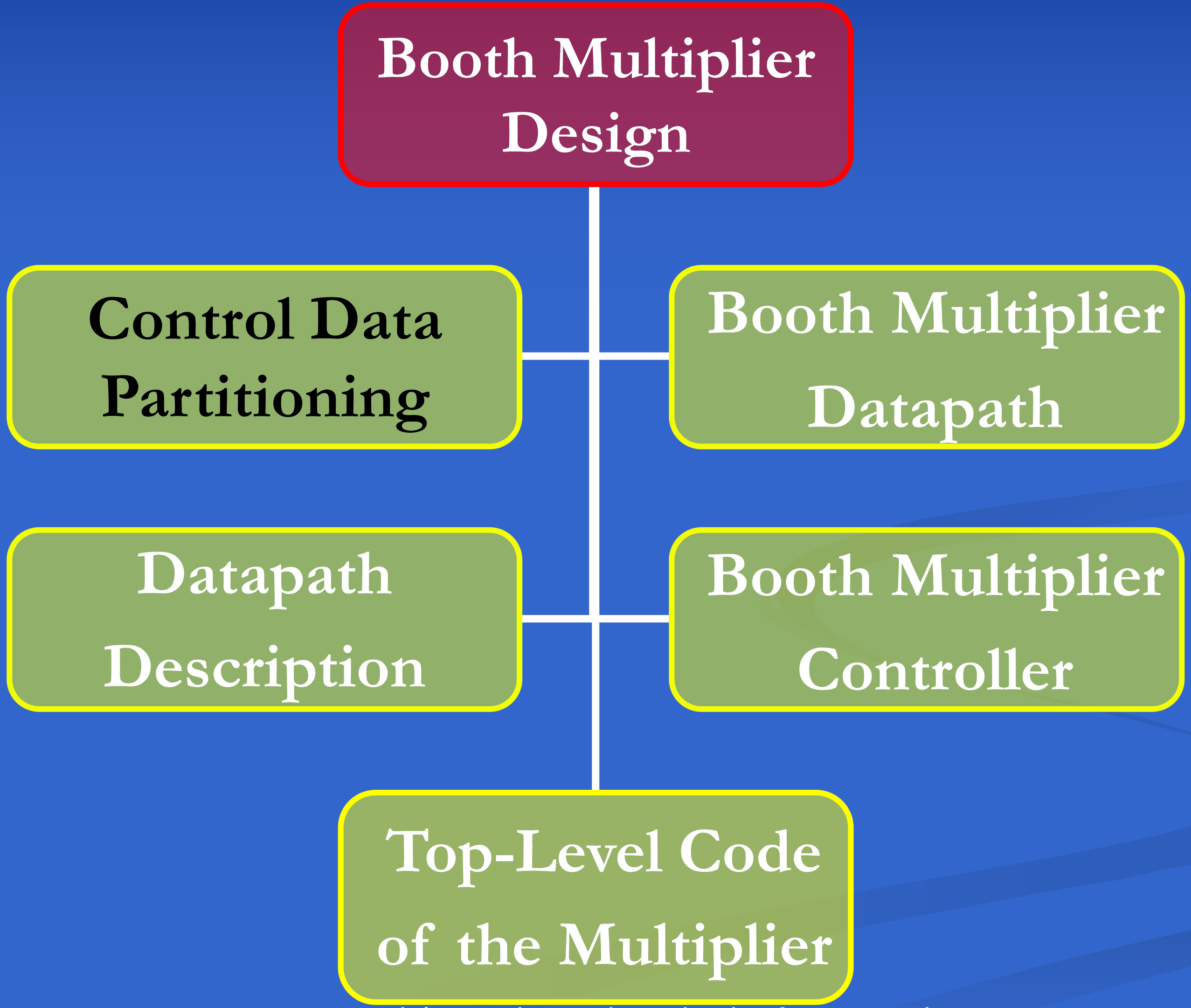
Booth Multiplier



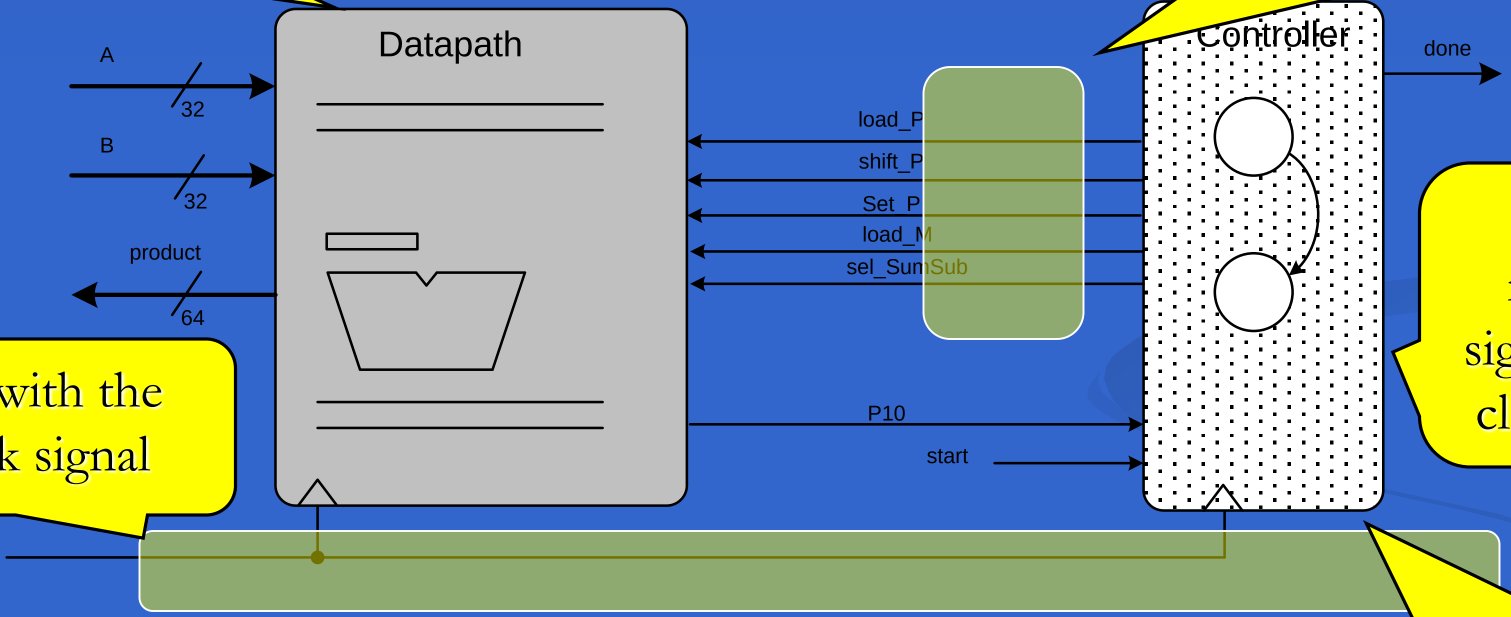
Booth Multiplier Design


Booth Multiplier Design
Control Data Partitioning
Booth Multiplier Datapath
Datapath Description
Booth Multiplier Controller
Top-Level Code of the Multiplier

Booth Multiplier Design


Booth Multiplier Design
Control Data Partitioning
Booth Multiplier Datapath
Datapath Description
Booth Multiplier Controller
Top-Level Code of the Multiplier

Control Data Partitioning



Data part consists of registers, logic units, and their interconnecting buses.

In each new state, several control signals are issued, and the components of the datapath start reacting to these signals.

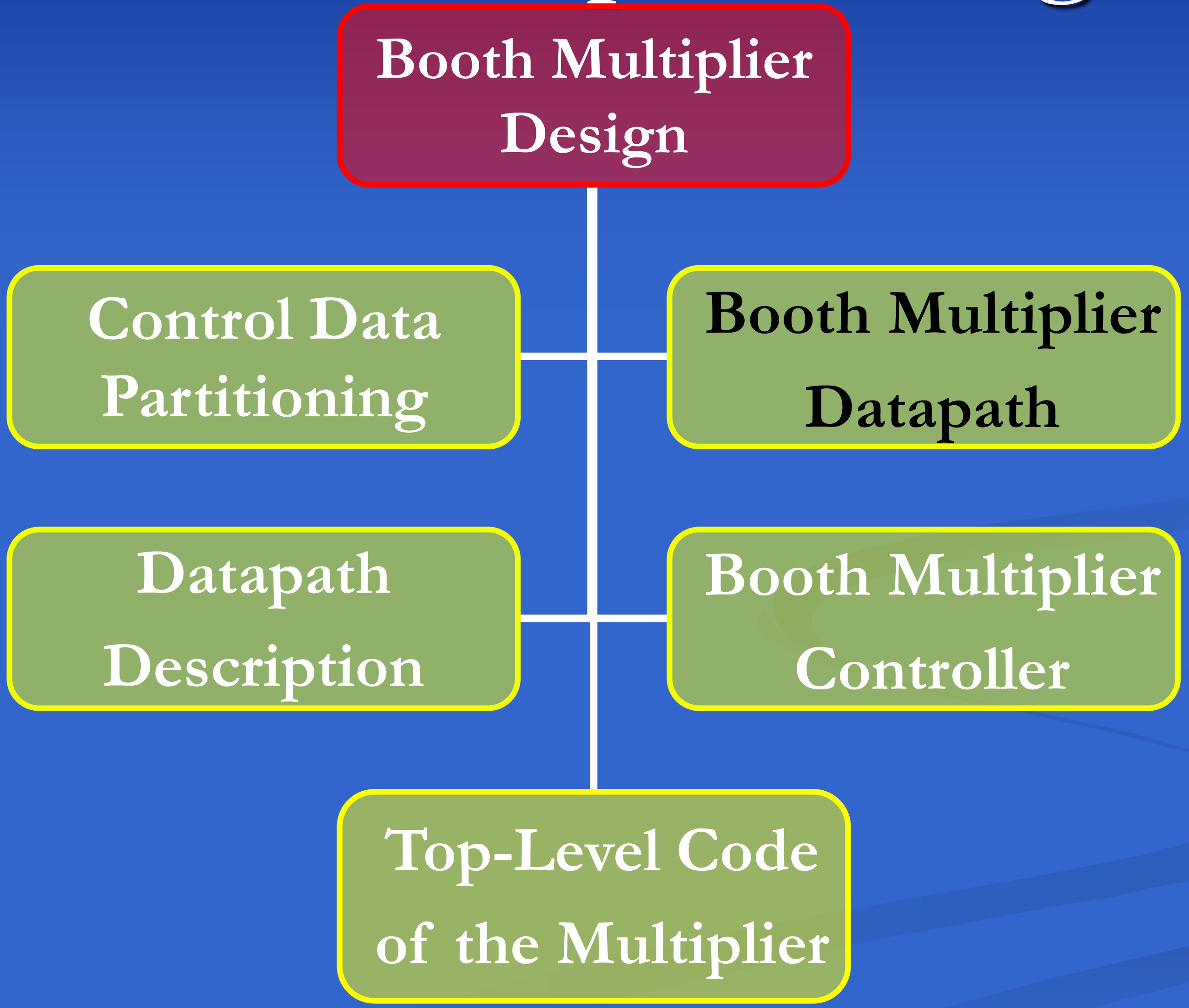
The controller is a state machine that issues control signals for control of what gets clocked into the data registers.

Triggered with the same clock signal

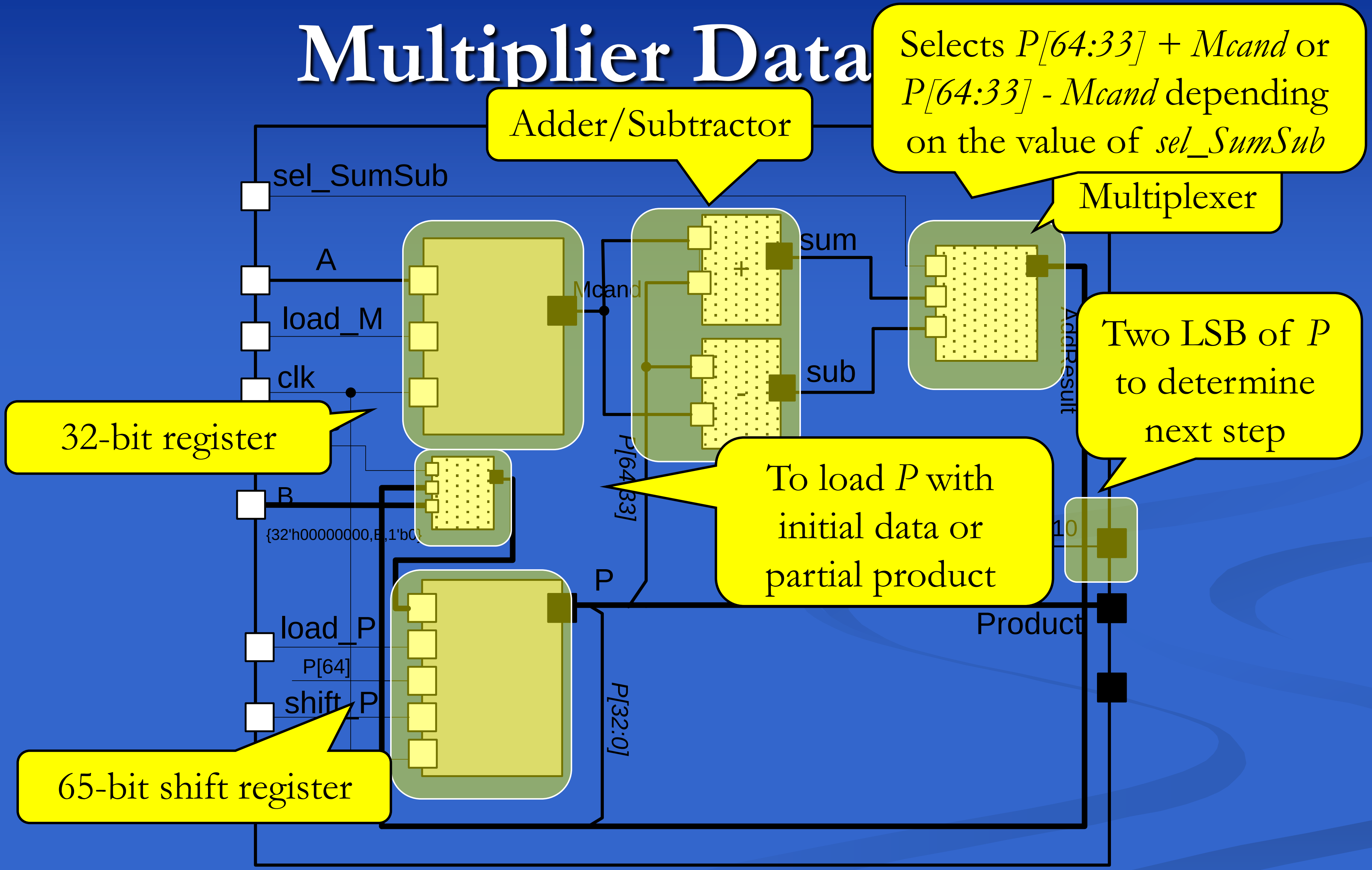
On the rising edge of the system clock, the controller goes into a new state.

- Datapath and Controller

Booth Multiplier Design

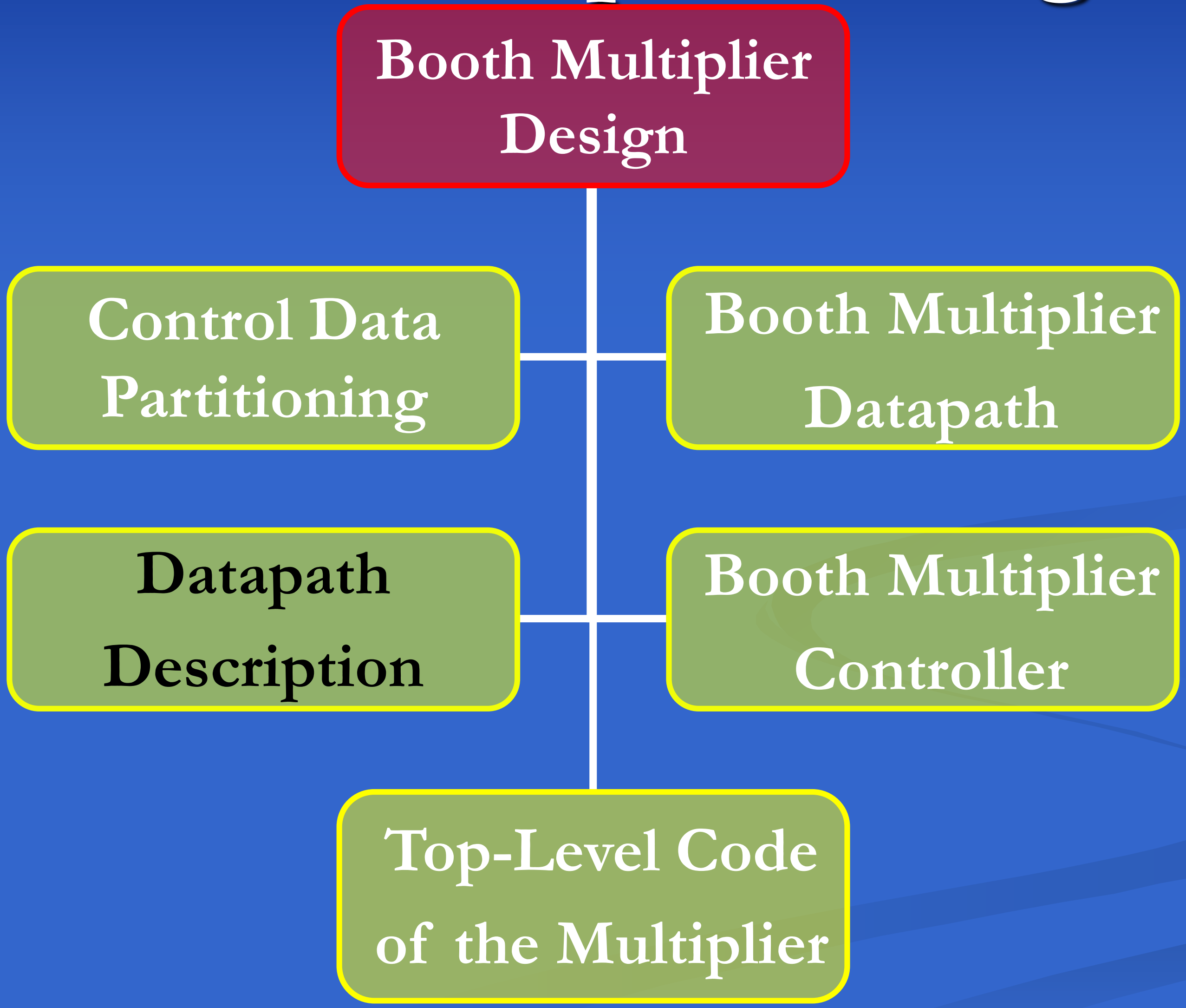

Booth Multiplier Design
Control Data Partitioning
Booth Multiplier Datapath
Datapath Description
Booth Multiplier Controller
Top-Level Code of the Multiplier

Multiplier Data


Adder/Subtractor
Selects *P[64:33]* + *Mcand* or *P[64:33]* - *Mcand* depending on the value of *sel_SumSub*
Multiplexer
sel_SumSub
A
load_M
clk
Mcand
sum
sub
AddResult
32-bit register
B
{32'h00000000,B,1'b0}
P[64:33]
To load *P* with initial data or partial product
Two LSB of *P* to determine next step
10
P
Product
load_P
P[64]
shift_P
P[32:0]
65-bit shift register

- Multiplier Block Diagram

Booth Multiplier Design


Booth Multiplier Design
Control Data Partitioning
Booth Multiplier Datapath
Datapath Description
Booth Multiplier Controller
Top-Level Code of the Multiplier

Datapath Description

```
ENTITY datapath is
      PORT( clk, Set_P, load_M, load_P,
            shift_P, preset, sel_SumSub : IN std_logic;
            A, B : IN std_logic_vector (31 downto 0);
            Product : OUT std_logic_vector (63 downto 0);
            P10 : OUT std_logic_vector (1 downto 0));
END ENTITY;

ARCHITECTURE procedural OF datapath IS

      SIGNAL sum, sub, AddResult, Mcand : std_logic_vector ( 31 downto 0);
      SIGNAL data, P : std_logic_vector (64 downto 0);
      SIGNAL ls : std_logic_vector (1 downto 0);

BEGIN
```

- Datapath Code

Represents 32-bit register *Mcand* for keeping input *A*

Datapath Description

```
PROCESS (clk) BEGIN
        IF(clk = '0' AND clk'EVENT) THEN
                IF (load_M = '1') THEN Mcand <= A;
                END IF;
        END IF;
END PROCESS;
PROCESS (clk) BEGIN
        IF(clk = '0' AND clk'EVENT) THEN
                CASE (ls)  IS
                        WHEN "01" => P <=  P(64) & P(64 DOWNTO 1);
                        WHEN "10" => P <= data;
                        WHEN OTHERS => P <= P;
                END CASE;
        END IF;
END PROCESS;
..............................
```

Implements the 65-bit shift-register for keeping the result and partial products

- Datapath Code (continued)

Datapath Description

Two LSB of *P* going to controller to determine the next step

```
P10 <= P(1 downto 0);
sum <= P(64 downto 33) + Mcand;
sub <= P(64 downto 33) - Mcand;

AddResult <= sum WHEN sel_SumSub = '1' ELSE sub;
data <= ((“00000000000000000000000000000000”) & B & '0') WHEN Set_P = '1'
                    ELSE (AddResult & P(32 downto 0)) ;

Product <= P(64 downto 1);

ls <= load_P & shift_P;

END ARCHITECTURE procedural;
```

The *Mcand* is added or subtracted to/from upper 32 bits of *P*

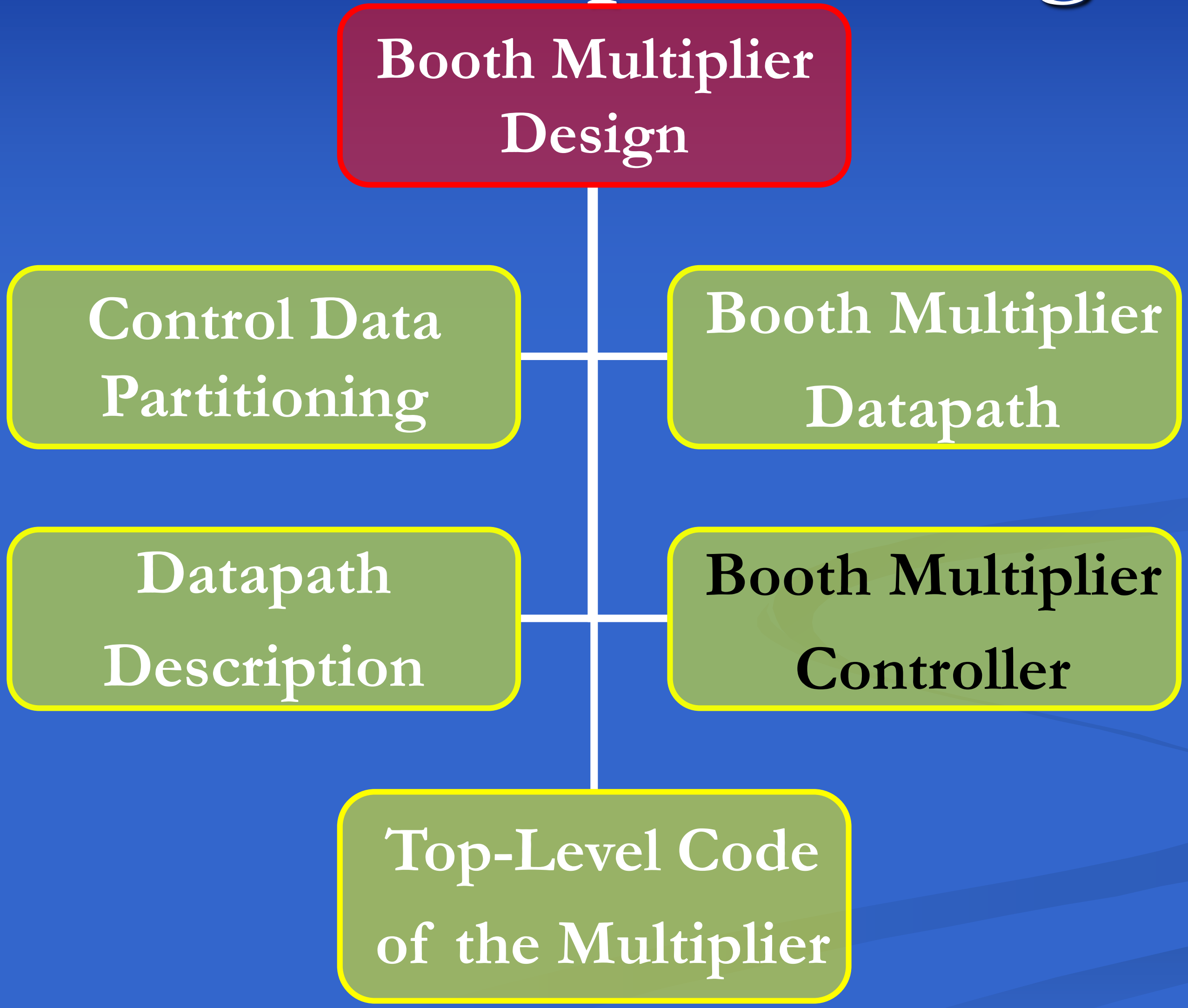
sel_SumSub selects *sum* or *sub* to store in the *P*

Iinitial data or intermediate data going to *P* based on *Set_P*

The final result will be placed on the *Product*

- Datapath Code (continued)

Booth Multiplier Design


Booth Multiplier Design
Control Data Partitioning
Booth Multiplier Datapath
Datapath Description
Booth Multiplier Controller
Top-Level Code of the Multiplier

Multiplier Controller

States of Multiplier

```
idle
Check_2bits
Sum_Sub
Shift
```

Multiplier waits for *start* while loading *M* and *P*

Checking 2 LSB of *P* to specify the next step

Multiplier adds or subtracts *Mcand* to/from *P[64:33]* based on *P10*

Multiplier shifts *P* one place to right

- Multiplier Control States

Multiplier Controller

```
ENTITY controller IS PORT (clk, start : IN std_logic;
                          P10 : IN std_logic_vector (1 downto 0);
                          Set_P, load_M, load_P : OUT std_logic;
                          shift_P, sel_SumSub, done : OUT std_logic);
END ENTITY;

ARCHITECTURE procedural OF controller IS
      TYPE state IS (idle, Check_2bits, Sum_Sub, Shift);
      SIGNAL current, nextState : state;
      SIGNAL CntValue : std_logic_vector (5 downto 0);
      SIGNAL preset, cntEn, cntZero : std_logic;
BEGIN
..............................
```

States of multiplier

- Controller Code

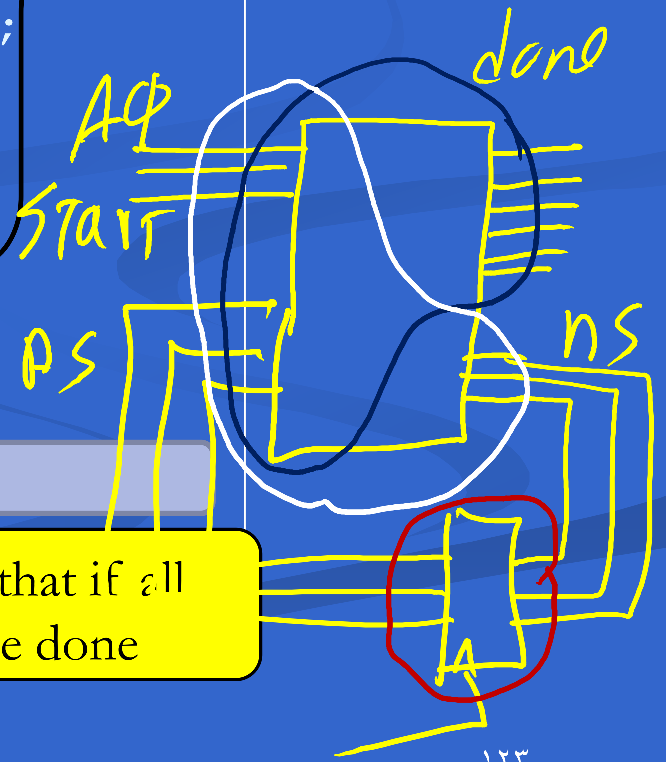
Multiplier Controller

```
PROCESS (clk) BEGIN
        IF(clk = '0' AND clk'EVENT) THEN
                IF (preset = '1') THEN
                        CntValue <= "100000";
                ELSE
                        IF (cntEn = '1') THEN
                                CntValue <= CntValue - 1;
                        ELSE
                                CntValue <= CntValue;
                        END IF;
                END IF;
        END IF;
END PROCESS;

cntZero <= '1' WHEN (CntValue = "000000") ELSE '0';
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Implements the 6-bit down counter for counting number of steps

Indicates that if all steps are done

- Controller Code



Multiplier Controlle

```
PROCESS (current, start, P10) BEGIN
  Set_P <= '0';       load_M <= '0';    load_P <= '0';
  shift_P <= '0';     sel_SumSub <='0'; cntEn <= '0';
  done <= '0';        preset <= '0';
CASE ( current ) IS
        WHEN idle =>
                IF (start = '0') THEN
                        done <= '1';
                ELSE
                        Set_P <= '1'; load_P <= '1';
                        load_M <= '1'; preset <= '1';
                END IF;
        WHEN Sum_Sub =>
                IF (P10 = "01") THEN
                        load_P <= '1'; sel_SumSub <= '1';
                ELSIF (P10 = "10") THEN
                        load_P <= '1';
                END IF;
```

Process block to issue control signals and make state transitions

All control signal outputs are set to their inactive values.

To initialize *P* register

To load *Mcand*

To preset the counter to 32

If *P10* is 01, *sel_SumSub* is one, indicating that *Mcand* should be added to upper bits of *P*

If *P10* is 10, *sel_SumSub* remains zero, indicating that *Mcand* should be subtracted from upper bits of *P*

Controller Code (continued)

Multiplier Controller

P will be shifted to the right and counter will count down, showing that one step is done.

```
        WHEN Shift =>
               cntEn <= '1';
               shift_P <= '1';
         WHEN OTHERS =>
               Set_P <= '0';     load_M <= '0';     load_P <= '0';
                shift_P <= '0';   sel_SumSub <='0'; cntEn <= '0';
                done <= '0';      preset <= '0';
END CASE;
END PROCESS;
```

- Controller Code (continued)

Multiplier Controller

```
PROCESS (current, start, P10, cntZero) BEGIN
      CASE ( current ) IS
            WHEN idle =>
                  IF (start = '0') THEN
                        nextState <= idle;
                  ELSE
                        nextState <= Check_2bits;
                  END IF;
            WHEN Check_2bits =>
                  IF (cntZero = '0') THEN
                        IF (P10 = "00" or P10 = "11") THEN
                              nextState <= Shift;
                        ELSIf(P10 = "10" or P10 = "01") THEN
                              nextState <= Sum_Sub;
                        END IF;
                  ELSE
                        nextState <= idle;
                  END IF;
```

Another always block for state transition

If *P10* is 11 or 00, *P* should only be shifted, otherwise, *Mcand* should be added or subtracted to/from it before shifting operation.

To check the number of multiplication steps

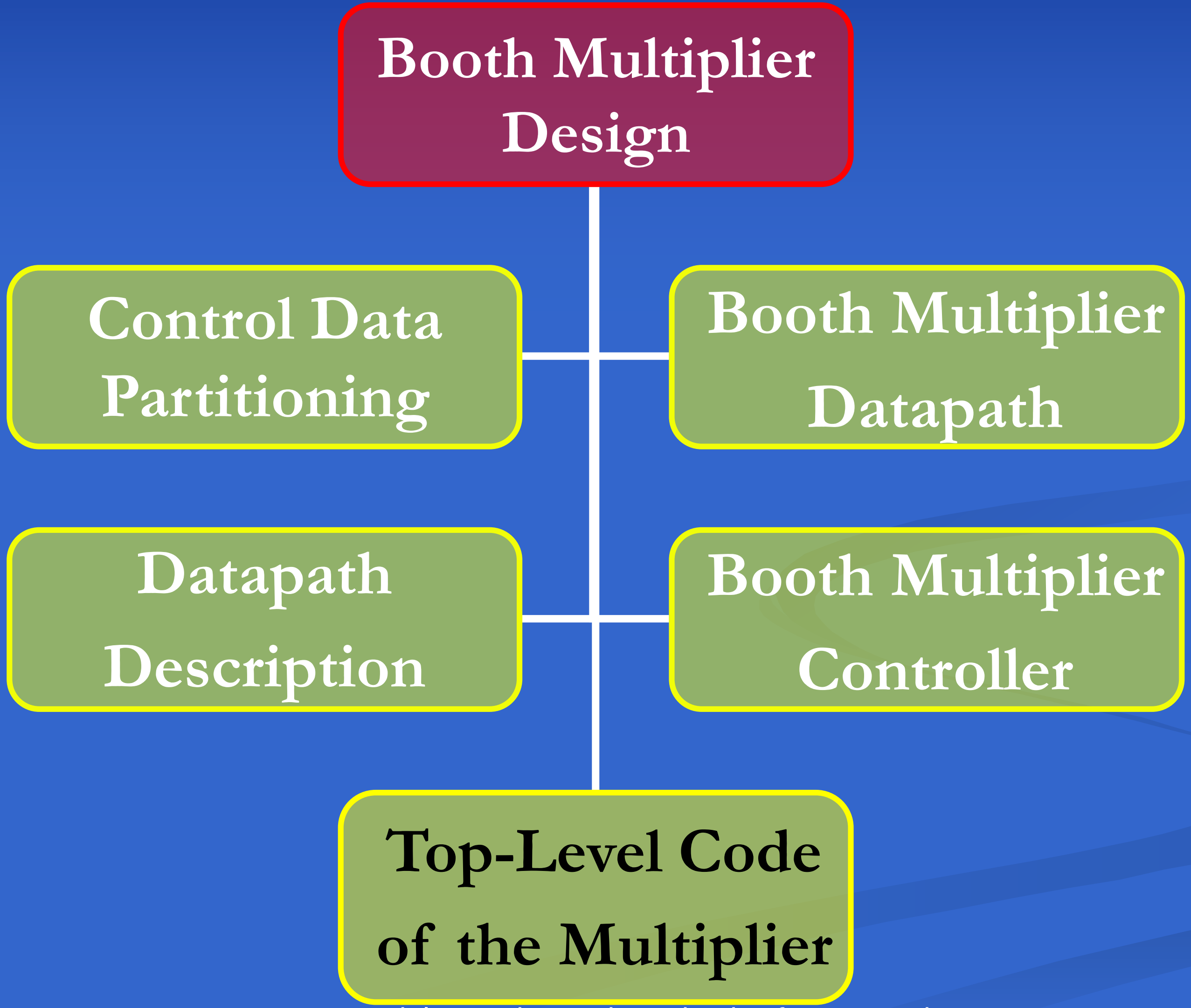
- Controller Code

Multiplier Controller

```
        WHEN Sum_Sub =>
                nextState <= Shift;
        WHEN Shift =>
                nextState <= Check_2bits;
        END CASE;

END PROCESS;

PROCESS(clk) BEGIN
        IF( clk = '1' and clk'event ) THEN
                current <= nextState;
        END IF;
END PROCESS;

END ARCHITECTURE procedural;
```

- Controller Code

Booth Multiplier Design


Booth Multiplier Design
Control Data Partitioning
Booth Multiplier Datapath
Datapath Description
Booth Multiplier Controller
Top-Level Code of the Multiplier

Booth Multiplier
VHDL Implementation

```
ENTITY booth_mult IS
   PORT (mc, mp : IN std_logic_vector (7 downto 0);
         clk, start : IN std_logic;
         prod : OUT std_logic_vector (15 downto 0);
         busy : OUT boolean);
END ENTITY booth_mult;
ARCHITECTURE behavioral OF booth_mult IS
   SIGNAL A, M : std_logic_vector (mc'RANGE);
   SIGNAL Q : std_logic_vector (mc'LENGTH DOWNTO 0);
   SIGNAL sum, dif : std_logic_vector(mc'RANGE);
   SUBTYPE cnt IS INTEGER RANGE 0 TO mc'LENGTH;
   SIGNAL count : cnt := 0;
BEGIN
   . . . . . . . . . . . . .
END ARCHITECTURE behavioral;
```

- Booth Algorithm VHDL Code

Booth Multiplier VHDL Implementation

```
BEGIN
   sum <= A + M;
   dif <= A - M;
   prod <= A & Q (mc'LENGTH DOWNTO 1);
   busy <= (count < mc'LENGTH);
   Counter: PROCESS (clk) BEGIN
     IF (clk = '1' AND clk'EVENT) THEN
       IF (start = '1') THEN count <= 0;
       ELSIF (count < mc'LENGTH) THEN count<= count + 1;
         END IF;
      END IF;
   END PROCESS;
   . . . . . . . . . . . . . . . . . .
END ARCHITECTURE behavioral;
```

- Booth Algorithm VHDL Code (Continued)

Booth Multiplier VHDL Implementation

```
RegClocking: PROCESS (clk) BEGIN
    IF (clk = '1' AND clk'EVENT) THEN
        IF (start = '1') THEN
            A <= (OTHERS => '0');
            M <= mc;
            Q <= mp & '0';
        ELSIF (count < mc'LENGTH) THEN
        . . . . . . . . . . . . . . .
        . . . . . . . . . . . . . . .
        END IF;
    END IF;
END PROCESS;
```

- Booth Algorithm VHDL Code (Continued)

Booth Multiplier VHDL Implementation

```
        . . . . . . . . . . . . . . .
        ELSIF (count < mc'LENGTH) THEN
            CASE Q(1 DOWNTO 0) IS
              WHEN "01" =>           --ADD AND SHIFT
                Q <= sum(0) & Q(Q'LEFT DOWNTO 1);
                A <= sum(sum'LEFT) &
                     sum(sum'LEFT DOWNTO 1);
              WHEN "10" =>           --SUBTRACT AND SHIFT
                Q <= dif(0) & Q(Q'LEFT DOWNTO 1);
                A <= dif(dif'LEFT) &
                     dif(dif'LEFT DOWNTO 1);
              WHEN OTHERS =>         --SHIFT ONLY
                Q <= A(0) & Q(Q'LEFT DOWNTO 1);
                A <= A(A'LEFT) & A(A'LEFT DOWNTO 1);
            END CASE;
          END IF;
        END IF;
      END PROCESS;
```

Handshaking

- Completely independent from data handling
- Wrappers and Interfacing Utilities help to implement handshaking
- Simple handshaking
- Register file
- Data 32-bit block

Types of handshaking

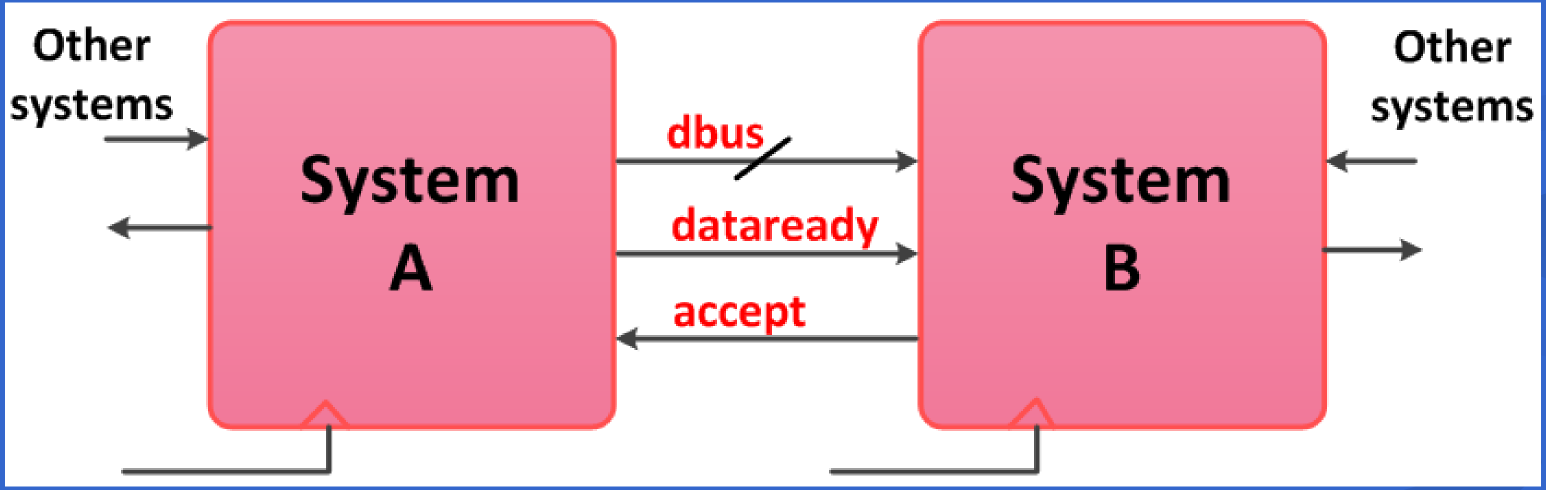
- Handshaking between two systems
- Handshaking for accessing a shared bus
- Memory handshaking
- DMA mode or burst mode

Why Handshaking

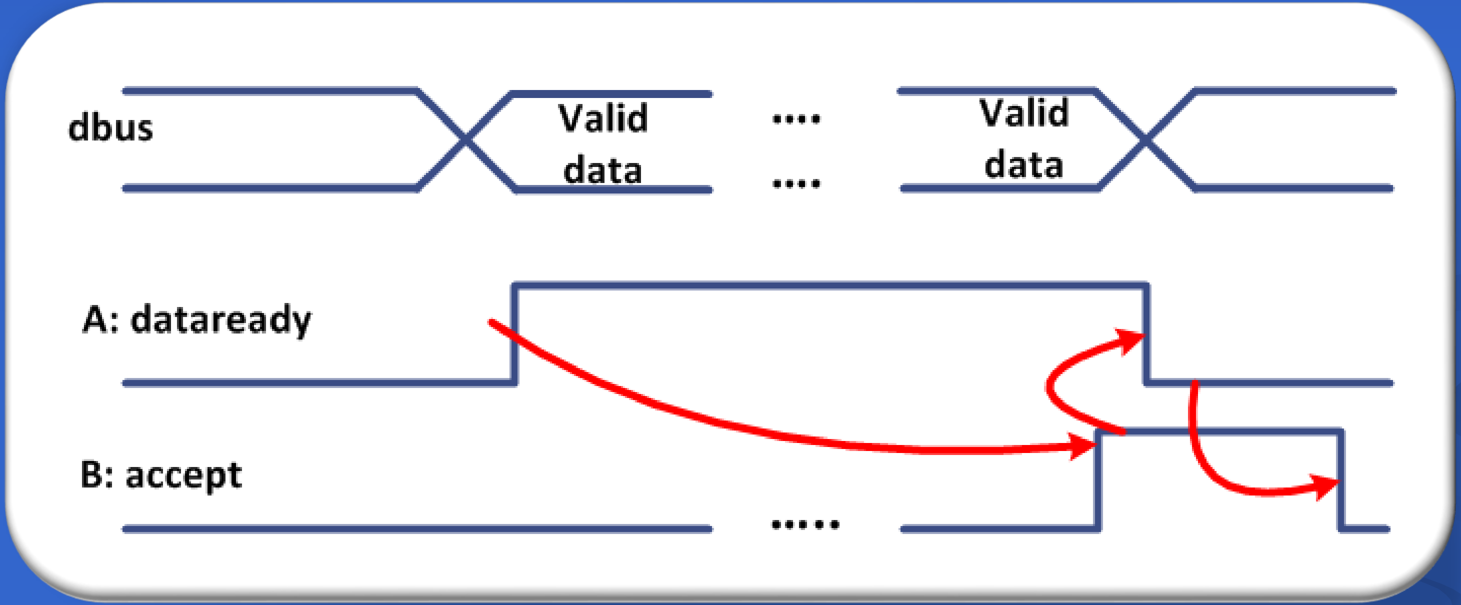
- Two systems want to communicate data and they don’t necessarily have the same timing
- The systems have to send some signals before the actual data is transmitted
- Handshaking implementation is a part of the control of the system

Between Two Systems Handshaking

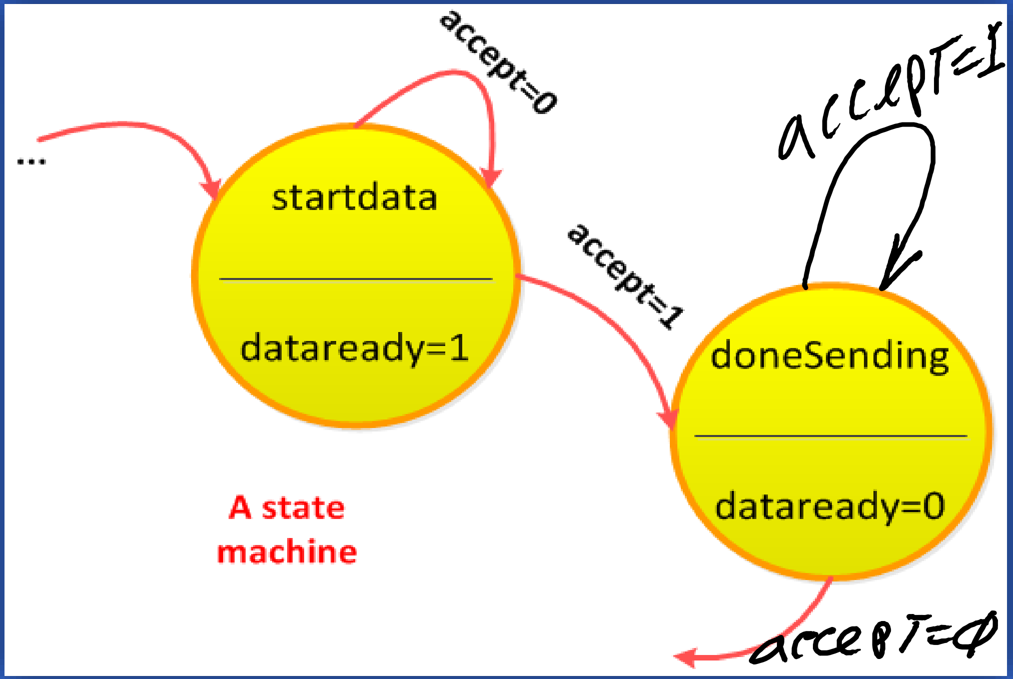
- Each system has its own clocking
- They have to have certain signals to talk

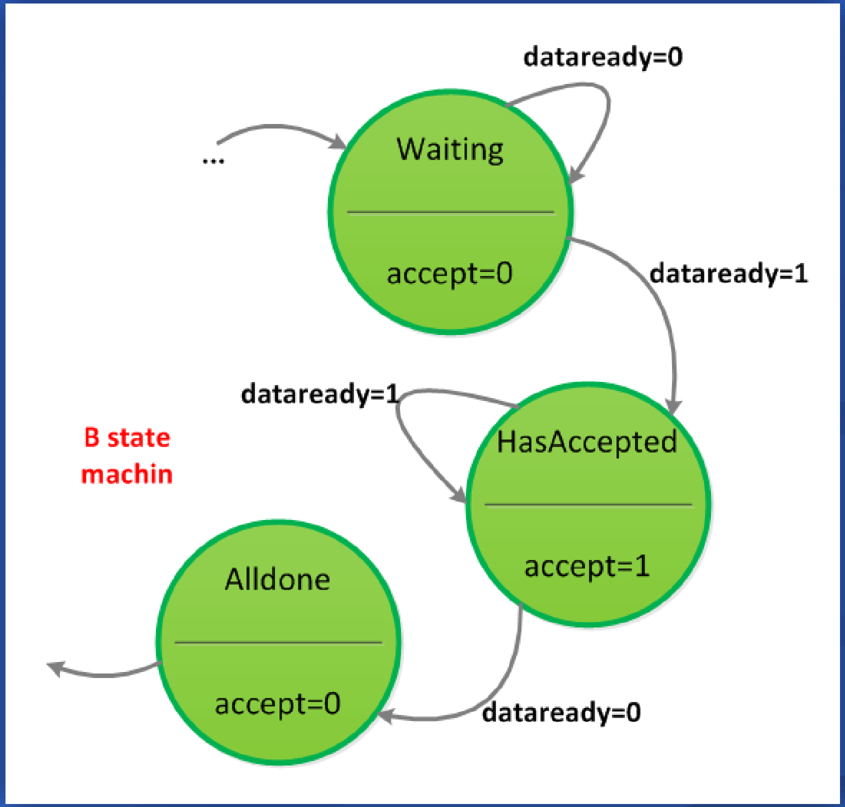


Fully Responsive Handshaking



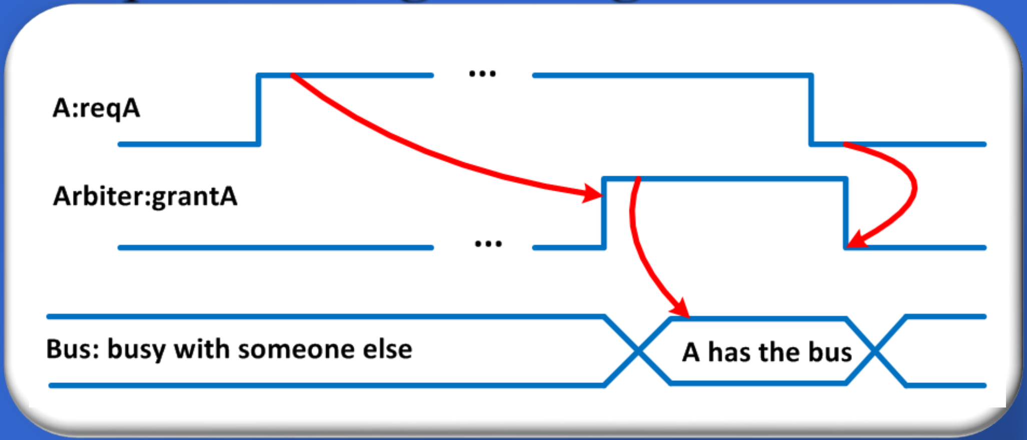
Fully Responsive Handshaking

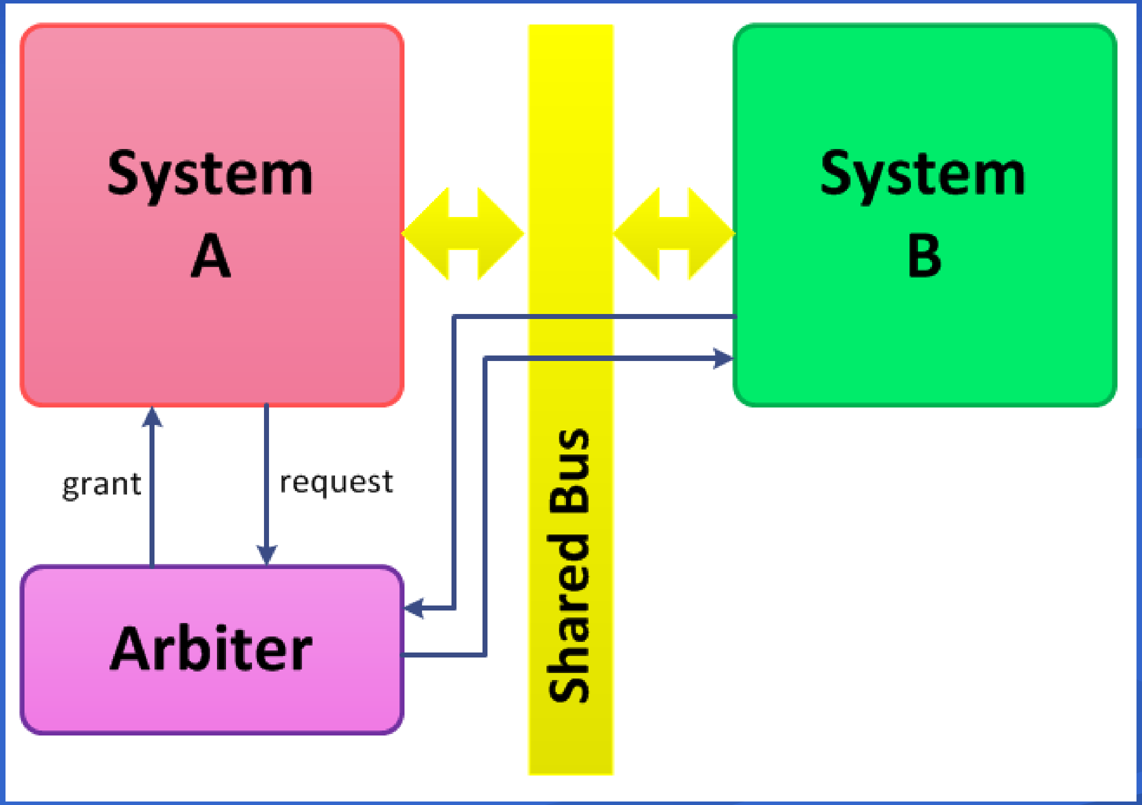




Handshaking for Accessing a Shared Bus

- Using an arbiter to assure that none of the systems will simultaneously access the shared bus
- Each system has to have its own request and grant signals

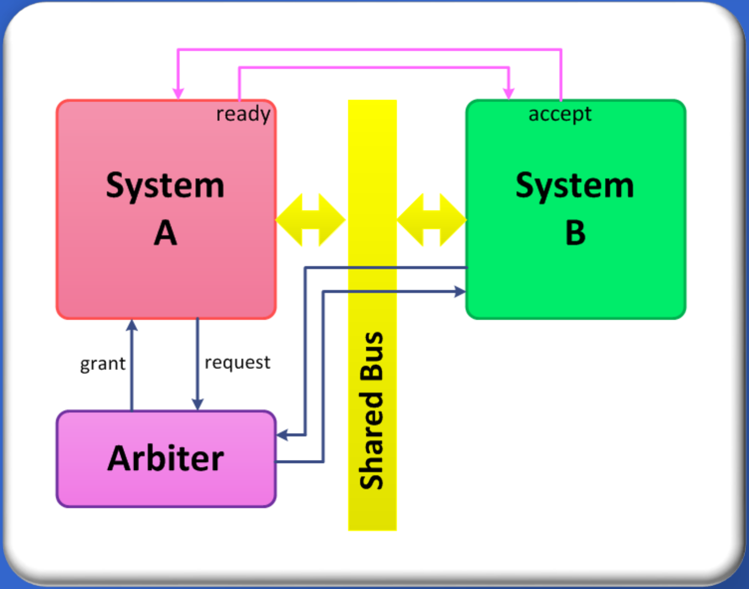


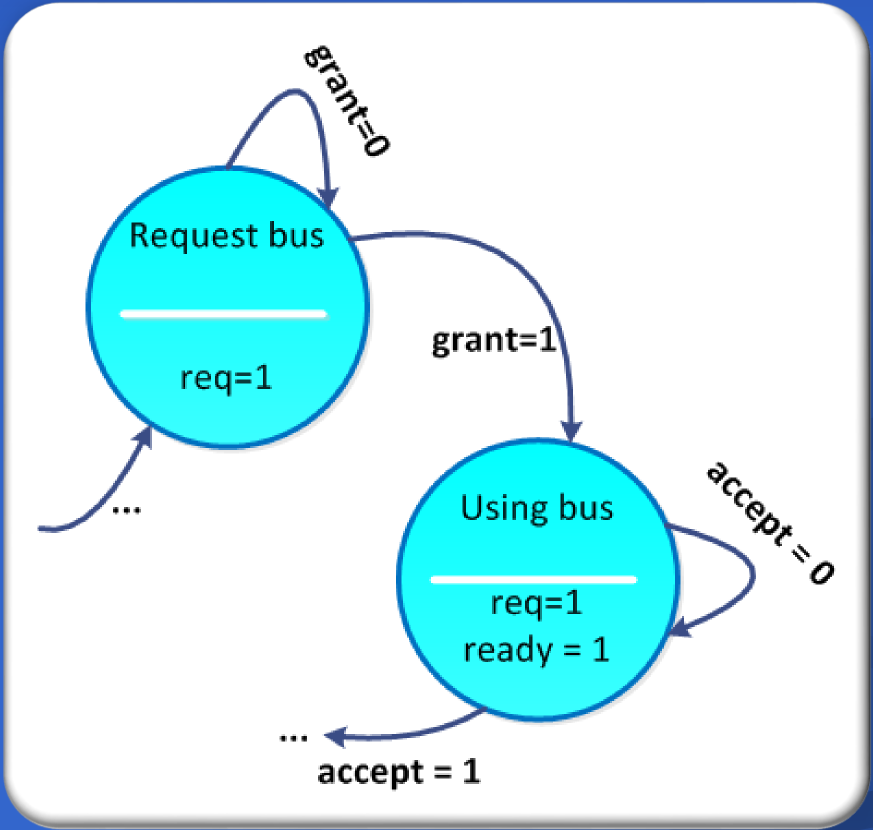


Two Level Handshaking

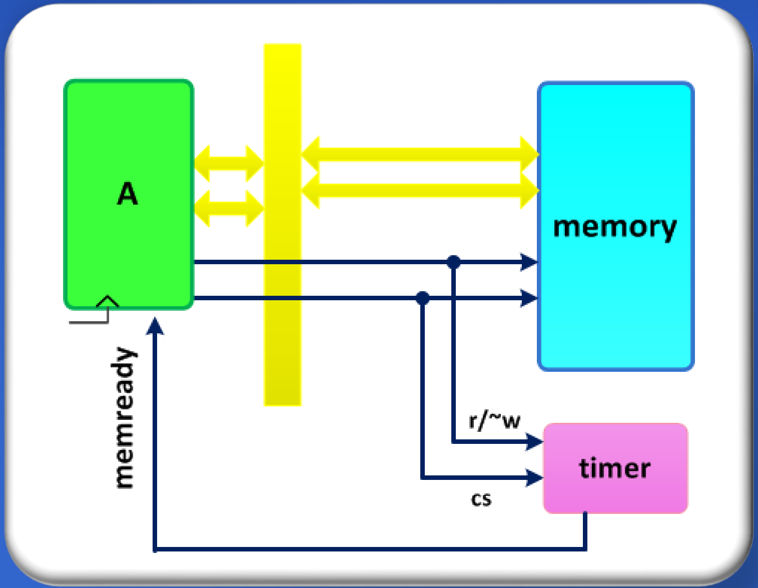
- Assume that system A wants to send some data to B through a shared bus
- At first A should talk to the arbiter and catches the bus by issuing a request
- Once it puts the data on bus it informs system B by issuing ready
- After data is picked up by B, A removes its request

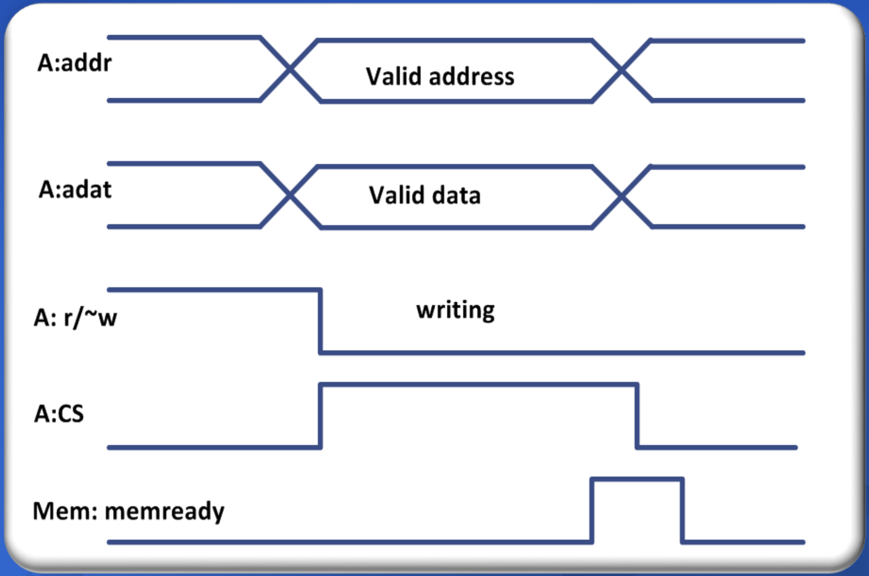
Two level handshaking

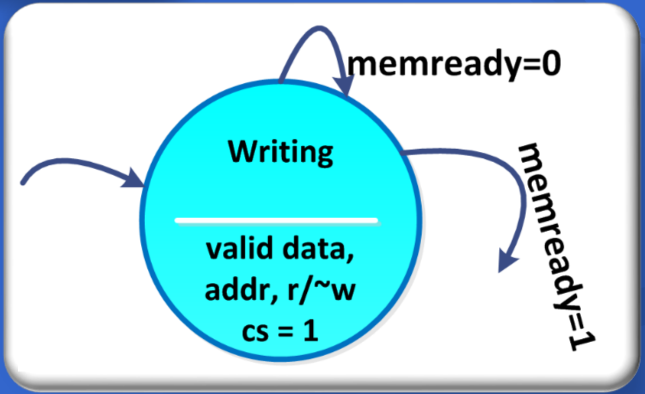




Handshaking Type Three: memory handshaking





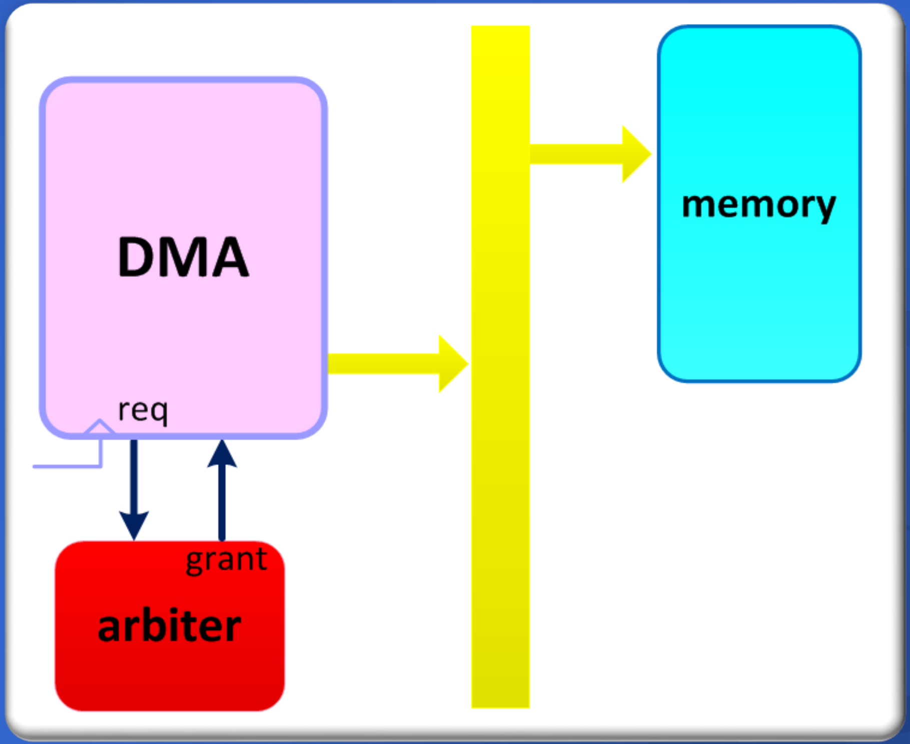


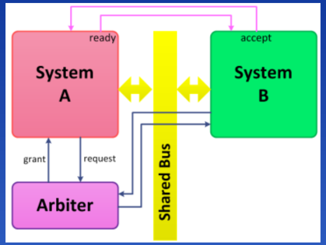
Combining Type Two and Three of Handshaking

- We can combine type two and three of handshaking
- At first A should deal with arbiter and gets the permission of using the bus
- Then it should send signals to the memory and waits for memready

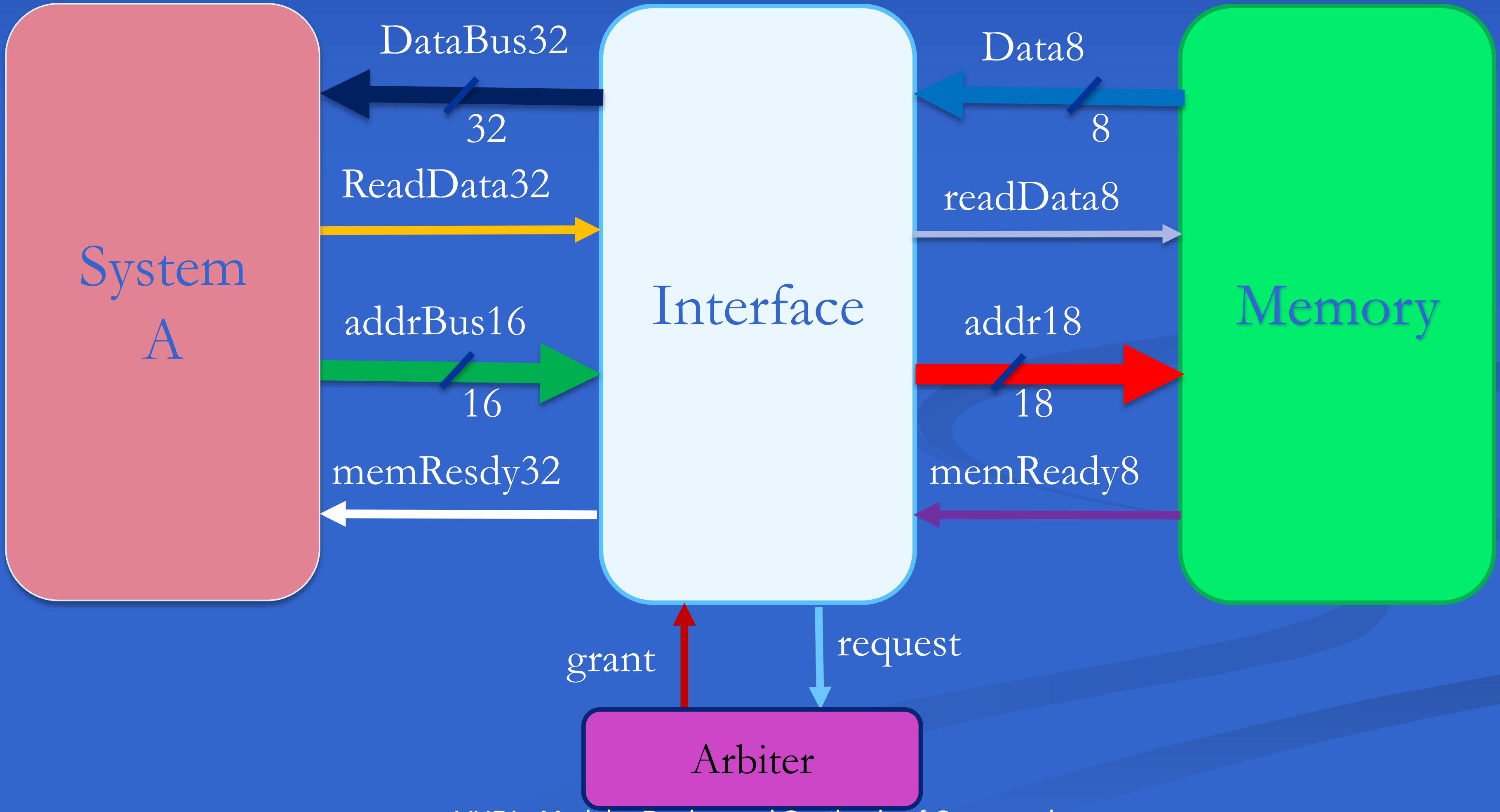
Handshaking Type Four: DMA Mode Or Burst Mode

- Burst writing or DMA writing or block writing

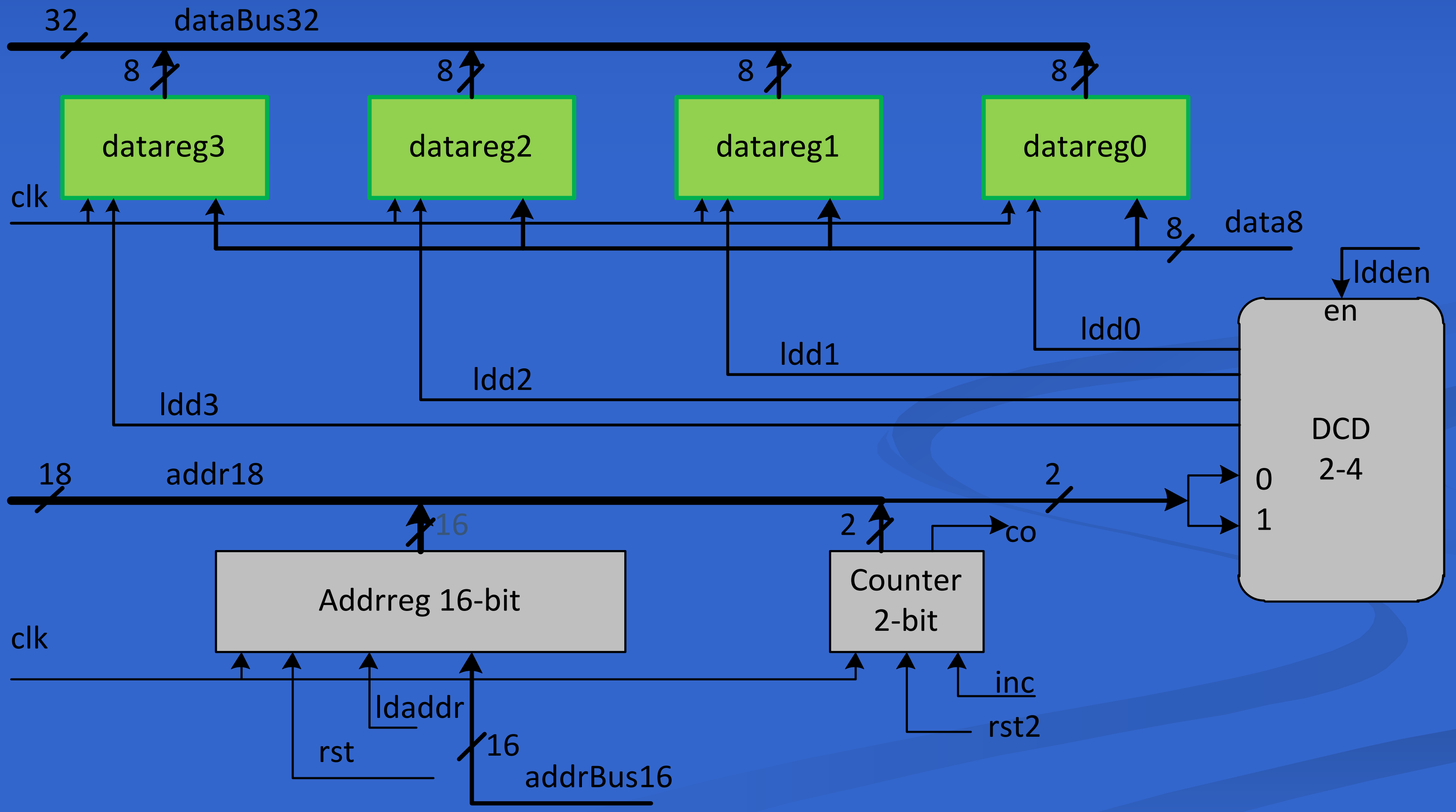




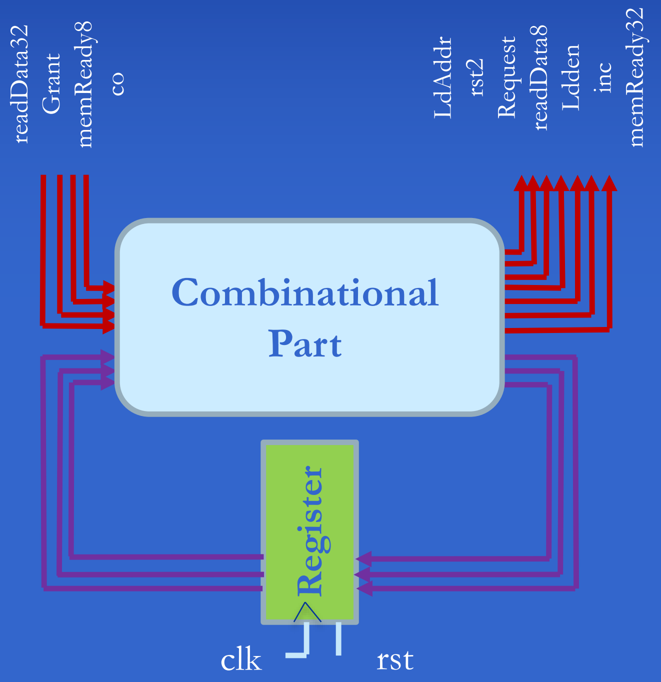
Memory Interface: Design Example

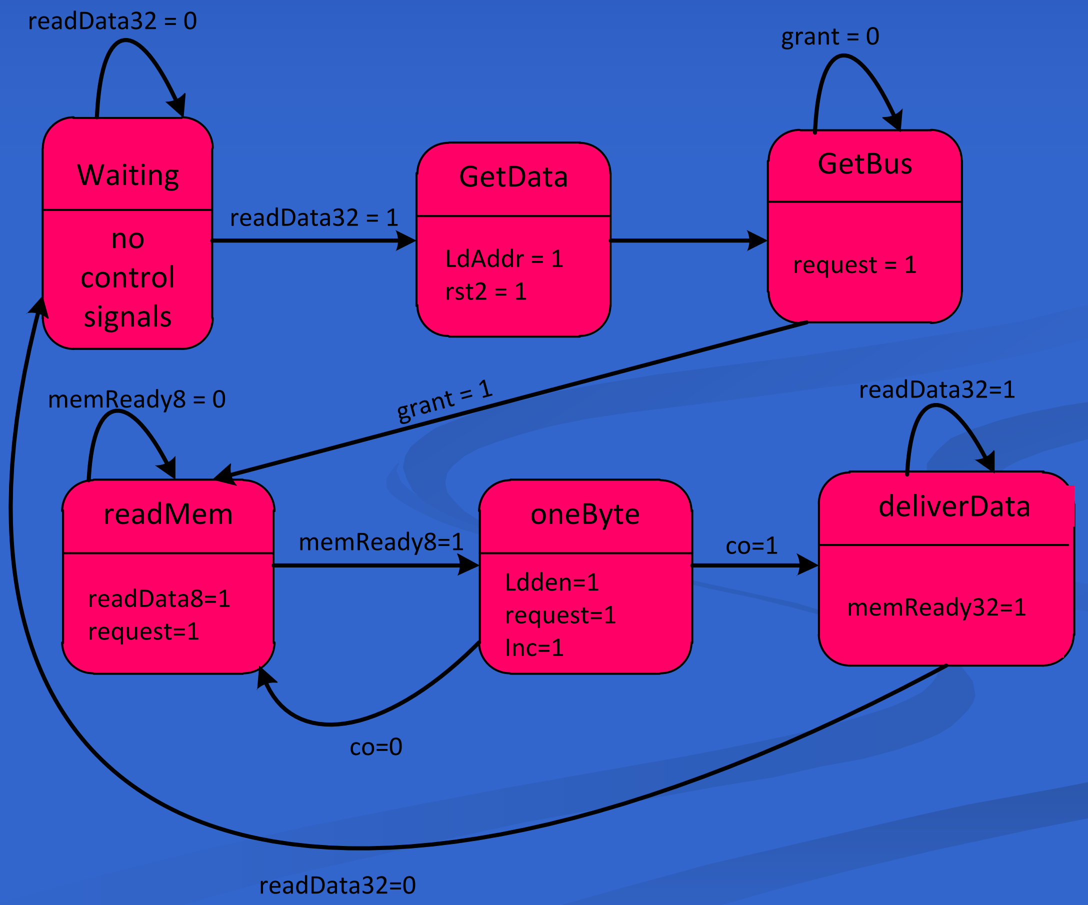


Memory Interface: Datapath & Controller Partitioning

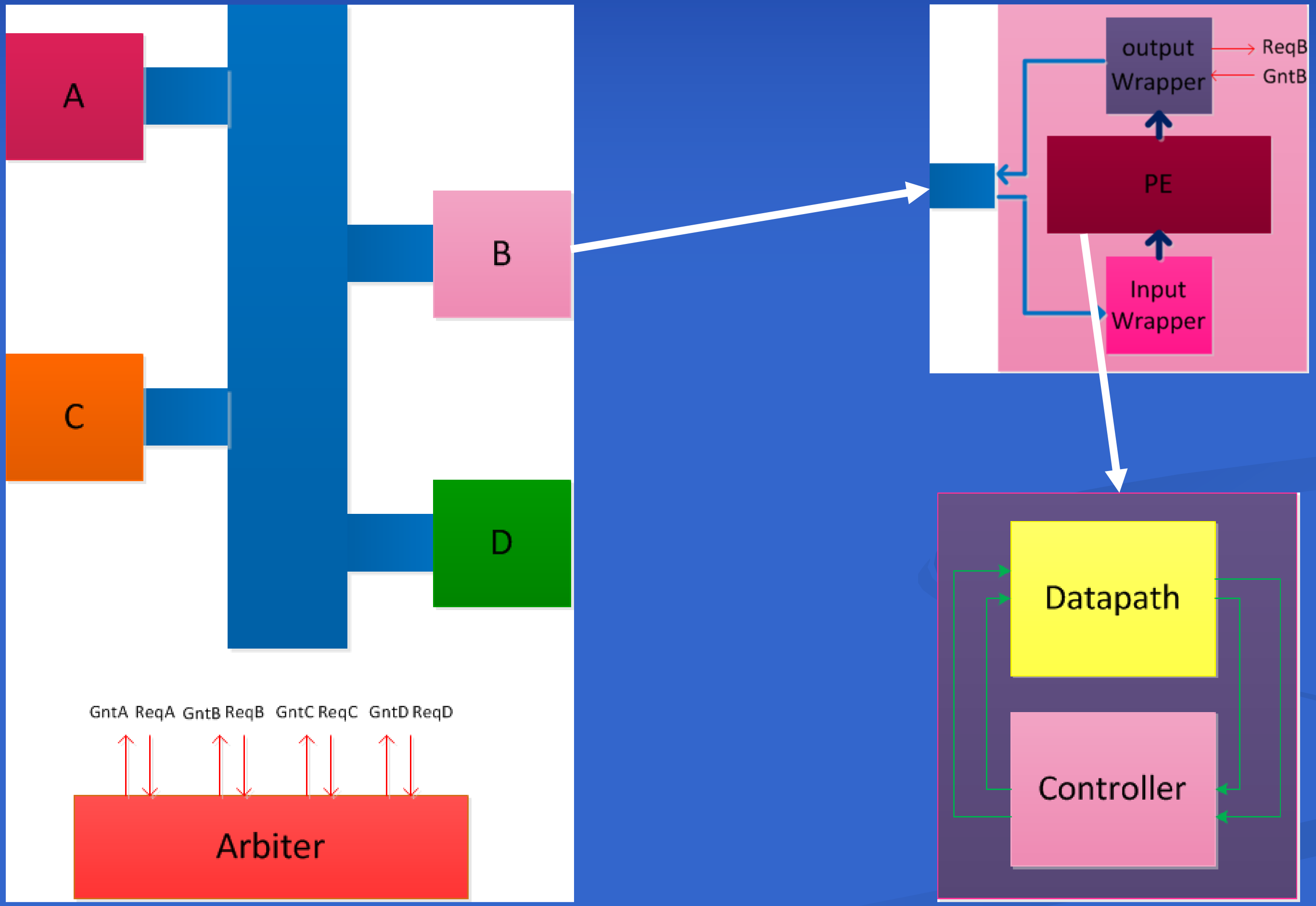


Memory Interface: State Machine

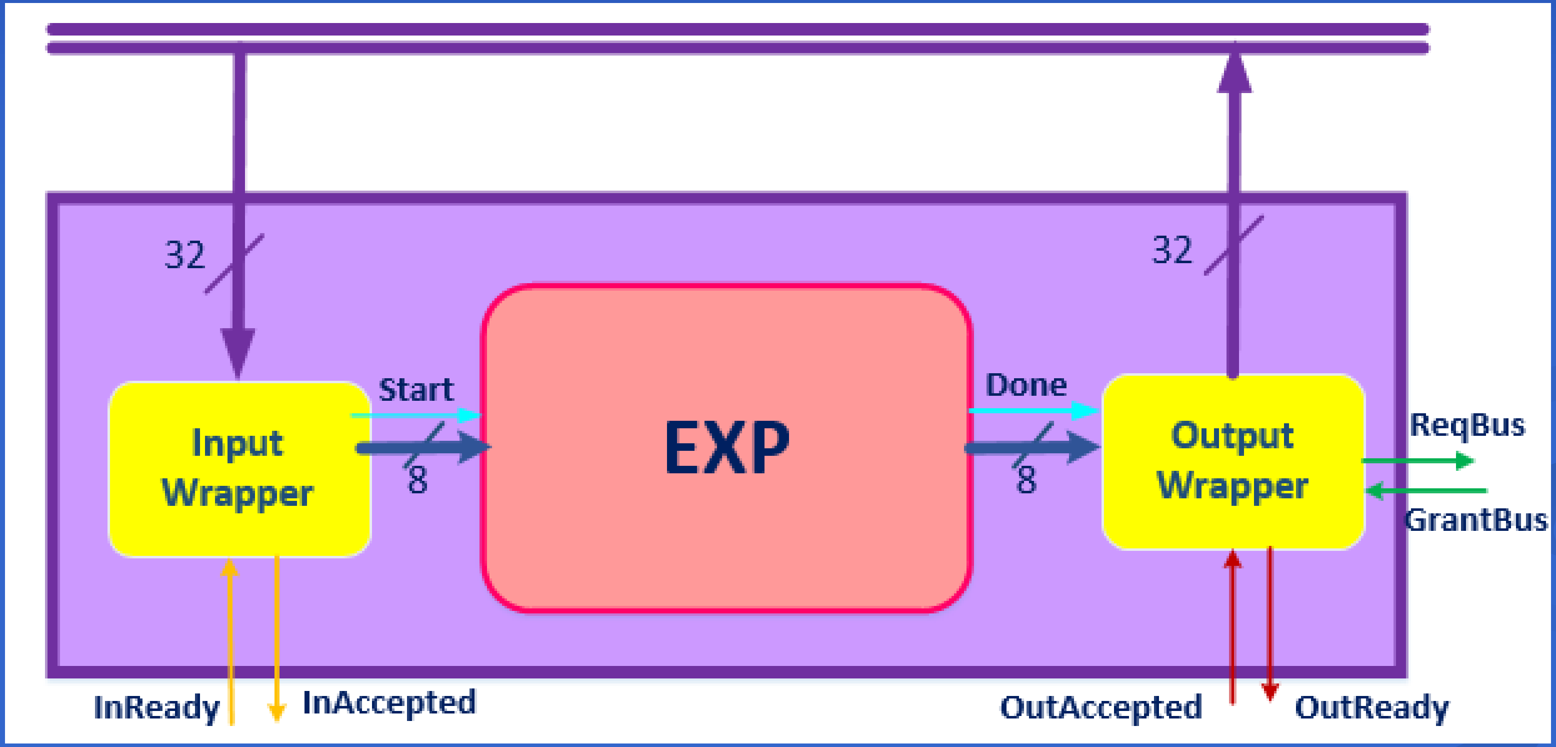




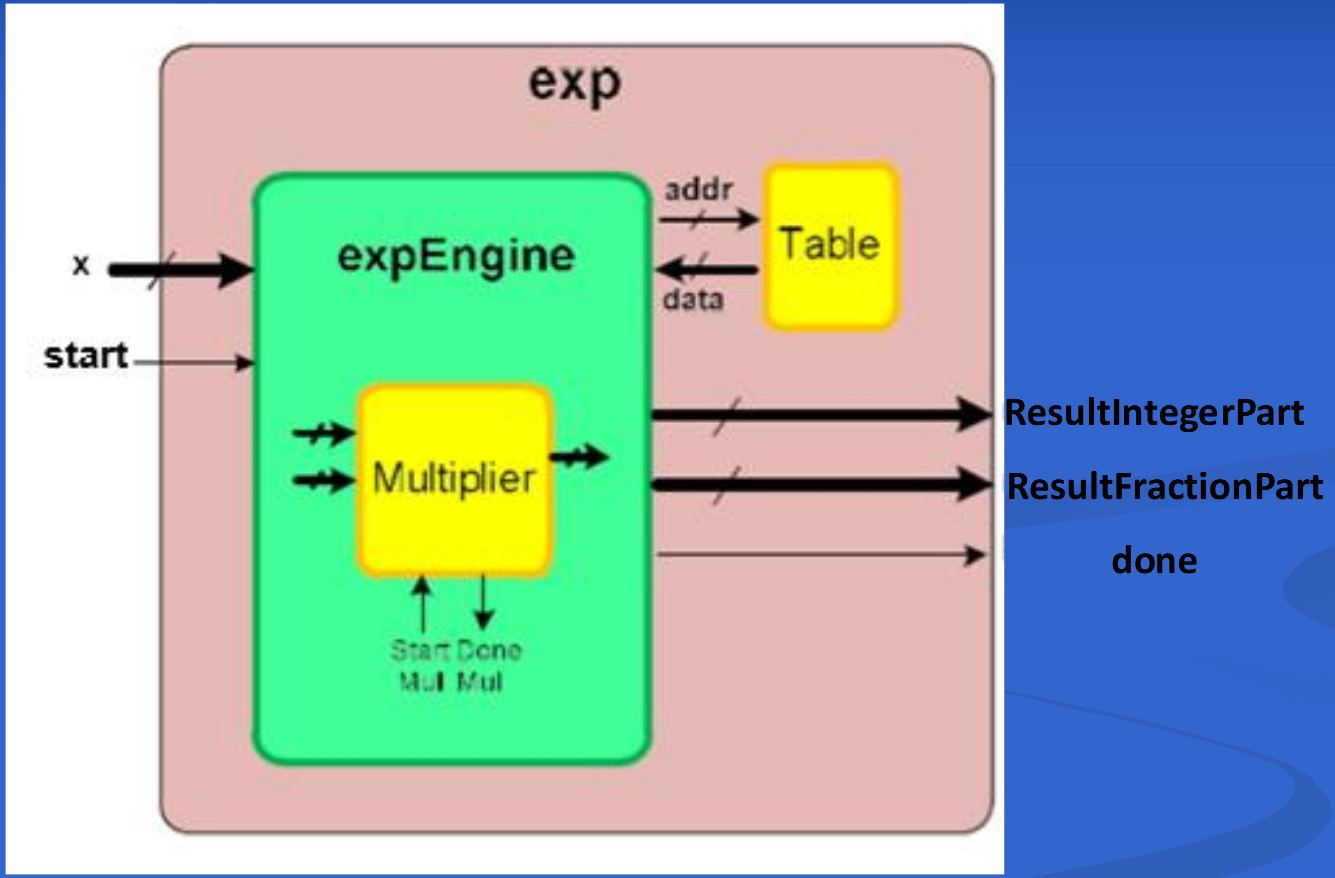
Exponential Module



Exponential Module



Exponentiation Module

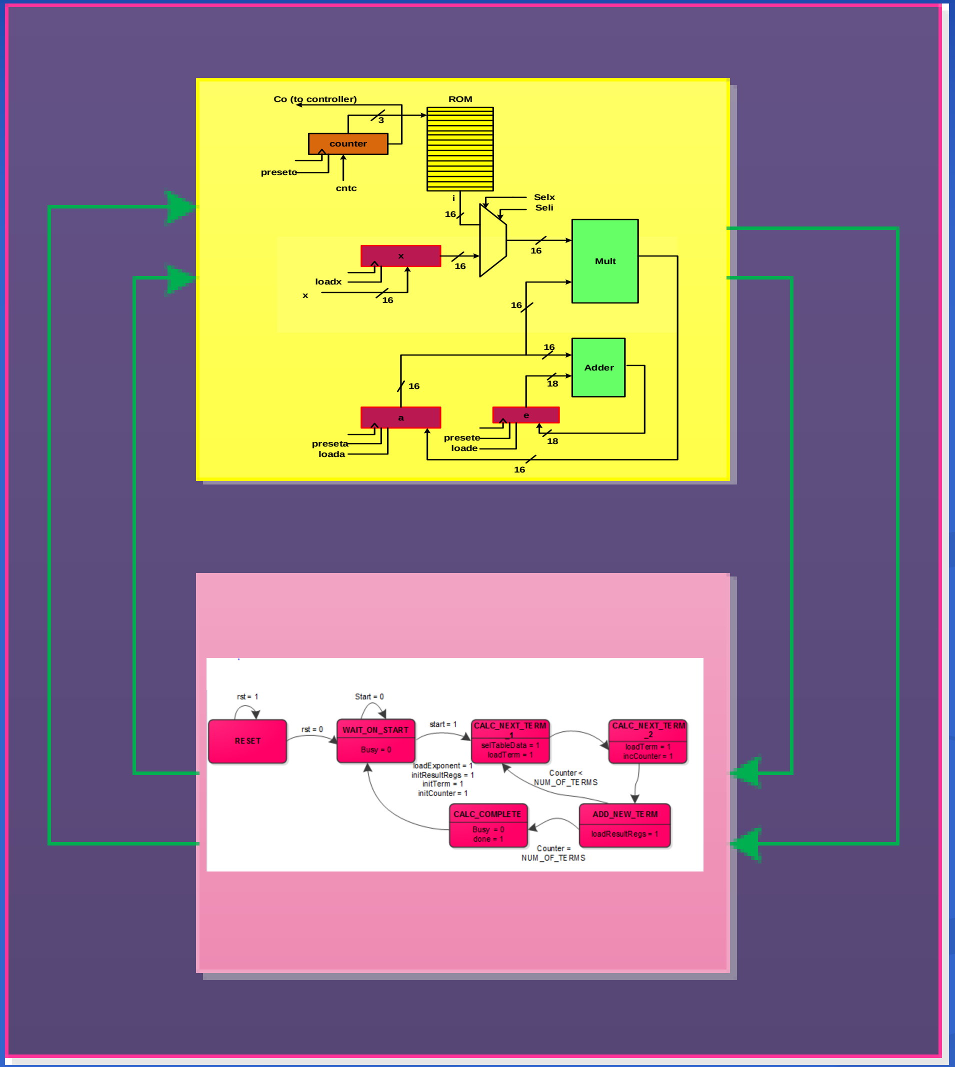


Exponential Function Algorithm

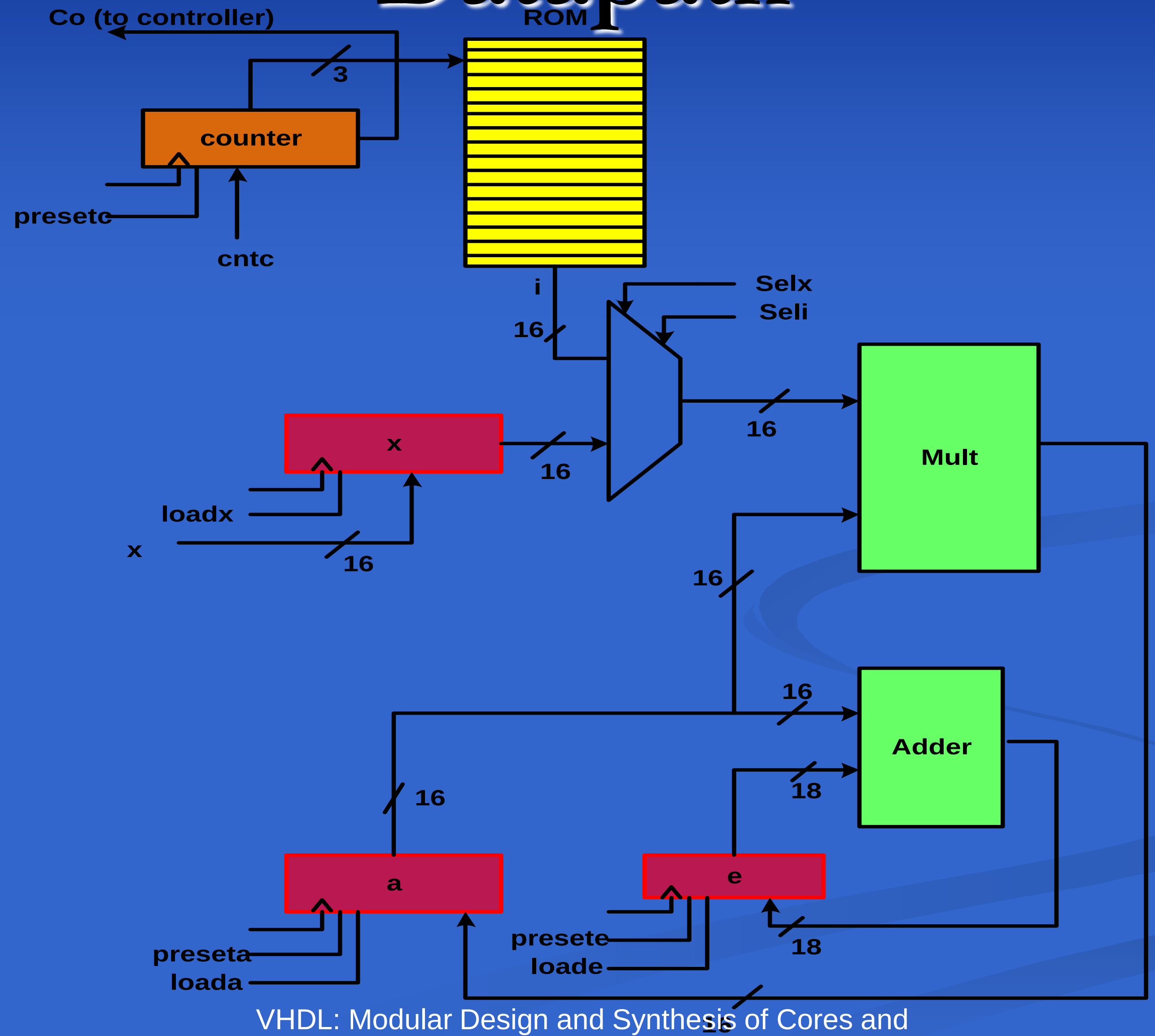
$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

```
e = 1;
a = 1;
for( i = 1; i < n; i++ ) {
    a = a × x × ( 1 / i );
    e = e + a;
}
```

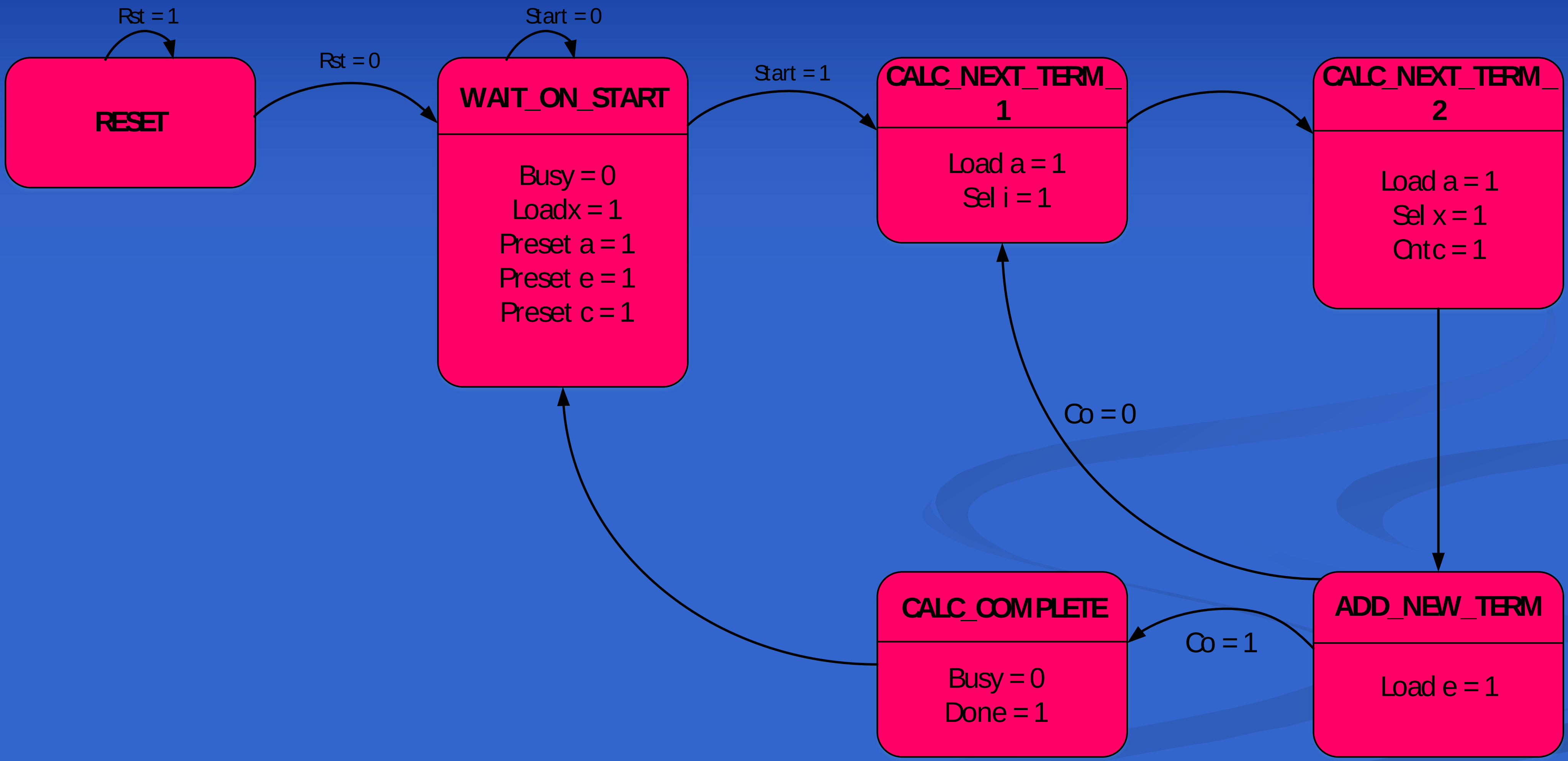
Datapath / Controller



Datapath


Co (to controller)
ROM
3
counter
presetc
cntc
i
16
Selx
Seli
16
x
16
loadx
x
16
Mult
16
16
Adder
16
18
a
e
presete
loade
18
preseta
loada
16

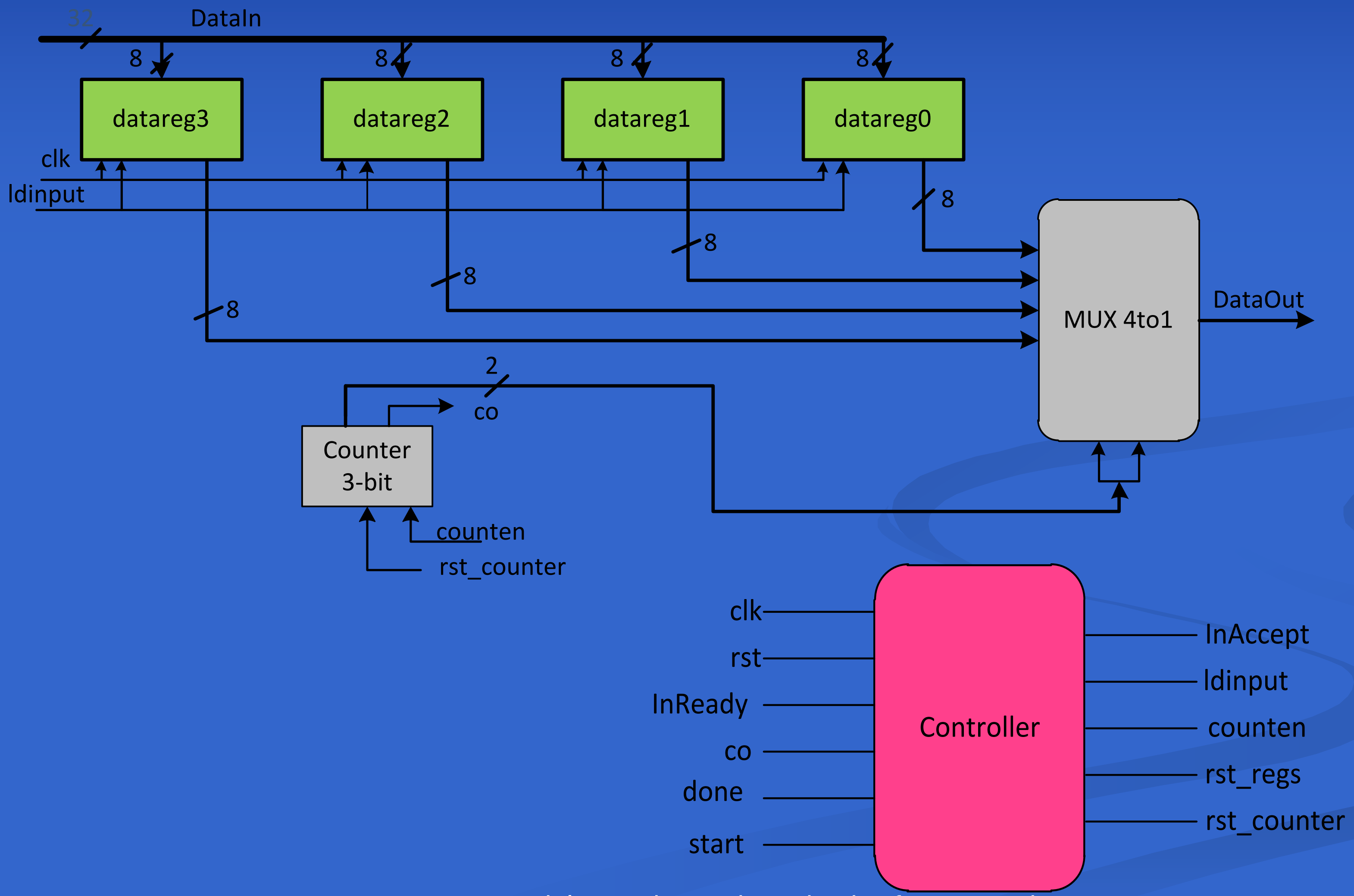
Controller



Rst = 1
RESET
Rst = 0
Start = 0
WAIT_ON_START
Busy = 0
Loadx = 1
Preset a = 1
Preset e = 1
Preset c = 1
Start = 1
CALC_NEXT_TERM_1
Load a = 1
Sel i = 1
CALC_NEXT_TERM_2
Load a = 1
Sel x = 1
Cntc = 1
Co = 0
ADD_NEW_TERM
Load e = 1
Co = 1
CALC_COMPLETE
Busy = 0
Done = 1

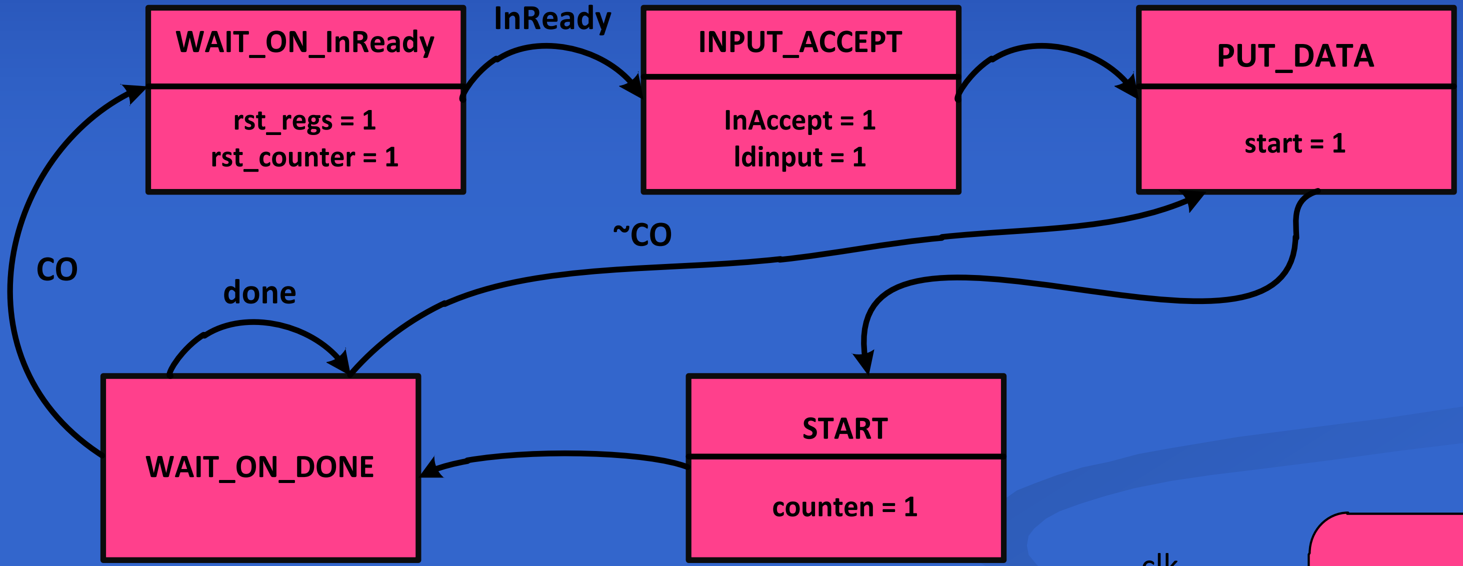
Input Wrapper:

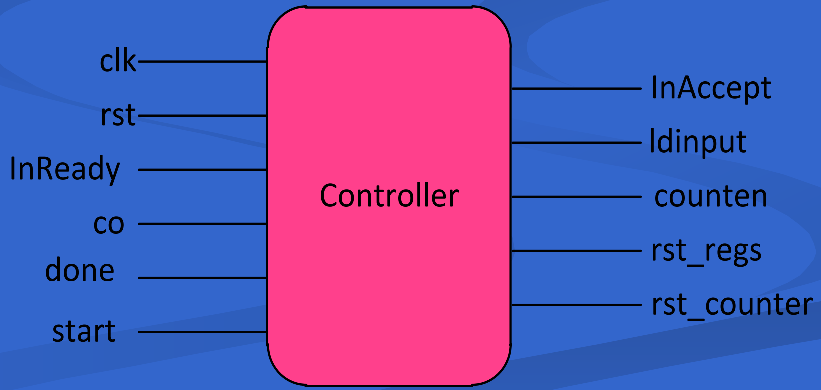
Datapath & Controller Partitioning



Input Wrapper:

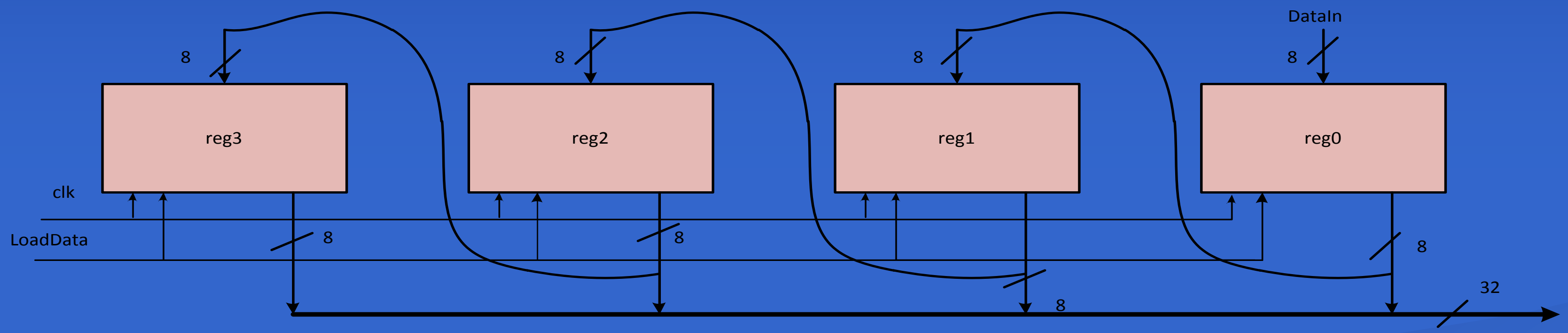
State Machine

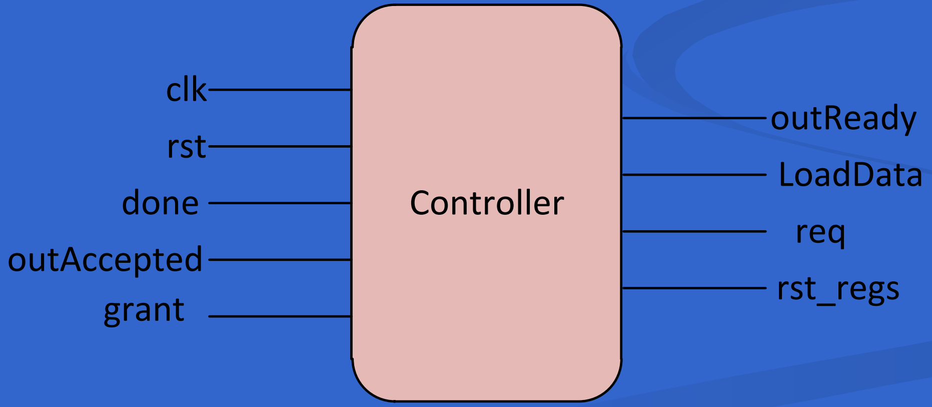




Output Wrapper:

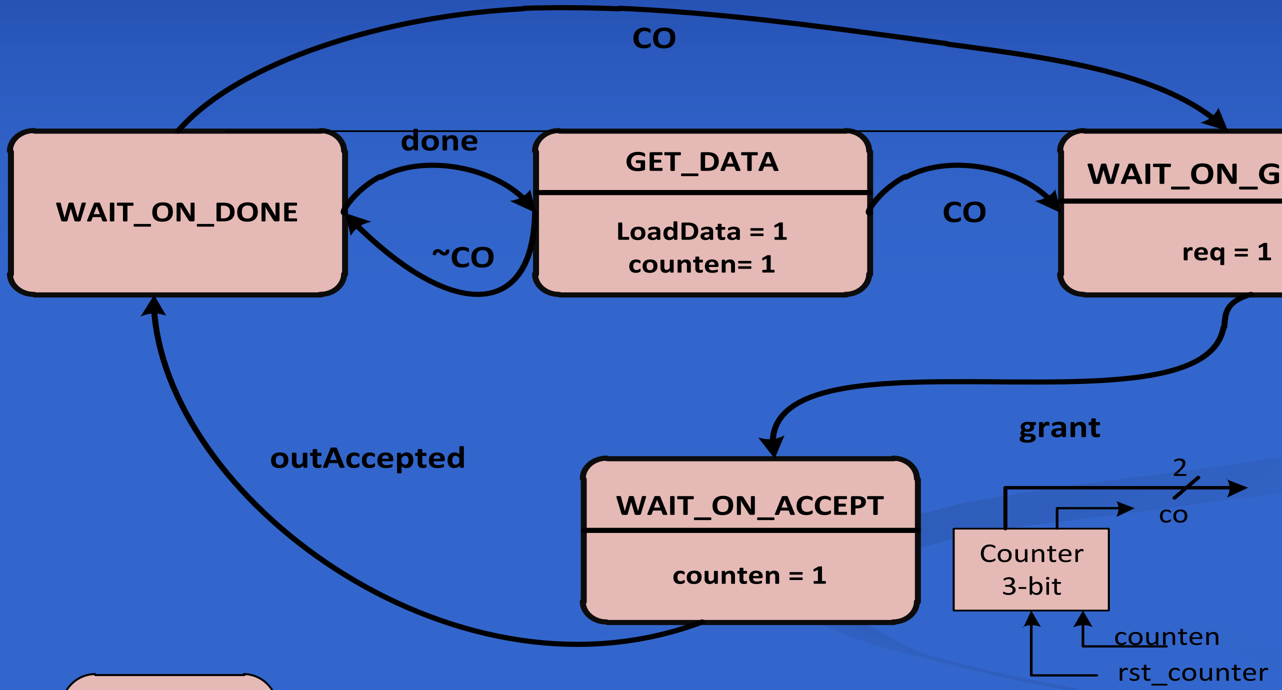
Datapath & Controller Partitioning

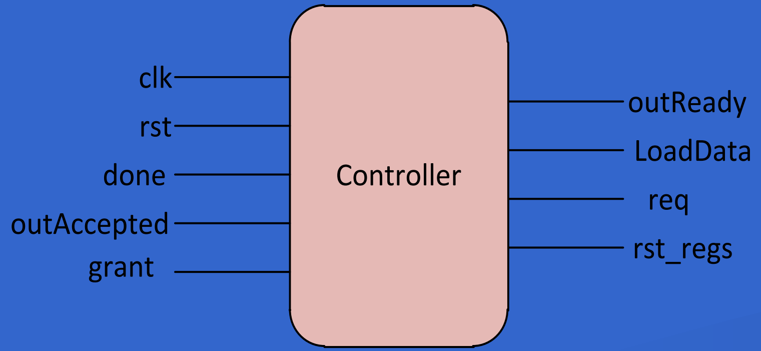




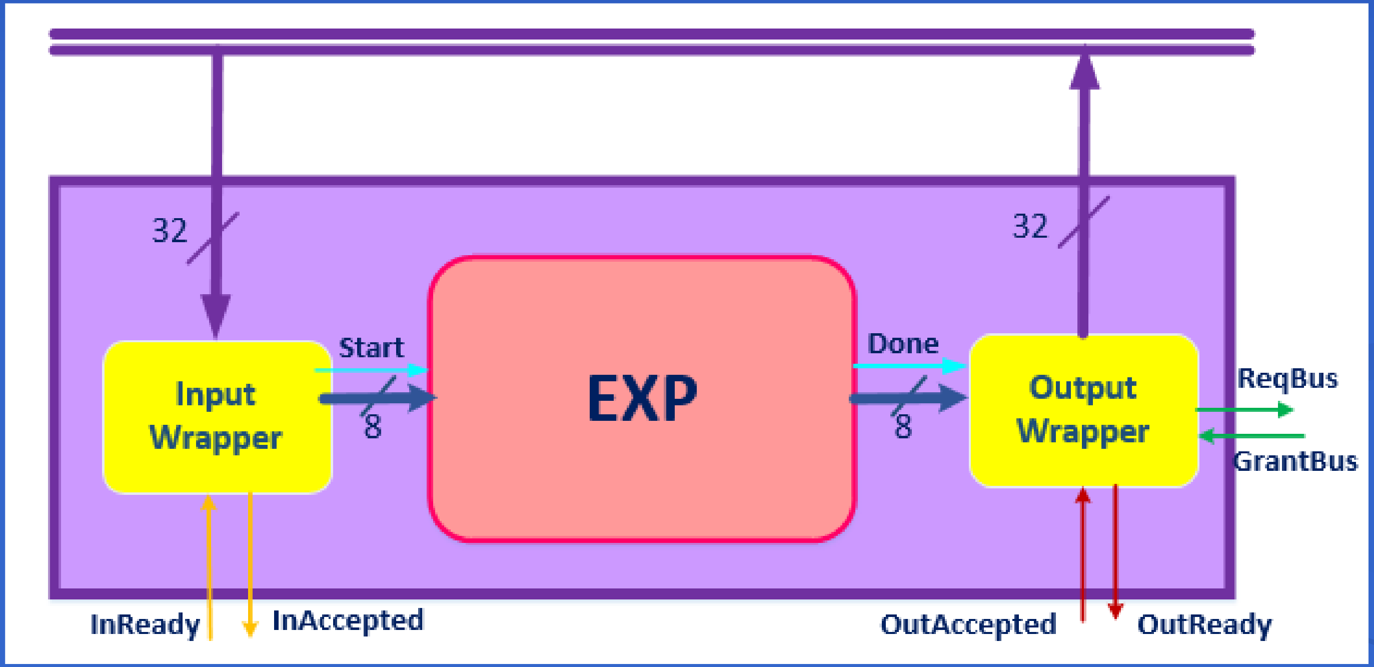
Output Wrapper:

State Machine





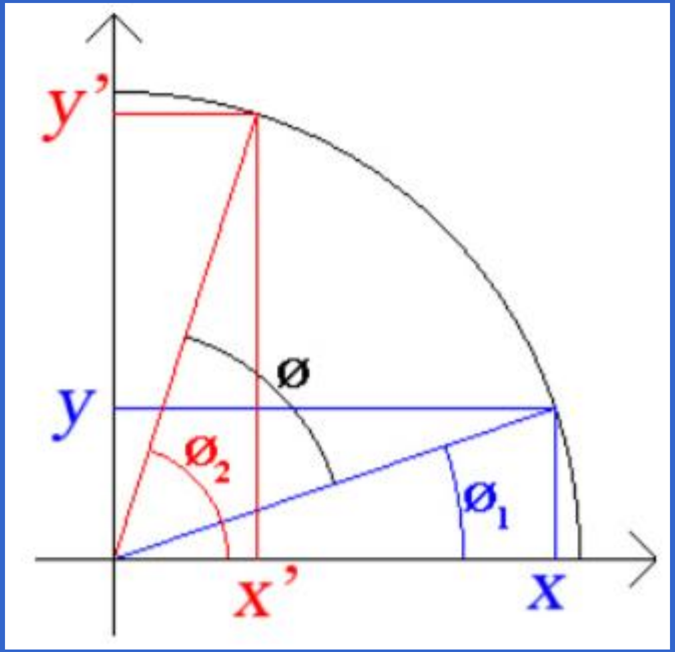
Complete System



CORDIC

- The CORDIC algorithm is an iterative technique based on the rotation of a vector which allows many transcendental and trigonometric functions to be calculated
- It is achieved using only shifts, additions/subtractions and table look-ups which map well into hardware

CORDIC



$$x' = x\cos\phi - y\sin\phi$$
$$y' = y\cos\phi + x\sin\phi$$

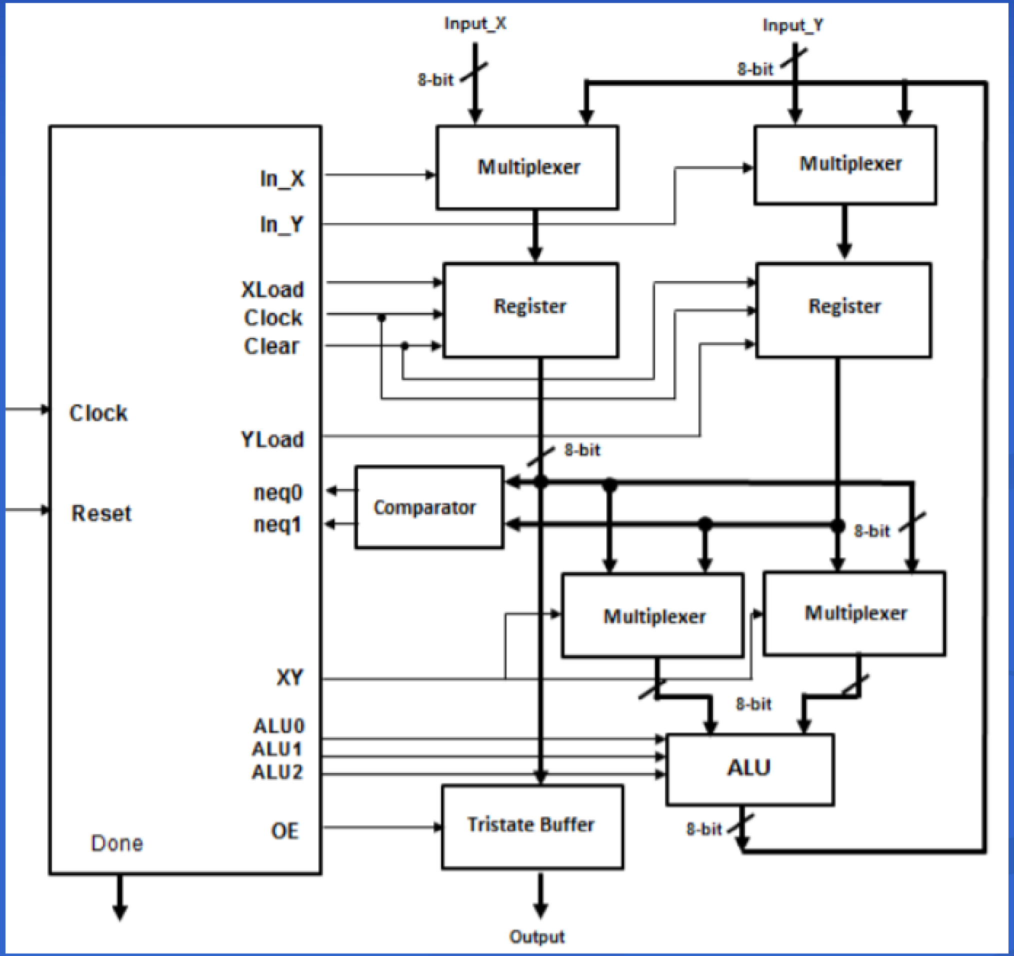
$$x' = \cos\phi \cdot [x - y\tan\phi]$$
$$y' = \cos\phi \cdot [y + x\tan\phi]$$

$$x_{i+1} = K_i\left[x_i - y_i \cdot d_i \cdot 2^{-i}\right]$$
$$y_{i+1} = K_i\left[y_i + x_i \cdot d_i \cdot 2^{-i}\right]$$

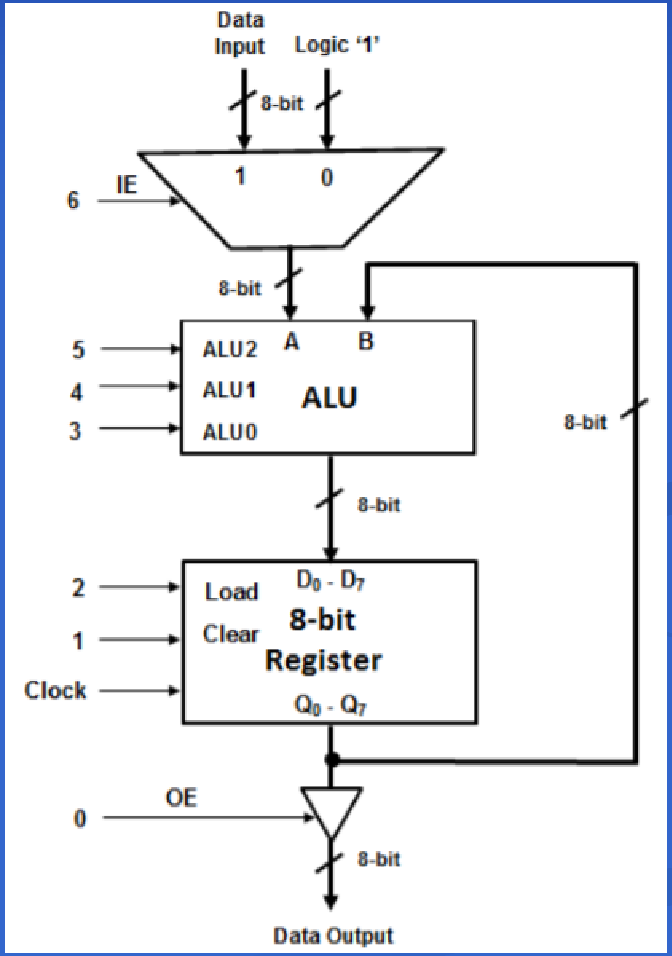
CORDIC Advantages

- Number of gates required in hardware implementation on an FPGA, are minimum and hardware complexity is greatly reduced
- Cost of a CORDIC hardware implementation is less as only shift registers, adders and look-up table (ROM) are required
- Delay involved during processing is comparable to that of a division or square-rooting operation
- No multiplication and only addition, subtraction and bit-shifting operation

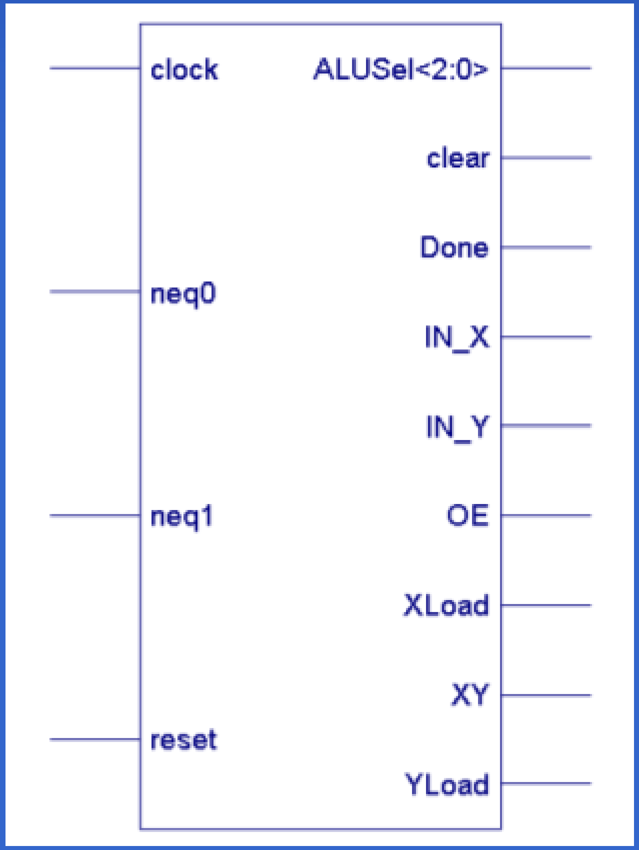
CORDIC Architecture

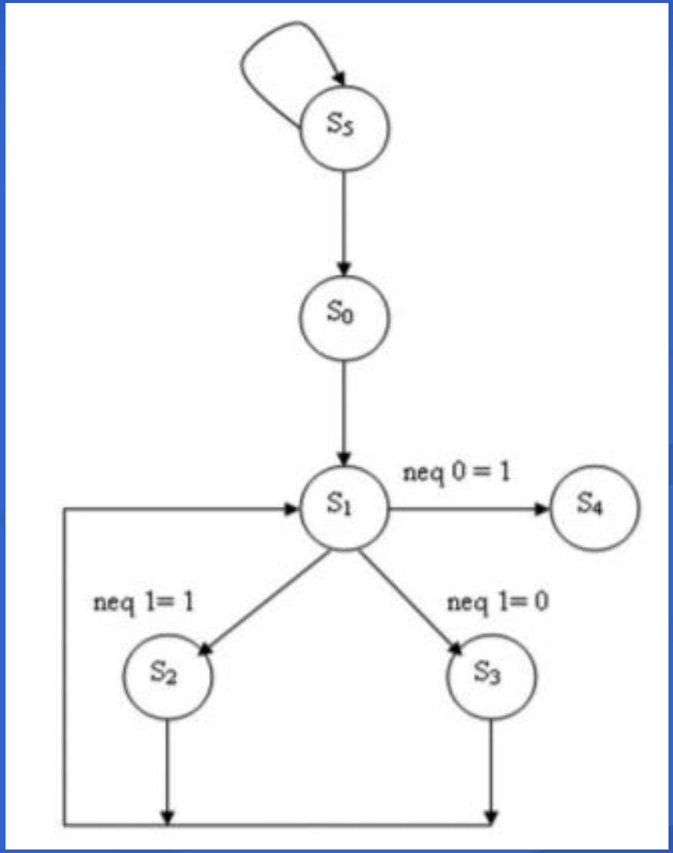


General Datapath



Controller


clock
ALUSel<2:0>
clear
Done
neq0
IN_X
IN_Y
neq1
OE
XLoad
XY
reset
YLoad


S5
S0
S1
neq 0 = 1
S4
neq 1= 1
neq 1= 0
S2
S3

- Accelerator right
- Left
- Extended instructions
- Near memory
- filter FIR from register file (rf in the wrapper)

The other category is interfaces or wrappers

- Wrappers
- Data sizing form ICEEP notes, that includes hand-shaking and arbitration
- A wrapper only for handshaking, another for data sizing only for burst connections, and the other for rf handling for extended insts rf can be the rf of the processor mapped to this rf

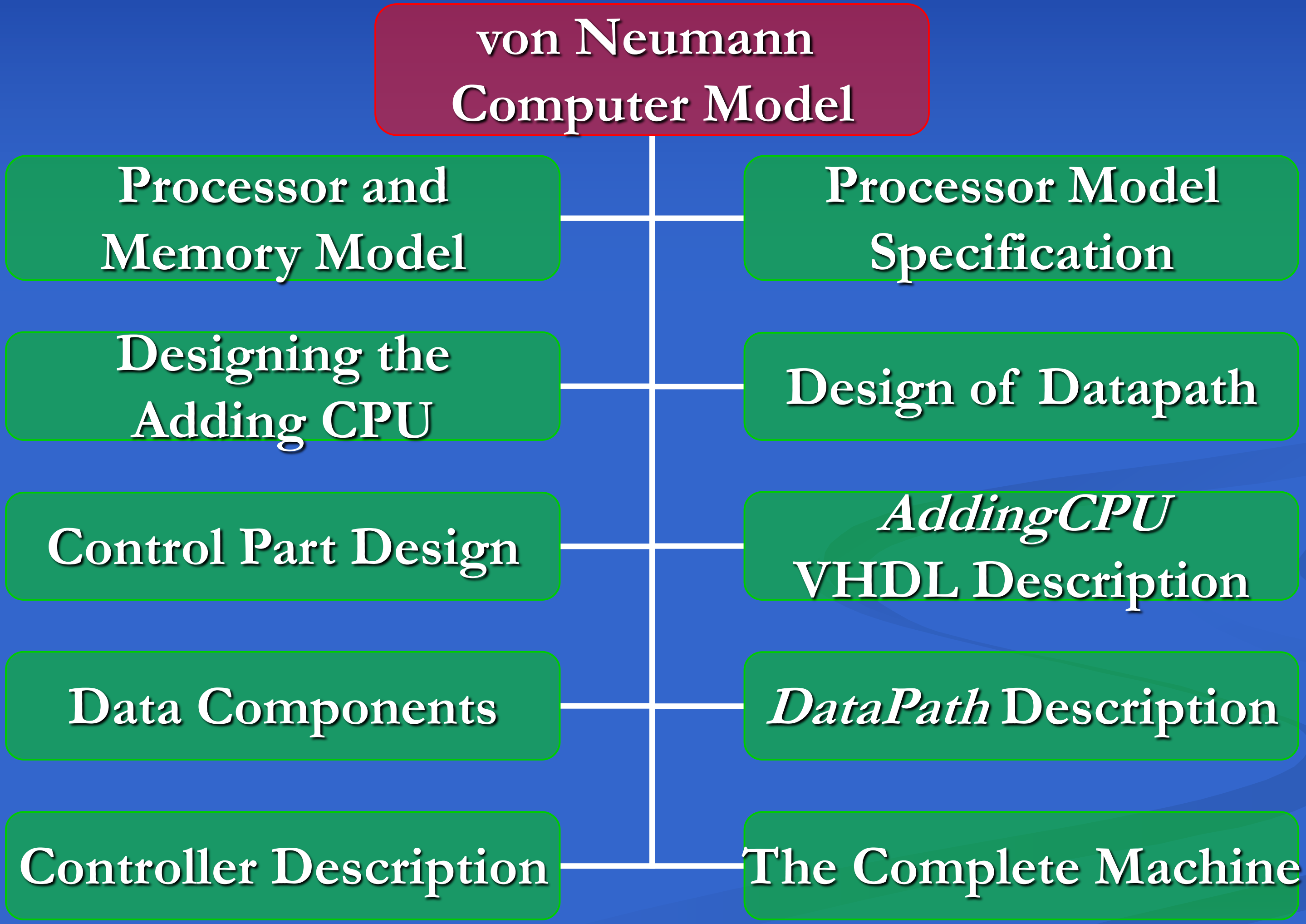
Interfacing hardware

- LRU, very different from other structures
- DMA, using arbitration handles the memory reads and writes
- Not arithmetic hardwares

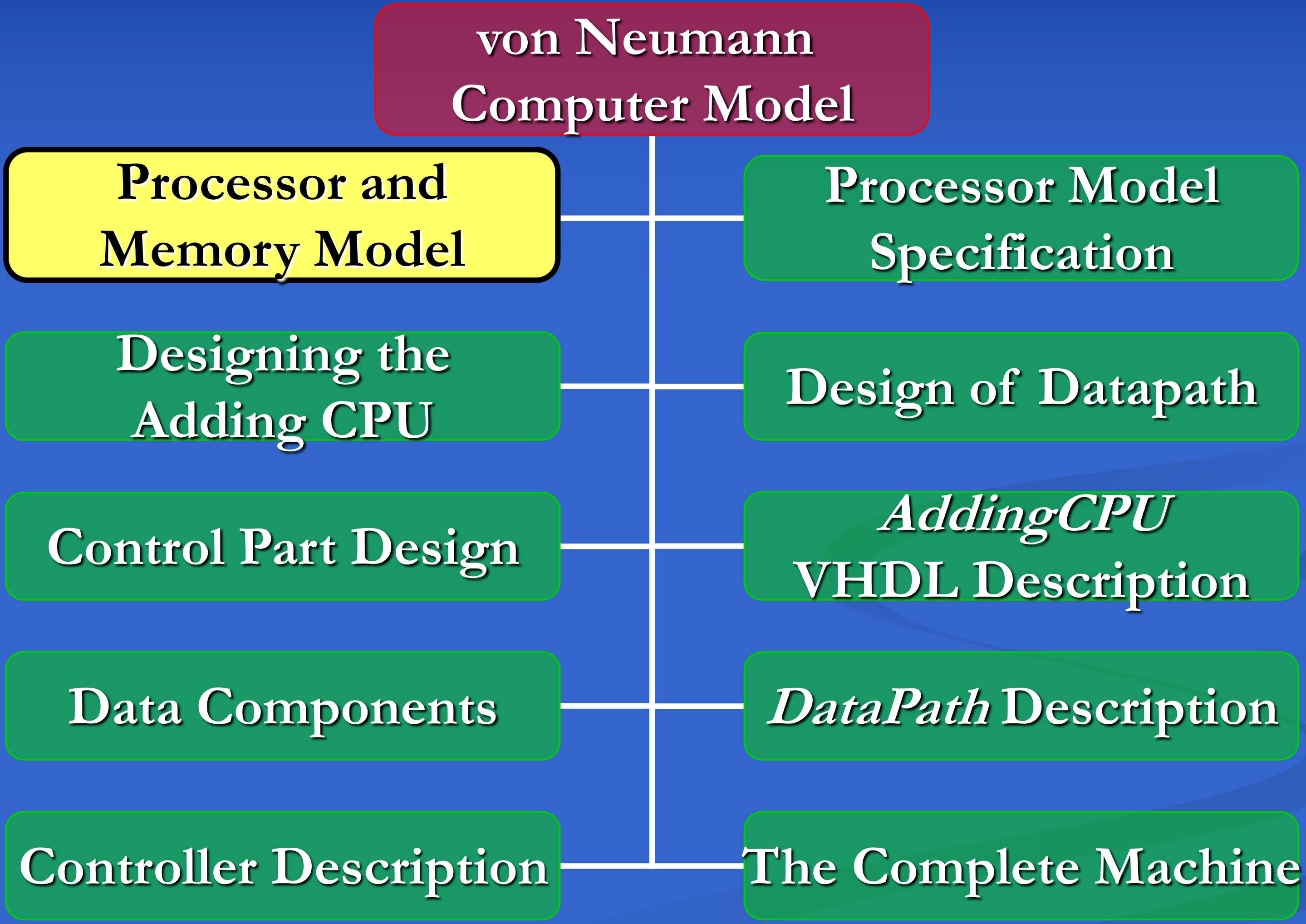
CPUs

- Von Neumann architcetures
- They have instructions, a processor with 4 instructions, it only accesses the memory, instruction fetch
- Next step is the Somayeh processor which is a processor that does the arithmetic work like sine, cosine, it only loads the program and has a pc to fetch the instruction
- Program counter or the sequencer
- SAYEH Processor

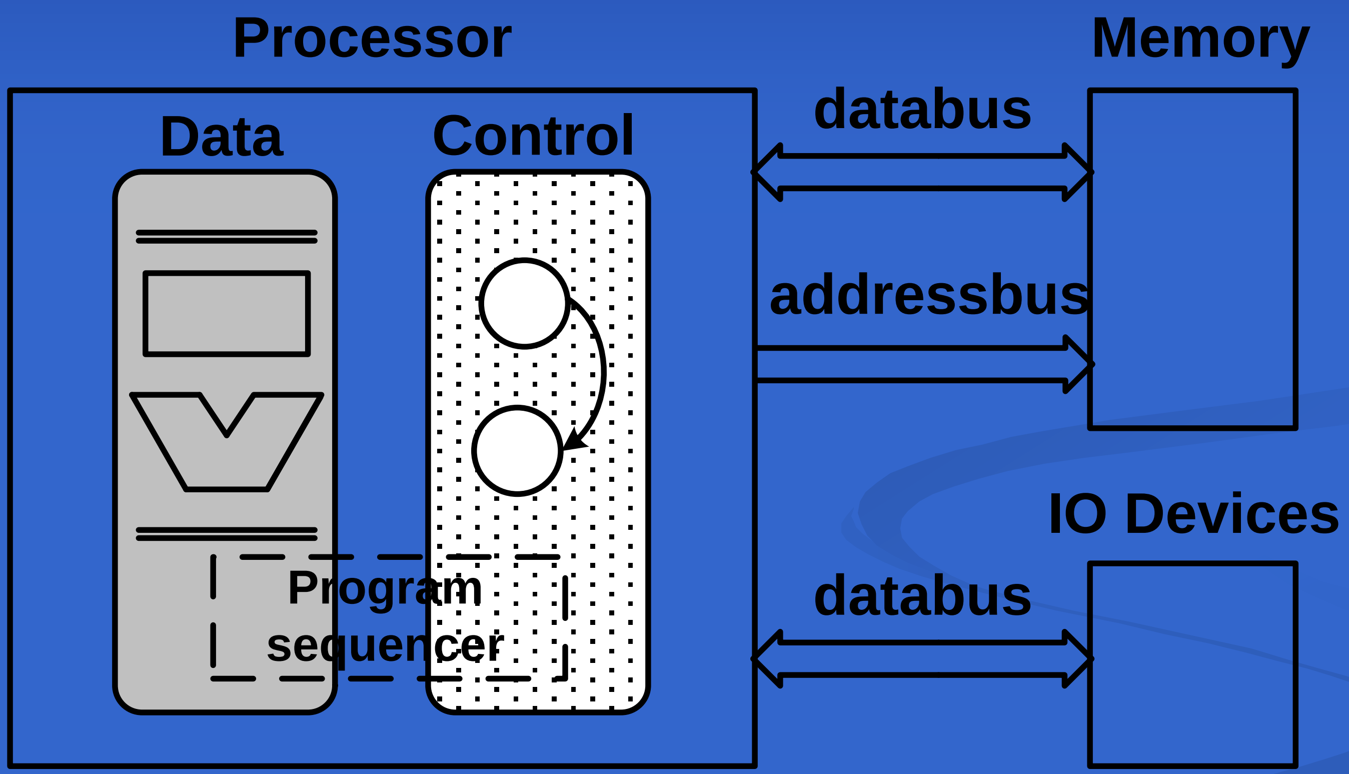
von Neumann Computer Model



Processor and Memory Model

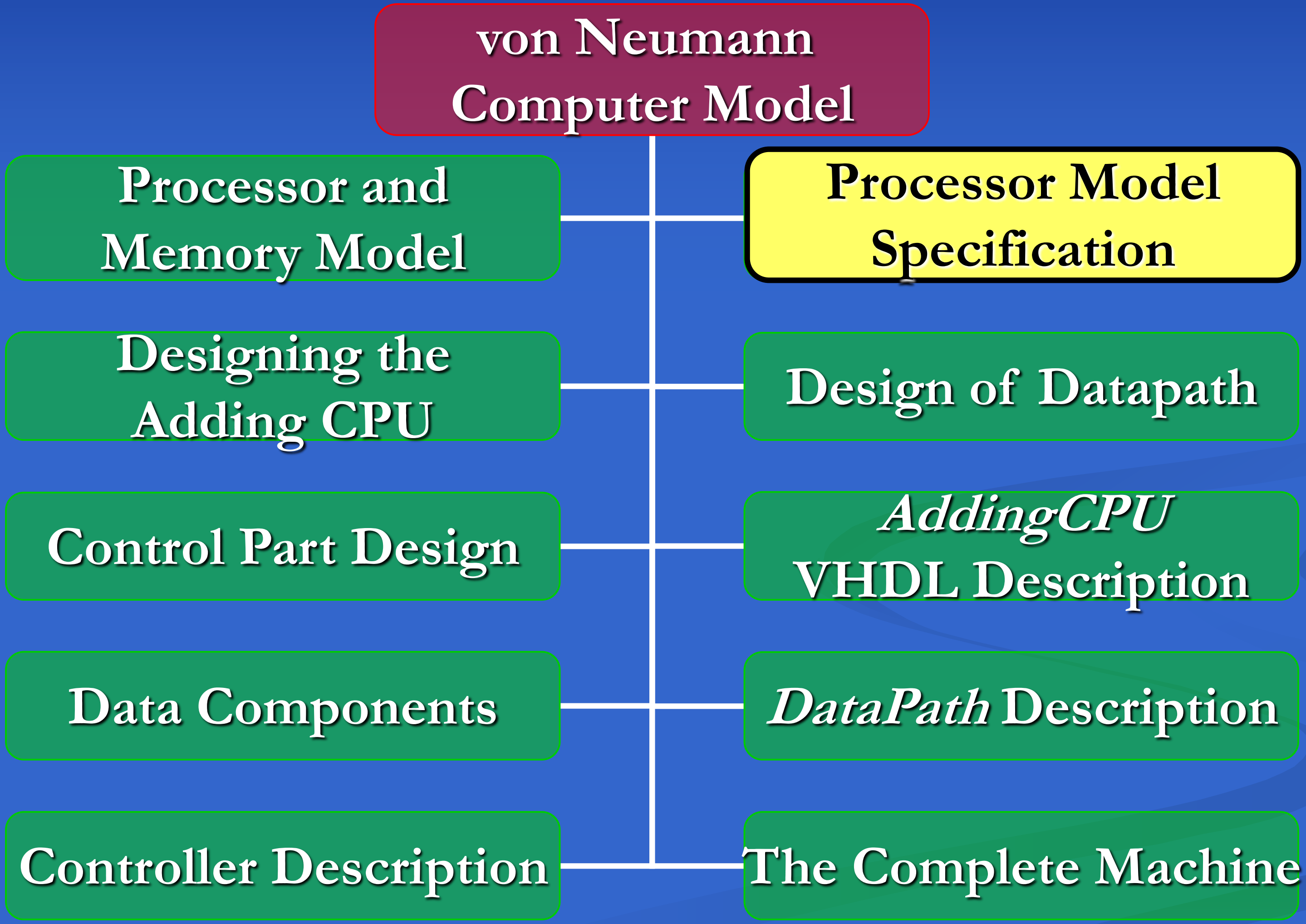


Processor and Memory Model

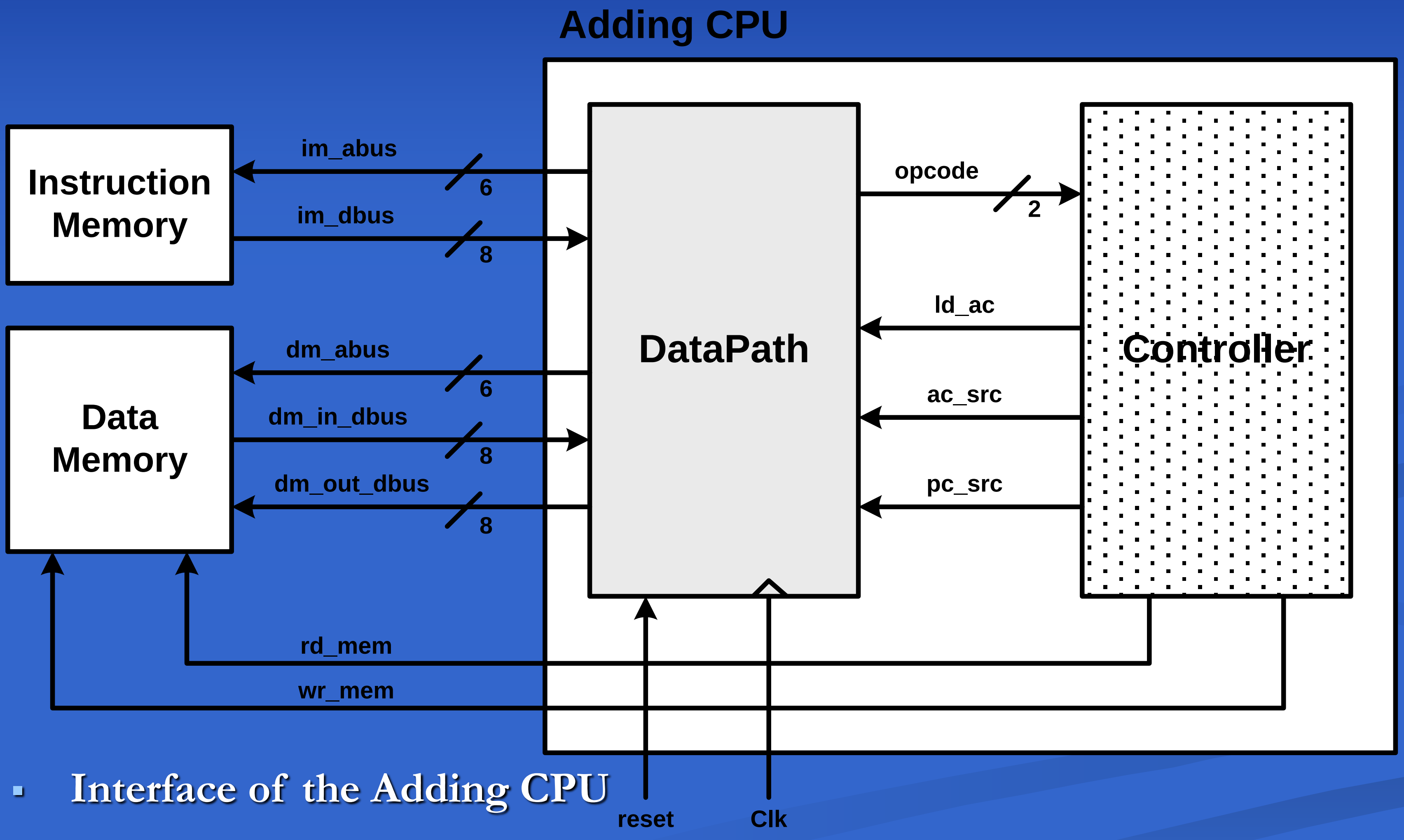


- von Neumann Process Model

Processor Model Specification

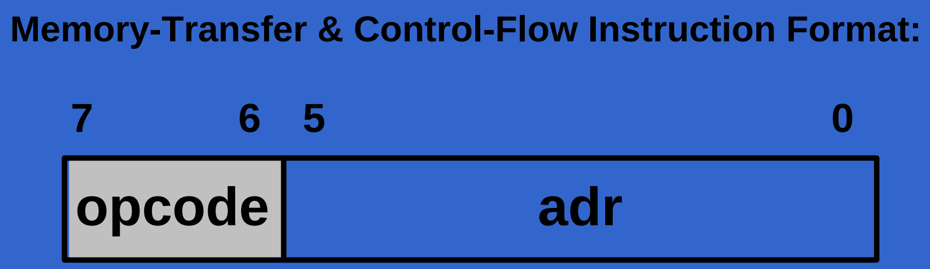


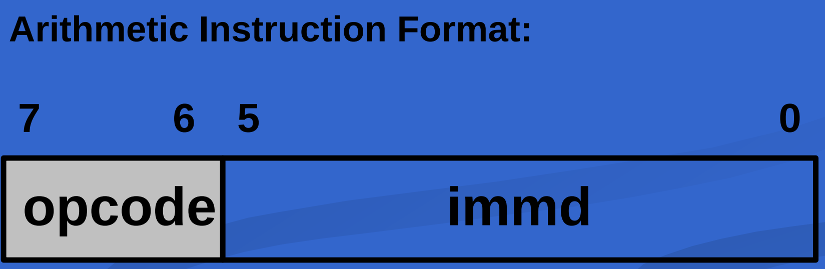
Processor Model Specification



- Interface of the Adding CPU

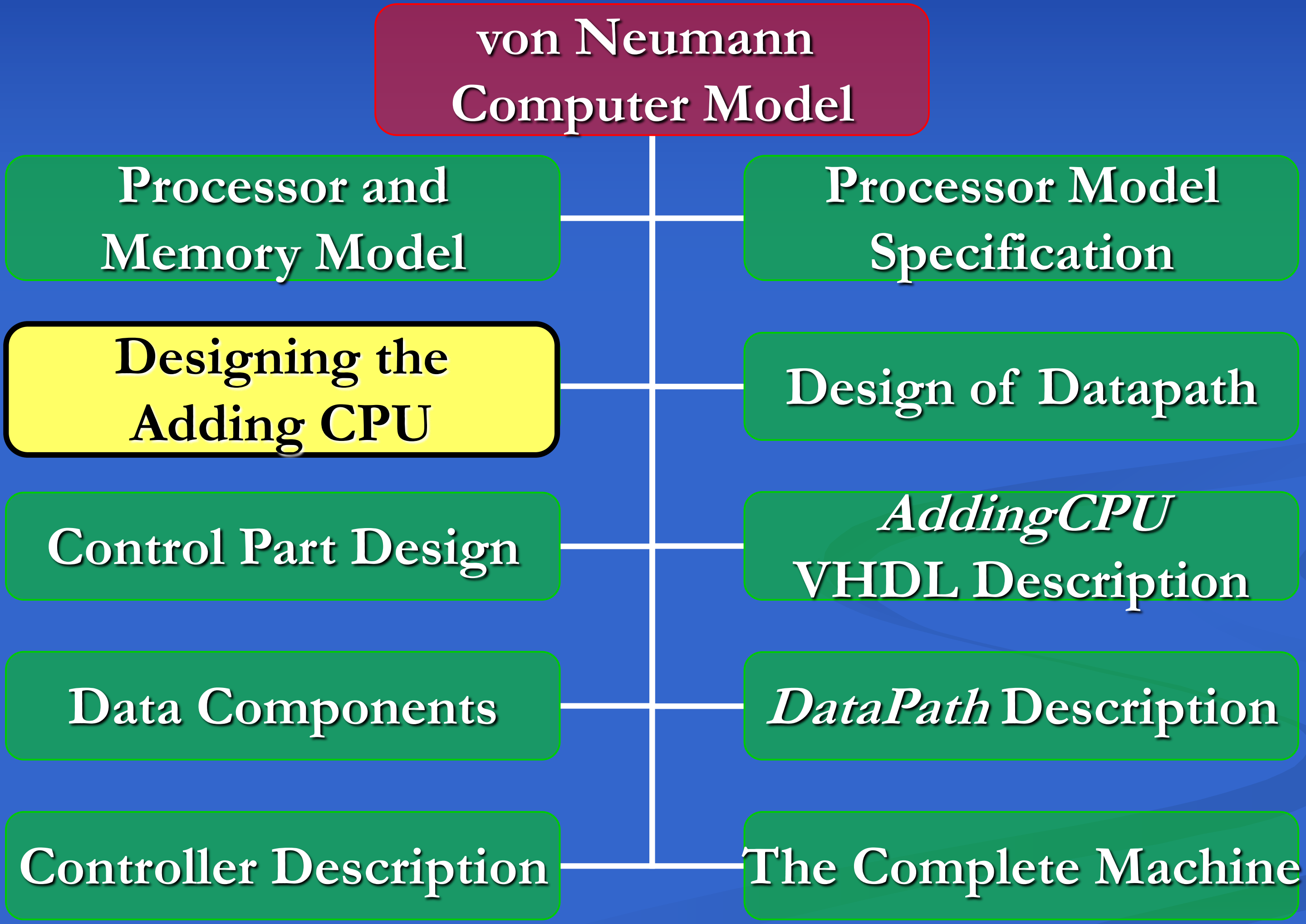
Processor Model Specification



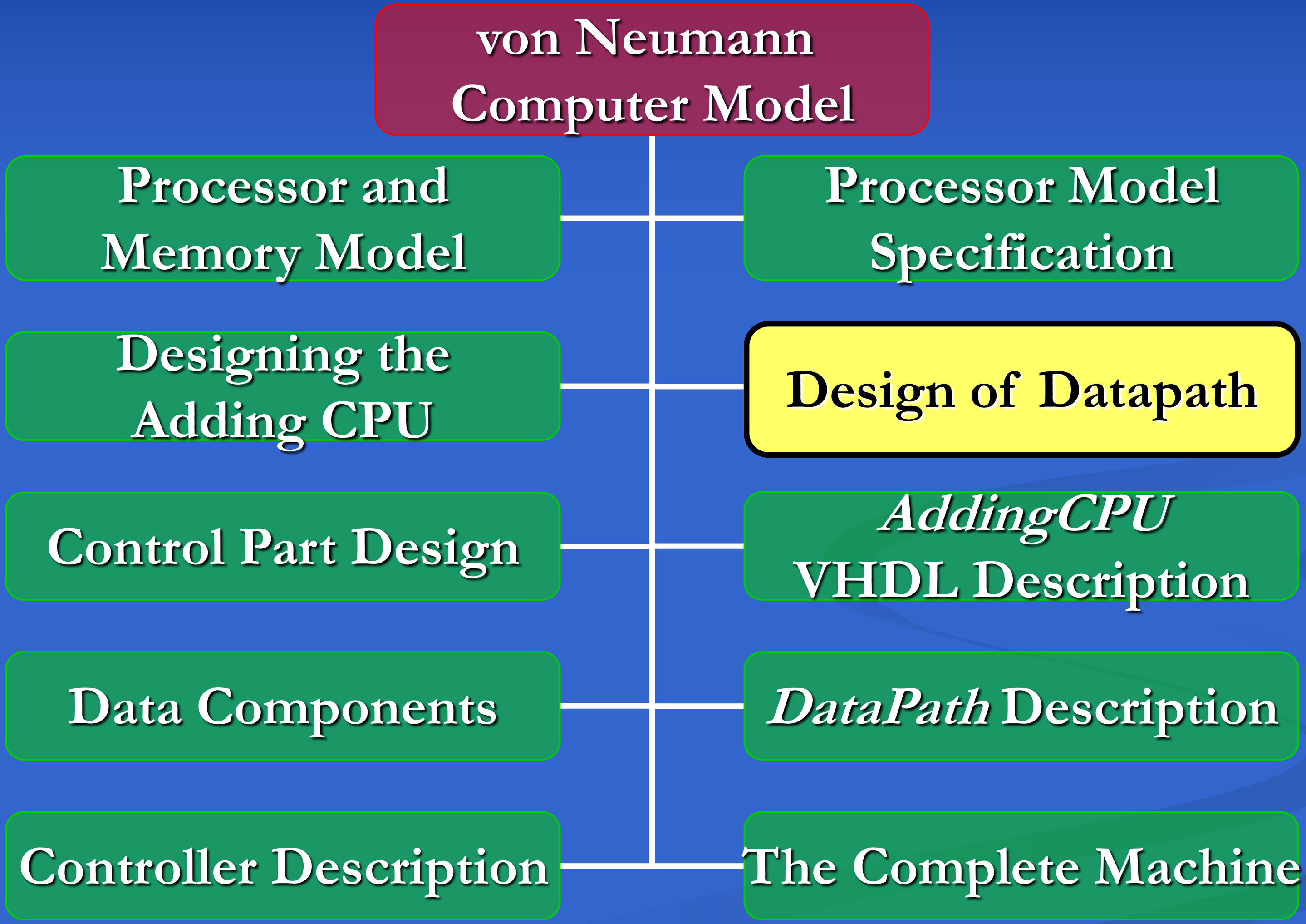


- Instruction Format

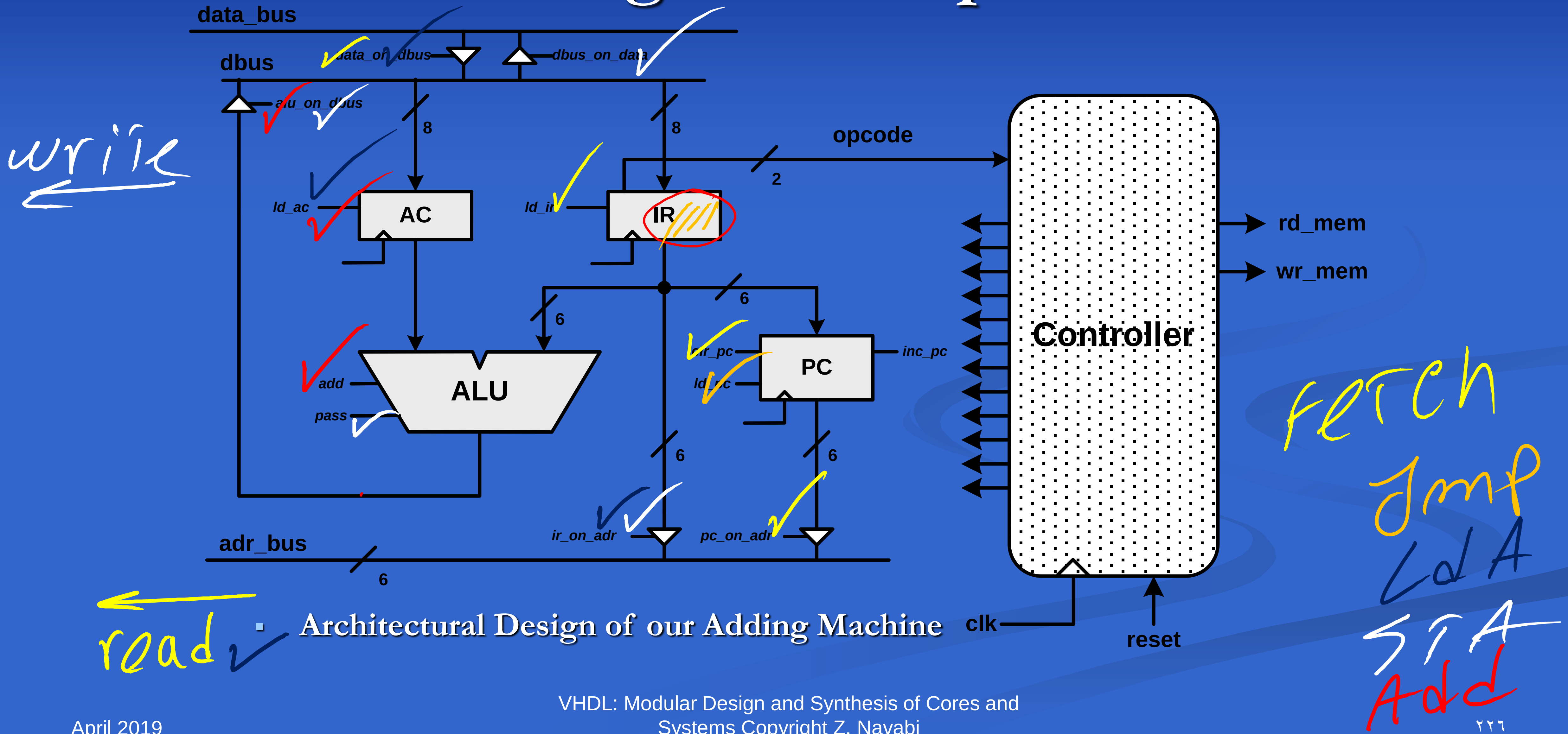
Designing the Adding CPU



Design of Datapath

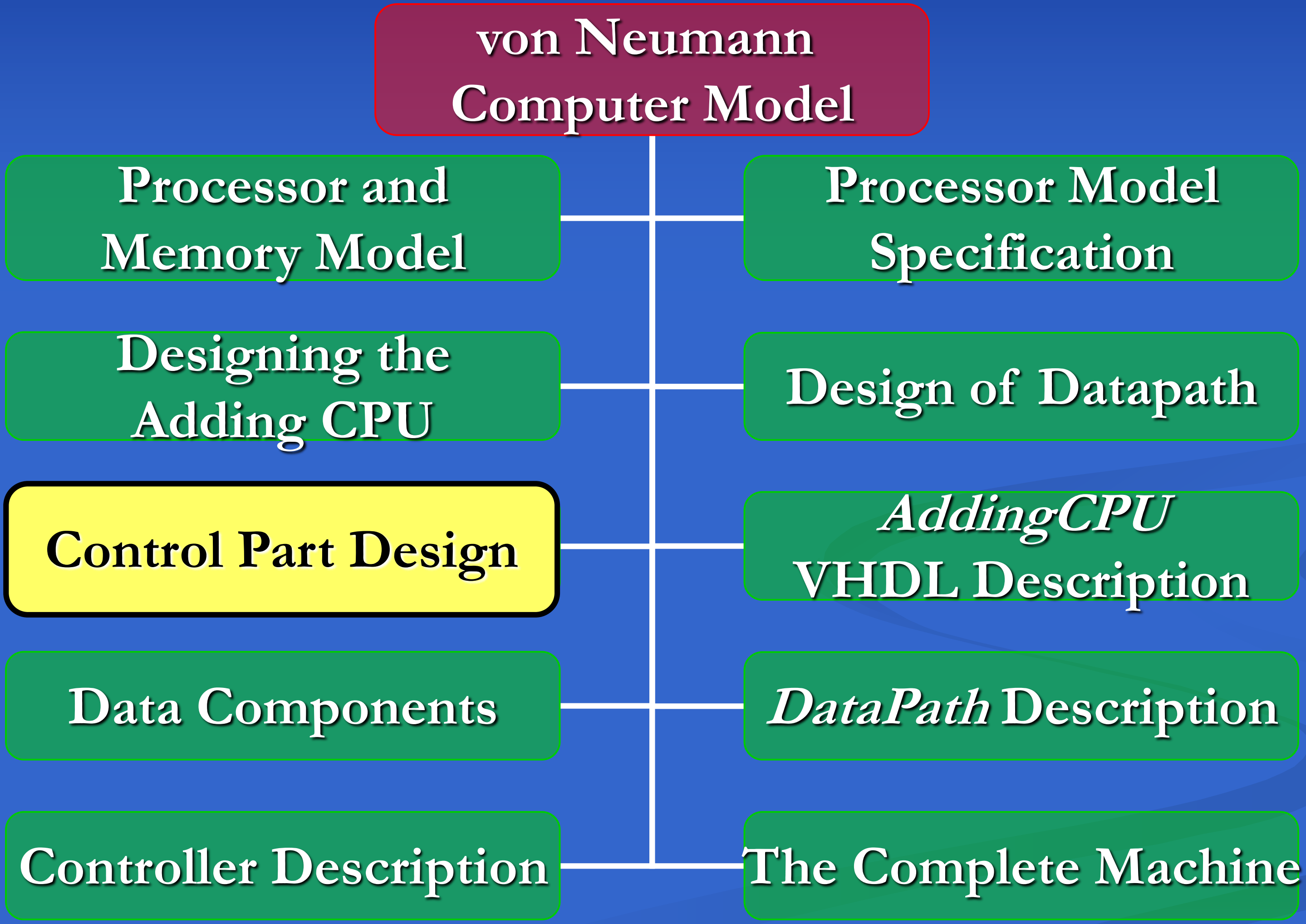


Design of Datapath

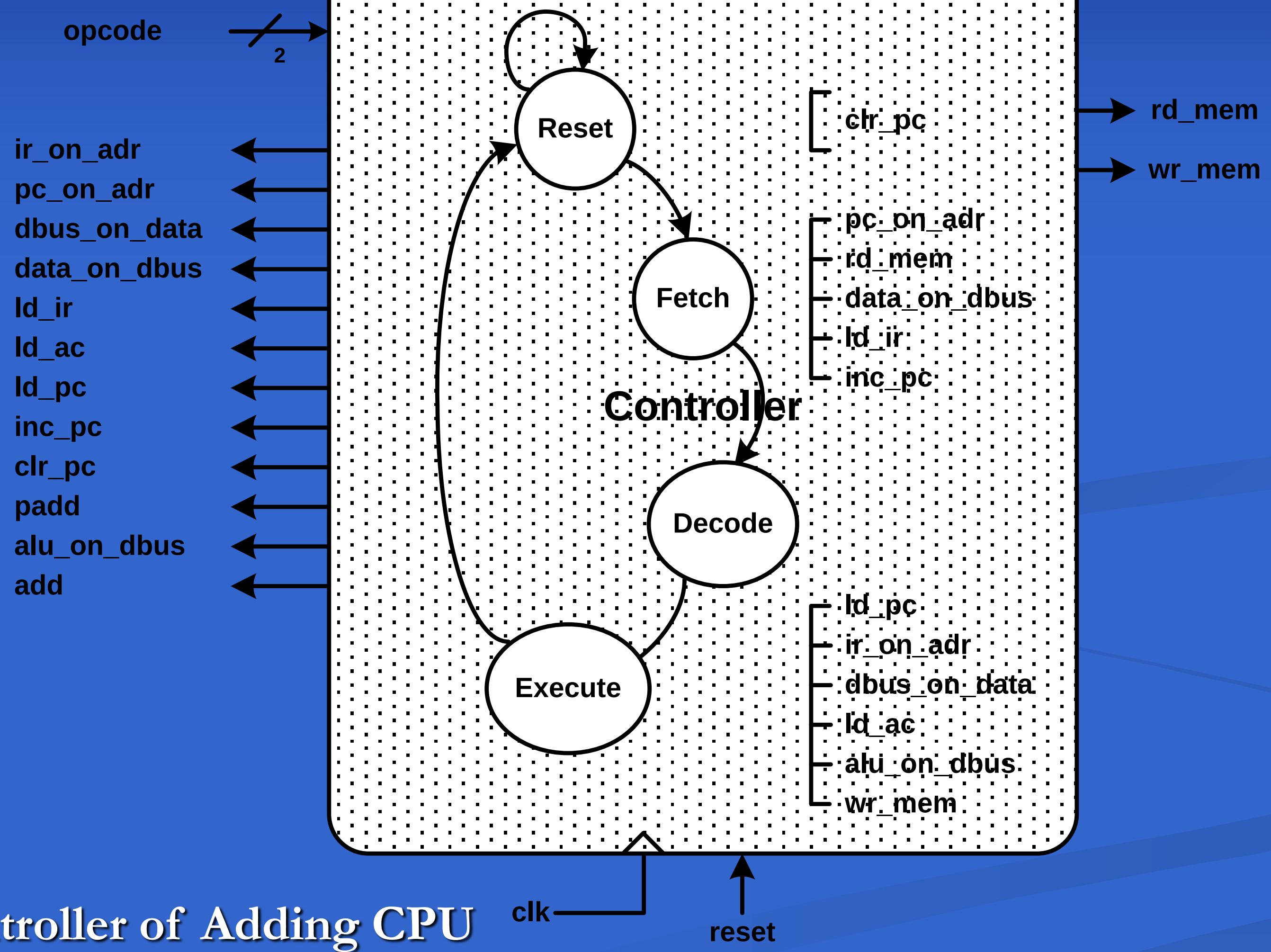


- Architectural Design of our Adding Machine

Control Part Design

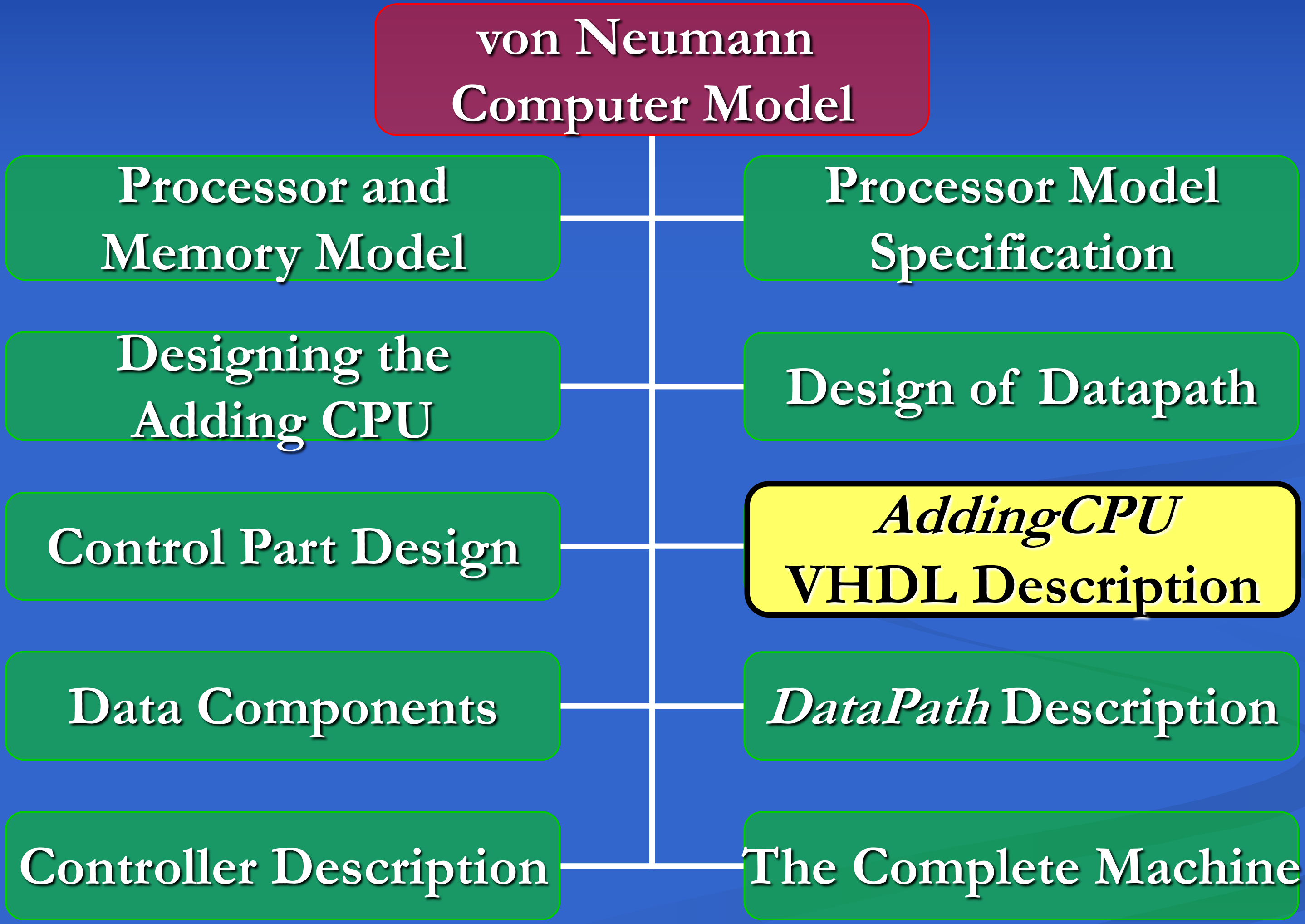


Control Part Design

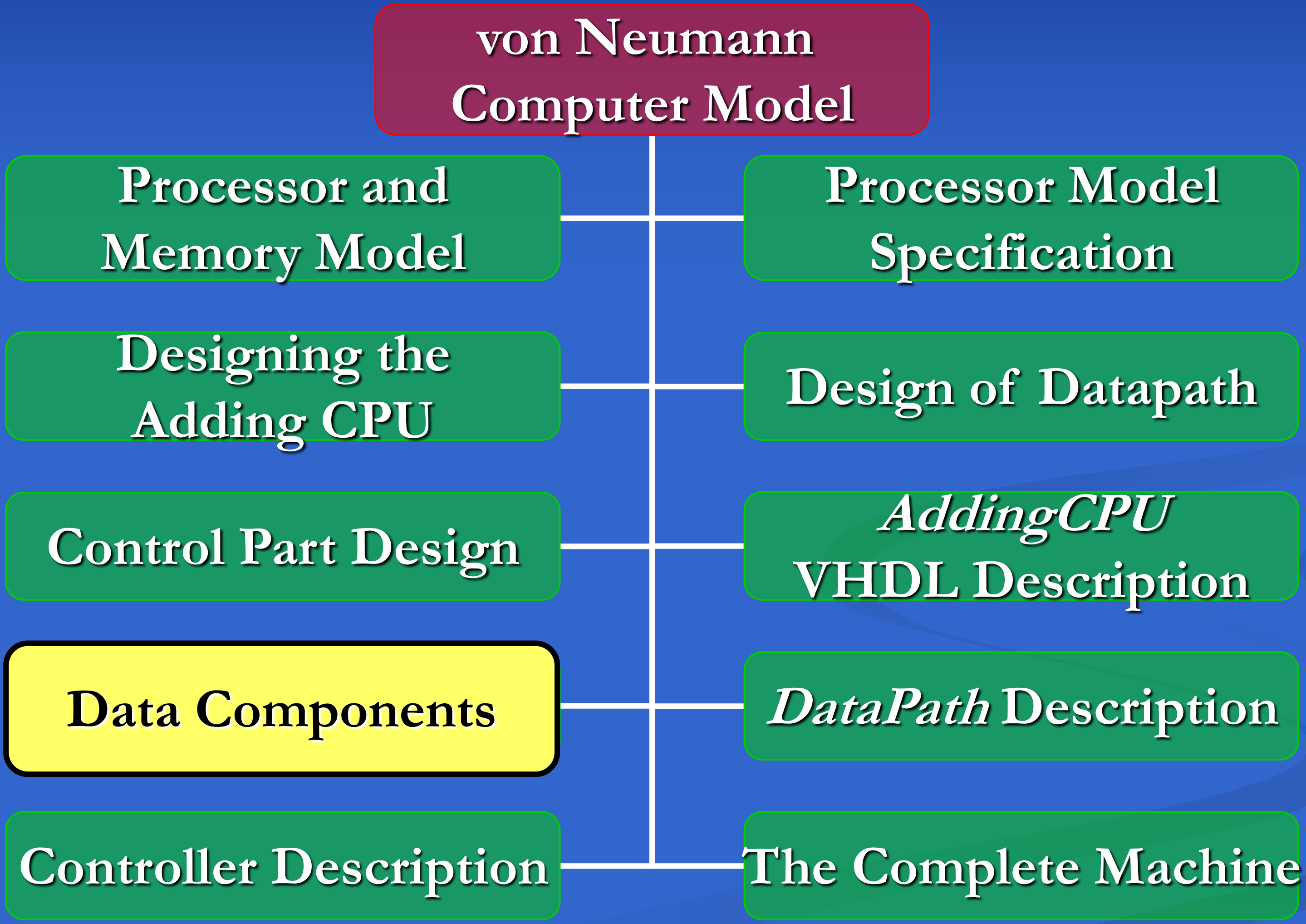


- Controller of Adding CPU

AddingCPU VHDL Description



Data Components



Data Components

```
ENTITY AC IS
   PORT (data_in : IN std_logic_vector(7 DOWNTO 0);
         load, clk : IN std_logic;
         data_out : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE procedural OF AC IS BEGIN
   PROCESS (clk) BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF load = '1' THEN
            data_out <= data_in;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;
```

- Datapath Components of the Adding Machine

Data Components

```
ENTITY IR IS
   PORT (data_in : IN std_logic_vector(7 DOWNTO 0);
         load, clk : IN std_logic;
         data_out : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE procedural OF IR IS BEGIN
   PROCESS (clk) BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF load = '1' THEN data_out <= data_in; END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;
```

- Datapath Components of the Adding Machine (Continued)

Data Components

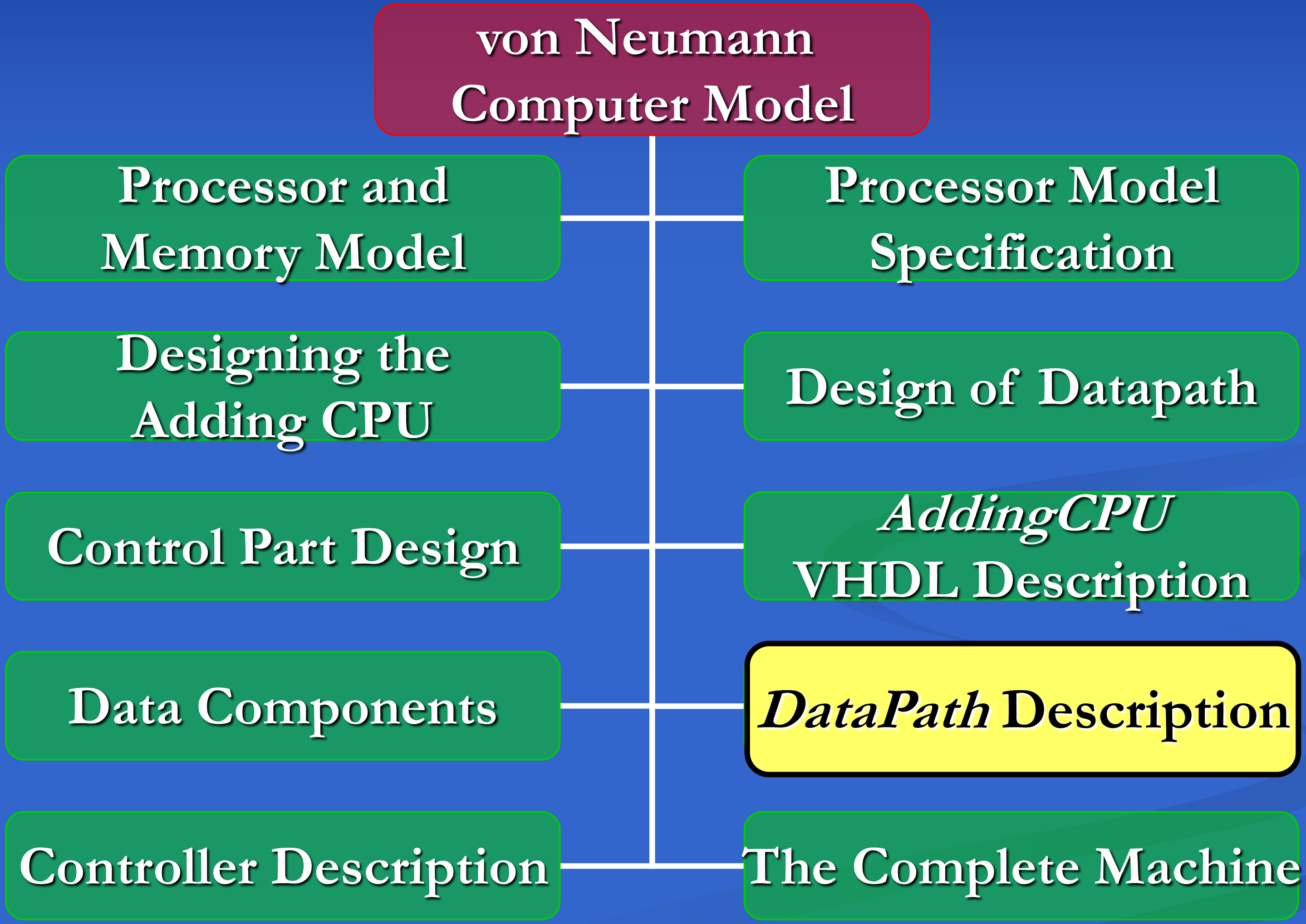
```
ENTITY PC IS
   PORT (data_in : IN std_logic_vector(5 DOWNTO 0);
         load, inc, clr, clk : IN std_logic;
         data_out : OUT std_logic_vector(5 DOWNTO 0));
END ENTITY ;
ARCHITECTURE procedural OF PC IS
   SIGNAL pc : std_logic_vector(5 DOWNTO 0);
BEGIN
   PROCESS (clk) BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF clr = '1' THEN pc <= (OTHERS => '0');
         ELSIF load = '1' THEN pc <= data_in;
         ELSIF inc = '1' THEN  pc <= pc + 1; END IF;
      END IF;
   END PROCESS;
   data_out <= pc;
END ARCHITECTURE;
```

Data Components

```
ENTITY ALU IS
   PORT (a, b : IN std_logic_vector(7 DOWNTO 0);
         pass, add : IN std_logic;
         alu_out : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
ARCHITECTURE functional OF ALU IS
   SIGNAL alu_res : std_logic_vector(7 DOWNTO 0);
BEGIN
   PROCESS (a, b, pass, add) BEGIN
      IF pass = '1' THEN alu_res <= a;
      ELSIF add = '1' THEN alu_res <= a + b;
      ELSE alu_res <= (OTHERS => '0');
      END IF;
   END PROCESS;
   alu_out <= alu_res;
END ARCHITECTURE;
```

- Datapath Components of the Adding Machine (Continued)

DataPath Description



DataPath Description

```
ENTITY datapath IS
   PORT (ir_on_adr, pc_on_adr, dbus_on_data : IN std_logic;
         data_on_dbus, ld_ir, ld_ac, ld_pc : IN std_logic;
         inc_pc, clr_pc,
            pass, add, alu_on_dbus, clk : IN std_logic;
         adr_bus : OUT std_logic_vector(5 DOWNTO 0);
         op_code : OUT std_logic_vector(1 DOWNTO 0);
         data_bus : INOUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
ARCHITECTURE structural OF datapath IS
   SIGNAL dbus, ir_out, a_side :
                std_logic_vector(7 DOWNTO 0);
   SIGNAL alu_out, b_side : std_logic_vector(7 DOWNTO 0);
   SIGNAL pc_out : std_logic_vector(5 DOWNTO 0);
   . . . . . . . . . . . . . . . .
END ARCHITECTURE;
```

- Adding CPU Datapath Description

DataPath Description

```
ARCHITECTURE structural OF datapath IS
BEGIN
   IR :  ENTITY WORK.IR(procedural)
         PORT MAP (dbus, ld_ir, clk, ir_out);
   PC :  ENTITY WORK.PC(procedural)
         PORT MAP (ir_out(5 DOWNTO 0), ld_pc, inc_pc,
                   clr_pc, clk, pc_out);
   AC :  ENTITY WORK.AC(procedural)
         PORT MAP (dbus, ld_ac, clk, a_side);
   ALU : ENTITY WORK.ALU(functional)
         PORT MAP (a_side, b_side, pass, add, alu_out );

   b_side <= '0'&'0'&ir_out(5 DOWNTO 0);
   . . . . . . . . . . . . . . . . . . .
END ARCHITECTURE;
```

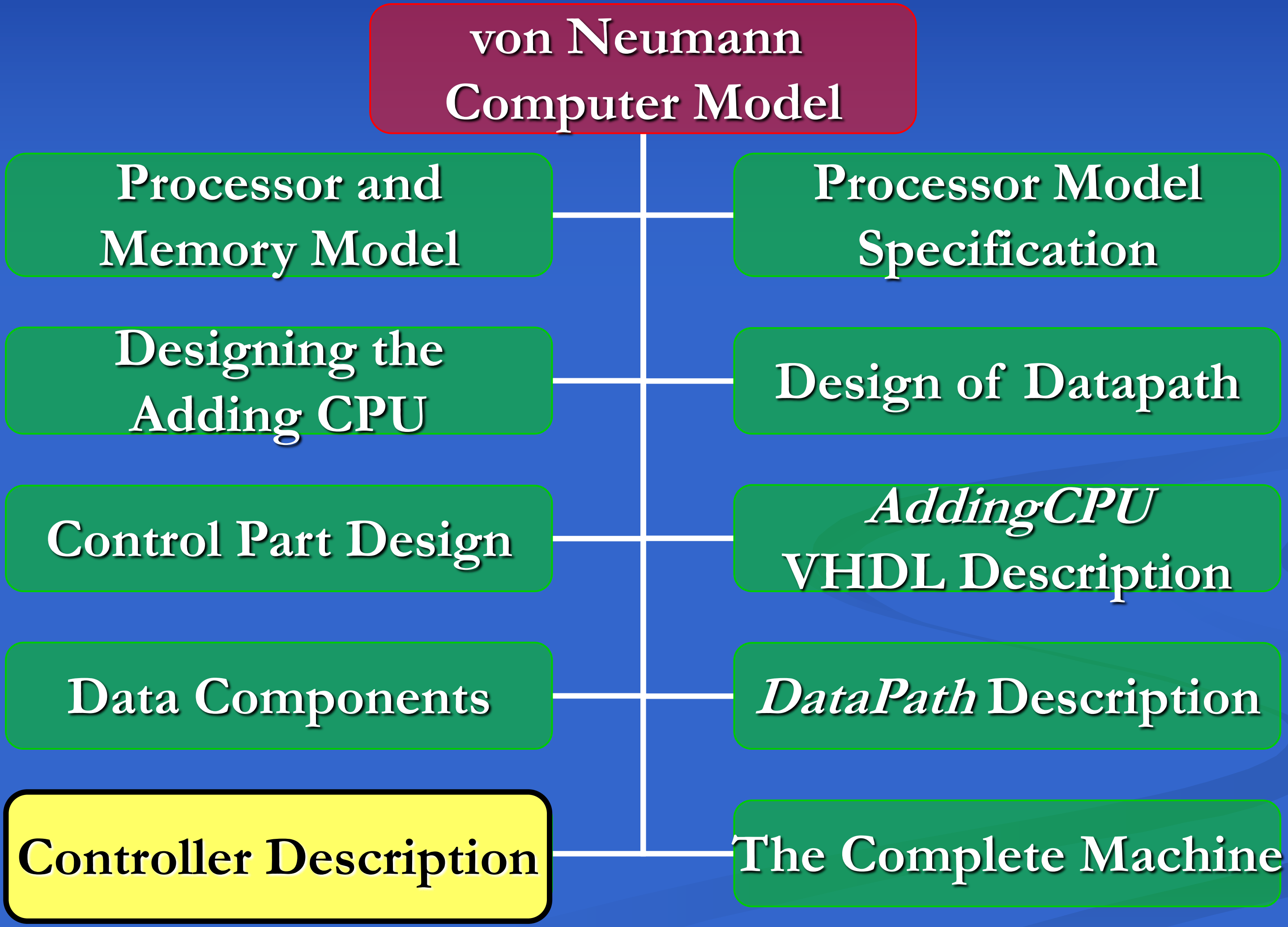
- Adding CPU Datapath Description (Continued)

DataPath Description

```
ARCHITECTURE structural OF datapath IS
BEGIN
   . . . . . . . . . . . . . . . . .
   adr_bus <= ir_out(5 DOWNTO 0) WHEN ir_on_adr = '1'
                 ELSE (OTHERS => 'Z');
   adr_bus <= pc_out WHEN pc_on_adr = '1' ELSE
                 (OTHERS => 'Z');
   dbus <= alu_out WHEN alu_on_dbus = '1' ELSE
              (OTHERS => 'Z');
   data_bus <= dbus WHEN dbus_on_data = '1' ELSE
                  (OTHERS => 'Z');
   dbus <= data_bus WHEN data_on_dbus = '1' ELSE
              (OTHERS => 'Z');
   op_code <= ir_out(7 DOWNTO 6);
END ARCHITECTURE;
```

- Adding CPU Datapath Description (Continued)

Controller Description



Controller Description

```
ENTITY controller IS
   PORT (rst, clk : IN std_logic;
         op_code : IN std_logic_vector(1 DOWNTO 0);
         rd_mem, wr_mem : OUT std_logic;
         ir_on_adr, pc_on_adr : OUT std_logic;
         dbus_on_data, data_on_dbus, ld_ir : OUT std_logic;
         ld_ac, ld_pc, inc_pc, clr_pc,pass : OUT std_logic;
         add, alu_on_dbus : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF controller IS
   TYPE state IS (Reset, Fetch, Decode, Execute);
   SIGNAL present_state, next_state : state;
BEGIN
   . . . . . . . . . . . . . . . . . .
END ARCHITECTURE;
```

- Controller VHDL Code

Controller Description

```
ARCHITECTURE procedural OF controller IS
BEGIN
    PROCESS (clk)--Sequential
    BEGIN
        IF clk = '1' AND clk'EVENT THEN
            IF rst = '1' THEN
                present_state <= Reset;
            ELSE
                present_state <= next_state;
            END IF;
        END IF;
    END PROCESS;
    --
    . . . . . . . . . . . . . . . .
END ARCHITECTURE;
```

- Controller VHDL Code (Continued)

Controller Description

```
PROCESS (present_state, rst) --Combinational
BEGIN
   rd_mem <= '0'; wr_mem <= '0'; ir_on_adr <= '0';
   pc_on_adr <= '0'; dbus_on_data <= '0';
   data_on_dbus <= '0'; ld_ir <= '0'; pass <= '0';
   ld_ac <= '0'; ld_pc <= '0'; inc_pc <= '0';
   clr_pc <= '0'; add <= '0'; alu_on_dbus <= '0';
   CASE present_state IS
      WHEN Reset  =>
         IF rst = '1' THEN
            next_state <= Reset;
         ELSE
            next_state <= Fetch;
         END IF;
         clr_pc <= '1';
     . . . . . . . . . . . . . . .
```

- Controller VHDL Code (Continued)

Controller Description

```
PROCESS (present_state, rst, op_code) --Combinational
BEGIN
     . . . . . . . . . . . . . . . .
     WHEN Fetch  =>
         next_state <= Decode;
         pc_on_adr <= '1';
         rd_mem <= '1';
         data_on_dbus <= '1';
         ld_ir <= '1';
         inc_pc <= '1';
      WHEN Decode  =>
         next_state <= Execute;
      . . . . . . . . . . . . . . . .
 END PROCESS;
  . . . . . . . . . . . . . . .
```

- Controller VHDL Code (Continued)

Controller Description

```
. . . . . . . . . . . . . . . .
WHEN Execute =>
   next_state <= Fetch;
   CASE  op_code IS
      WHEN "00" =>
         ir_on_adr <= '1'; rd_mem <= '1';
         data_on_dbus <= '1'; ld_ac <= '1';
      WHEN "01" =>
         dbus_on_data <= '1';alu_on_dbus<= '1';
         pass <= '1'; wr_mem <= '1';
         ir_on_adr <= '1';
      WHEN "10" =>
         ld_pc <= '1';
      WHEN "11" =>
         add <= '1'; alu_on_dbus <= '1';
         ld_ac <= '1';
```

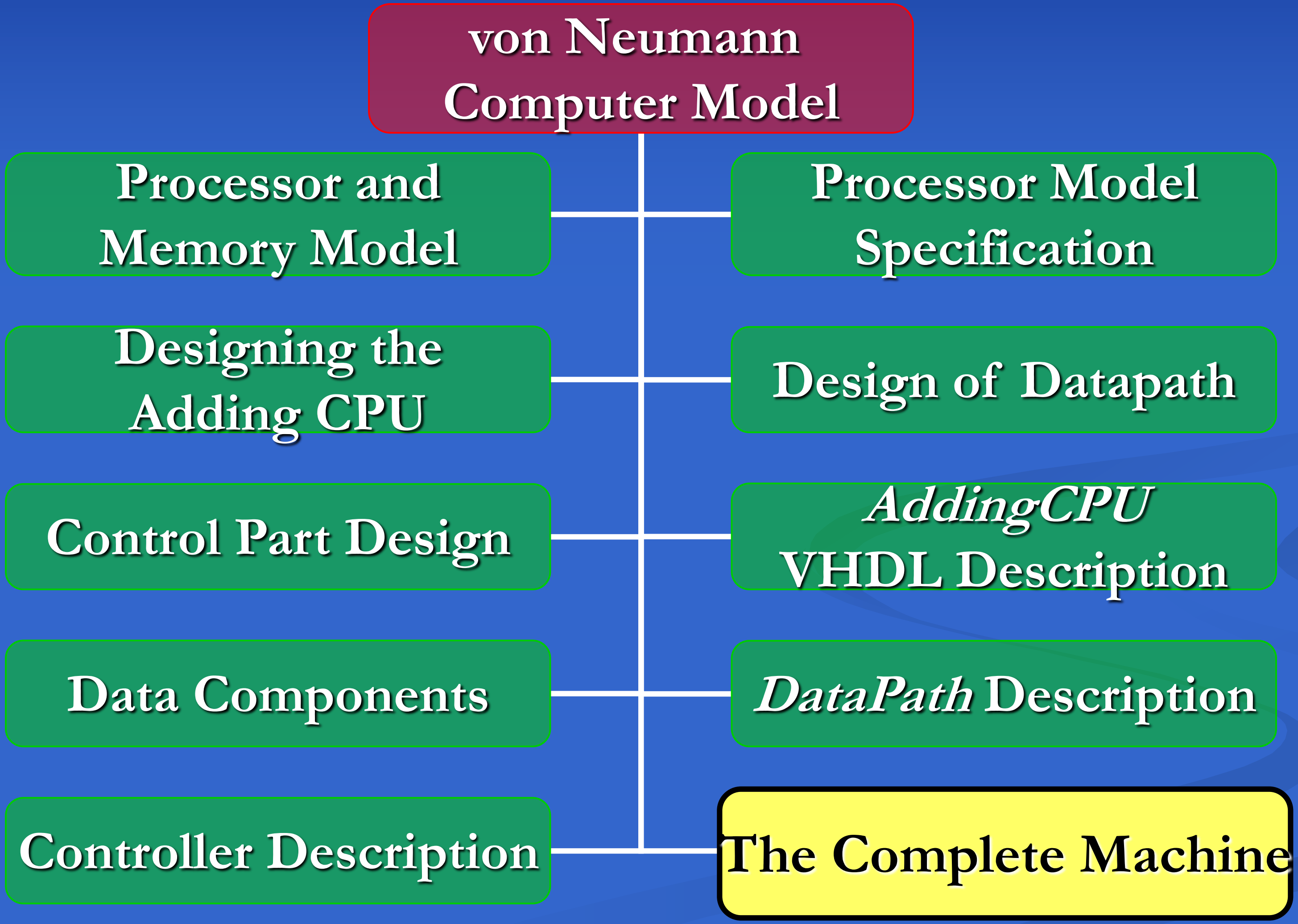
- Controller VHDL Code

Controller Description

```
          . . . . . . . . . . . . . . . .
          WHEN OTHERS =>
                    rd_mem <= '0'; pc_on_adr <= '0';
                    pass <= '0';
                    ir_on_adr <= '0'; wr_mem <= '0';
                    ld_ac <= '0';
                    dbus_on_data <= '0';data_on_dbus<='0';
                    ld_ir <= '0'; alu_on_dbus <= '0';
                    add <= '0';
                    inc_pc <= '0'; clr_pc <= '0';
                    ld_pc <= '0';
              END CASE;
           WHEN OTHERS => next_state <= Reset;
       END CASE;
    END PROCESS;
END ARCHITECTURE;
```

- Controller VHDL Code

The Complete Machine



The Complete Machine

```
ENTITY addingCPU IS
   PORT (reset, clk : IN std_logic;
         adr_bus : OUT std_logic_vector(5 DOWNTO 0);
         rd_mem, wr_mem : OUT std_logic;
         data_bus : INOUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE structural OF addingCPU IS
   SIGNAL ir_on_adr, pc_on_adr, dbus_on_data : std_logic;
   SIGNAL data_on_dbus, ld_ir, ld_ac, ld_pc : std_logic;
   SIGNAL inc_pc, clr_pc : std_logic;
   SIGNAL pass, add, alu_on_dbus : std_logic;
   SIGNAL op_code : std_logic_vector(1 DOWNTO 0);
BEGIN
   . . . . . . . . . . . . . . . .
END ARCHITECTURE;
```

- AddingCPU Top-level Description

The Complete Machine

```
ARCHITECTURE structural OF addingCPU IS
BEGIN
   CU: ENTITY WORK.Controller
       PORT MAP (reset, clk, op_code, rd_mem,
                 wr_mem, ir_on_adr, pc_on_adr,
                 dbus_on_data, data_on_dbus,
                 ld_ir, ld_ac, ld_pc, inc_pc,
                 clr_pc, pass, add, alu_on_dbus );
   DP: ENTITY WORK.DataPath
       PORT MAP (ir_on_adr, pc_on_adr, dbus_on_data,
                 data_on_dbus,ld_ir, ld_ac, ld_pc,
                 inc_pc, clr_pc, pass, add,
                 alu_on_dbus, clk, adr_bus, op_code,
                 data_bus );
END ARCHITECTURE;
```

- AddingCPU Top-level Description (Continued)

Summary

- This chapter presented VHDL code and descriptions for several hardware components.

- Emphasized on synthesizable cores
- Considered situations that a core model was to be for evaluation purposes only

- We discussed:
  - Individual stand-alone component descriptions
  - Design partitioning
  - Putting sub-components of a system together for formation of complete systems.

Acknowledgment

Slides developed by:

Homa Alemzadeh

Edited December 2017, by:

Bahar Behazin

Last edited April 2019, by:

Saba Yousefzadeh