Lecture 10- High-Level Synthesis Optimizations

Benjamin Carrion Schaefer
Associate Professor
Department of Electrical and Computer Engineering

ERIK JONSSON SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
The University of Texas at Dallas

Objectives of this Lecture

- High Level Synthesis optimizations
  - FU constraint
  - Global options
  - Synthesis directives (pragmas)
- Loops
  - Unroll
  - Fold (pipeline)
- Functions
  - Inline, goto, functional unit conversion
- Arrays
  - RAM, regsiters, ROM, logic
- Advanced Synthesis directives
- Design Space Exploration

Ref: B. Carrion Schaefer, High-Level Synthesis Made Easy, 1st Edition,– Chapters 5,6

Lecture + Homework Synchronization

- Lecture 1: Intro
- Lecture 2: FPGA architecture
  - Full adder schematic & RTL
  - HW1 (FA)
- Lecture 3: FPGA programing
  - DE1-SoC board
  - HW2 (LEDs)
- Lecture 4: FPGA IPs
  - IP on DE1-SoC board
  - HW3
- Lecture 5: Physical Design
  - Physical Design Chip Planner
  - HW4 (Timing)
- Lecture 6: Clocks &Power
- Lecture 7: Pwr est.
  - Clock and Power Estimation VGA
  - HW5 (VGA)
- Lecture 8+9: High-Level Synthesis
  - High-Level Synthesis (CyberWorkBench)
  - HW6 (ave8)
- Lecture 10+11: HLS optimizations
  - HLS optimizations
  - HW7 (bubble sort)
- Lecture 12: Embedded Processor
  - Embedded Processor (ARM)
  - HW8 (linux+app)
- Lecture 13: HW/SW Co-design
  - HW/SW co-design
  - HW9
- Lecture 14: FPGA applications

Making C code synthesizable

- Non-synthesizble constructs:
  - scanf, printf, system calls

```
int idata;
int odata;
main( ){
 int tmp = 0;
 scanf("%d",&idata);
 while(tmp < 5) {
    idata += tmp;
    tmp++;
 }
 odata = idata;
 printf("%d",odata);
}
```

```
in ter(7..0) idata;
out ter(8..0) odata;
process main( ){
var(2..0) tmp = 0;
var(7..0) idatai ;
idatai = idata ;
while(tmp < 5) {
    idatai += tmp;
    tmp++;
}
odata = idatai;
}
```

```
in reg(7..0) idata;
out reg(8..0) odata;
process main( ){
var(2..0) tmp = 0;
var(7..0) idatai ;
idatai = idata ;
while(tmp < 5) {
    idatai += tmp;
    tmp++;
}
odata = idatai;
}
```

Preparing ANSI-C vs. BDL

- Use '#define' to replace enhanced syntax to ANSI-C syntax

```
in ter(7..0) idata;
out ter(8..0) odata;
process main( ){
var(2..0) tmp = 0;
var(7..0) idatai ;
idatai = idata ;
while(tmp < 5) {
   idatai += tmp;
   tmp++;
}
odata = idatai;
}
```

➡

```
#ifdef C
  #define in
  # define out
  #define ter(x)  int
  #define var(x) int
  #define process
#endif

in ter(7..0) idata;
out ter(8..0) odata;
process main( ){
var(2..0) tmp = 0;
var(7..0) idatai ;
idatai = idata ;
while(tmp < 5) {
   idatai += tmp;
   tmp++;
}
odata = idatai;
}
```

%gcc foo.c –DC –o foo.exe

Arithmetic Operations

- u – unsigned upper – bit width resulting from expressions
- s – signed lower – sign type resulting from expressions

A op B	+	-	*	/
u(0:n) op u(0:m)	unsigned max(n,m)+1	signed max(n,m)+1	unsigned n+m	unsigned n
s(0:n) op s(0:m)	signed max(n,m)+1	signed max(n,m)+1	signed n+m	signed n+1
s(0:n) op u(0:m)	signed max(n,m+1)+1	signed max(n,m+1)+1	signed n+(m+1)	signed n
u(0:n) op s(0:m)	signed max(n+1,m)+1	signed max(n+1,m)+1	signed (n+1)+m	signed n+1

Example – Bubble Sort

- Modify the original C code to be synthesizable
- The modified code should be used for both simulation and synthesis
- Declare input and output port as follows:

  in var(7..0) indata;

  out var(7..0) otdata;
- **Hints:**
  - Disable #include statements
  - Specify process as top function
  - Display file IO statements

The following style using "ifdef" statements is recommended:

#ifdef C
 Effective only for C-simulation
#else
 Effective only for HLS
#endif

```c
#include <stdio.h>
#include <stdlib.h>

#define  DATANUM  8

int main(void){

    int b_ary[DATANUM] ;
    int cnti ,cntj ,tmp ;

    FILE  *fp1, *fp2 ;

    if ((fp1 = fopen("indata.txt", "r")) == NULL) {
        fprintf(stderr, "file indata.txt not open \n");
        exit(1);
    }

    if ((fp2 = fopen("otdata.txt", "w")) == NULL) {
        fprintf(stderr, "file otdata.txt not open \n");
        exit(1);
    }

    for(cnti = 0 ; cnti < DATANUM ; cnti++){
        if (fscanf(fp1, "%d", &b_ary[cnti]) == EOF) exit(0);
    }

    for(cnti = 0 ; cnti < DATANUM ; cnti++){
        for(cntj = DATANUM-1 ; cntj > cnti ; cntj--){
            if(b_ary[cntj] < b_ary[cntj-1]){
                tmp            = b_ary[cntj] ;
                b_ary[cntj]    = b_ary[cntj-1] ;
                b_ary[cntj-1]  = tmp ;
            }
        }
    }

    for(cnti = 0 ;cnti < DATANUM ;cnti++){
        fprintf(fp2, "%d\n", b_ary[cnti]) ;
    }
    fclose(fp1) ;
    fclose(fp2) ;
    return 0 ;
}
```

Example – Bubble Sort

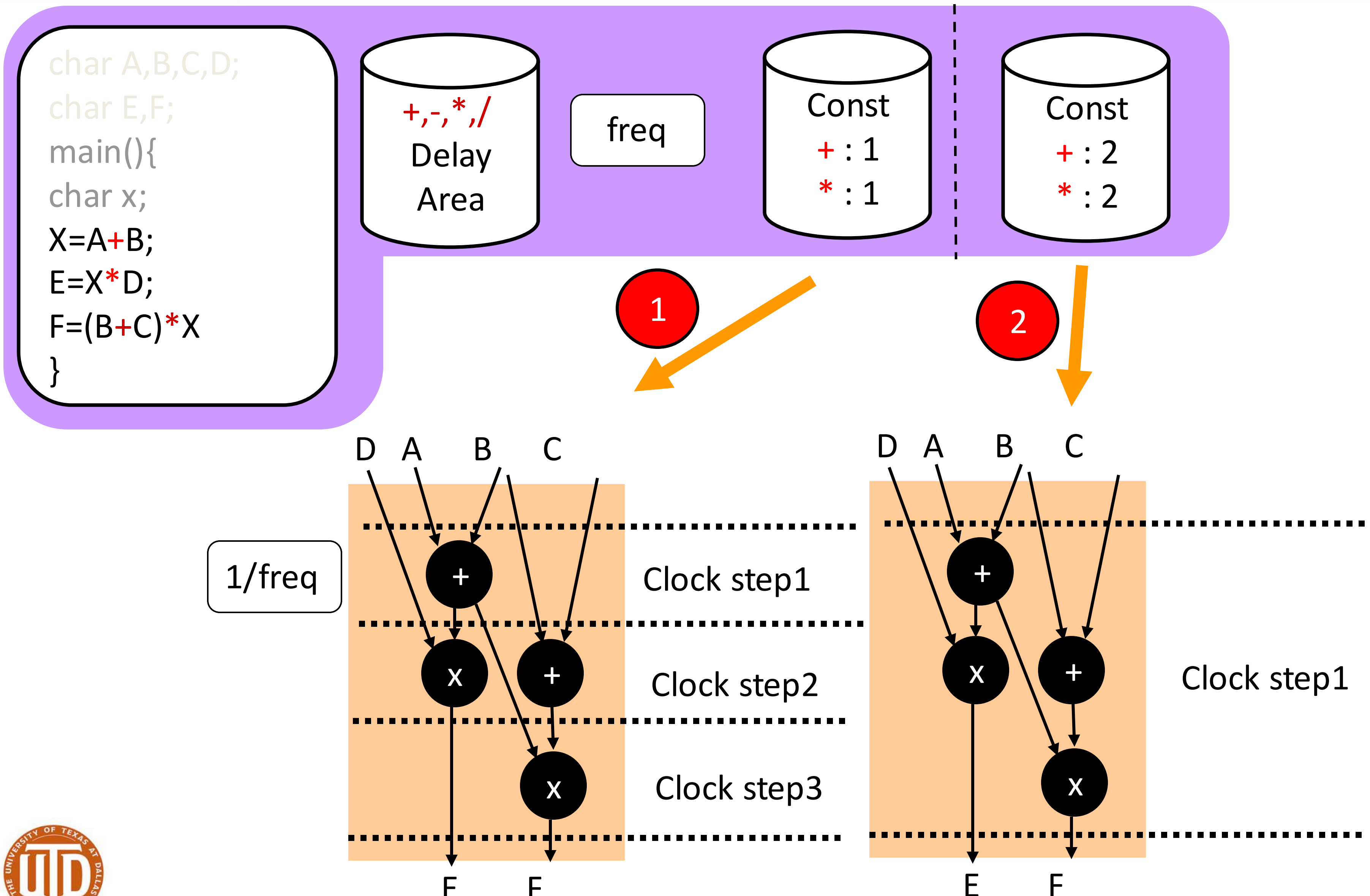
```
#include <stdio.h>
#include <stdlib.h>

#define  DATANUM  8

int main(void){

        int b_ary[DATANUM] ;
        int cnti ,cntj ,tmp ;

        FILE  *fp1, *fp2 ;

        if ((fp1 = fopen("indata.txt", "r")) == NULL) {
                fprintf(stderr, "file indata.txt not open \n");
                exit(1);
        }

        if ((fp2 = fopen("otdata.txt", "w")) == NULL) {
                fprintf(stderr, "file otdata.txt not open \n");
                exit(1);
        }

   for(cnti = 0 ; cnti < DATANUM ; cnti++){
                if (fscanf(fp1, "%d", &b_ary[cnti]) == EOF) exit(0);
        }

                    :                              :
```

```
#ifdef C
#include <stdio.h>
#include <stdlib.h>
#endif

#define  DATANUM  8

#ifndef C
in  var(7..0) indata ;
out var(7..0) otdata ;
#endif

#ifdef C
int main(void){
#else
 process main(){
#endif
        int b_ary[DATANUM] ;
        int cnti ,cntj ,tmp ;
        #ifdef C
        FILE  *fp1, *fp2 ;

        if ((fp1 = fopen("indata.txt", "r")) == NULL) {
                fprintf(stderr, "file indata.txt not open \n");
                exit(1);
        }

        if ((fp2 = fopen("otdata.txt", "w")) == NULL) {
                fprintf(stderr, "file otdata.txt not open \n");
                exit(1);
        }
        #endif
```

```
#ifdef C
#include <stdio.h>
#include <stdlib.h>
#endif
// #define  DATANUM  8
#ifndef C
in  var(7..0) indata ,index;
out var(7..0) otdata ;
#endif

#ifdef C
int main(void){
#else
process main(){
#endif
        int b_ary[100] ;
        int cnti ,cntj ,tmp ;
        #ifdef C
        int index_in;
        #endif

        #ifdef C
        FILE  *fp1, *fp2 ,*fp3 ;

        if ((fp1 = fopen("indata.txt", "r")) == NULL) {
                fprintf(stderr, "file indata.txt not open \n");
                exit(1);
        }

        if ((fp2 = fopen("otdata.txt", "w")) == NULL) {
                fprintf(stderr, "file otdata.txt not open \n");
                exit(1);
        }
        if ((fp3 = fopen("index.txt", "r")) == NULL) {
                fprintf(stderr, "file index.txt not open \n");
                exit(1);
        }
```

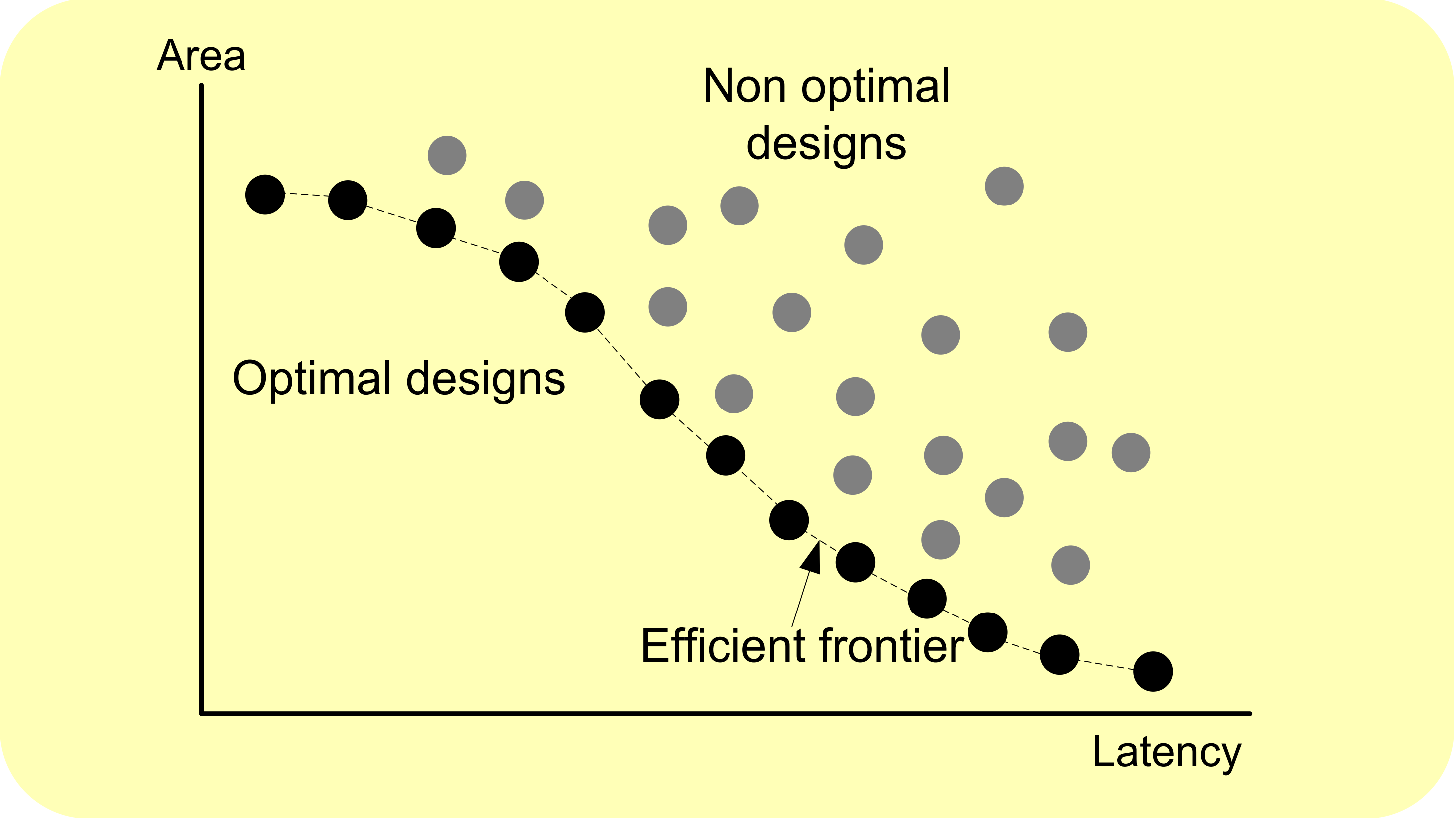
```
#ifdef C
        fscanf(fp3,"%d",&index) ;
#else
        index_in=index;
#endif

    for(cnti = 0 ; cnti < index_in ; cnti++){
                #ifdef C
                if (fscanf(fp1, "%d", &b_ary[cnti]) == EOF) exit(0);
                #else
                 b_ary[cnti] = indata ;
                #endif
        }

    for(cnti = 0 ; cnti < index_in ; cnti++){
                for(cntj = index_in-1 ; cntj > cnti ; cntj--){
                        if(b_ary[cntj] < b_ary[cntj-1]){
                                tmp          = b_ary[cntj] ;
                                b_ary[cntj]   = b_ary[cntj-1] ;
                                b_ary[cntj-1] = tmp ;
                        }
                }
        }

    for(cnti = 0 ;cnti < index_in ;cnti++){
                #ifdef C
                fprintf(fp2, "%d\n", b_ary[cnti]) ;
                #else
                otdata = b_ary[cnti] ;
                #endif
        }
        #ifdef C
        fclose(fp1) ;
        fclose(fp2) ;
        return 0 ;
        #endif
}
```

Benefits of HLS: Automatic Alternative Architecture Generation



Benefits of HLS: Architectural Design Space Exploration

- HLS allows to generate multiple functional equivalent hardware implementations → Out of all only interested in Pareto-optimal designs
- What other options apart from FU constraints?



Controlling the Resulting HW

1. Through the resource constraint file
   - Limit the number of FUs that can be instantiated
2. Though global synthesis options → apply to the entire design
   - Maximum frequency
   - Map all arrays to memory or registers
   - FSM encoding
3. Local synthesis directives (pragmas). Comments with keyword (e.g., // Cyber unroll_times = all)
   - **Loops synthesis** (e.g., unroll, partial, no unroll, pipeline)
   - **Functions synthesis** (e.g., inline, goto)
   - **Array synthesis** (e.g., memory or registers)

1. Resource Constraint Control

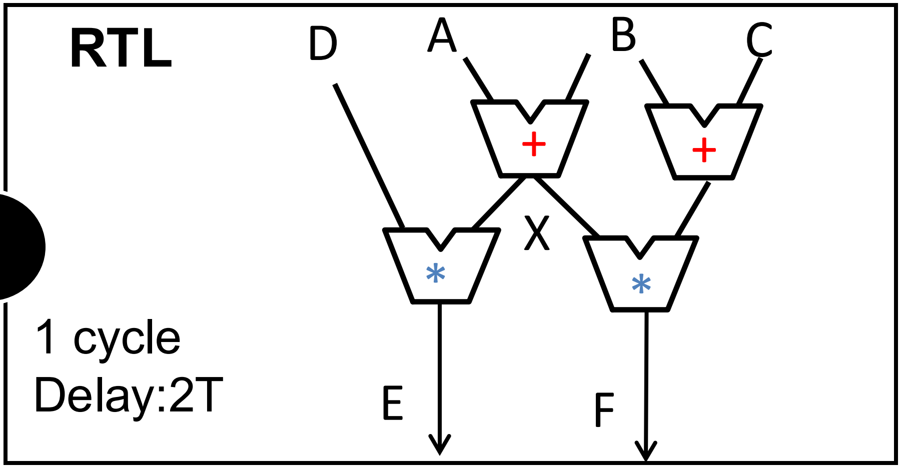
Behavioral Description in C

```
char A,B,C,D;
char E,F;
main(){
char X;
X = A + B;
E = X * D;
F = (B + C) * X;
}
```

FU constraints

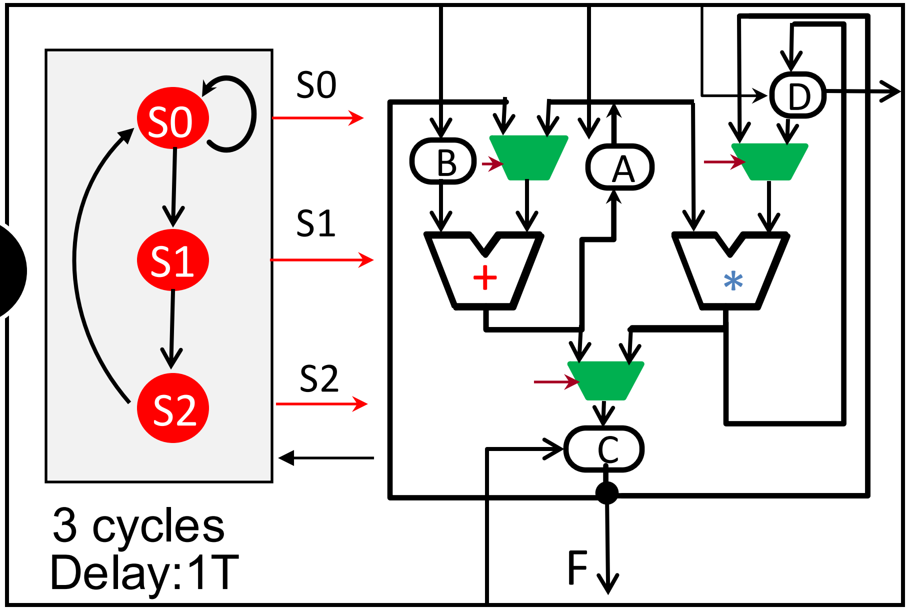
1

+ : 2
* : 2

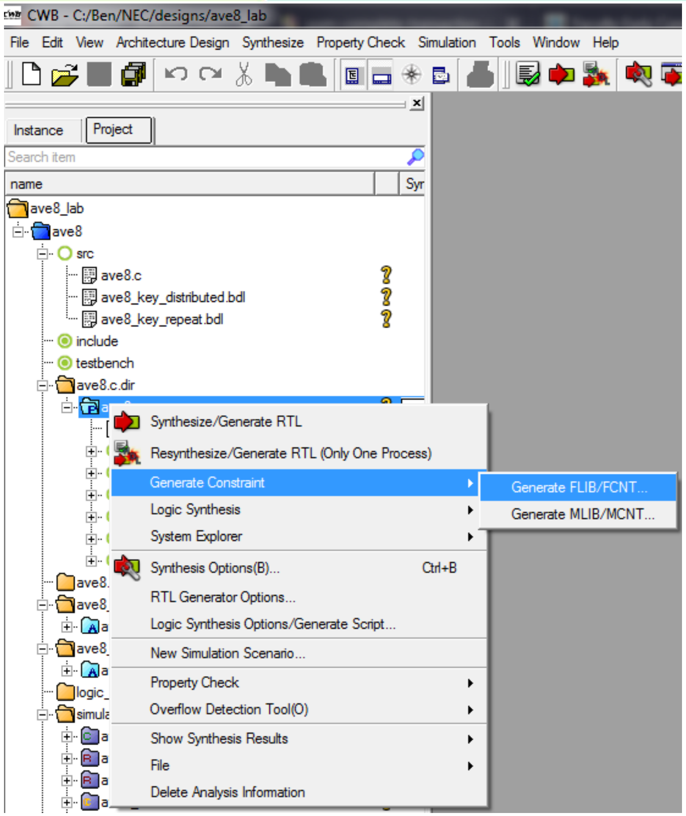


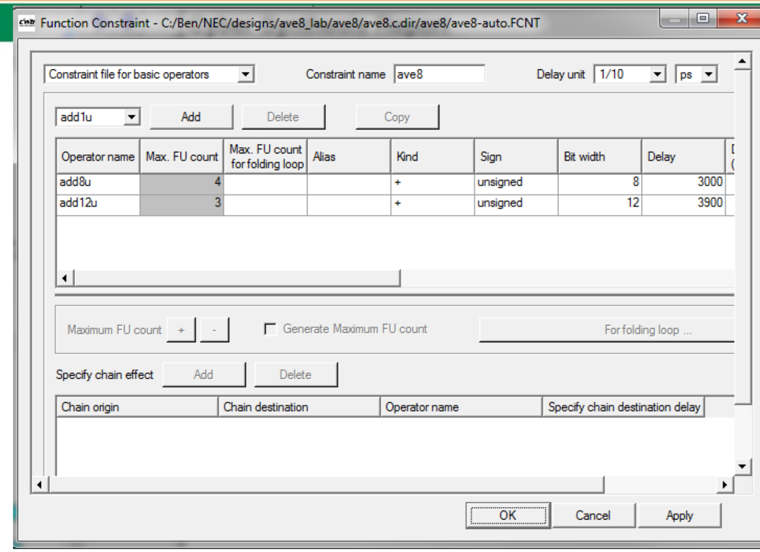
2

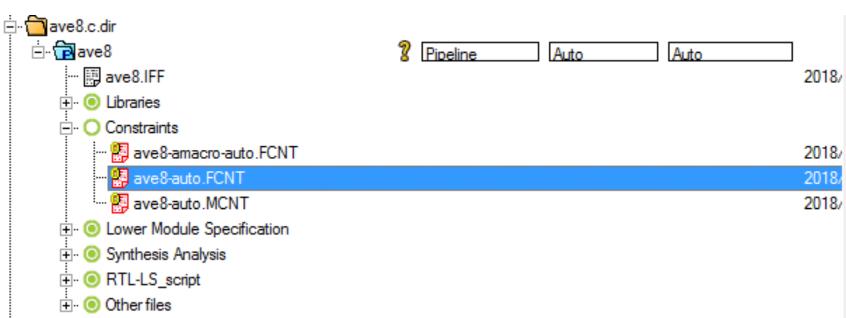
+ : 1
* : 1



1. Resource Constraint (FNCT)





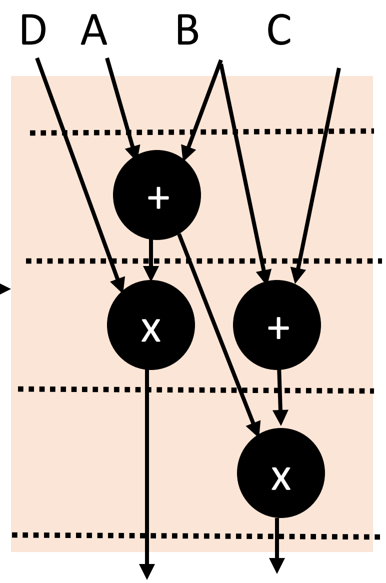



```
#@VERSION{3.00}
#@CNT{ave8}
#@KIND{BASIC_OPERATOR}
#@CLK 200000
#@UNIT 1/10ps
@FCNT{
        NAME  add8u
        LIMIT  4  AUTO
#       COMMENT
}
@FCNT{
        NAME  add12u
        LIMIT  3  AUTO
#       COMMENT
}
#@HASH{b6836427a6977baf58df8e5ddf1f14d7}
#@END{ave8}
```

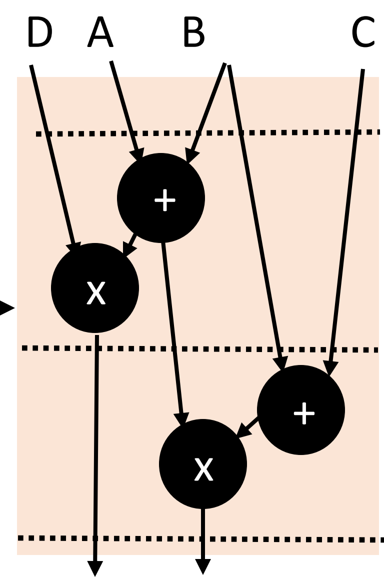
2. Global Options: Synthesis Mode

1. Automatic synthesis
2. Pipelined
3. <u>Manual synthesis</u>

```
#ifdef C
  #define $
#endif
int A,B,C,D;
int E,F;
main(){
int X;
X = A + B;
$
E = X * D;
$
F = (B + C) * X;
}
```

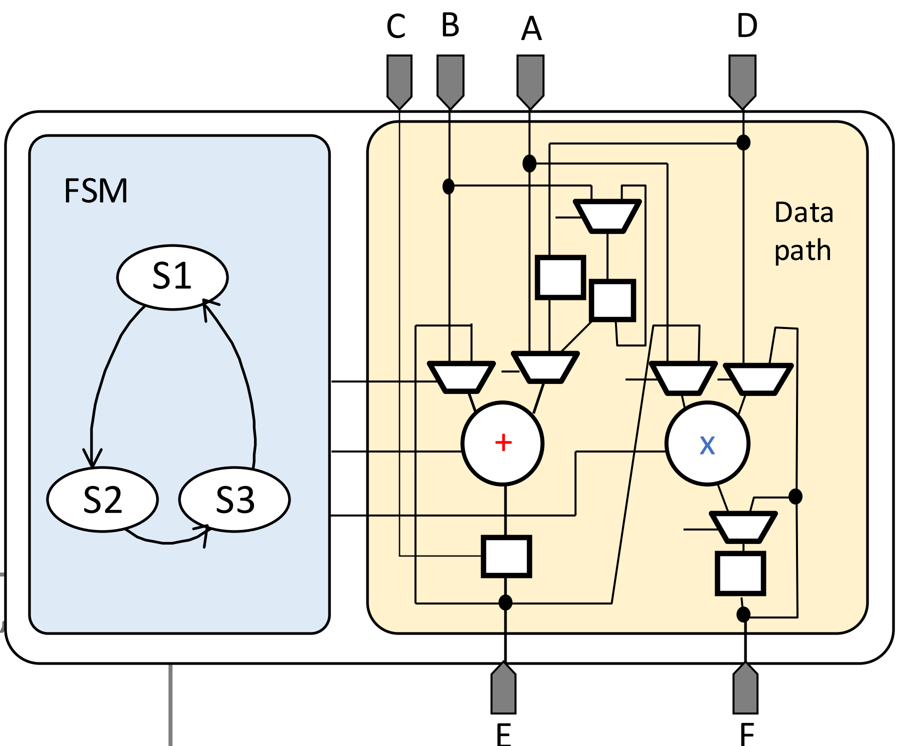



```
#ifdef C
  #define $
#endif
int A,B,C,D;
int E,F;
main(){
int X;

X = A + B;
E = X * D;
$
F = (B + C) * X;
}
```

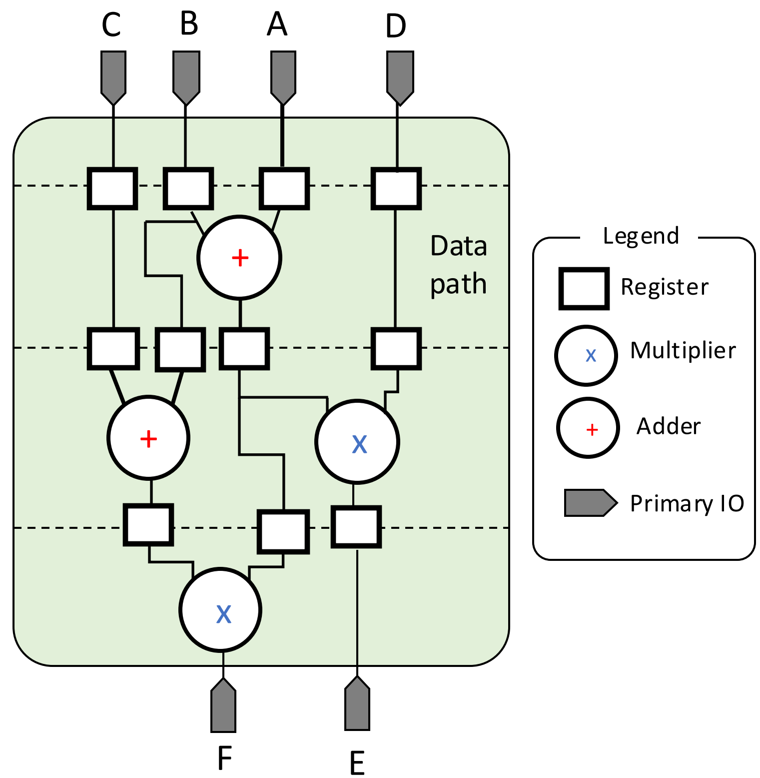


2. Global Options: Pipelining

- Pipelining whole descriptions → Automatic pipeline mode
- Pipelining some loops in a description → Loop folding



Sequential Circuit



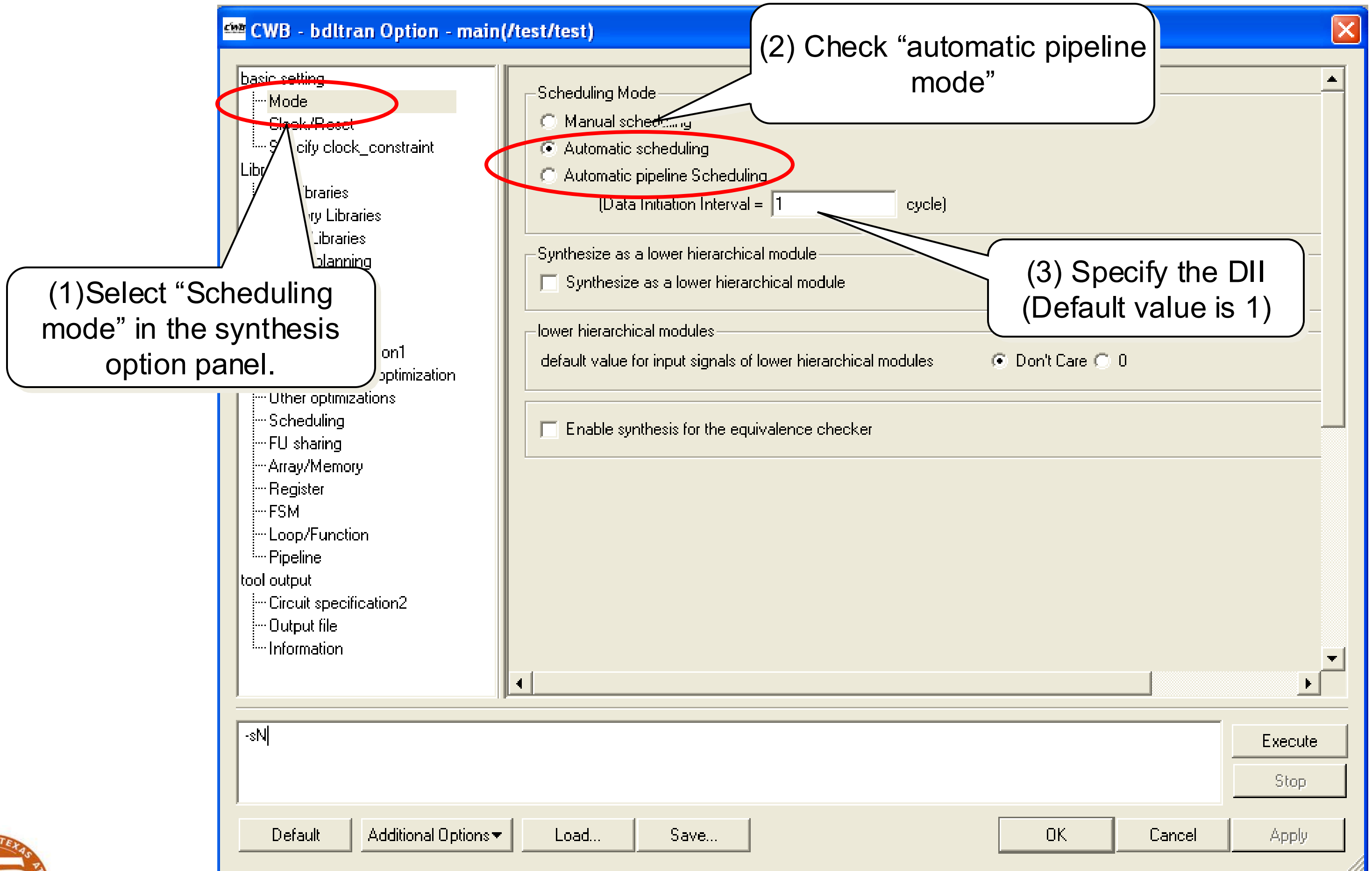
Pipelined circuit

All steps are executed simultaneously.
(High throughput circuit)

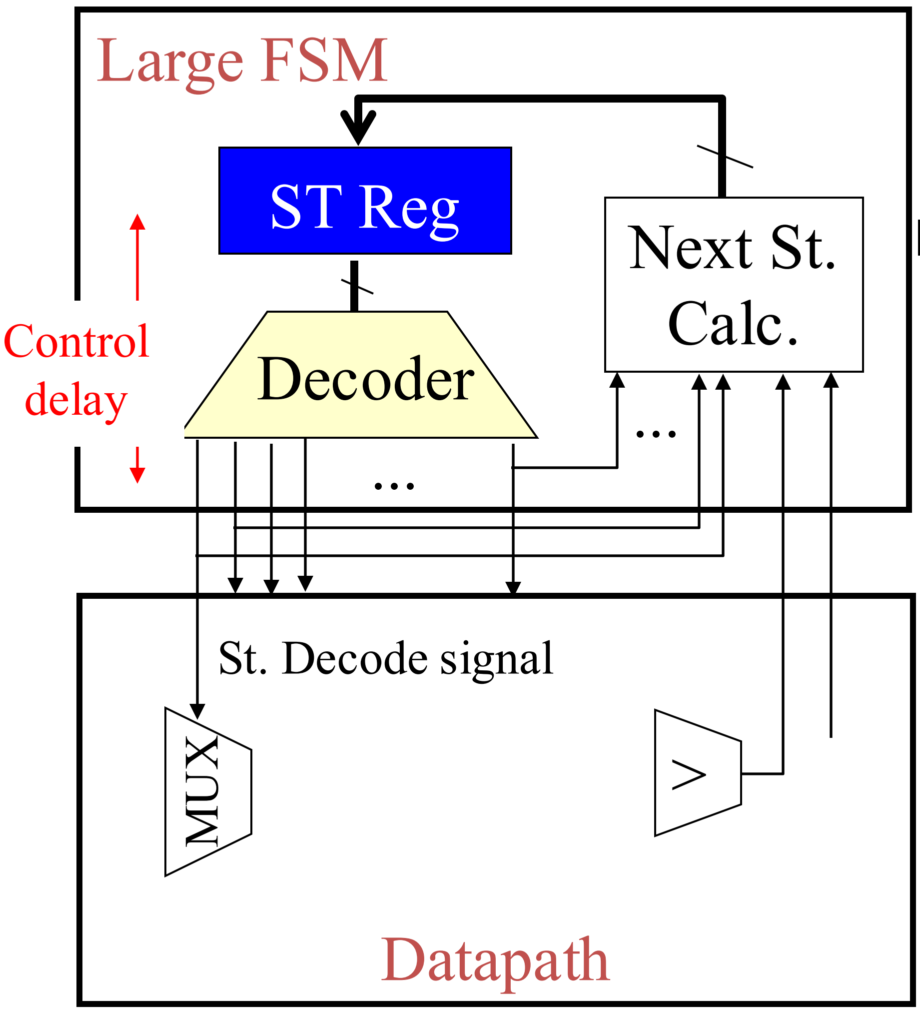
```
int A,B,C,
int E,F;
main(){
int X;
X = A + B;
E = X * D;
F = (B + C) * X;
}
```

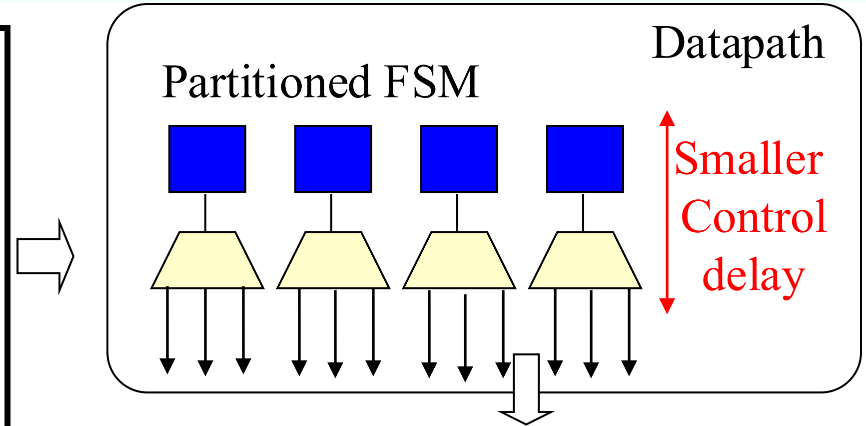
Synthesis Mode Control

- Set the global synthesis mode to pipeline
  - All loops in the description must be unrolled



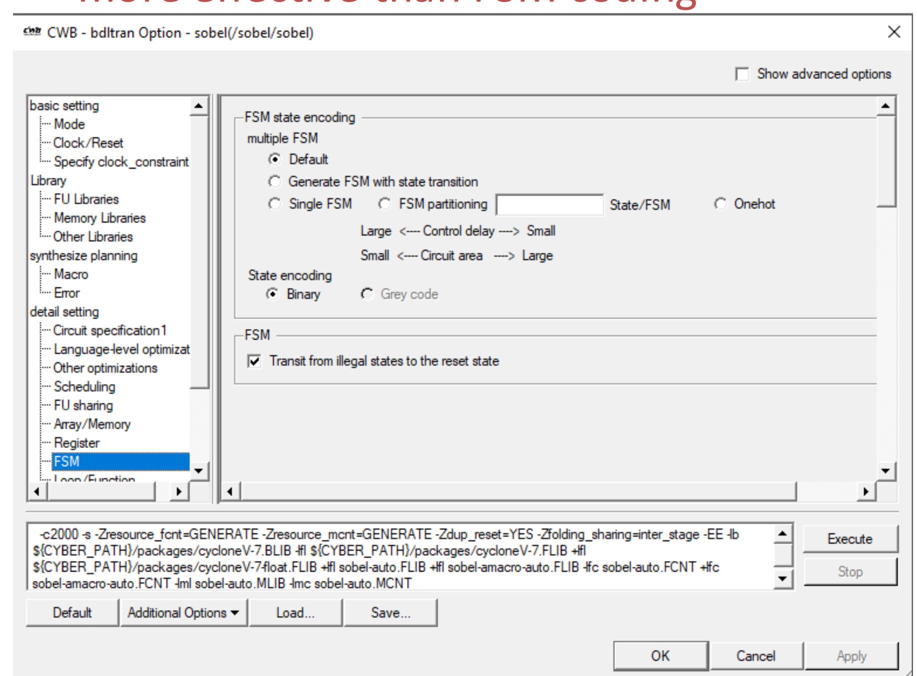
FSM Partitioning for Delay





Arbitrary FSM delay control

more effective than FSM coding



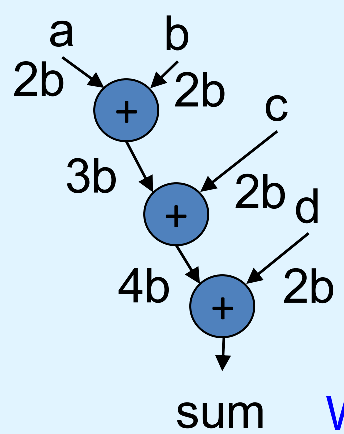
Automatic Bitwidth Reduction and Overflow check

- Automatic bitwidth optimization
- Overflow check

sum=a+b+c+d:

where a,b,c,d are unsigned 2 bit
Sum bit width**?**

Bit analysis



Using interval arithmetic
(Range Analysis)
Input (2bit) [0~3]
sum=[0~3]+ [0~3]+ [0~3]+ [0~3]
= [0~12]
→ 4bit[0~15] are OK

What should the bitwidth of sum be?

Function declaration func (int a, int b);
Function call func (x(7bit), y(6bit)) ➔ function bit width body is adjusted automatically to 7 and 6 bits

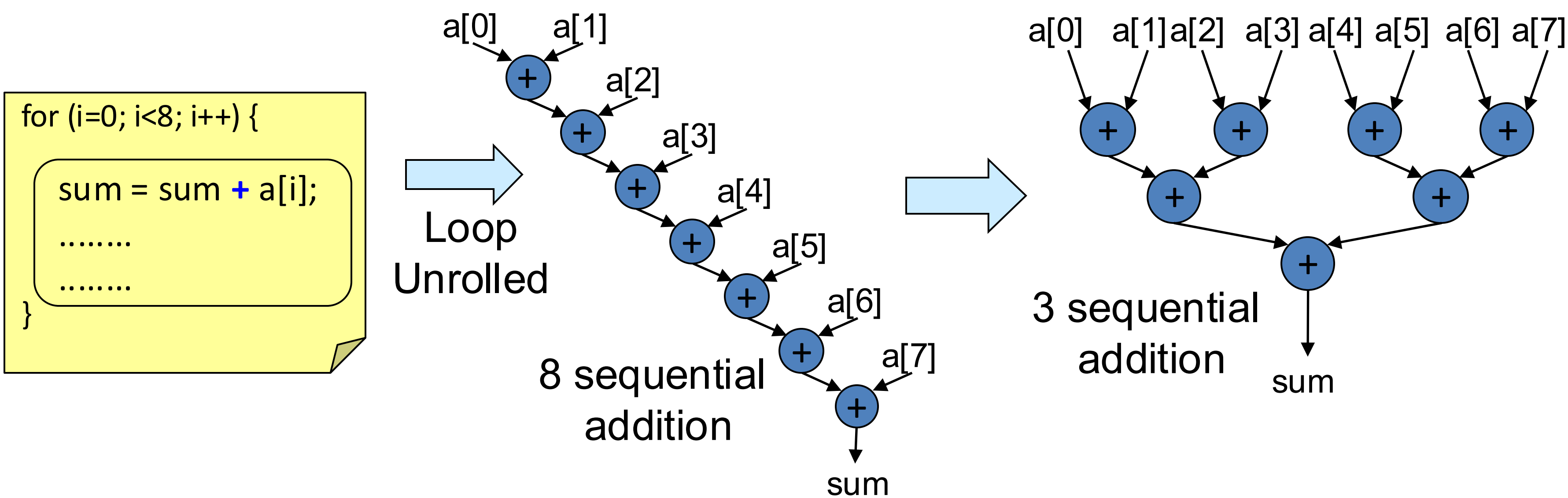
Automatic Bitwidth Optimization

- Only need to specify custom bitwidth at IOs
- Internal signals declare as ‘int’ (for up to 32 bits)

```
in  ter(0:8)  in0;
out ter(0:8)  out0;

/* Global variables */
int buffer[8]  = {0, 0, 0, 0, 0, 0, 0, 0};
ave8(){
  /* Local variables declaration */
   int  out0_v, sum,  i;

 /* Shift data to accommodate new input to be read */
      for (i = 7; i > 0; i--) {
          buffer[i] = buffer[i- 1];

  /* Read new input into fifo */
   buffer[0] = in0;

  /* Add up all the numbers in the fifo */
for (i= 0; i< 8; i++) {
          sum += buffer[i];

   out0=sum/8;
}
```

Data Dependencies: Arithmetic & Logic Operations

- Tree Height Reduction: with automatic bit length adjustment

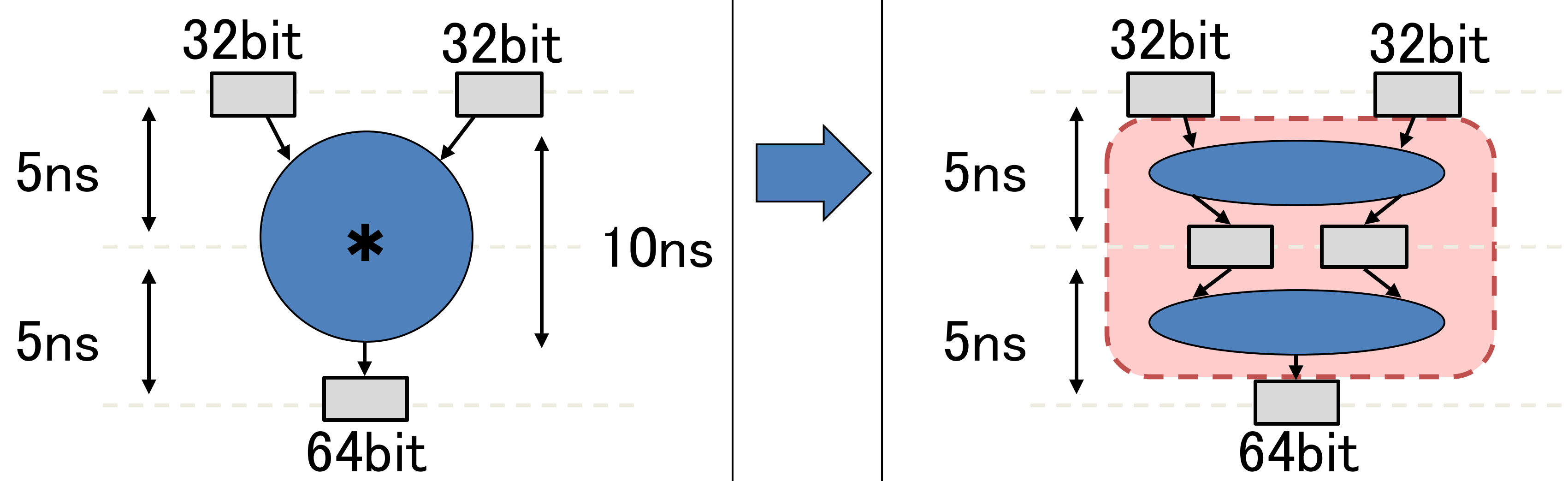


CoreGenerator/MegaWizard Interface

- Automatic interface with FPGA tools

Example where operator delay is larger than target delay

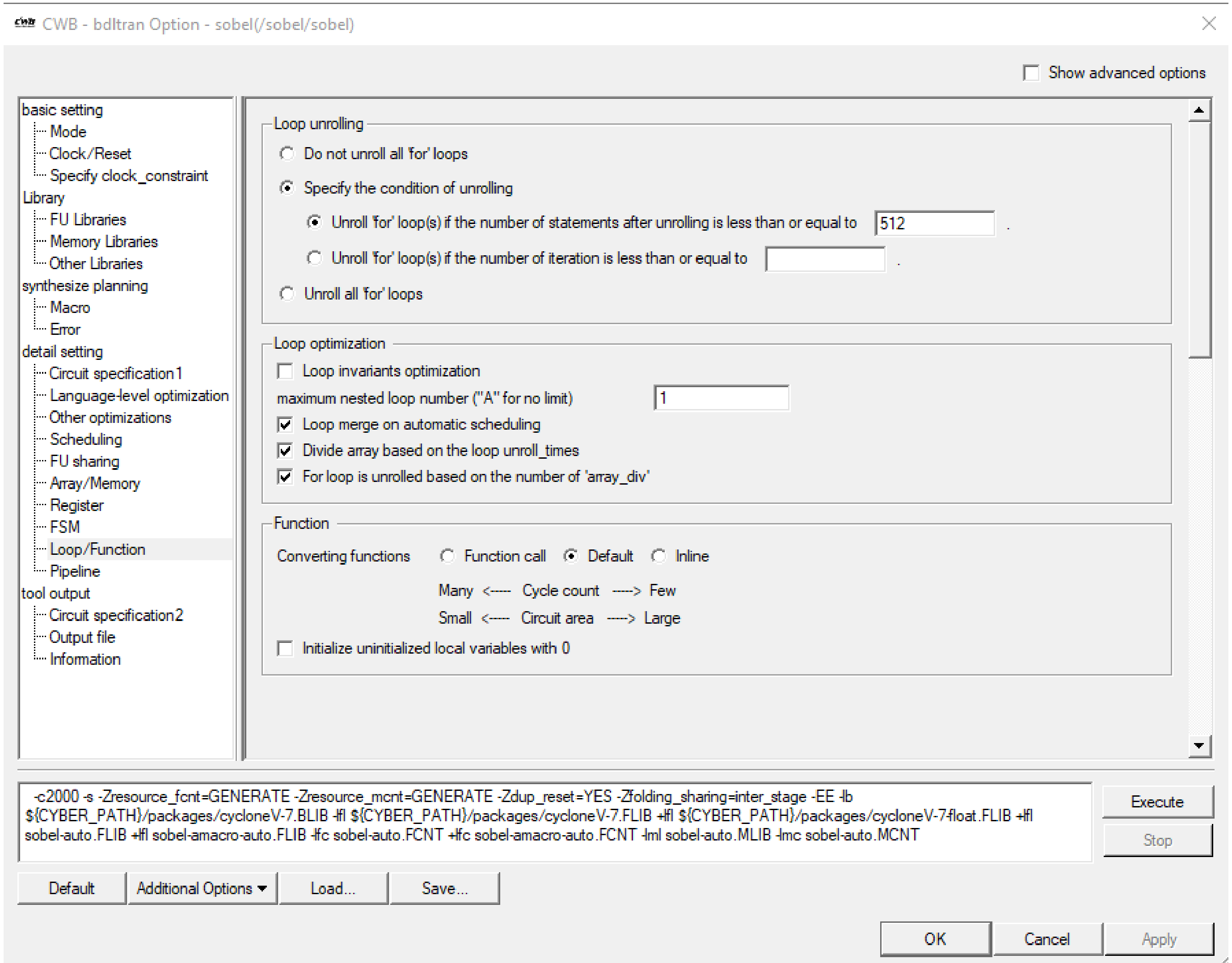
Support of automatic pipelining of arithmetic units to meet delay



- Frequency improvement through pipelining
- DesignWare/COREgen IP automatic connection (Easy to Use)

Loop Unrolling

- Applies to all loops in the description



3. Synthesis Directives (pragmas)

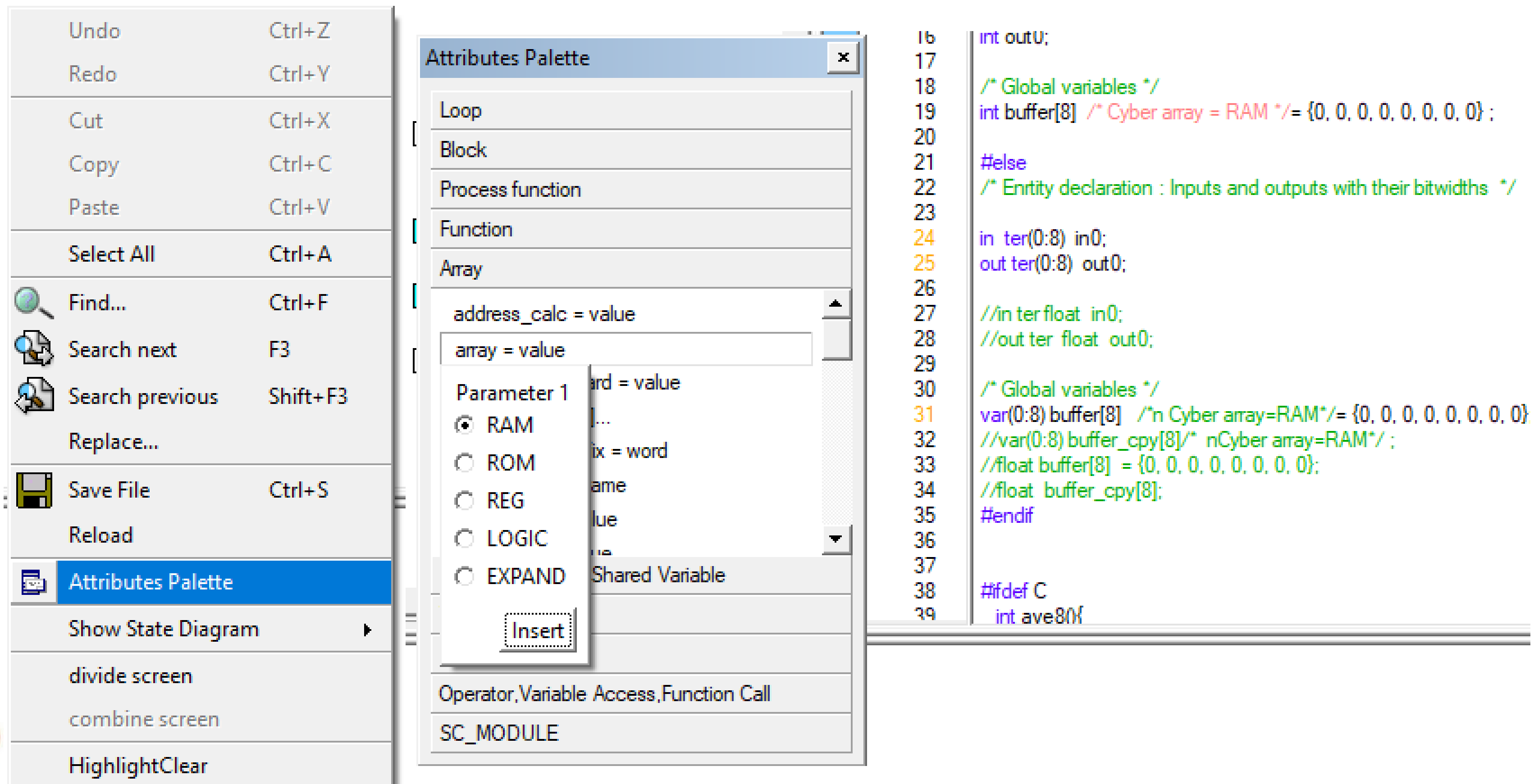
```
in  ter(0:8)  in0;
out ter(0:8)  out0;
var(0:8)  fifo[8] /* Cyber array = REG */= {0, 0, 0, 0, 0, 0, 0, 0};
process ave8(){
int  out0_v, sum,  i;
/* Cyber unroll_times =all */
 for (i = 7; i > 0; i--) {
          fifo[i] = fifo[i- 1];
}
   fifo[0] = in0;
   sum= fifo[0];
/* Cyber unroll_times =0 */
     for (i= 1; i< 8; i++) {
          sum += fifo[i];  }
       out0_v= sum / 8;
       out0 = out0_v;}
```

Synthesis directives:

- Arrays : Registers or Memory
- Loops: Unroll or not

Prgamas in CWB's GUI

- Right-Click on text editor where the attributed should be inserted→ attribute pallet

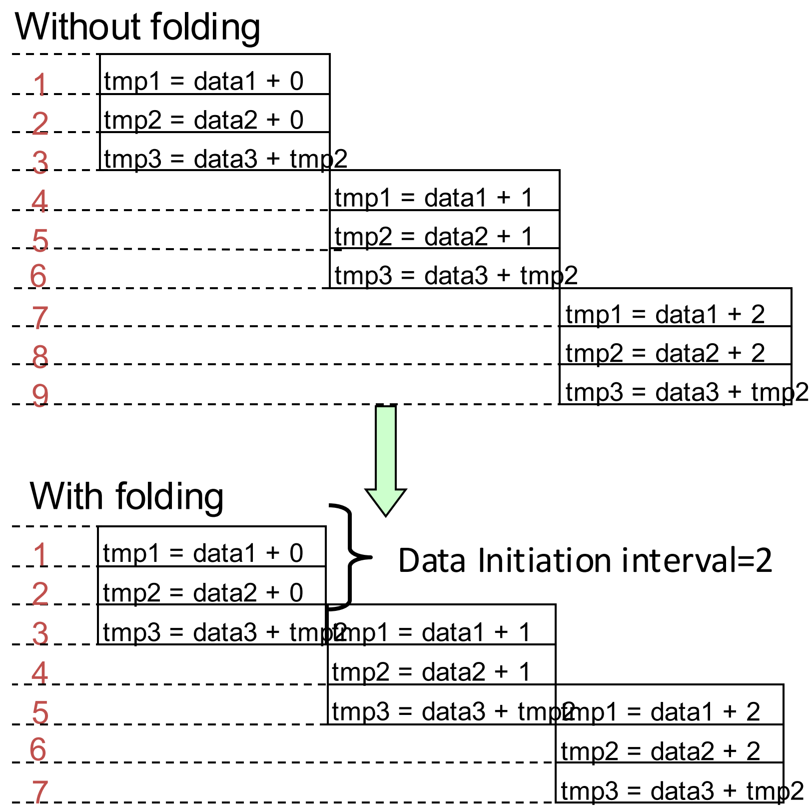


Loop folding

- Each loop can be pipelined using the “loop folding” attribute

DII specification

```
/* Cyber folding = 2 */
for( cnt = 0; cnt < 3; cnt++ )
{
    tmp1= data1 + cnt;

    tmp2 = data2 + cnt;

    tmp3 = data3 + tmp2;

}
```



Automatic Loop Pipelining Example

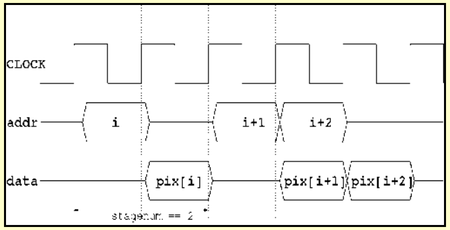
Specify Data Initiation Interval (DII)

```
/* Cyber folding=1 */
while(i<16) {
   a0 = in_pix[i];
   a1 = in_pix [i+1];
   a2 = in_pix [i+2];
   if (a0 + a1 > a2)
      o1 = (a0 + a1) / 4;
   else
      break;
   out_pix [j] = o1;
   i++;
}
```

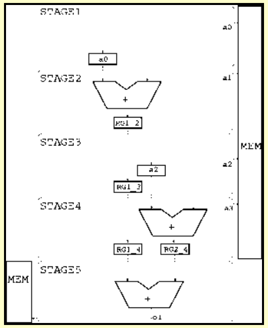
Array mapped to Memory

One port memory

Memory access timing



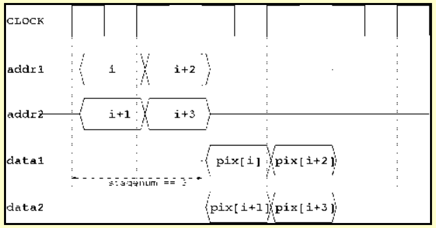
Block Diagram



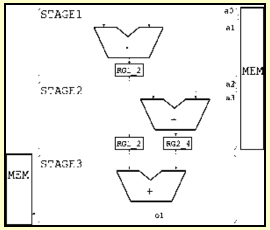
➔ In RTL design: two different designs in C automatic re-timing

Two Port Memory

Memory Access Timing



Block Diagram



Loop Unrolling

- Only "for" loops with constant # of iterations can be unrolled

▪ Unrolling all iterations

```
// Cyber unroll_times = all
for( cnt = 0; cnt < 10; cnt++){
   data += cnt;
}
```

After complete unrolling

```
data += 0;
data += 1;
…    ⋮
data += 9;
```

▪ Unrolling 2 times in parts

```
// Cyber unroll_times = 2
for( cnt = 0; cnt < 10; cnt++){
   data += cnt;
}
```

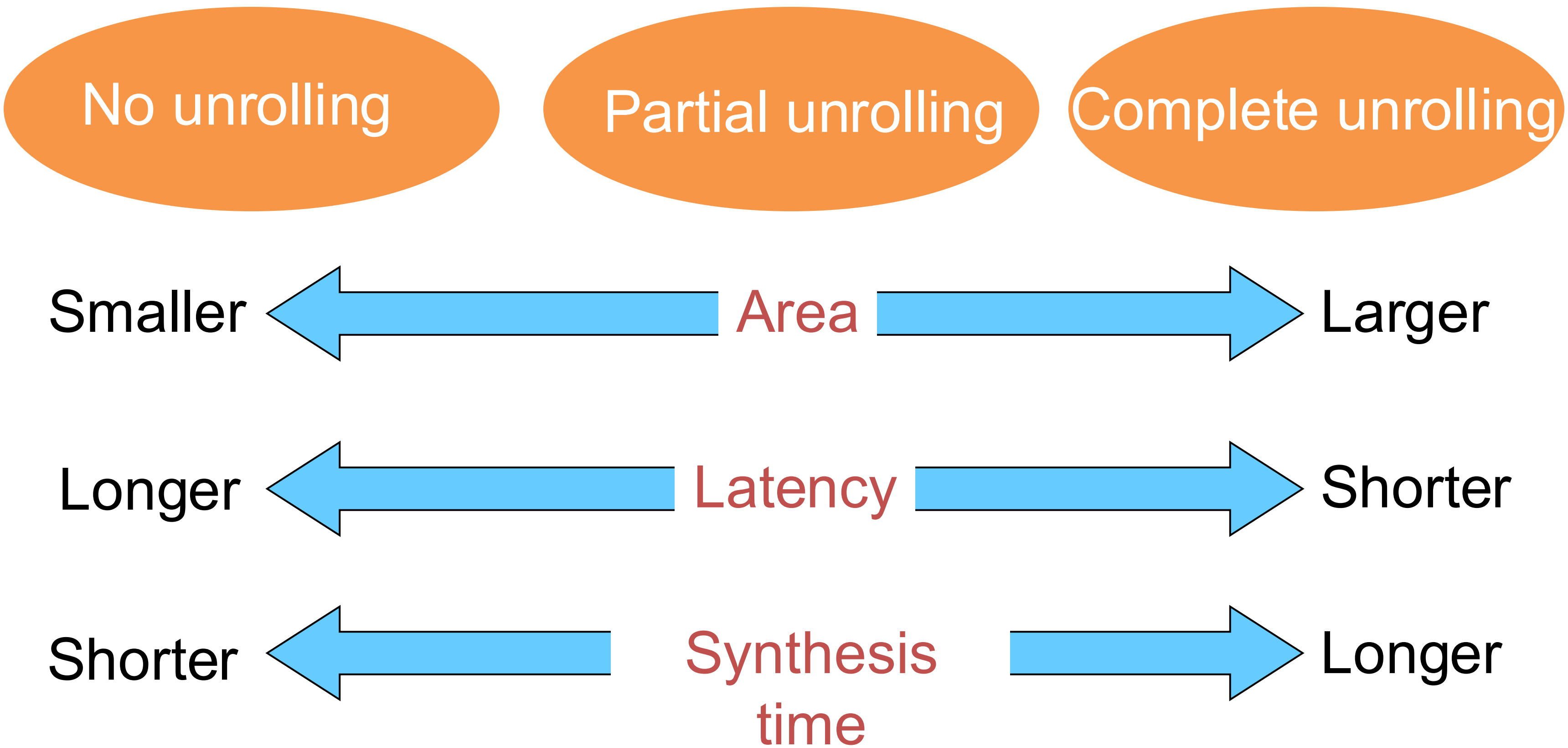
After partial unrolling

```
for( cnt = 0; cnt < 10; cnt += 2){
   data += cnt;
   data += cnt + 1;
}
```

<u># of unrolled iteration influences area and latency</u>

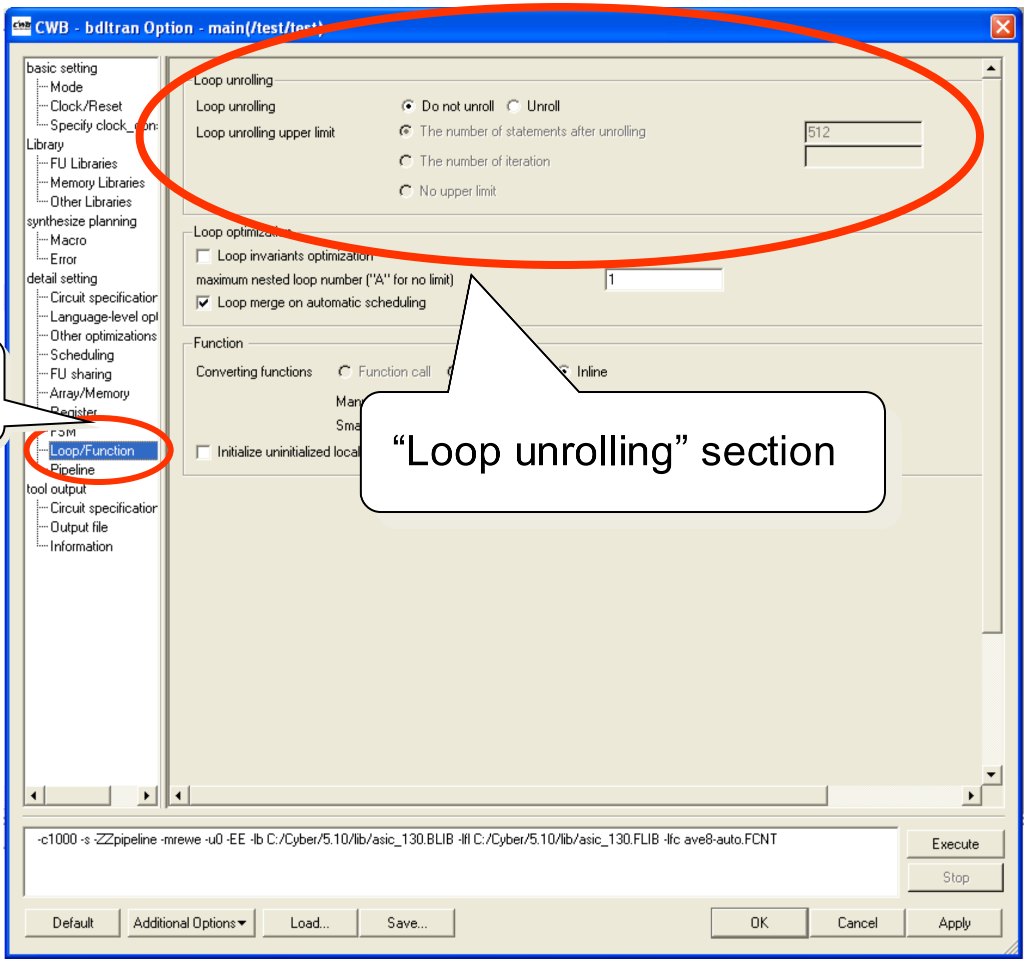
Impact of Loop Unrolling

- Selection and trend for loop unrolling



Loop Unrolling Control

- Global synthesis options
- Local synthesis directives (attributes-pragmas)
- Global Synthesis options example:



Loop Unrolling

- Control through attributes

Do not unroll: /* Cyber unroll_times = **0** */
Completely unroll: /* Cyber unroll_times = **all** */
Partially unroll: /* Cyber unroll_times = ***N****/

* N should be a natural number

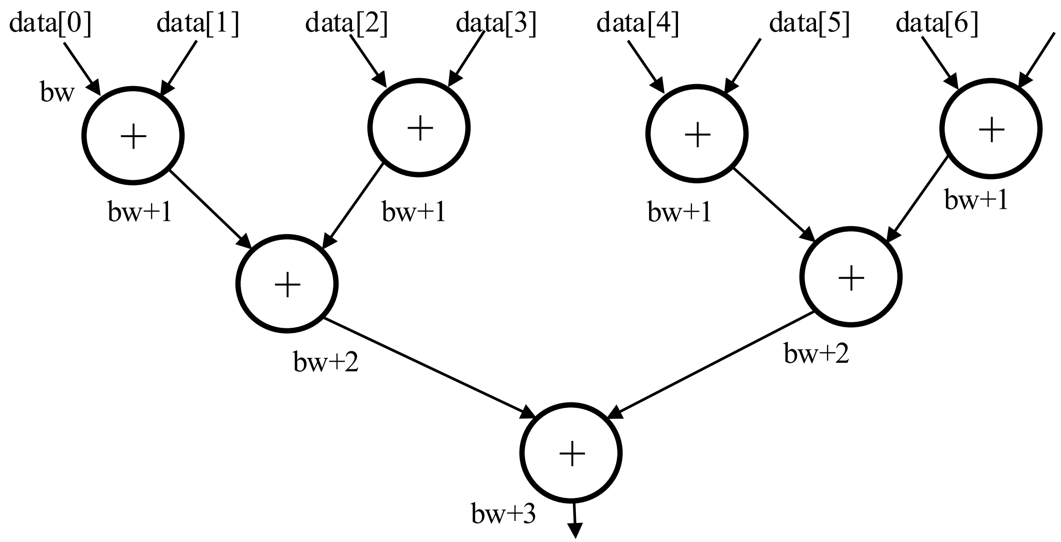
➔ The attributes are specified at the "for" loop as follows:

```
/* Cyber unroll_times = 0 */
for(cnt = 0; cnt<1024; cnt++) {
   ...
   ...
}
```

Tree Height Reduction

- Average of 8 numbers computation

```
// Cyber unroll_times=all
for(x=0; x<8;x++)
        sum +=data[x];
sum=sum/8;
```



Resource Allocation Output

- FLIB file only contains pre-characterized FUs
  - E.g., 4bit, 8bit, 12bit, 16bit, 20bit adders
- If inputs (data[]) declared as 16 bits what should the FCNT file contain after resource allocation?

```
// Cyber unroll_times=all
for(x=0; x<8;x++)
        sum +=data[x];
sum=sum/8;
```

→

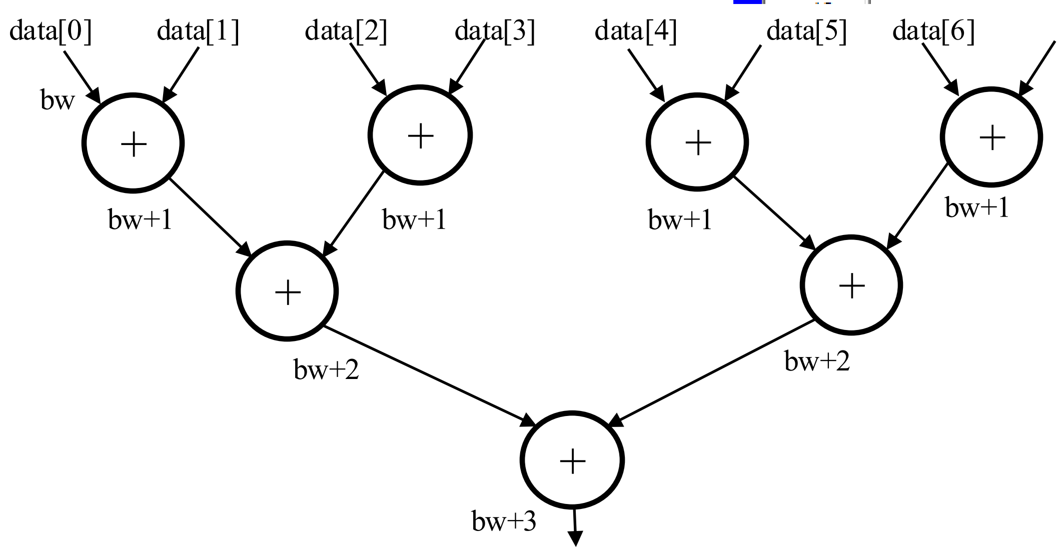
```
sum +=data[0];
sum +=data[1];
sum +=data[2];
sum +=data[3];
sum +=data[4];
sum +=data[5];
sum +=data[6];
sum +=data[7];
sum=sum/8;
```

Resource Allocation Example

- 7 adders needed
  - Add16u 4
  - Add20u 3
- But based on tree
  - Add16u 4
  - Add17u 2
  - Add 18u 1

Quality of Result(2015/10/23 09:01:27) | ave8-auto.FCNT(ave8)

```
#@VERSION{3.00}
#@CNT{ave8}
#@KIND{BASIC_OPERATOR}
#@CLK 200000
#@UNIT 1/10ps
@FCNT{
        NAME    add16u
        LIMIT   4
#       COMMENT
}
@FCNT{
        NAME    add20u
        LIMIT   3
#       COMMENT
}
#@HASH{c88a0d33a58dfe30a9e4e9eab617cdb1}
#@END{ave8}
```



Resource Allocation Example

- CWB's QOR report file shows the actual FUs used

Quality of Result(2015/10/23 09:01:27) | ave8-auto.FCNT(ave8)

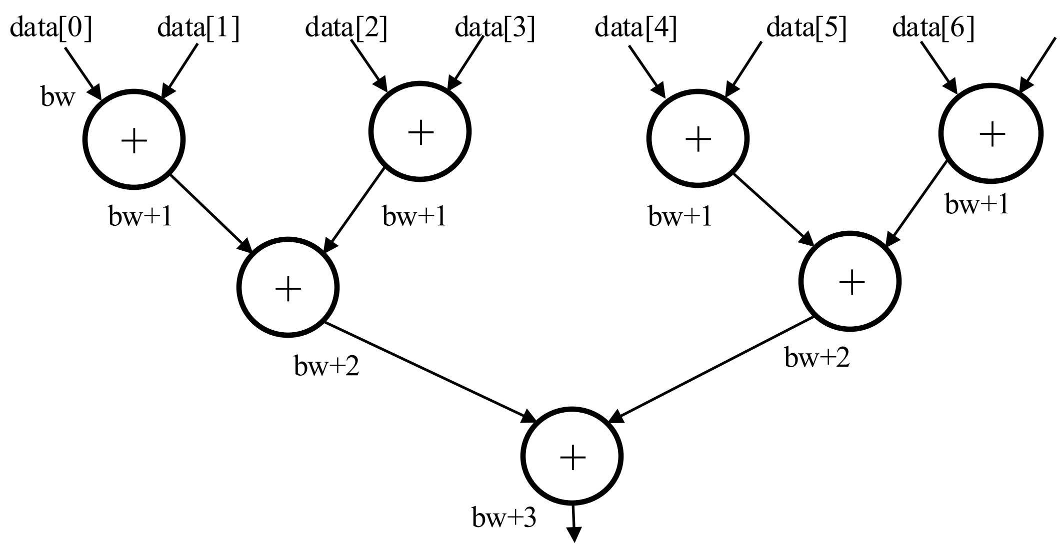
- Summary
- Area
- Functional Unit
- Register
- Memory
- Latency
- FSM
- Delay
- Wire
- Primary Port

Functional Unit

FU Name	Kind	Sign	Bit Width	Area	Reg	Delay (ns)	Count
add16u	+	unsigned	(16,16) 17	16	0	1.22	4
add20u_19	+	unsigned	(18,18) 19	20	0	1.28	1
add20u_19_18	+	unsigned	(17,17) 18	20	0	1.28	2

Unused Functional Units

None



Functions Synthesis

Implementations of function

(1) inline expansion

(2) goto conversion

(3) conversion into functional unit

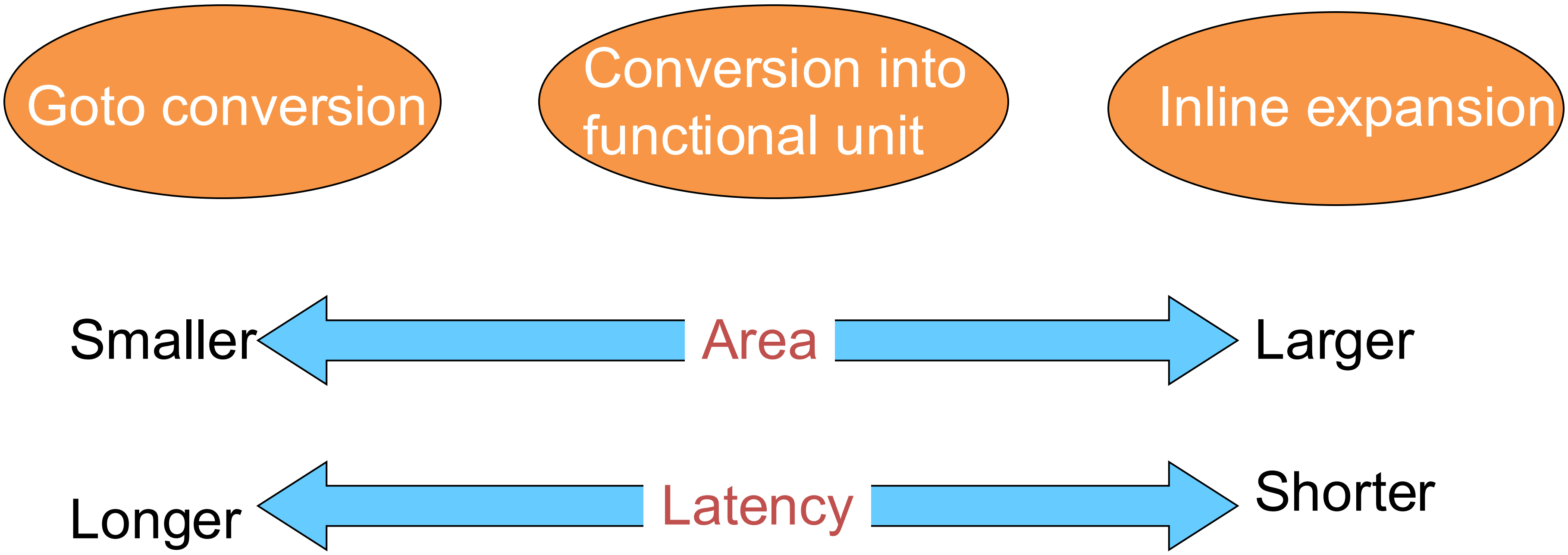
b) Synthesis option for function implementation

c) Synthesis attribute for function implementation

d)Convert into functional units

Function Synthesis Impact

- The selection and trend for function implementation types are as follows:



Function Implementation: (1) Inline expansion

- Inline expansion expands a function definition at all corresponding function call directly
- Inline expansion tends to make latency shorter but tends to make area larger.

```
process main( ){
   ...
   data1 = func(a,b);
   ...
   data2 = func(e,f);
}

int func(int x, int y){
   int z;
   z = (x + y) / 2;
   return z;
}
```

```
process main( ){
   ...
   data1 = (a + b)/2;
   ...
   data2 = (e + f)/2;
}
```

Function body is expanded at all function calls.

Function Implementation: (2) Goto conversion

- Goto conversion implements a function as only 1 instance and covert all corresponding function calls with labels and goto statements to activate the single function instance

```
process main( ){
   ...
   data1 = func(a,b);
   ...
   data2 = func(e,f);
}

int func(int x, int y){
   var(7..0) z;
   z = (x + y) / 2;
   return z;
}
```

For each function call, the block starting from label "L_func" is executed.
The function definition is implemented only in the block.

```
...
F_func = 0; goto L_func;
ST1_03 :
...
F_func = 1; goto L_func;
ST1_05 :
...
L_func:
 switch(F_func) {
    case 0: x = a; y = b; break;
    case 1: x = e; y = f;  break;
 }
 z = (x + y) / 2;
 switch(F_func) {
   case 0: data1 = z; goto ST1_03;
break;
   case 1: data2 = z; goto ST1_05;
break;
 }
```

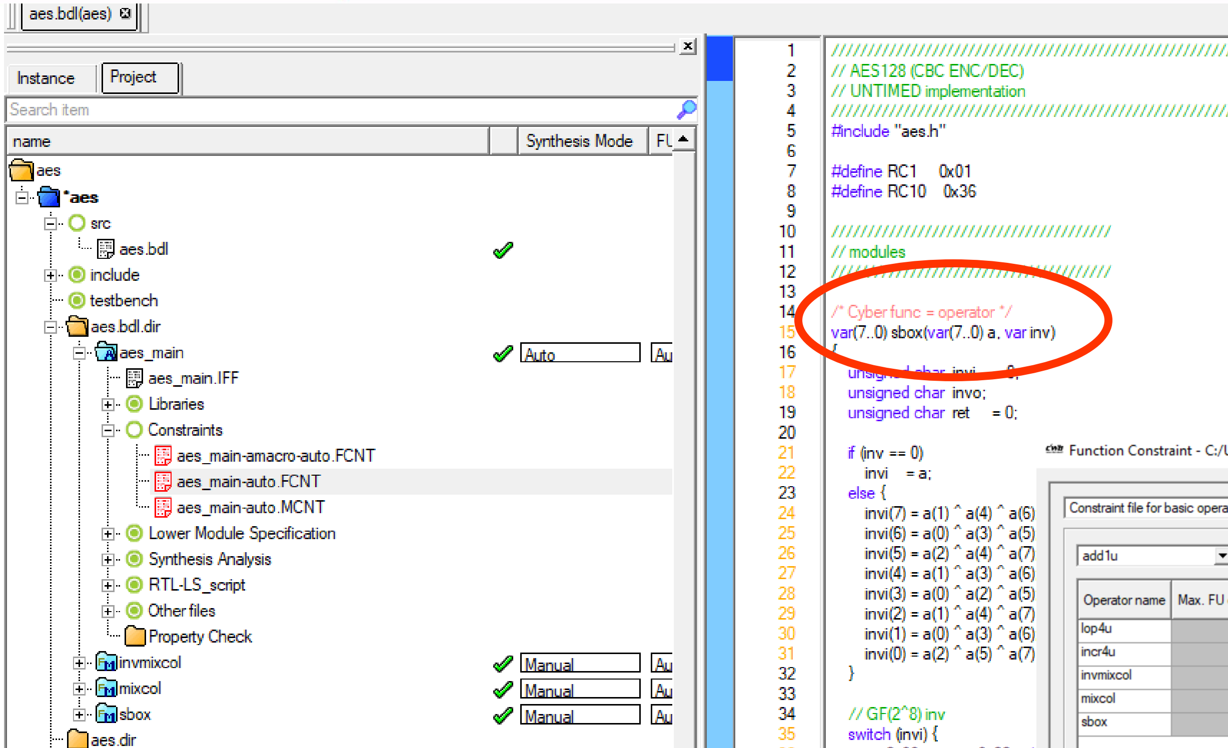
- Because a function definition is implemented only in 1 block, area tends to be smaller. **But** latency tends to be longer because extra operations are required for pre/post processing.

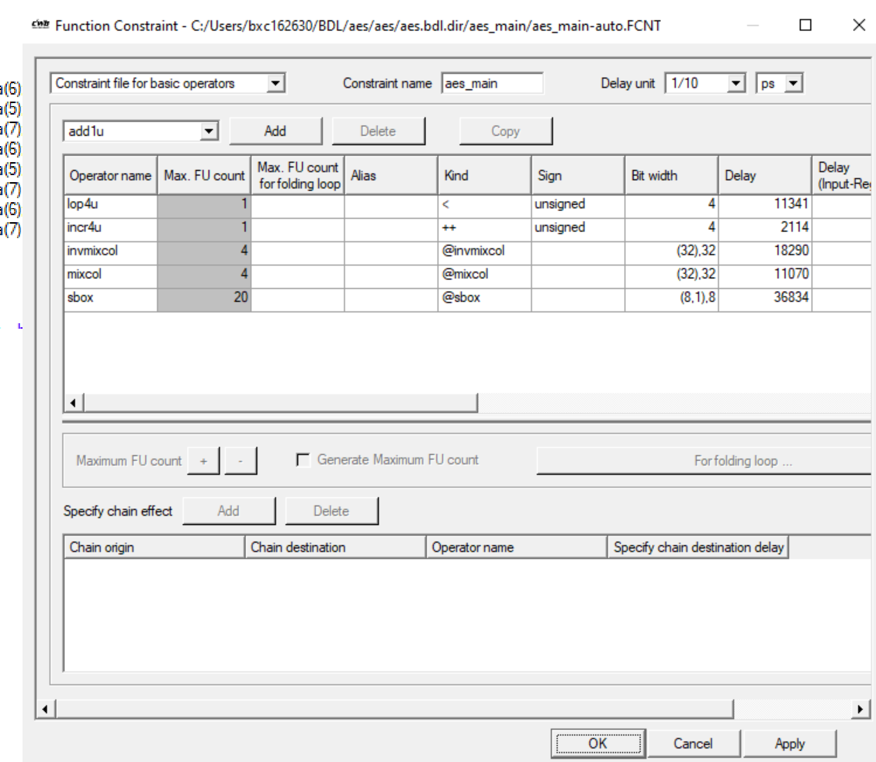
Function Implementations: (3) Conversion into FU

- A function definition is treated as a functional unit and the # of instances can be controlled
- Limit of # of instances is defined in a constraint file.
  - inline expansion → # of instances = # of function call
  - goto conversion → # of instances = only 1

Benefits:
- It becomes easier to control the trade-off between area and latency
- Each function definition can be implemented flexibly in a different way
- Converted functional units can be treated as hierarchical circuits

Function Operators

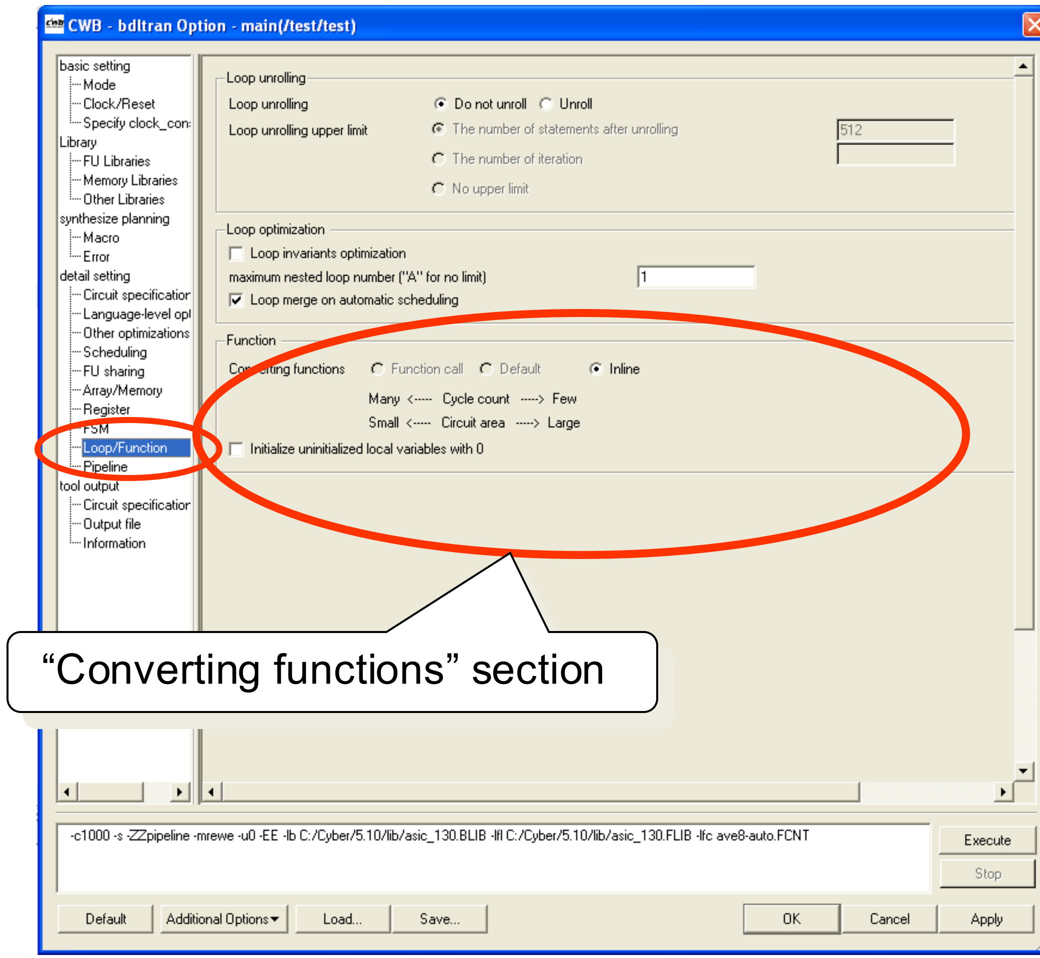




- Declare function as operators
  // **Cyber func=operator**
- Each function synthesized separately
- FCNT file generated in top function

Synthesis Option for Function Implementations

- Default synthesis mode:
  - In automatic scheduling mode, a function is implemented with either inline expansion or goto conversion based on # of calls and # operations in its body.



Synthesis Attribute for Function Implementation

inline expansion : /* Cyber func = **inline** */
goto conversion : /* Cyber func = **goto** */
conversion into functional unit : /* Cyber func = **operator** */

➔Attributes are specified at the function definitions as follows:

```
/* Cyber func = inline */
int funcA(int data_a, int data_b ){
   …
   …
}
```

How to specify Conversion into Functional Unit

- Functional Unit Constraint FILE (FCNT) file for each function declared as an operator will be generated
- Can modify that FCNT file to specify the number of Functions to be instantiated

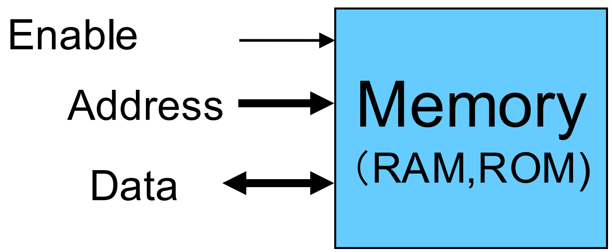
```
/* Cyber func = operator */
int filter7(int data_a, int data_b ){
   …
   …
}
```

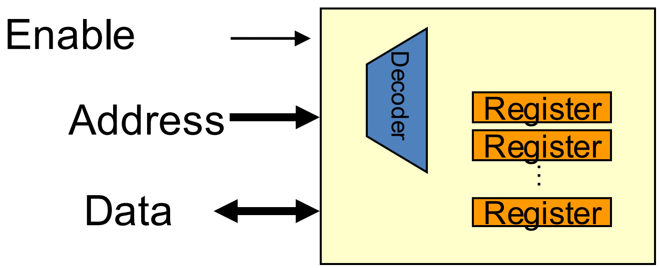
Array Synthesis

- Array implementation (memory, register array)
- Array implementation (combinational circuit, variable expansion)
- Synthesis option for array implementation
- Synthesis attribute for array implementation
- How to specify memory assignments
- How to specify # of decoders for register arrays

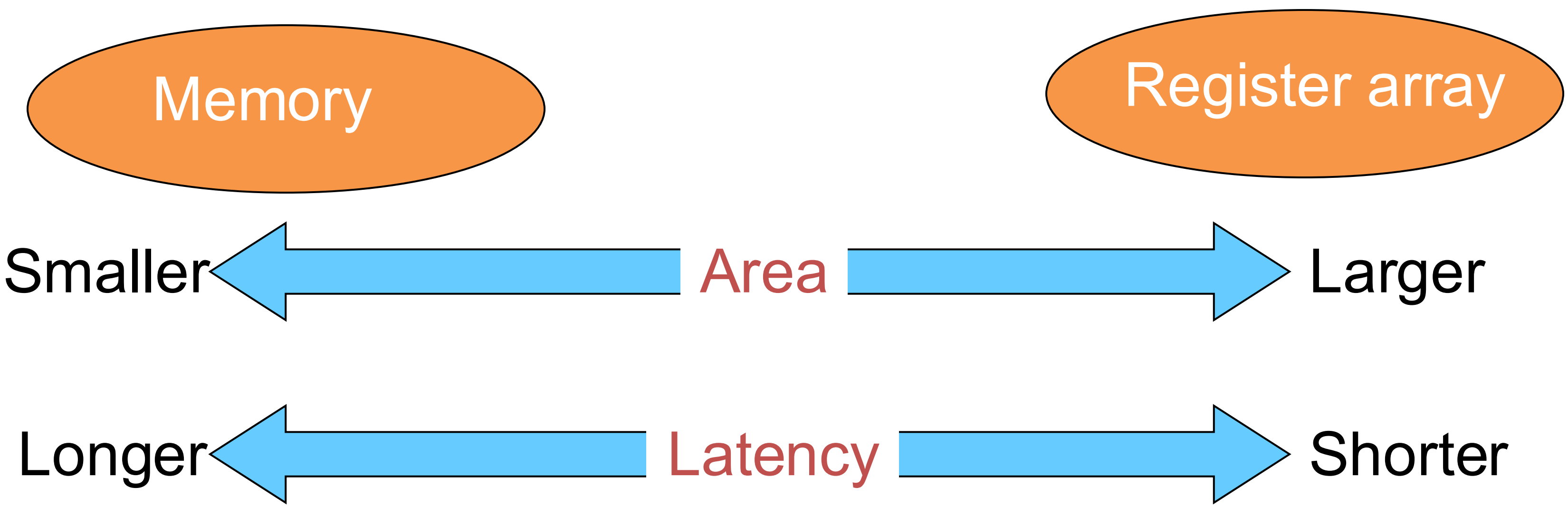
Impact of Array Synthesis

- RAM vs. Registers





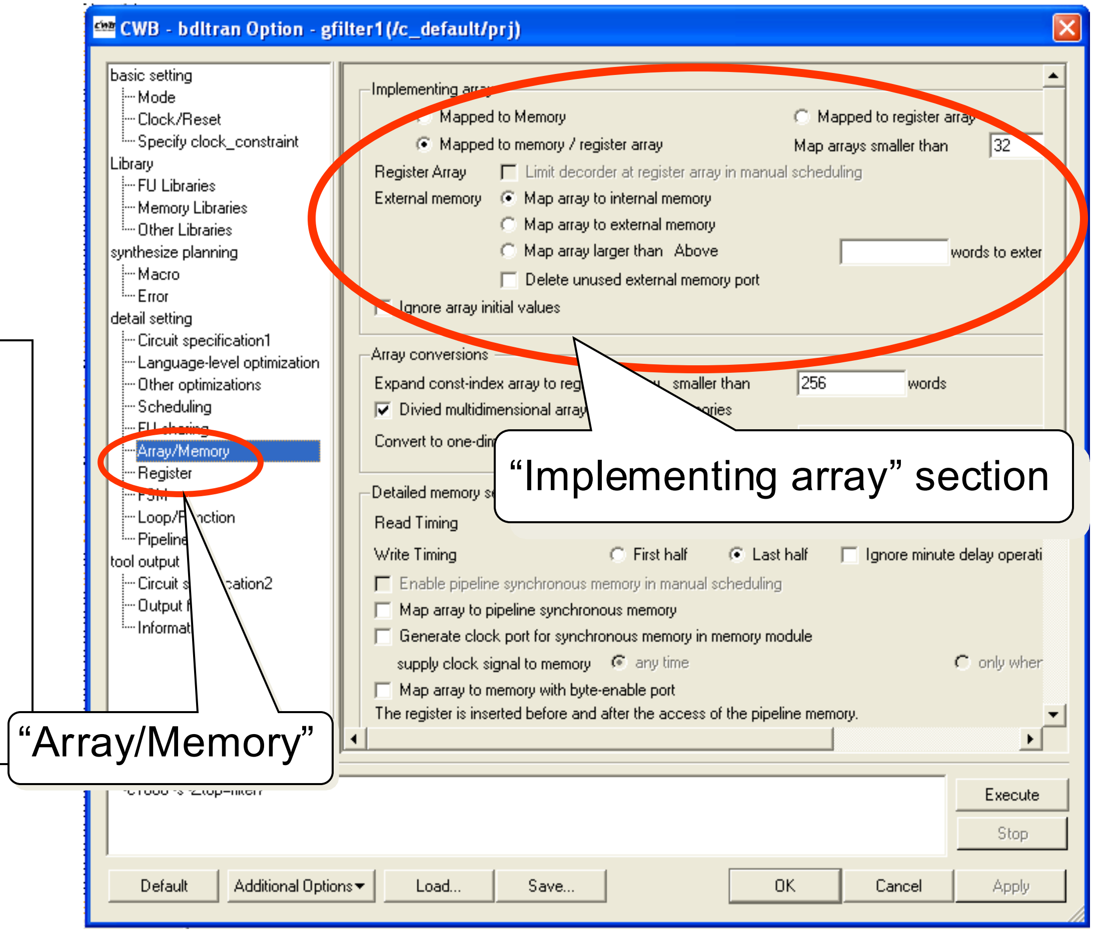
* Array is implemented as register-file with decorders and registers.



Synthesis Option for Array Implementation

int data[1024]

- Default (impl. is decided based on size of array automatically)
- All arrays are mapped to memory
- All array are mapped to register array



Synthesis Attribute for Array Implementation

```
int  data[30] /* Cyber array = RAM */;
```

Memory	/* Cyber array = **RAM** */
Read only memory	/* Cyber array = **ROM** */
Register array	/* Cyber array = **REG** */
Combinational circuit	/* Cyber array = **LOGIC** */
Variable expansion	/* Cyber array = **EXPAND** */

Advanced Array Attributes

- Multiple ports for decoder enabled register array

  int buffer128] /* Cyber array=REG, rw_port=4*/;

  - Generates registers with 4 rw_ports

Advanced Array Attributes

- Expand multi-dimensional arrays

  ```
  int buffer[2][128] /* Cyber array=REG, expand_dim=1 */;
  ```

  - Generates 2 arrays of size 128

  ```
  int buffer[2][128] /* Cyber array=REG, expand_dim=2 */;
  ```

  - Generates 128 arrays of size 2

Scheduling Blocks

- Allows to insert manually scheduled C code into automatically scheduled one.
  - Used e.g., to model interfaces

```
int data[8]  = {0, 0, 0, 0, 0, 0, 0, 0}
Bool done = false;
int ave8(int in0){

int, sum,  i, out0;
for (i = 7; i > 0; i--)
        data[i] = data[i- 1];

data[0] = in0;
sum= data[0];
for (i= 1; i< 8; i++) {
        sum += data[i];
out= sum / 8;

/* Cyber scheduling_block */
{
done=true;
return (out0);
$
done=false;
}
}
```

Alternative Ways of Specifying Pragmas

ave8.c

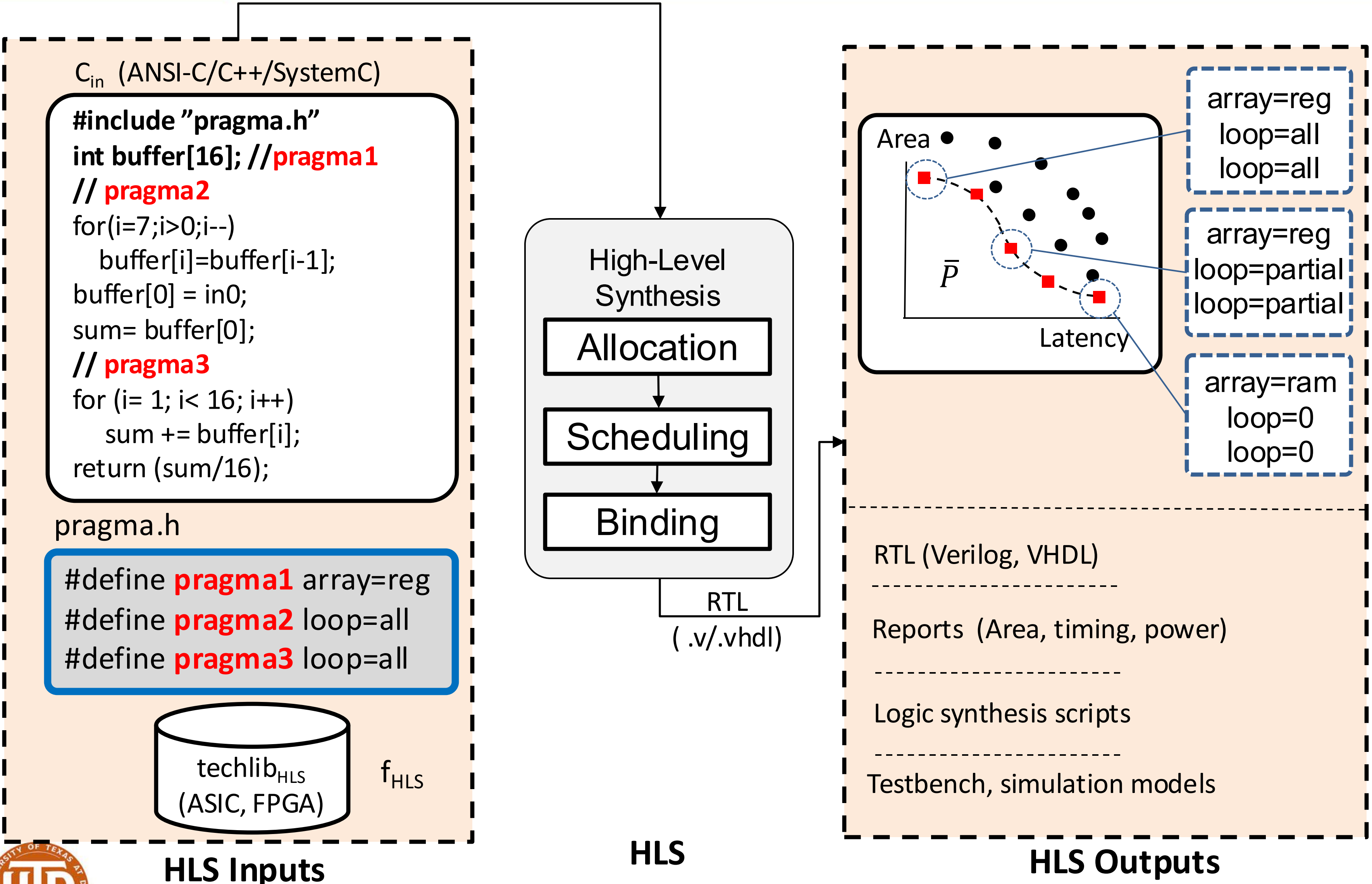
```
#include "attrs.h"
int data[8]  = {0, 0, 0, 0, 0, 0, 0, 0} /*  ATTR1 */;

int ave8(int in0){

int, sum,  i, out0;
// ATTR2
     for (i = 7; i > 0; i--)
         data[i] = data[i- 1];

data[0] = in0;
// ATTR3
sum= data[0];

for (i= 1; i< 8; i++) {
         sum += data[i];
out= sum / 8;
return (out0);}
```

- Set labels in C code
- Include header file where pragmas have been defined
  - Include in header file pragma values

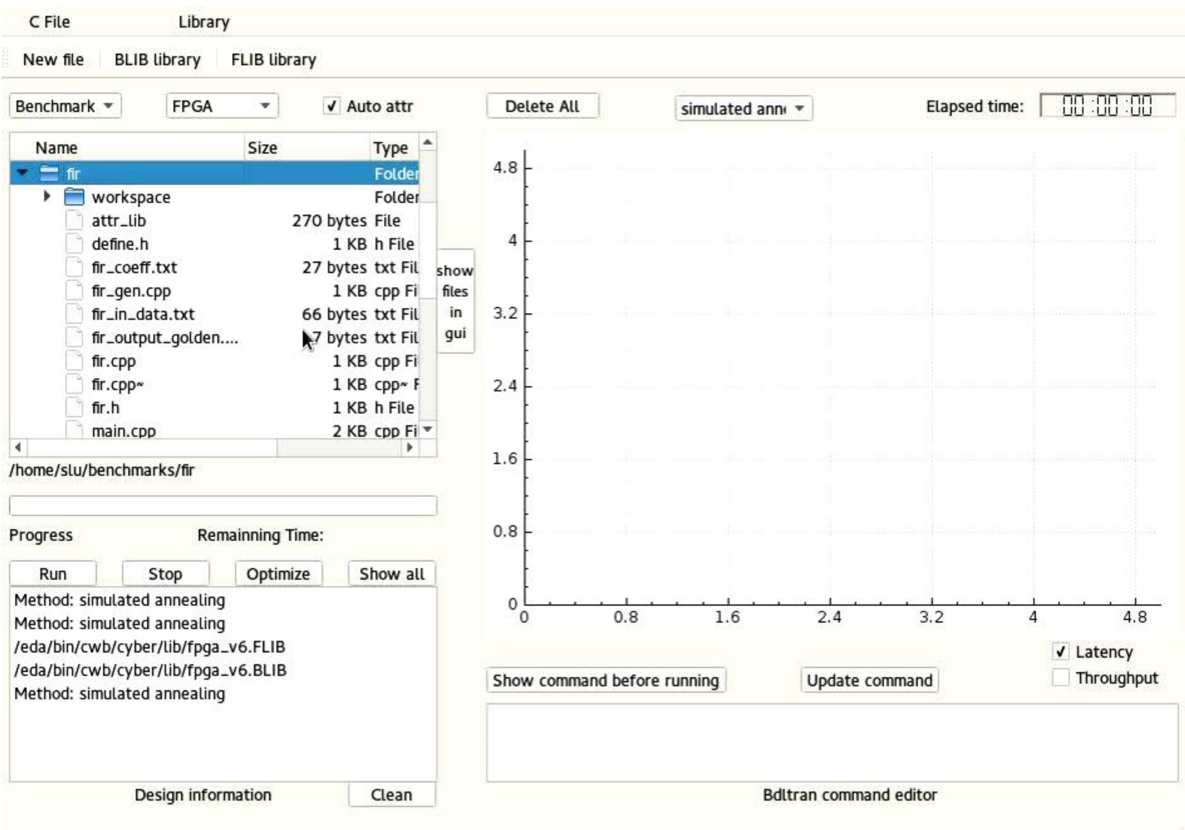
attrs.h

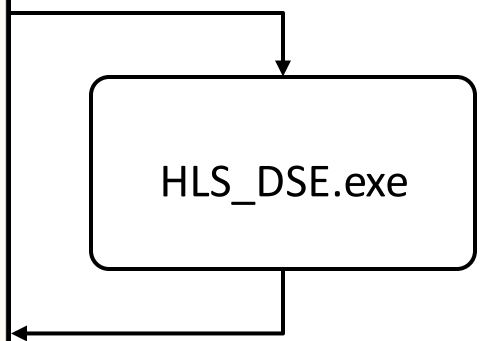
```
#define  ATTR1  Cyber array=REG
#define  ATTR2 Cyber unroll_times=all
#define  ATTR3 Cyber unroll_times=0
```

Automated HLS Design Space Exploration



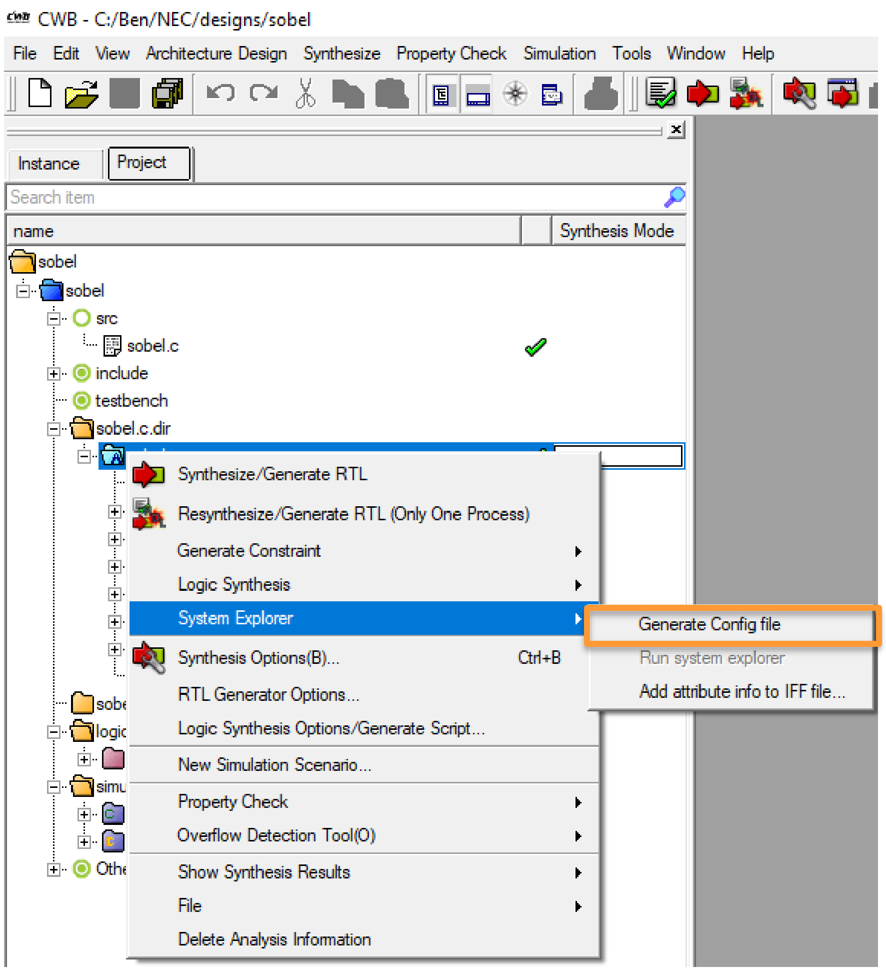
HLS Design Space Explorer

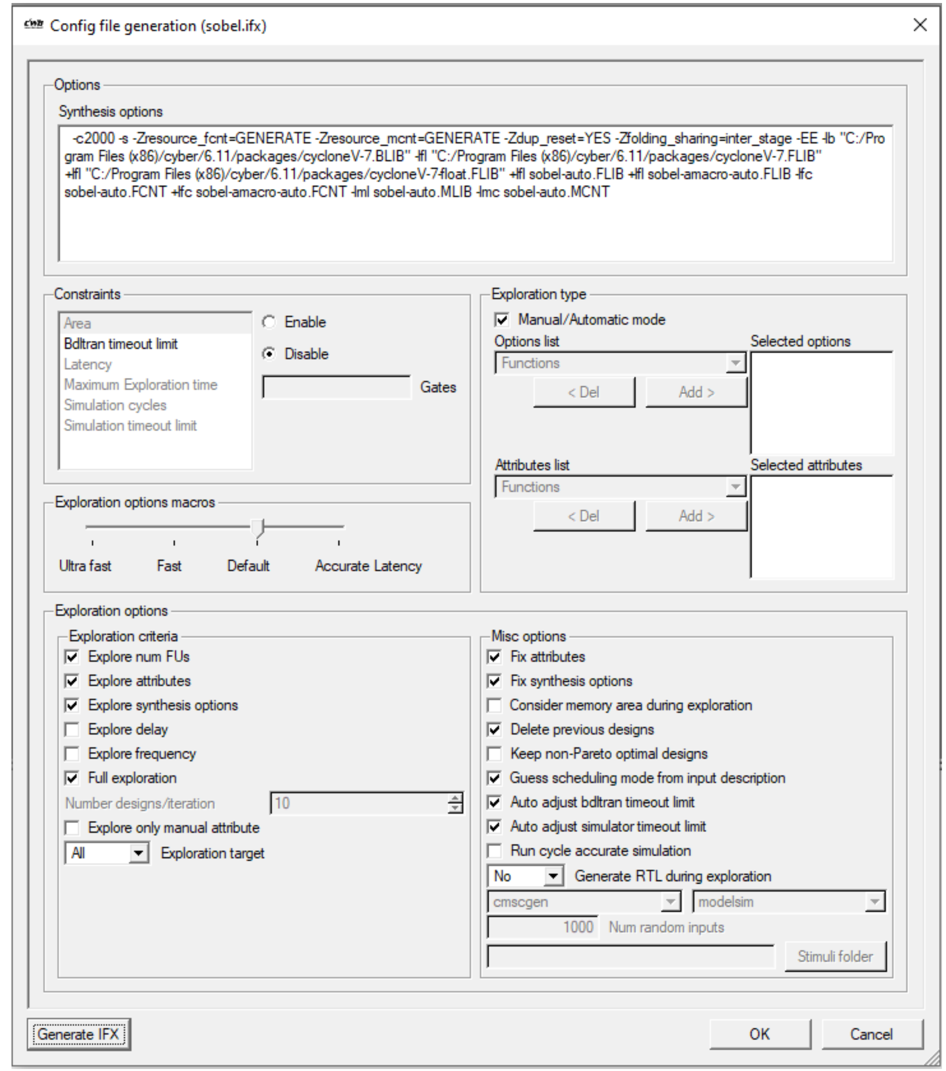




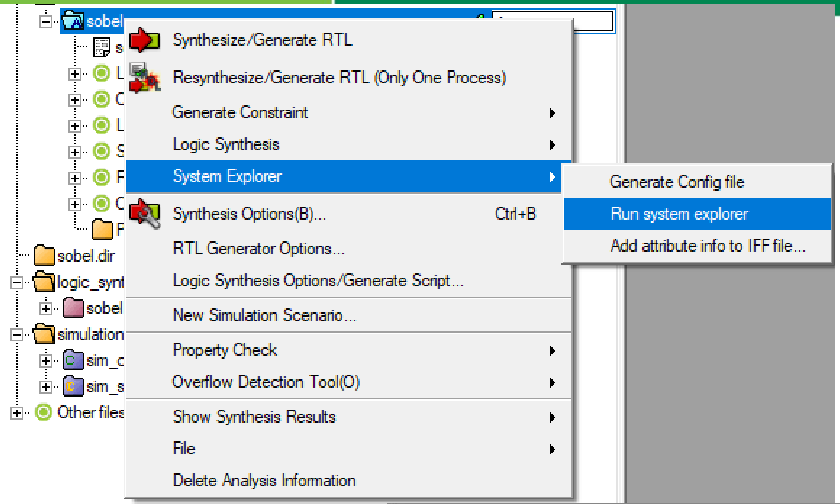
1. Simulated Annealer
2. Genetic Algorithm
3. Ant Colony
4. Predictive Model
5. <u>Own Heuristic</u>

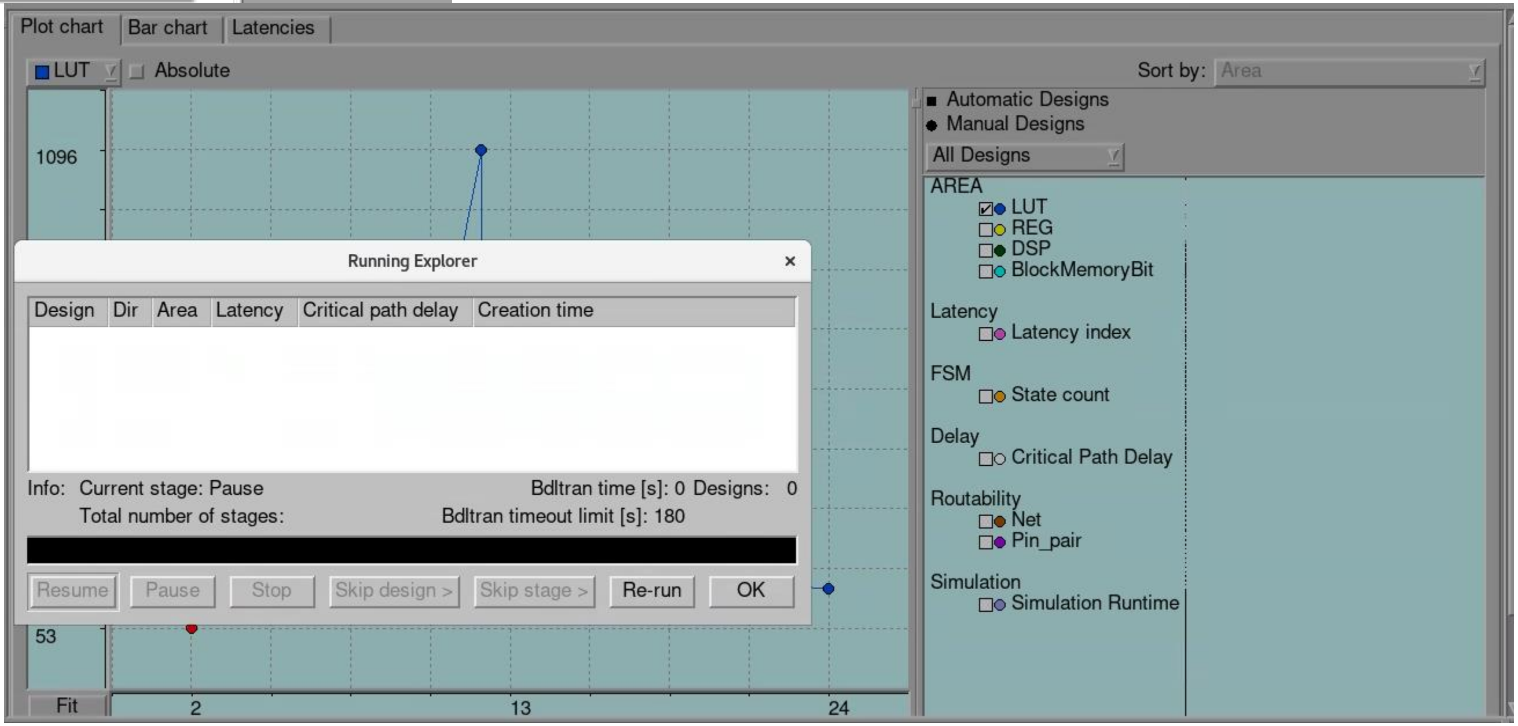
CyberWorkBench Explorer





CyberWorkBench Explorer





Own DSE

sobel.c

```
#include "attrs.h"
 char Gx[3][3] /* ATTR1 */  ={{1 ,0 ,-1},
                     { 2, 0, -2},
                     { 1, 0,-1}};
/* ATTR2 */
 for(rowOffset = -1; rowOffset <= 1; rowOffset++){
/* ATTR3 */
 for(colOffset = -1; colOffset <=1; colOffset++){
```

attrs.c

```
#define ATTR1 Cyber array=REG
#define ATTR2 Cyber unroll_times=all
#define ATTR3 Cyber unroll_times=all
```

lib_attrs.c

```
ATTR1 array REG
ATTR1 array EXPAND,array_index=const
ATTR1 array ROM
ATTR1 array LOGIC

ATTR2 unroll_times 0
ATTR2 unroll_times all

ATTR3 unroll_times 0
ATTR3 unroll_times all
```

Steps for own explorer

Step 1: Read lib_sobel.info with all attributes for each operation.

Step2: Generate new attrs.h header file with unique attributes combination.

Step3: Parse the new description. Make sure attrs.h is in the same folder as sobel.c:

%cpars sobel.c

Step4: Synthesize design calling HLS (bdltran)

%bdltran -c2000 -s sobel.IFF -lfl cycloneV.FLIB -lb cycloneV.BLIB

Step5: Read the area and latency of the new design and store sobel.QOR file or sobel.csv file and attrs.h file by moving it to either another folder or renaming the files.

Repeat step2 until all combinations are generated.

Step6: Generate report file with all the results and report optimal designs (area, latency and pragmas that lead to them).

Conclusions

- High Level Synthesis optimizations
  - Pragmas
  - Global synthesis options
  - Functional Unit Constraint files
- Main structures to optimize
  - Loops
  - Functions
  - Arrays
- Advanced Synthesis directives
- HLS Design Space Exploration