EIE4110 Introduction to VLSI and Computer-Aided Circuit Design

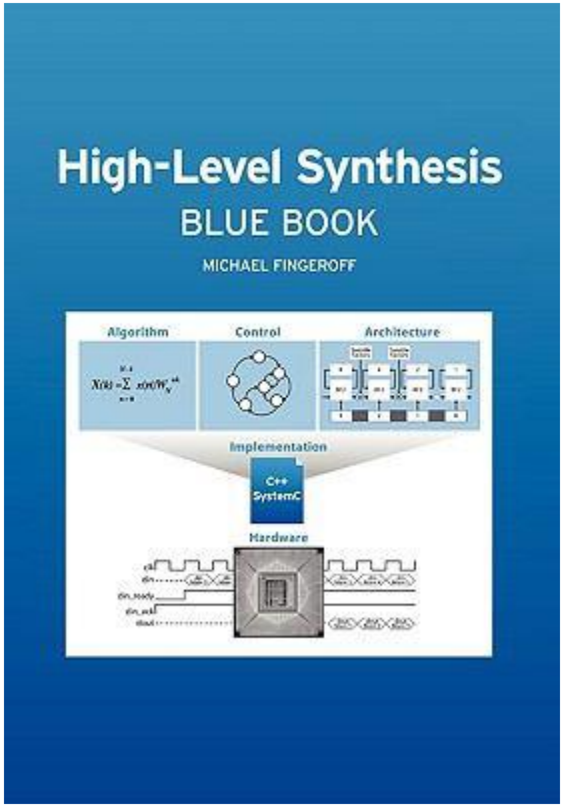
Behavioral Synthesis -High Level Synthesis

Objectives of this Lecture

- Introduction
  - What is High Level Synthesis ?
  - Target architecture
- High Level Synthesis
  1. Allocation
  2. Scheduling
  3. Binding
- HLS optimizations
  - Loop unrolling
  - Function inlining
  - Synthesis of arrays
- C for HW vs. C for SW
- Benefits of HLS
  - Automatic architecture re-targeting
  - Design Space Exploration
  - Less coding–>less verification–>less bugs
- SystemC

References

"High-Level Synthesis Blue Book", Michael Fingeroff

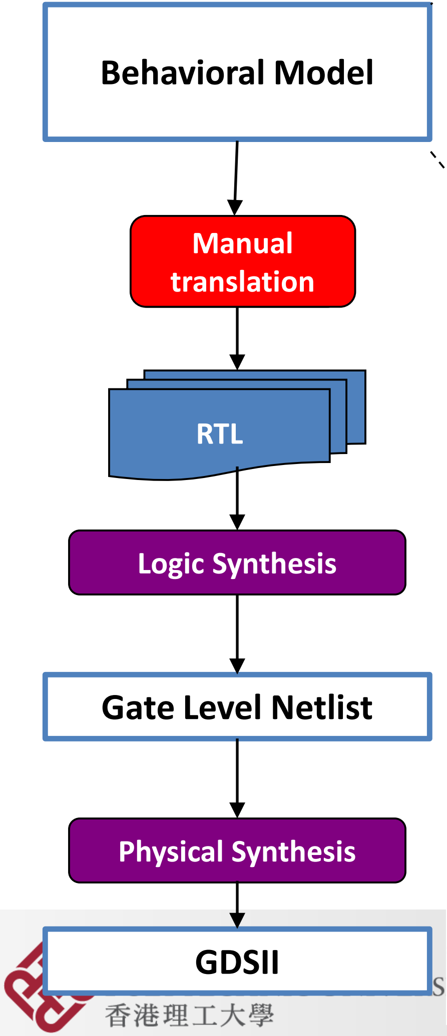


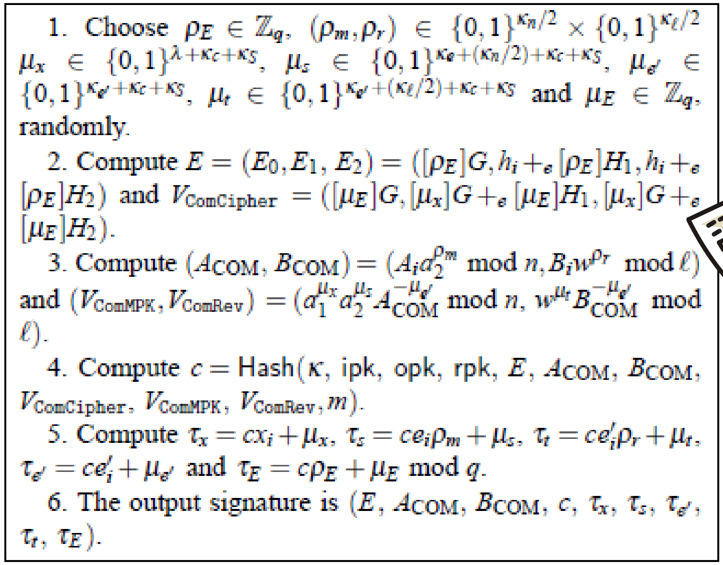
High-Level Synthesis

BLUE BOOK

MICHAEL FINGEROFF

VLSI Flow Overview





1. Choose $\rho_E \in \mathbb{Z}_q$, $(\rho_m, \rho_r) \in \{0,1\}^{\kappa_n/2} \times \{0,1\}^{\kappa_\ell/2}$ $\mu_x \in \{0,1\}^{\lambda+\kappa_c+\kappa_S}$, $\mu_s \in \{0,1\}^{\kappa_e+(\kappa_n/2)+\kappa_c+\kappa_S}$, $\mu_{e'} \in \{0,1\}^{\kappa_{e'}+\kappa_c+\kappa_S}$, $\mu_r \in \{0,1\}^{\kappa_{e'}+(\kappa_\ell/2)+\kappa_c+\kappa_S}$ and $\mu_E \in \mathbb{Z}_q$, randomly.

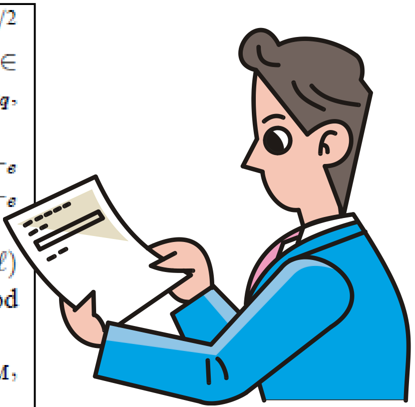
2. Compute $E = (E_0, E_1, E_2) = ([\rho_E]G, h_i +_e [\rho_E]H_1, h_i +_e [\rho_E]H_2)$ and $V_{\texttt{ComCipher}} = ([\mu_E]G, [\mu_x]G +_e [\mu_E]H_1, [\mu_x]G +_e [\mu_E]H_2)$.

3. Compute $(A_{\mathrm{COM}}, B_{\mathrm{COM}}) = (A_i a_2^{\rho_m} \bmod n, B_i w^{\rho_r} \bmod \ell)$ and $(V_{\texttt{ComMPK}}, V_{\texttt{ComRev}}) = (a_1^{\mu_x} a_2^{\mu_s} A_{\mathrm{COM}}^{-\mu_{e'}} \bmod n,\ w^{\mu_r} B_{\mathrm{COM}}^{-\mu_{e'}} \bmod \ell)$.

4. Compute $c = \mathsf{Hash}(\kappa, \mathsf{ipk}, \mathsf{opk}, \mathsf{rpk}, E, A_{\mathrm{COM}}, B_{\mathrm{COM}}, V_{\texttt{ComCipher}}, V_{\texttt{ComMPK}}, V_{\texttt{ComRev}}, m)$.

5. Compute $\tau_x = cx_i + \mu_x$, $\tau_s = ce_i\rho_m + \mu_s$, $\tau_r = ce_i'\rho_r + \mu_r$, $\tau_{e'} = ce_i' + \mu_{e'}$ and $\tau_E = c\rho_E + \mu_E \bmod q$.

6. The output signature is $(E, A_{\mathrm{COM}}, B_{\mathrm{COM}}, c, \tau_x, \tau_s, \tau_{e'}, \tau_r, \tau_E)$.

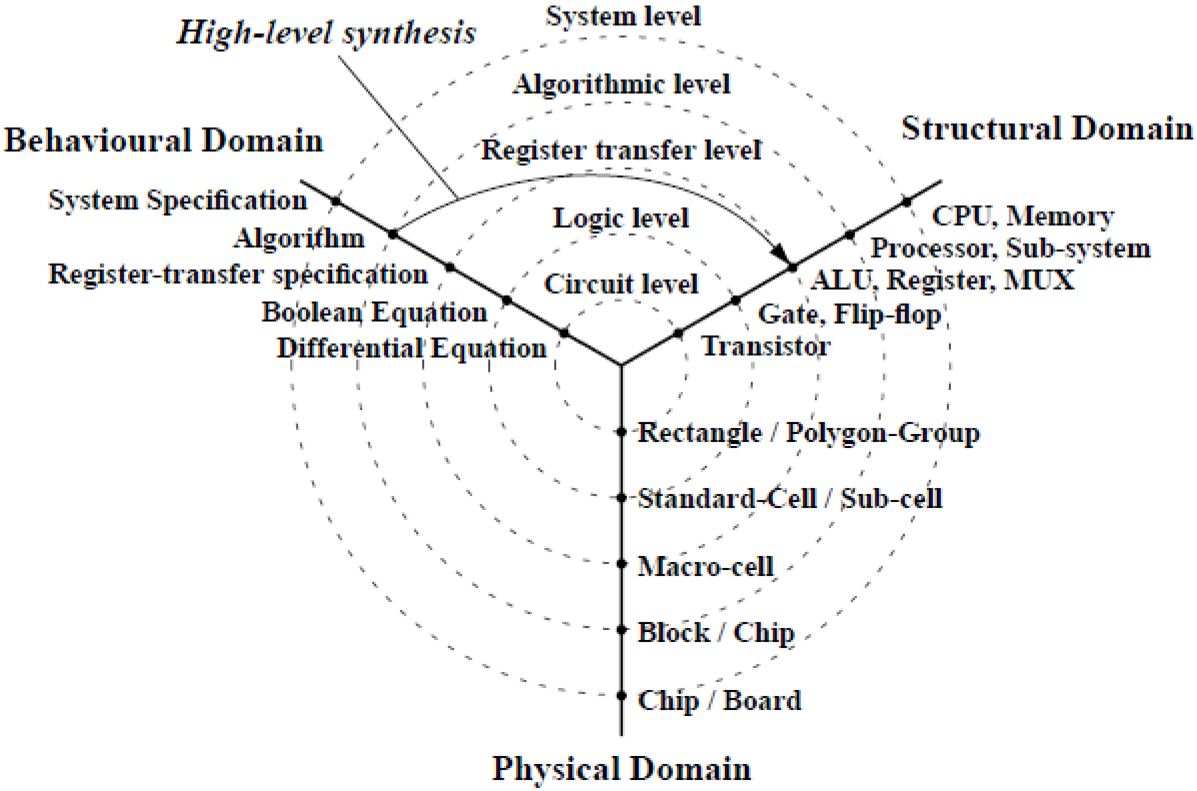


Software Algorithm Designer

Hardware Designer

Y-Chart

- VLSI abstraction levels



Gajski et al.

High Level Synthesis

- Definition

  "Automatic conversion of behavioral, untimed descriptions into hardware that implements that behavior"

- Main Steps
  - Allocation
    - Specify the hardware resources that will be necessary
  - Scheduling
    - Determine for each operation the time at which it should be performed such that no precedence contraint is violated
  - Binding
    - Provide a mapping from each operation to a specific functional unit and from each variable to a register

Typical Applications

Traditionally Fit for HLS

Data Intensive

Arithmetic operations
Simple algorithm

-FIR, FFT, ...

-secret key Encryption

-simple ECC, EDC

-graphic decoding

Traditionally **NOT** fit for HLS

Control Intensive

Arithmetic operation in Complex control

- Video Voice recognition
- Data compression
- Complex CODEC
- DRM
- Turbo ECC
- Public Key Encryption

Traditionally **NOT** fit for HLS

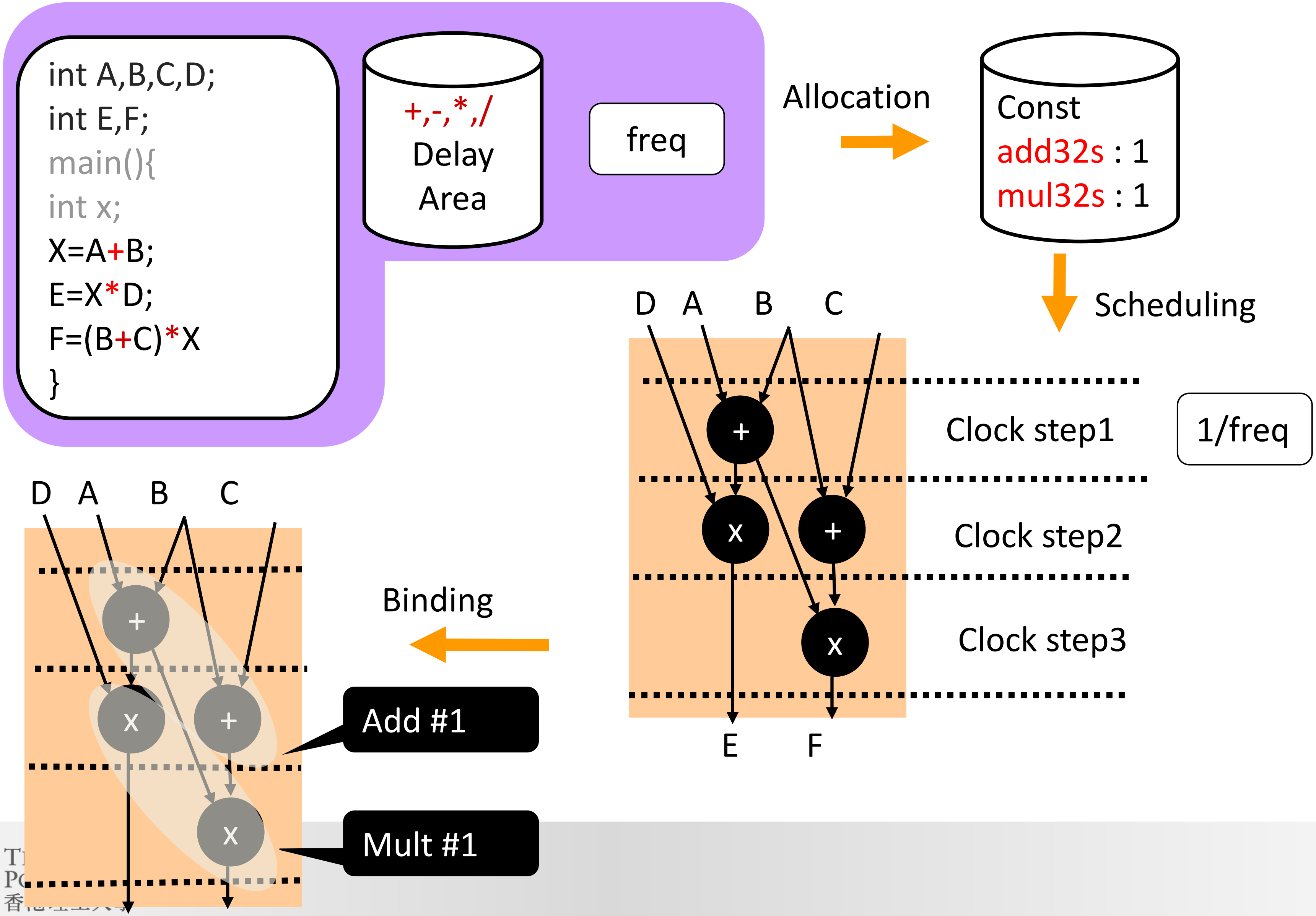
Controller

Sequencers

-USB I/F, ATA, UART,...
-PCI bus I/F, AMBA bus.
-DMA, TIMER,…
-SDRAM I/F, NAND flash,...

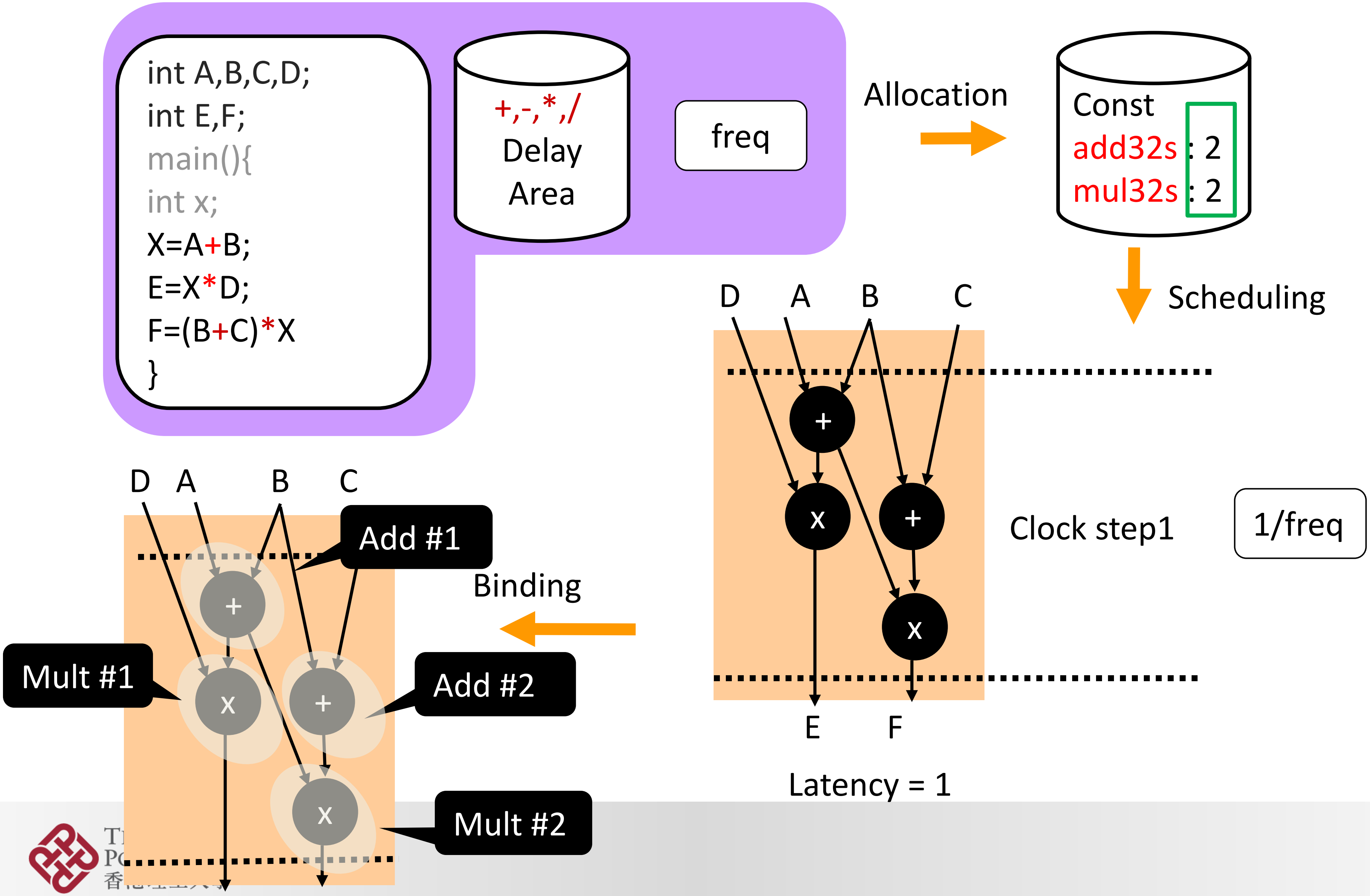
- Multiple internal synthesis Engines
- Multiple Synthesis directives (local and global)
- C syntax extension (implementation description)

High Level Synthesis Overview



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

High Level Synthesis Overview cont.



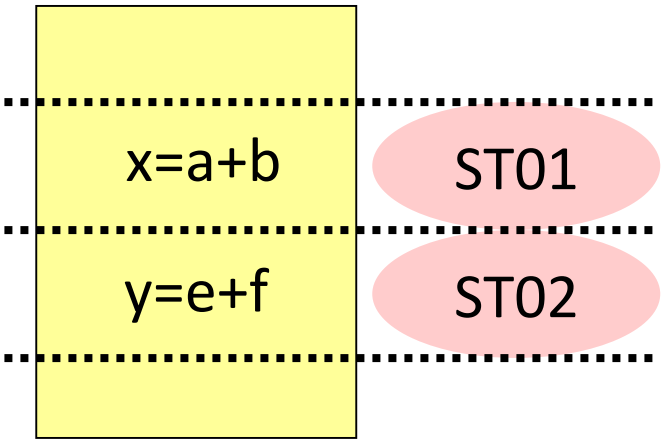
```
int A,B,C,D;
int E,F;
main(){
int x;
X=A+B;
E=X*D;
F=(B+C)*X
}
```

HLS Resources Constraints

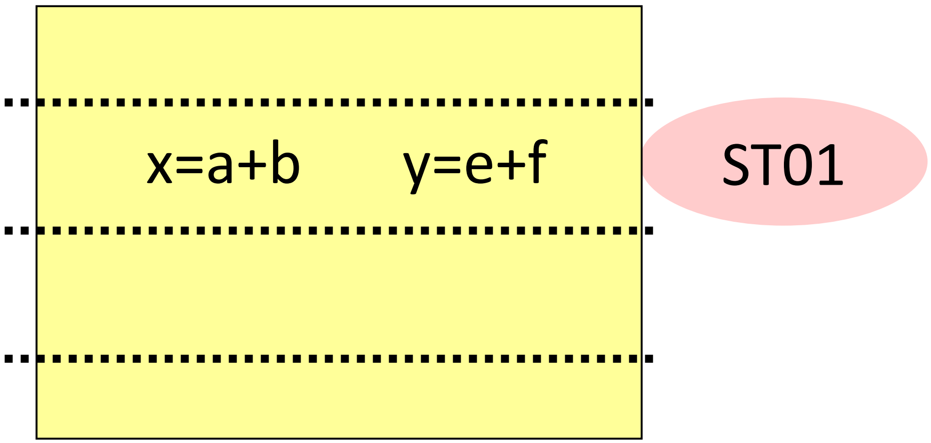
- Functional Unit Constraint file specifies how many FUs can be instantiated → Impacts the synthesized architecture

```
int main(){
  x = a+b;
  y = e+f;
}
```

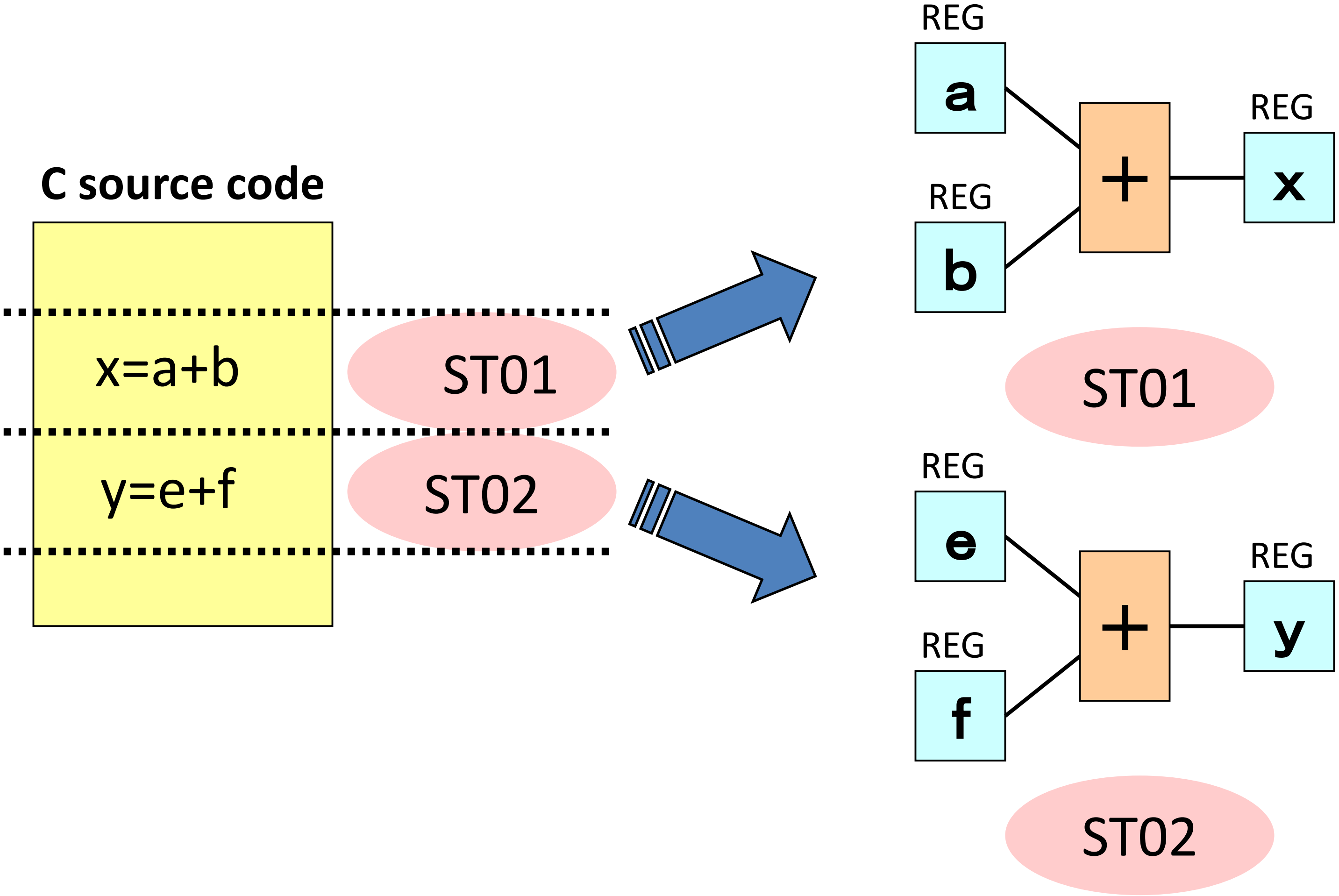
1 Adder



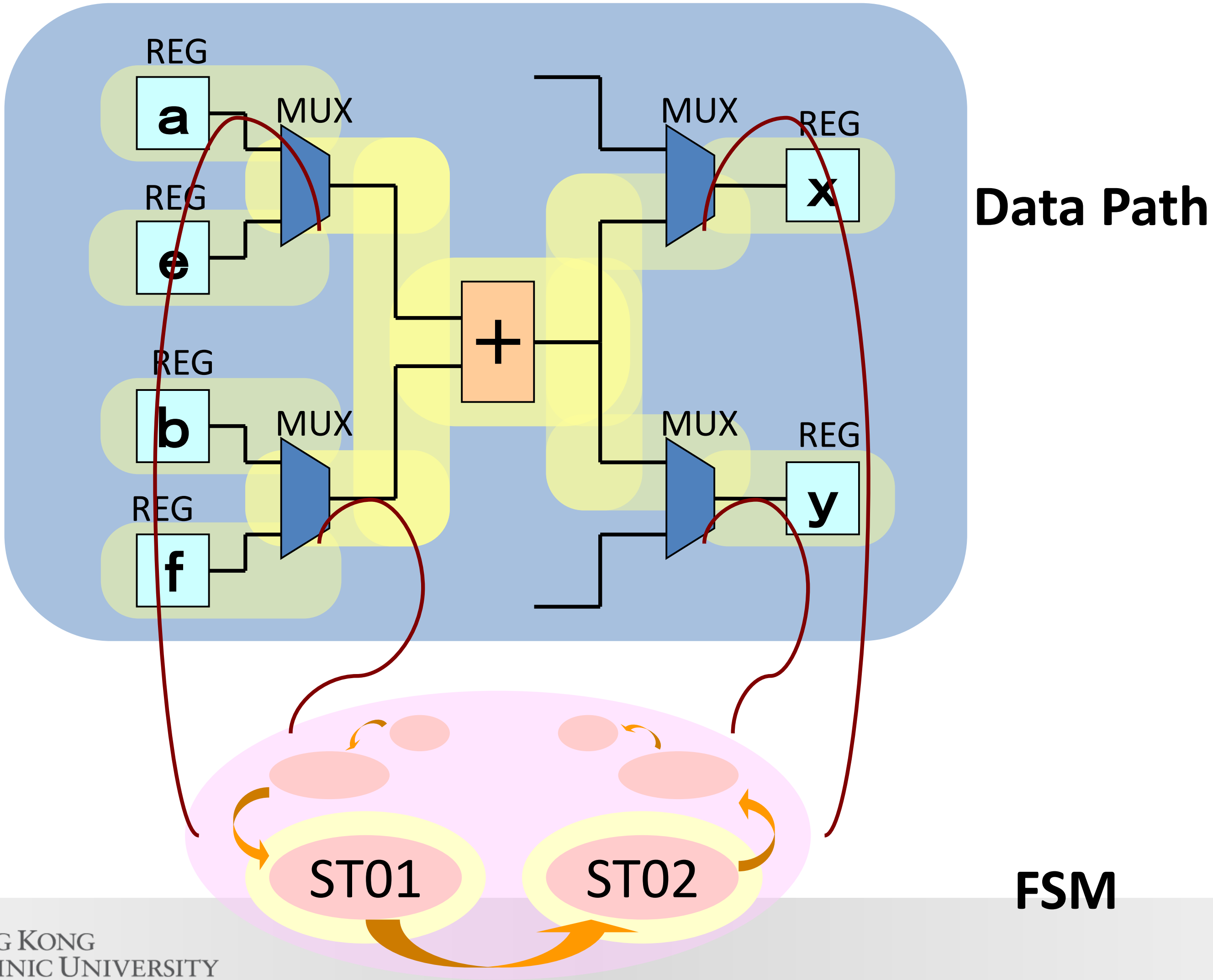
2 Adders



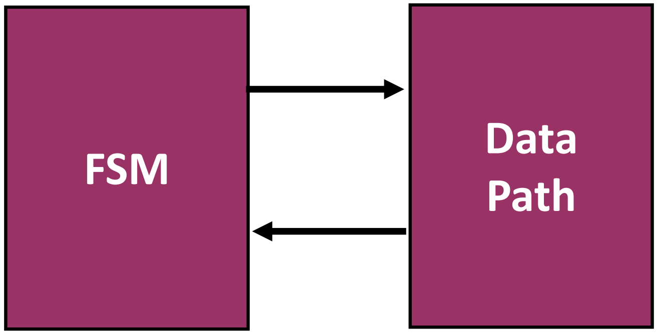
Resource Constraint: Min FUs (Resource Sharing)



High Level Synthesis – Min FUs (Resource Sharing)



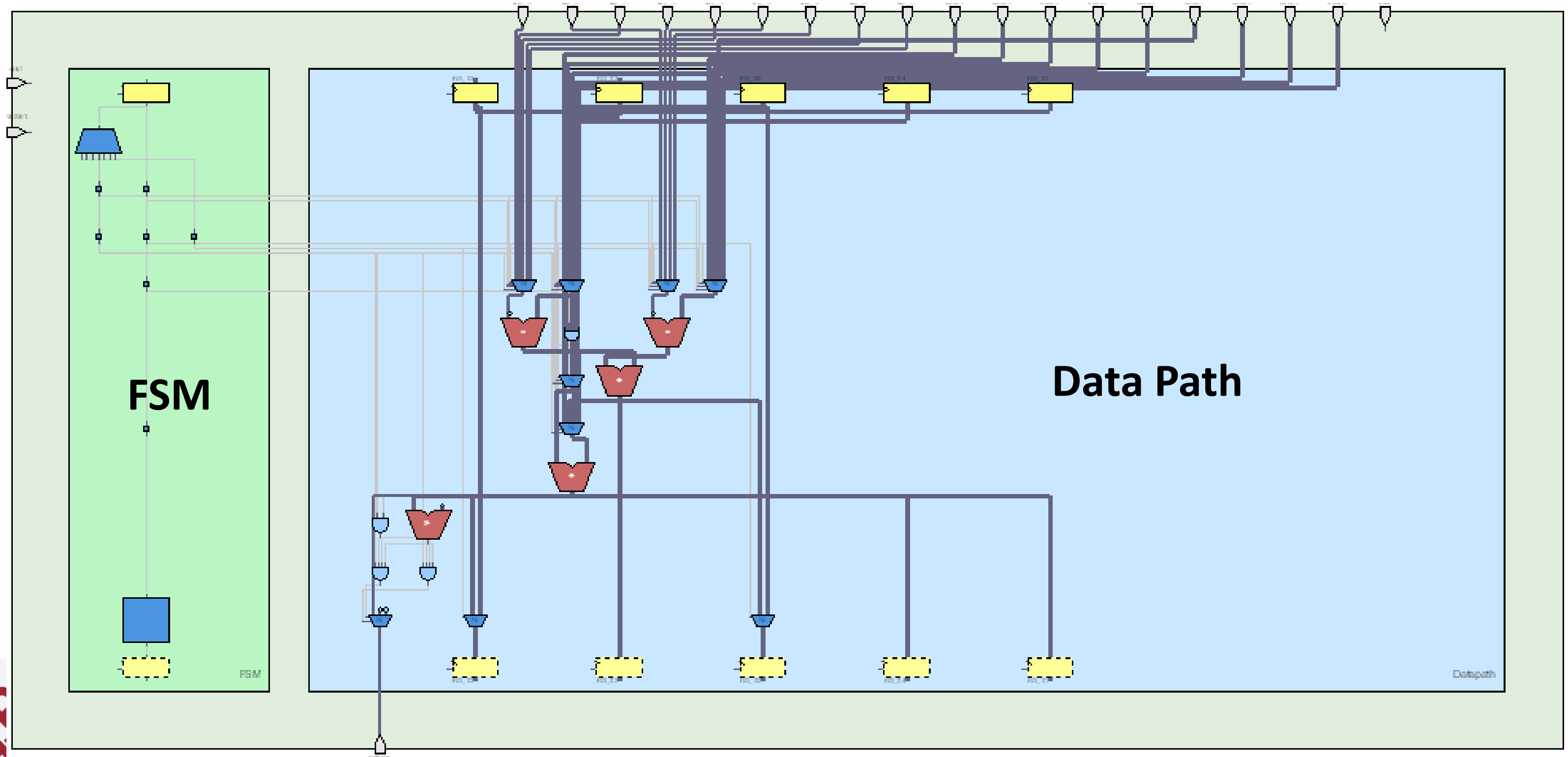
Target Architecture



Finite State Machine with Data Path

FIR Filter example

```
for(i=0;i<9;i++)
        sum += ary[i] * coeff[i] ;
```



Fmax (delay) vs. FUs delay

- Very important for good synthesis result
- Latency will change

```
int main(){
  x = a+b;
  y = e+f;
}
```

F = 100Mhz = 10ns

Delay adder =5ns

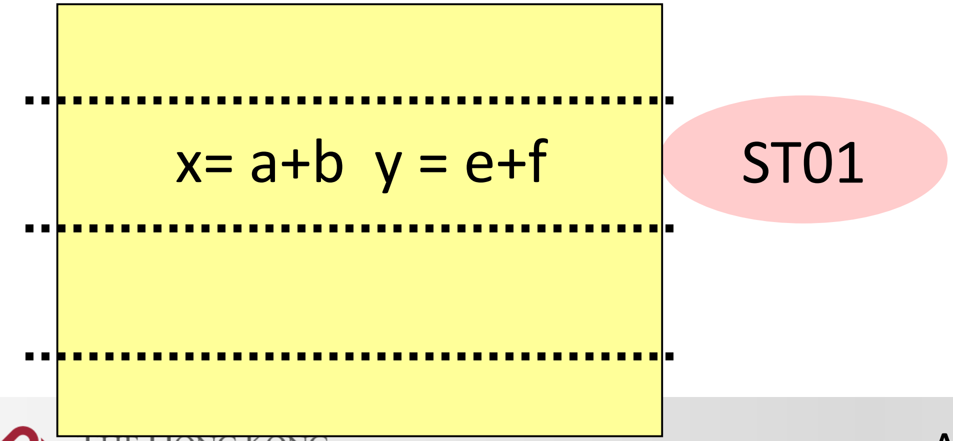
FCNT = 2 adders

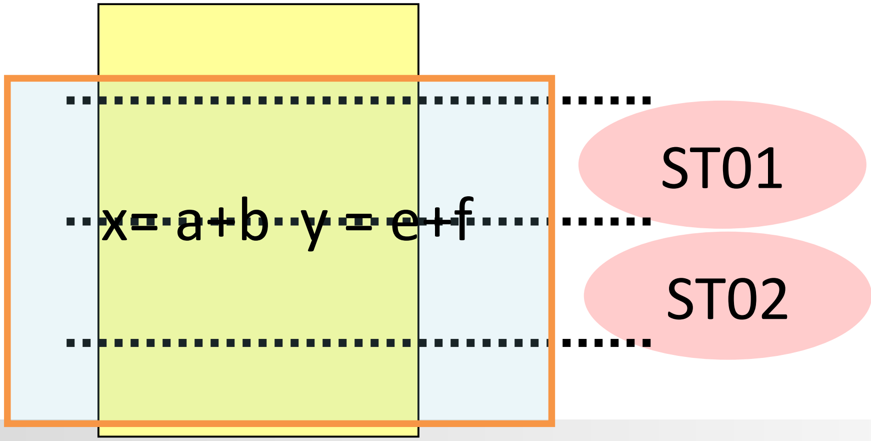
```
int main(){
  x = a+b;
  y = e+f;
}
```

F = 200Mhz = 5ns

Delay adder =6ns

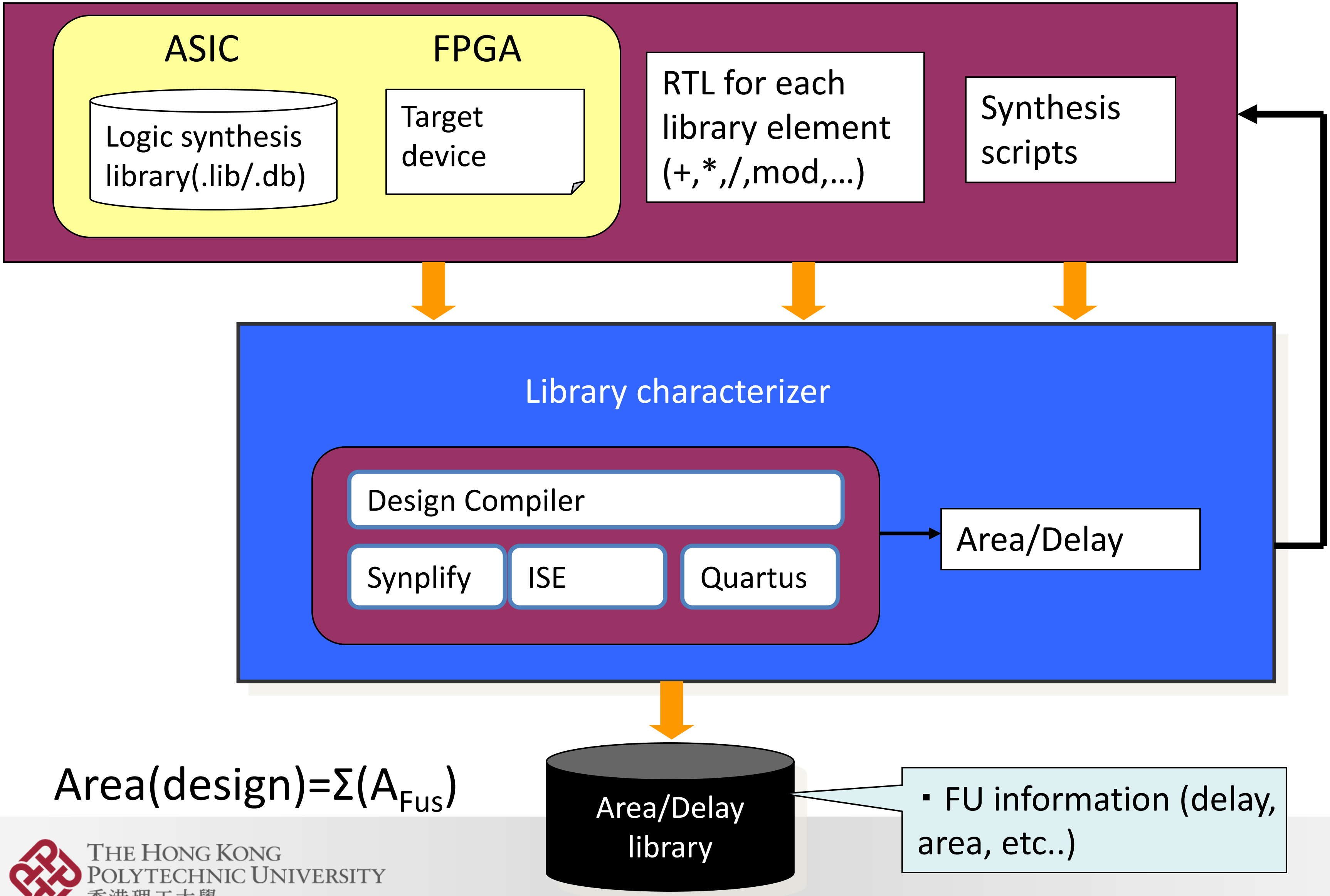
FCNT = 2 adders





Achieved: (1) Multi-cycle operation or (2) pipelining

Library Characterizer



- FU information (delay, area, etc..)

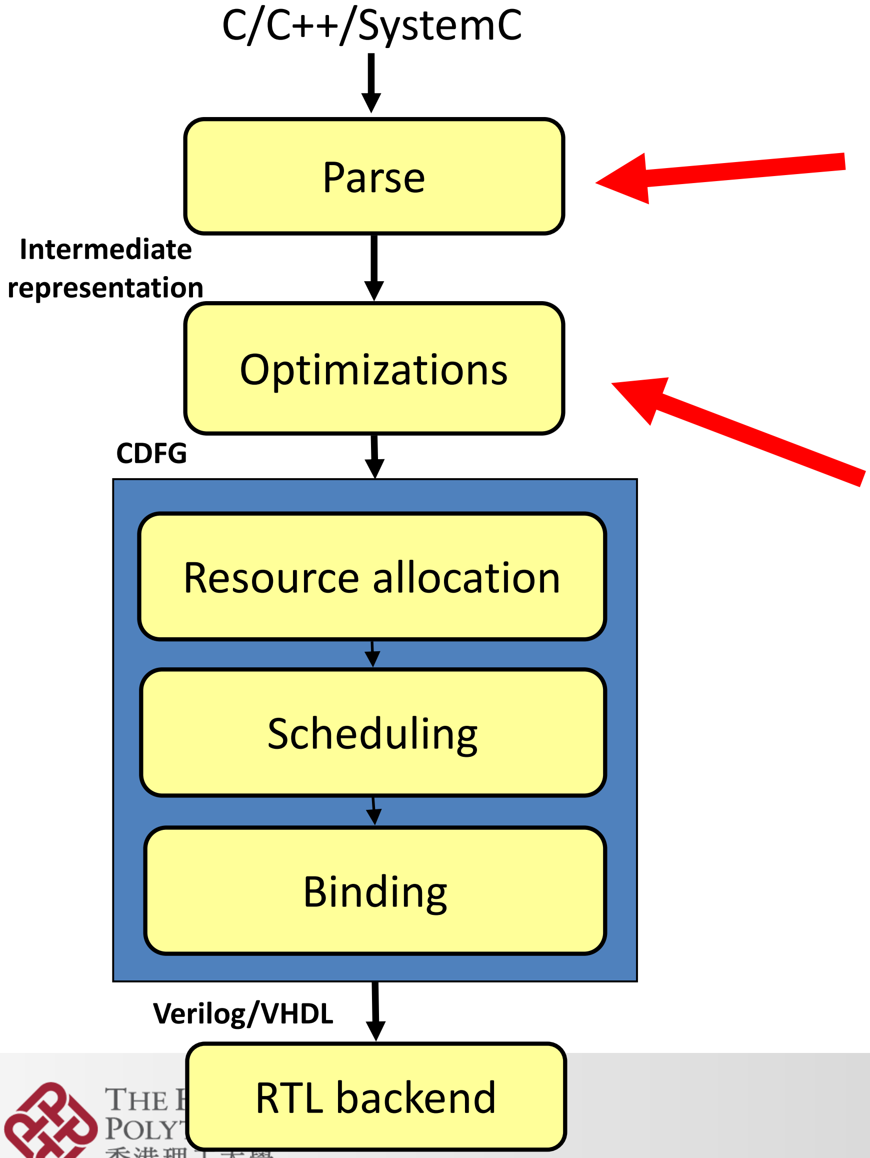
$$\text{Area(design)}=\Sigma(A_{\text{Fus}})$$

Library Format

- Vendor dependent. No standard format like in logic synthesis (.lib, .db).
- E.g.:

```
@FLIB {
   NAME      mul4s
   KIND      *
   BITWIDTH  4
   DELAY     264
  SIGN      SIGNED
   SYN_TOOL  ISE
   LSTAGE    1
}
@FLIB {
   NAME      mul8u
   KIND      *
   BITWIDTH  8
   DELAY     362
   SIGN      UNSIGNED
   SYN_TOOL  ISE
   LSTAGE    1
}
```

HLS Flow Overview



Compiler Front End
(lex and parse)

Compiler optimizations

Constant/Variable propagation, common sub-expression elimination, dead-code elimination, etc..

Synthesis transformations

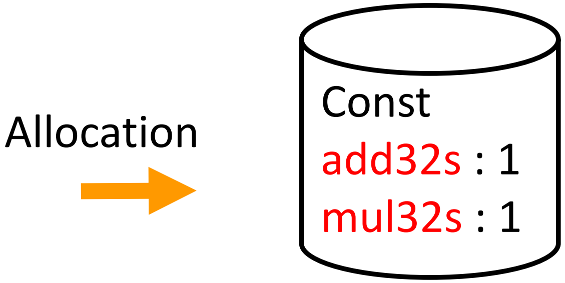
Loop unrolling, array expansion, speculation, conditional expansions, tree high reduction

---

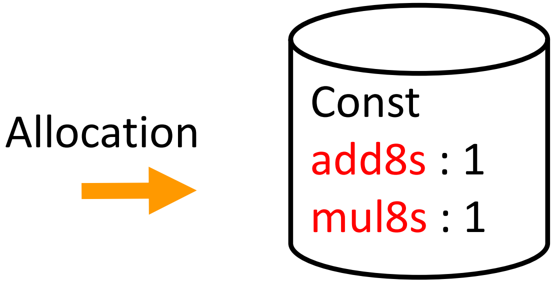
Resource Allocation

- Specify the hardware resources that will be necessary

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



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



➔ How to represent variable bitwidths in ANSI-C? (e.g. 12 bits, 17bits)

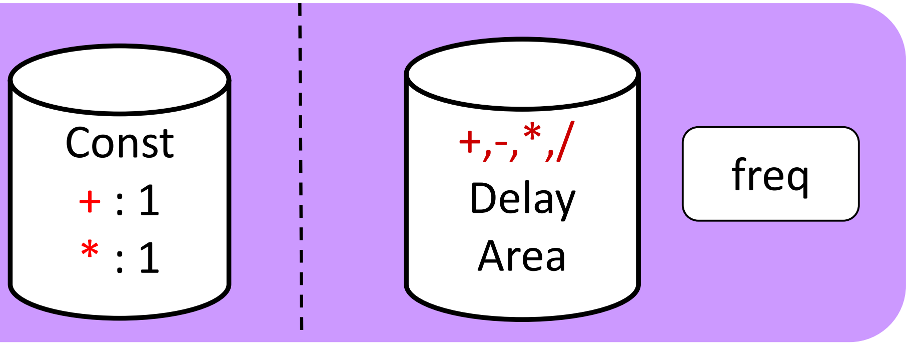
Scheduling : Constraint (Resources, Time)

- Resource constraint:
  - Given a set *O* of operations with a partial ordering, a set *K* of functional unit types, a type function, $\sigma: O \rightarrow K$, to map the operations into the functional unit types, and resource constraints m_k for each functional unit type.
  - Find a (optimal) schedule for the set of operations that obeys the partial ordering (Minimize latency)
    - ➔ ASAP, ALAP, List scheduling

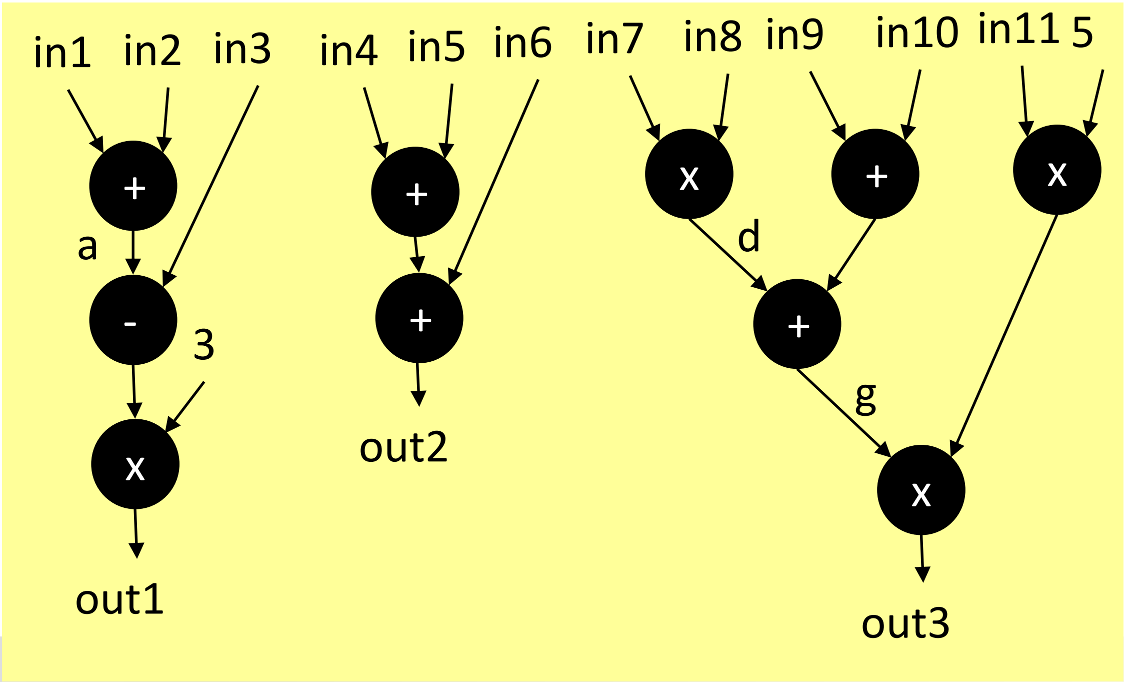
- Timing constraints
  - Minimize the resources given a fixed timing
    - ➔ Force directed scheduling

Example

```
int main(){
int in1, in2, in3, in4,…,in11;
int a, d, g;
int out1, out2, out3;
a= in1+in2;
out1 = (a-in3) * 3;
out2 = in4 + in5 + in6;
d = in7 * in8;
g = d + in9 + in10;
out3 = in11 * 5 * g;
}
```



CDFG

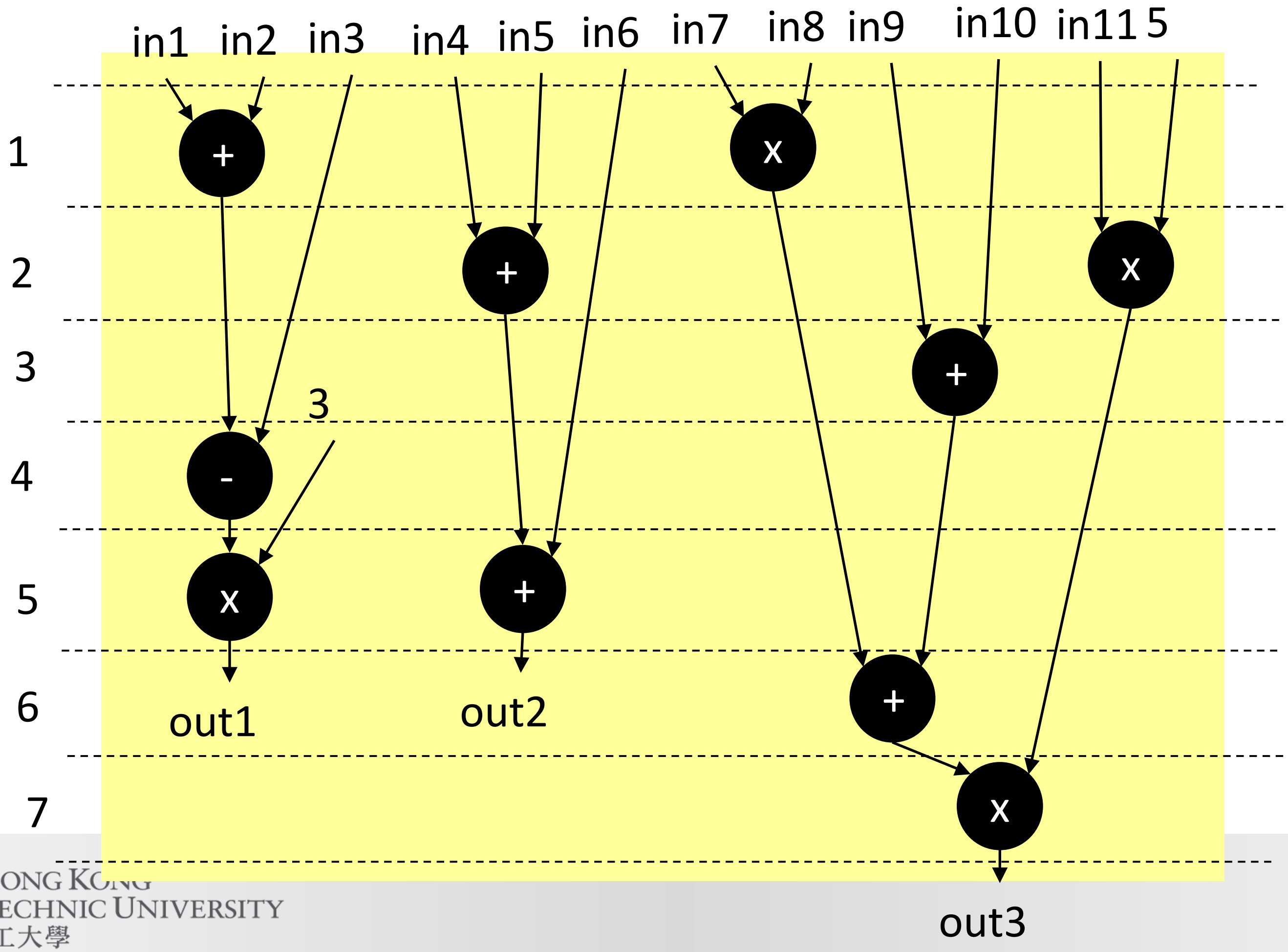


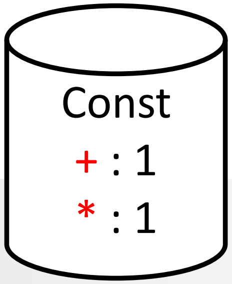
Scheduling (I): ASAP

- Map operations to their *earliest* possible start time not violating the precedence constraints
- Easy and fast to compute
  - Find longest path in a directed acyclic graph
  - No attemp to optimize ressource cost
- Gives the fastest possible schedule if unlimited amount of resources are available
- Gives an upper bound on execution speed

ASAP Scheduling

- Sort operation topologically according to their dependence
- Schedule operations in sorted order by placing them in the earliest possible control step



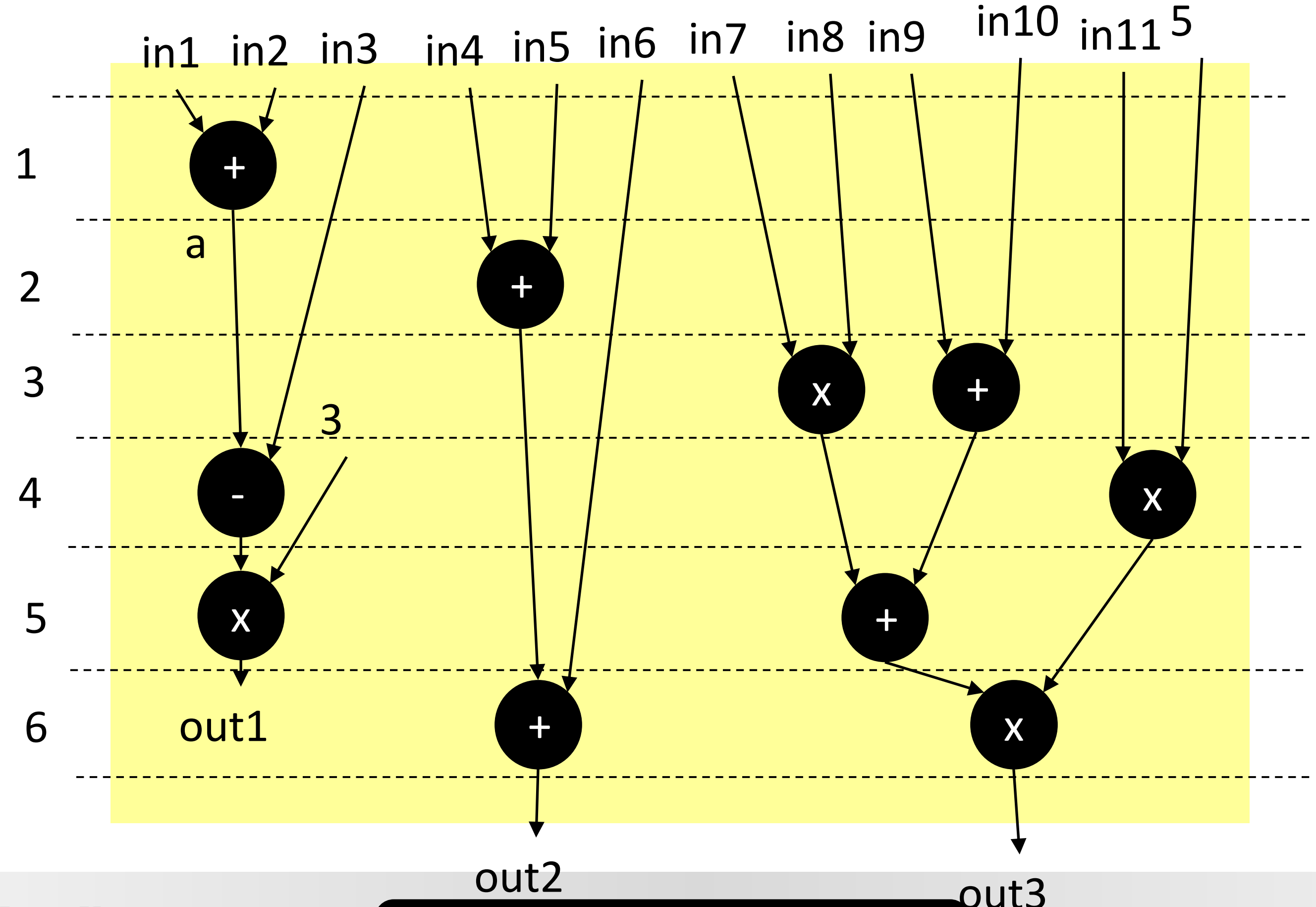


Scheduling (II): ALAP

- Map operations to their *latest* possible start time not violating the precedence constraints
- Easy and fast to compute
  - Find longest path in a directed acyclic graph
  - No attemp to optimize ressource cost

ALAP Scheduling

- Sort operations topologically according to their dependence
- Schedule operations in the reverse order by placing them in the latest possible control step

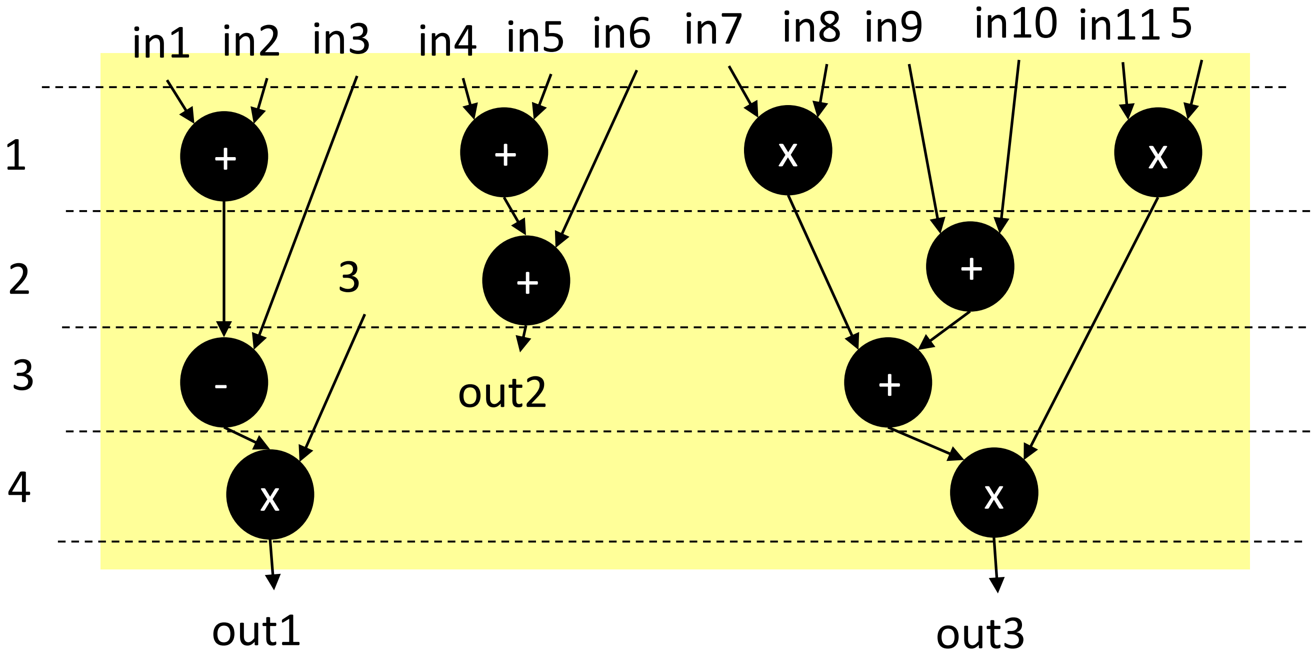


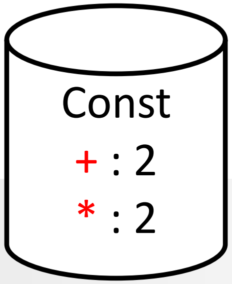
Const
+ : 1
* : 1

6 steps need → optimal

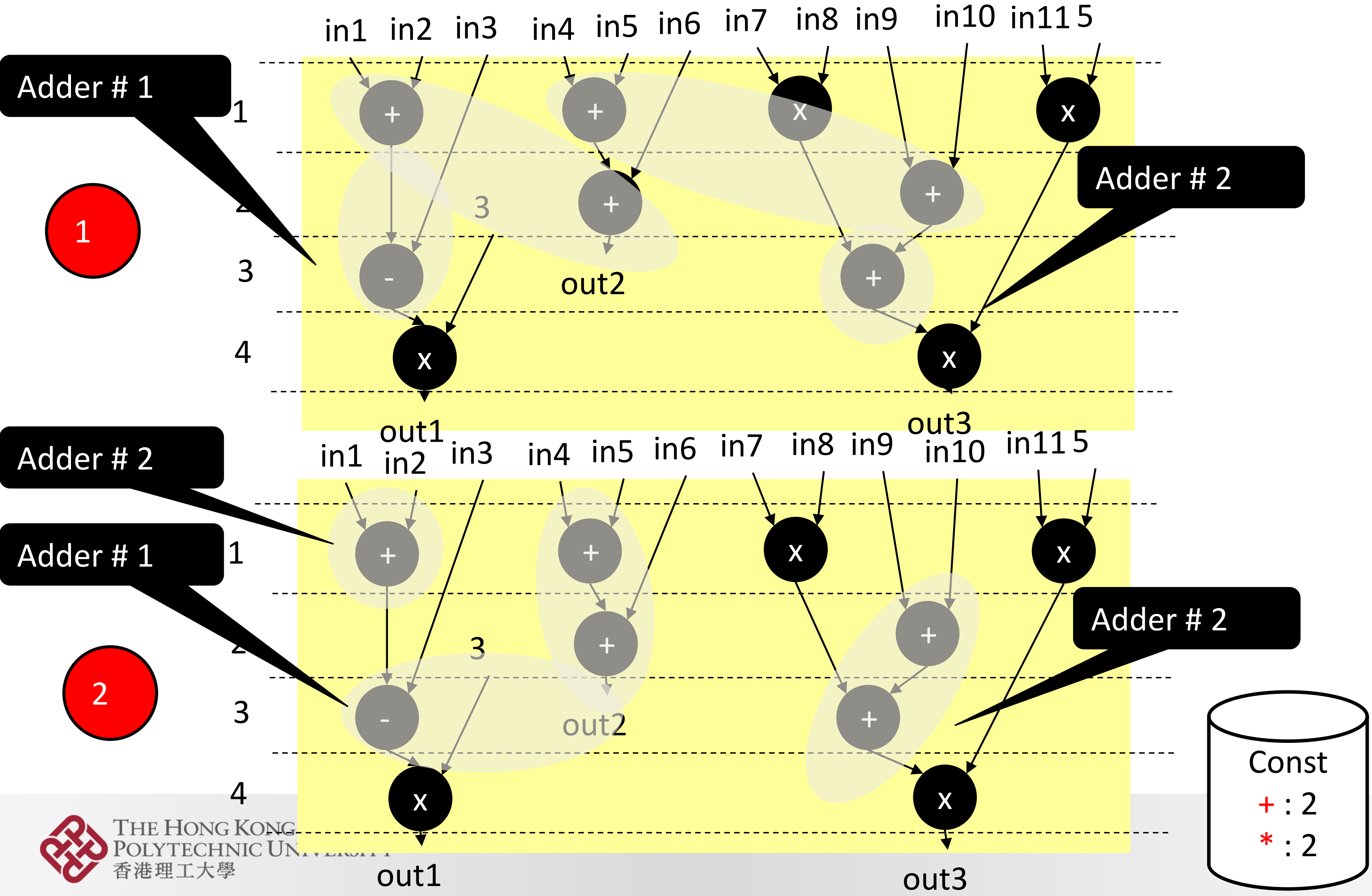
Binding for ASAP Example with new FU Constraint

- Assign Operators to particular FUs
- ASAP with 2 adders and 2 multipliers

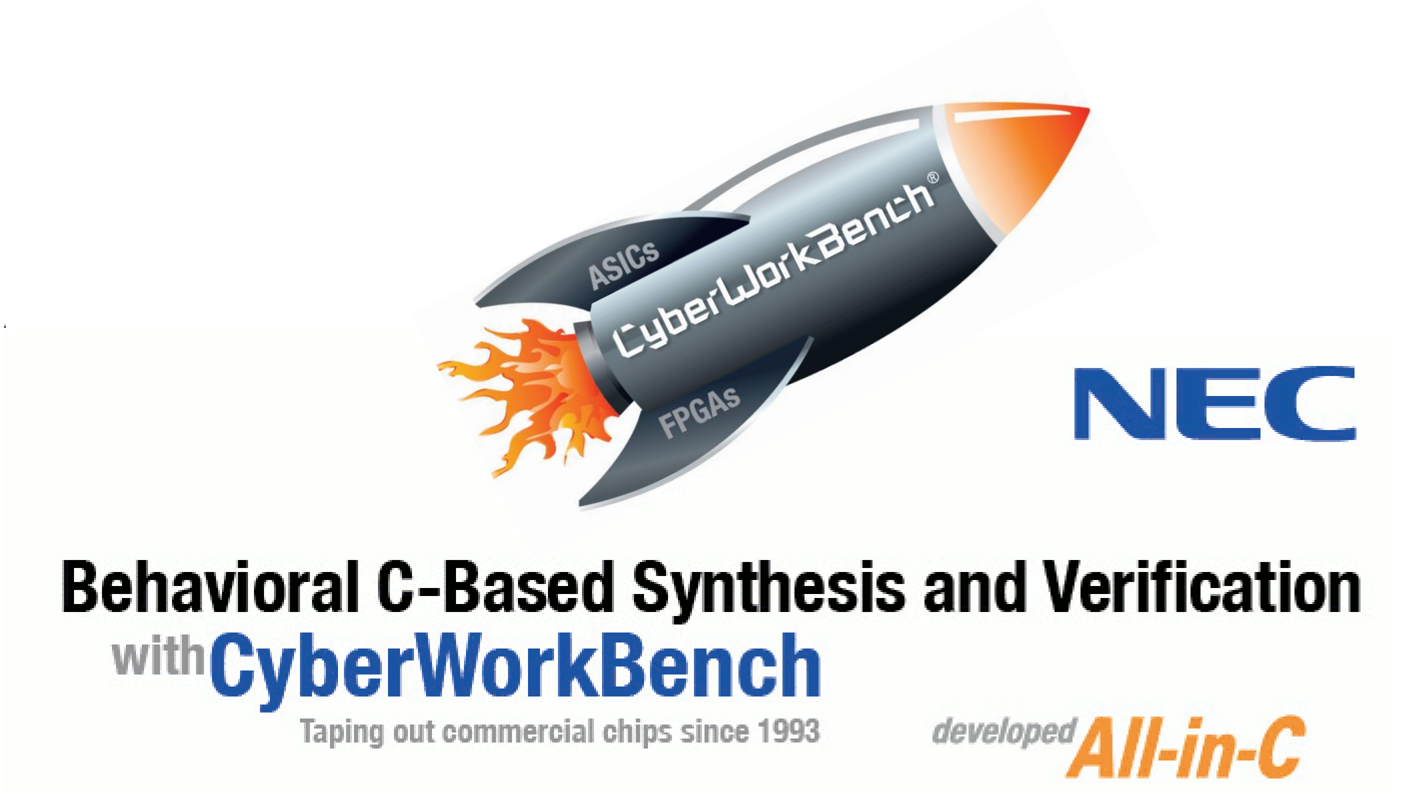




Binding Example con’t



NEC - CyberWorkBench


ASICs
CyberWorkBench®
FPGAs
NEC
Behavioral C-Based Synthesis and Verification
with CyberWorkBench
Taping out commercial chips since 1993
developed All-in-C

Video

HLS Optimizations

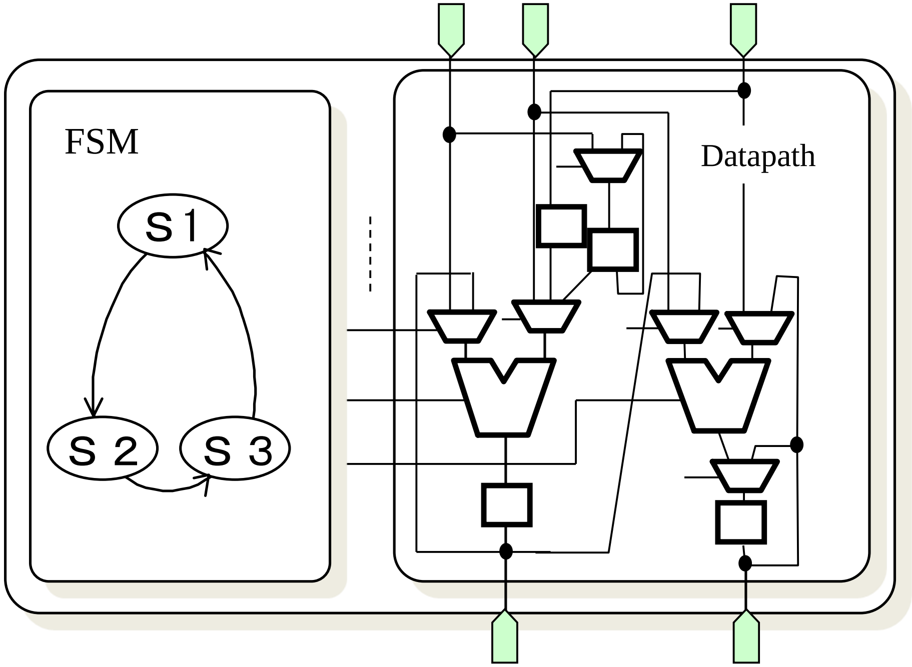
- Behavioral Synthesizer’s Optimizations to improve QoR
  - Loop pipelining
  - Loop parallelization
  - Automatic bitwidth reduction
  - Fixed point data types support (float to fixed point data type conversion)
  - Speculation
  - Tree high reduction
  - Support for ALU macros

How to control HLS optimizations

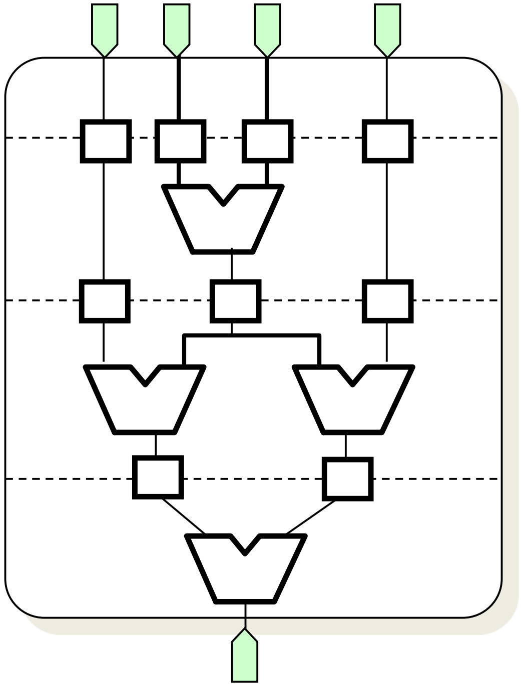
- Though **global synthesis options** (from the dialog window) → apply to the entire design
  - Maximum frequency
  - Map all arrays to memory or registers
- Through the **resource constraint file**
  - Limit the number of FUs that can be instantiated
- **Local synthesis directives** (pragmas). Comments with keyword (e.g. // Cyber unroll_times = all)
  - Unroll loops
  - Inline functions
  - Map single arrays to memory or registers

Automatic Pipeline Mode and Loop Folding

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



Sequential Circuit

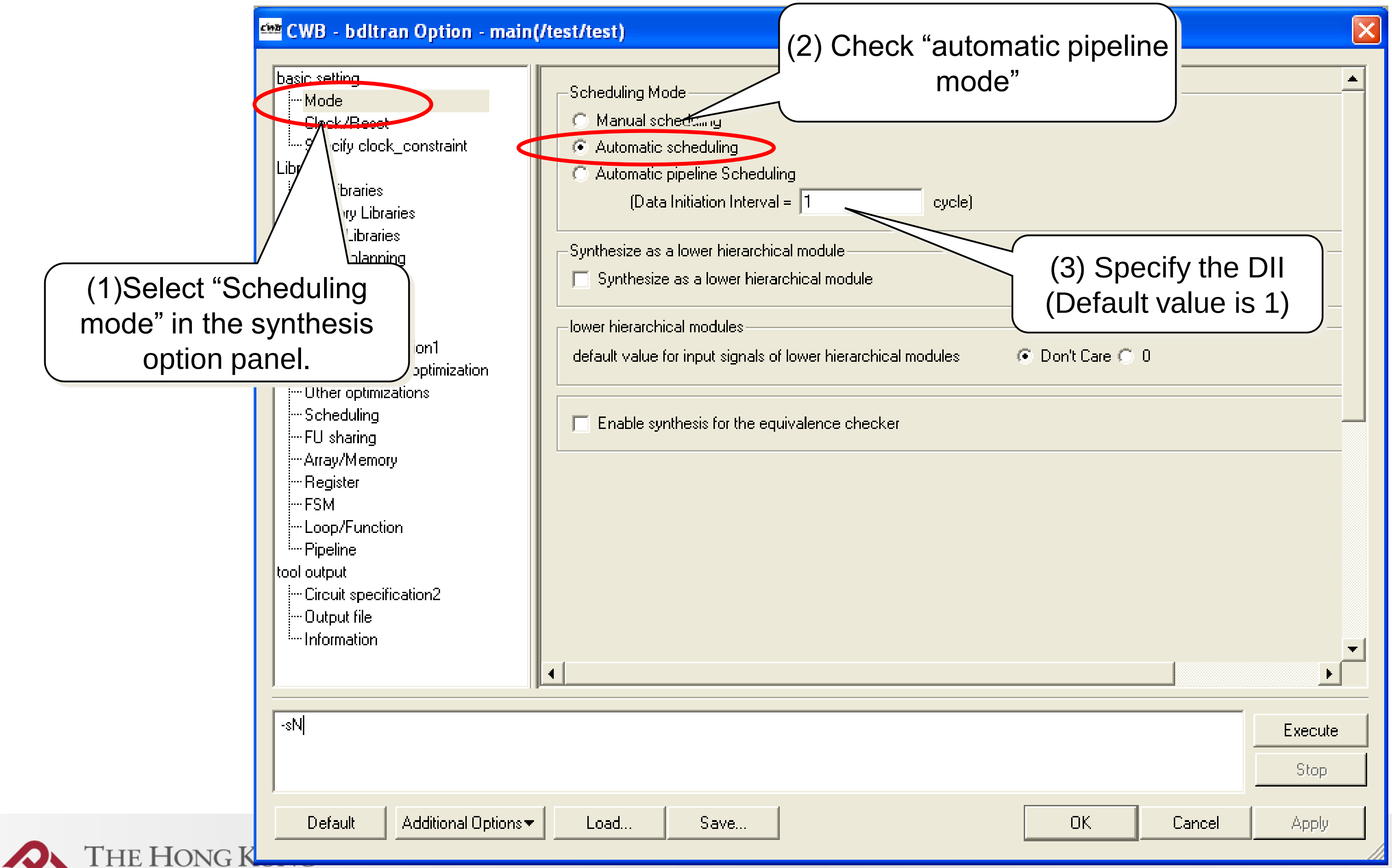


Pipelined circuit

All steps are executed simultaneously. (High throughput circuit)

Automatic Pipelining

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



CWB - bdltran Option - main(/test/test)

basic setting
- Mode
- Clock/Reset
- S cify clock_constraint

Libr
- braries
- ry Libraries
- Libraries
- planning
- on1
- optimization
- Other optimizations
- Scheduling
- FU sharing
- Array/Memory
- Register
- FSM
- Loop/Function
- Pipeline

tool output
- Circuit specification2
- Output file
- Information

Scheduling Mode

○ Manual scheduling

● Automatic scheduling

○ Automatic pipeline Scheduling

(Data Initiation Interval = 1 cycle)

Synthesize as a lower hierarchical module

□ Synthesize as a lower hierarchical module

lower hierarchical modules

default value for input signals of lower hierarchical modules ● Don't Care ○ 0

□ Enable synthesis for the equivalence checker

-sN

(1)Select "Scheduling mode" in the synthesis option panel.

(2) Check "automatic pipeline mode"

(3) Specify the DII (Default value is 1)

Loop folding

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

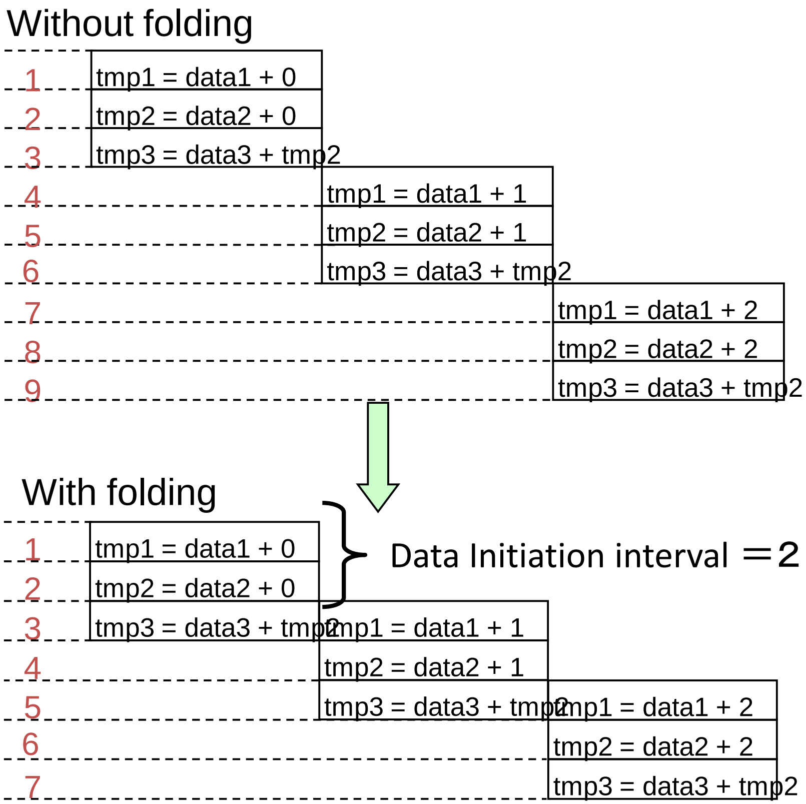
DII specification

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

    tmp2 = data2 + cnt;

    tmp3 = data3 + tmp2;

}
```



Loop Unrolling

- Only “for” loops with constant # of iterations can be unrolled.

- Unrolling all iterations

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

After complete unrolling

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

- Unrolling 2 times in parts

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

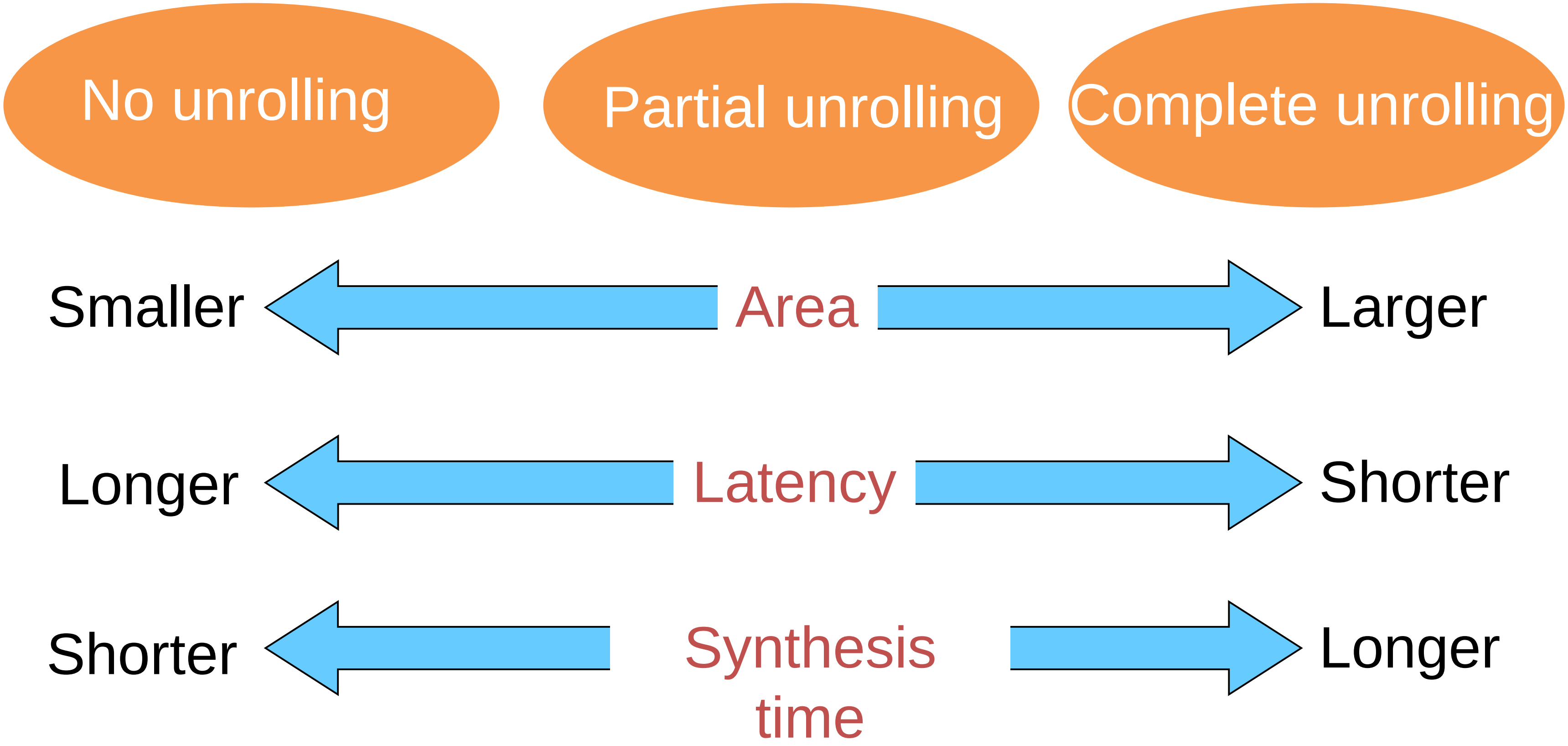
After partial unrolling

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

# of unrolled iteration influences area and latency

Impact of Loop Unrolling

- Selection and trend for loop unrolling



No unrolling

Partial unrolling

Complete unrolling

Smaller Area Larger

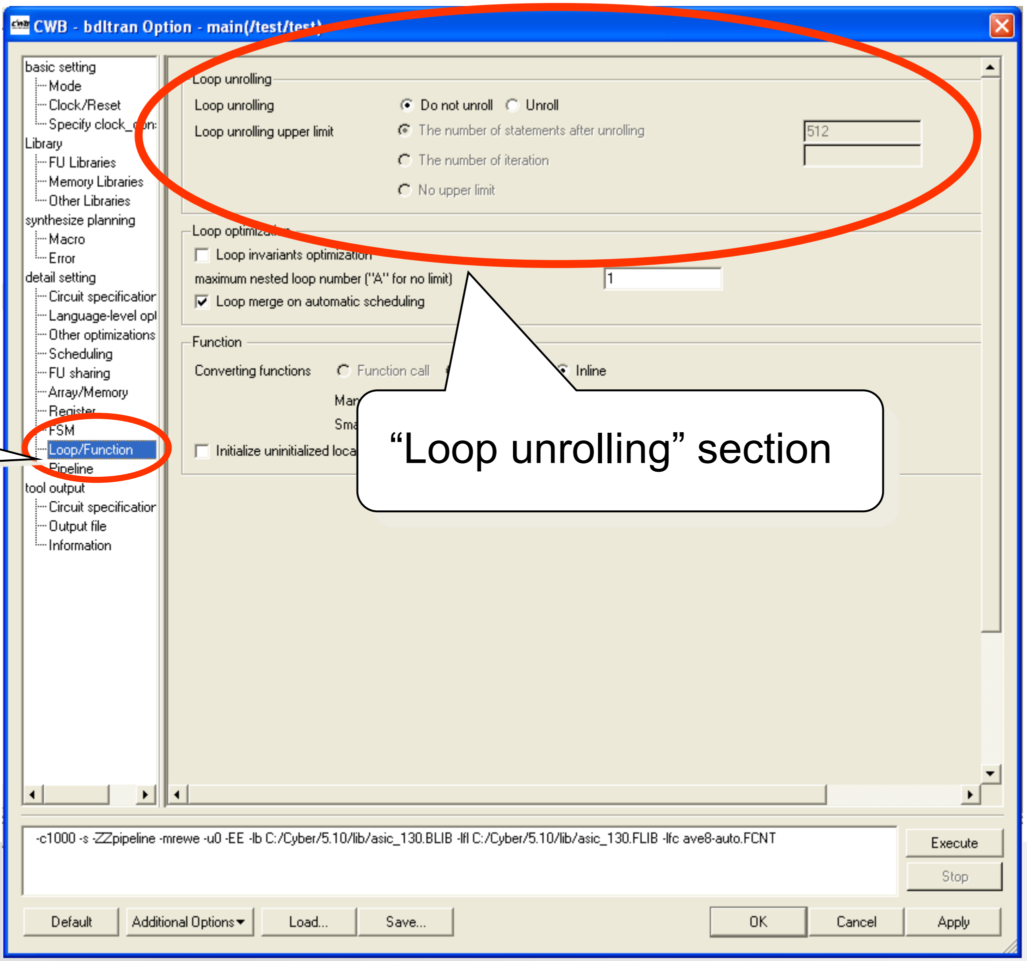
Longer Latency Shorter

Shorter Synthesis time Longer

Loop Unrolling Control

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

“Loop/Function”



Loop unrolling

Loop unrolling ● Do not unroll ○ Unroll

Loop unrolling upper limit ● The number of statements after unrolling 512

○ The number of iteration

○ No upper limit

Loop optimization

□ Loop invariants optimization

maximum nested loop number ("A" for no limit) 1

☑ Loop merge on automatic scheduling

Function

Converting functions ○ Function call ● Inline

Man

Sma

□ Initialize uninitialized loca

```
-c1000 -s -ZZpipeline -mrewe -u0 -EE -lb C:/Cyber/5.10/lib/asic_130.BLIB -lfl C:/Cyber/5.10/lib/asic_130.FLIB -lfc ave8-auto.FCNT
```

“Loop unrolling” section

Loop Unrolling

- Control through attributes

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

* 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++) {
   ...
   ...
}
```

Functions Synthesis

Implementations of function

(1) inline expansion

(2) goto conversion

(3) conversion into functional unit

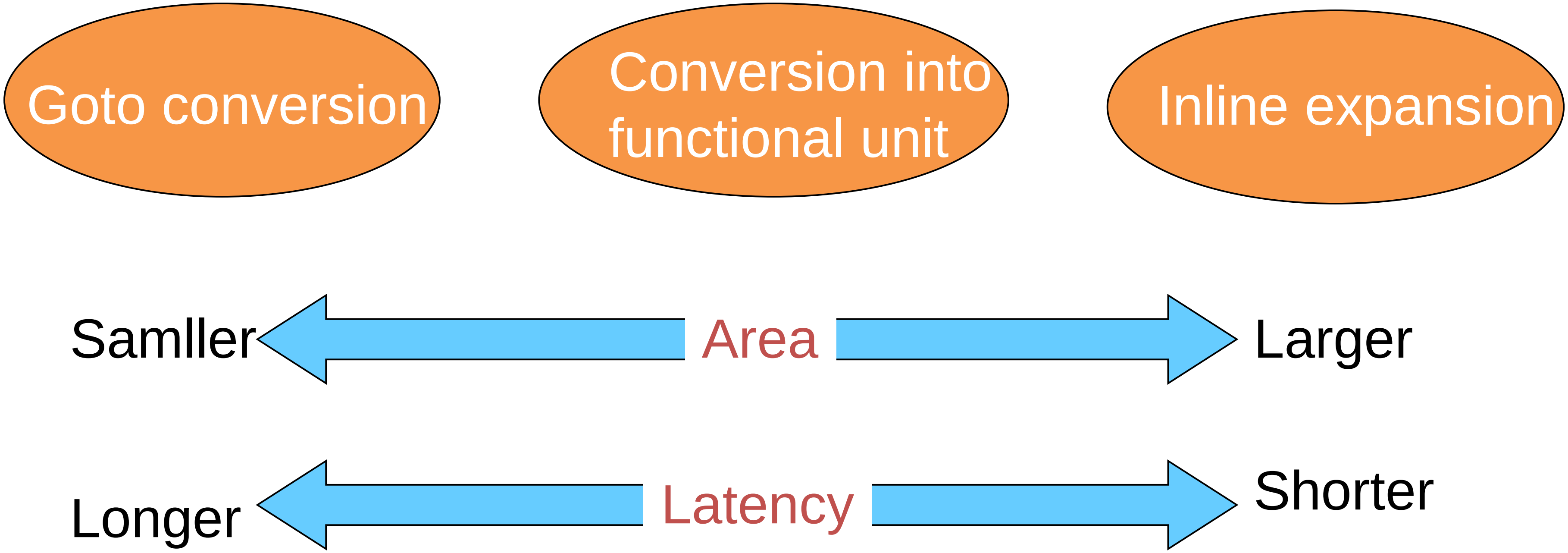
b) Synthesis option for function implementation

c) Synthesis attribute for function implementation

d)Convert into functional units

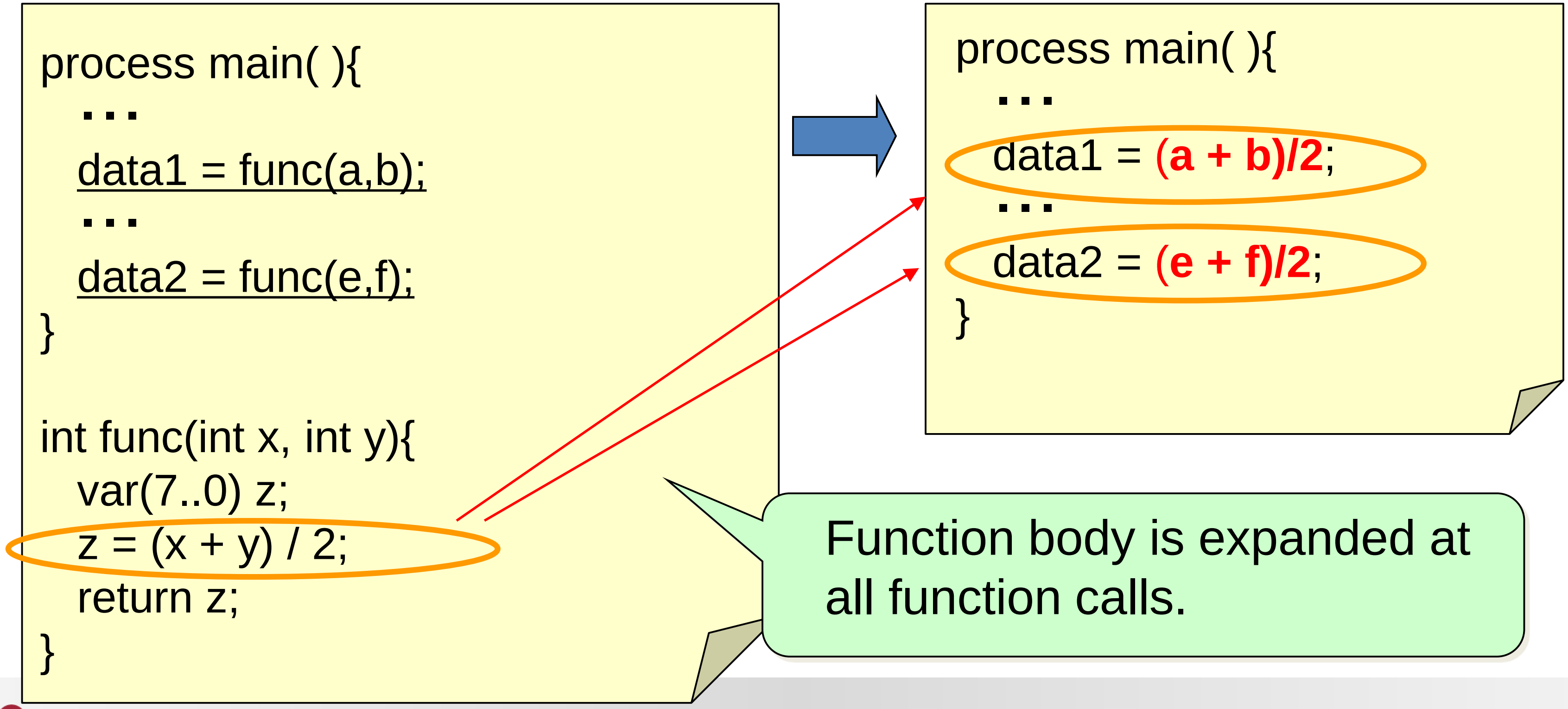
Function Synthesis Impact

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

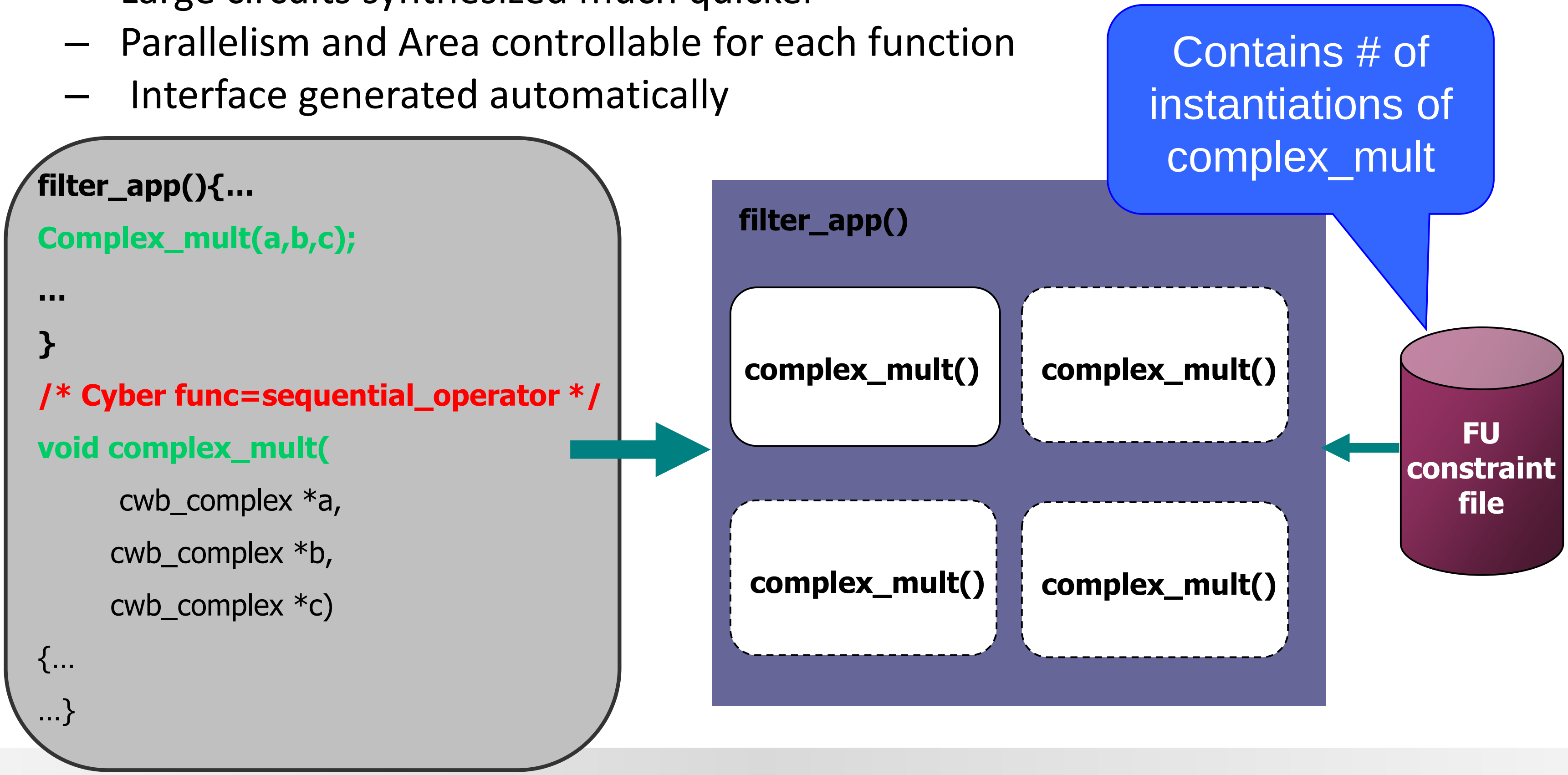
For each function call, the block starting from label “L_func” is executed.
The function definition is implemented only in the block.

```
…
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 functional Unit

- A function definition is treated as a functional unit and the # of instances can be controlled
- Benefits:
  - Large circuits synthesized much quicker
  - Parallelism and Area controllable for each function
  - Interface generated automatically

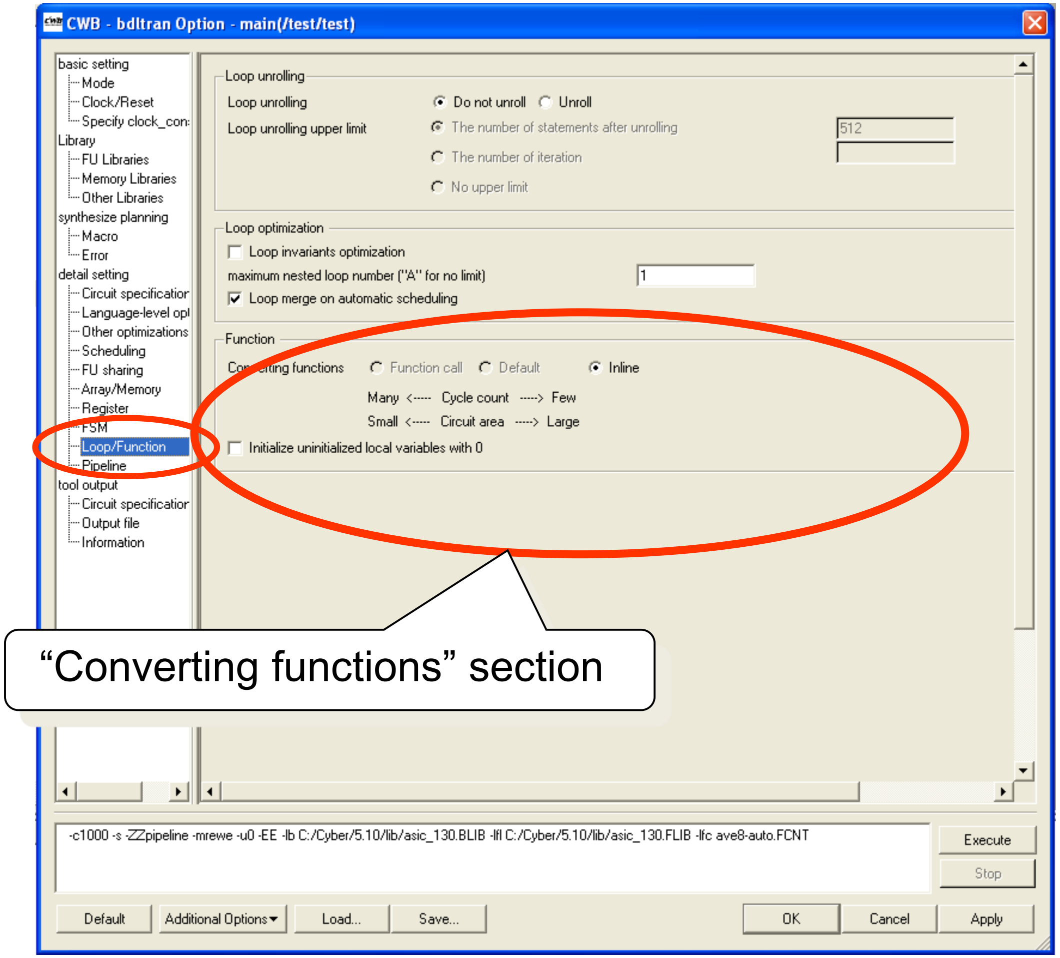


```
filter_app(){…
Complex_mult(a,b,c);
…
}
/* Cyber func=sequential_operator */
void complex_mult(
        cwb_complex *a,
       cwb_complex *b,
       cwb_complex *c)
{…
…}
```

Contains # of instantiations of complex_mult

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.



CWB - bdltran Option - main(/test/test)

Loop unrolling

Loop unrolling ● Do not unroll ○ Unroll

Loop unrolling upper limit ● The number of statements after unrolling 512 ○ The number of iteration ○ No upper limit

Loop optimization

□ Loop invariants optimization

maximum nested loop number ("A" for no limit) 1

☑ Loop merge on automatic scheduling

Function

Converting functions ○ Function call ○ Default ● Inline

Many <---- Cycle count ----> Few

Small <---- Circuit area ----> Large

□ Initialize uninitialized local variables with 0

```
-c1000 -s -ZZpipeline -mrewe -u0 -EE -lb C:/Cyber/5.10/lib/asic_130.BLIB -lfl C:/Cyber/5.10/lib/asic_130.FLIB -lfc ave8-auto.FCNT
```

"Converting functions" section

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:

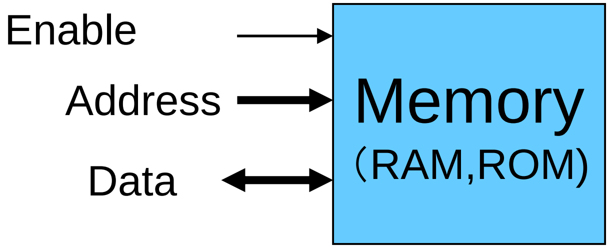
```
/* Cyber func = inline */
var(7..0) funcA( var(7..0) data_a, var(7..0) data_b ){
   …
   …
}
```

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



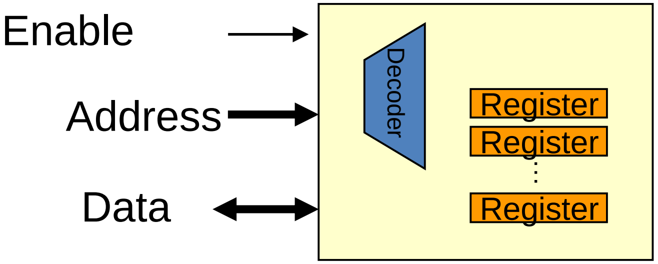
Enable

Address

Data

Memory

(RAM,ROM)



Enable

Address

Data

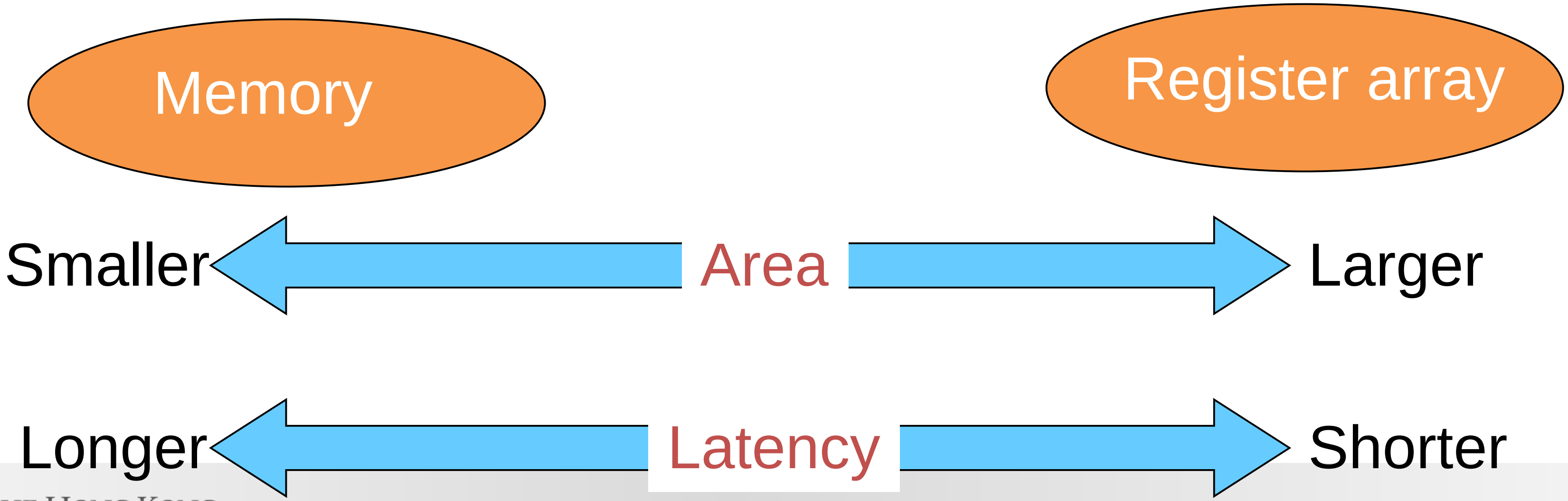
Decoder

Register

Register

Register

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



Memory

Register array

Smaller

Area

Larger

Longer

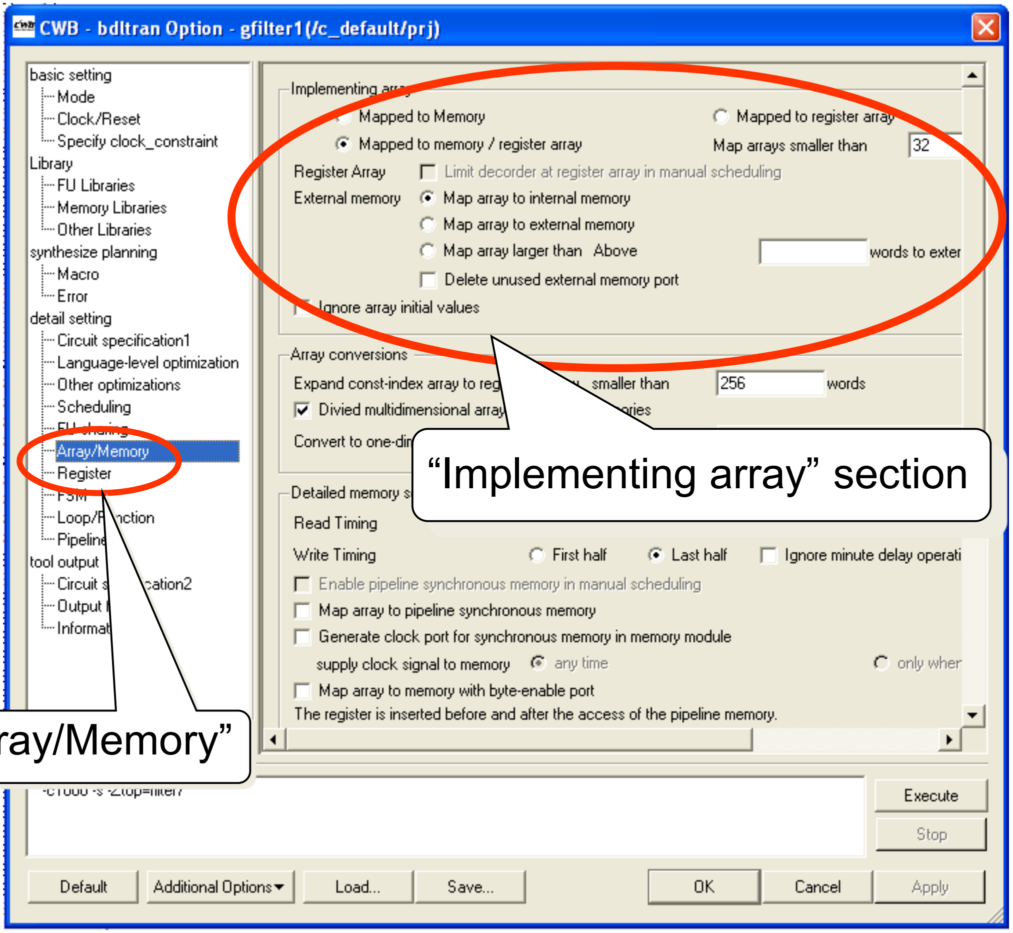
Latency

Shorter

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



Implementing arr

○ Mapped to Memory ○ Mapped to register array

● Mapped to memory / register array Map arrays smaller than 32

Register Array □ Limit decorder at register array in manual scheduling

External memory ● Map array to internal memory

○ Map array to external memory

○ Map array larger than Above words to exter

□ Delete unused external memory port

□ Ignore array initial values

Array conversions

Expand const-index array to reg smaller than 256 words

☑ Divied multidimensional array ories

Convert to one-dir

Detailed memory s

Read Timing

Write Timing ○ First half ● Last half □ Ignore minute delay operati

□ Enable pipeline synchronous memory in manual scheduling

□ Map array to pipeline synchronous memory

□ Generate clock port for synchronous memory in memory module

supply clock signal to memory ● any time ○ only wher

□ Map array to memory with byte-enable port

The register is inserted before and after the access of the pipeline memory.

-c1000 -s -Ztop=filter7

"Implementing array" section

"Array/Memory"

Synthesis Attribute for Array Implementation

```
var(7..0)  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** */

C for Hardware

- ANSI-C does not have custom data types. How to specify 12 or 17 bits?
- How to specify I/Os? (entity)

  ➔Need HW extensions
- CWB calls it BDL (Behavioral Description Language)
- Same as ANSI-C but need:
  - Rename synthesizable function as “process”
  - Declare Inputs and outputs

ANSI-C SW Example

```
int in0;
int out0;
int fifo[8] = {0, 0, 0, 0, 0, 0, 0, 0};
 main(){
int  out0_v, sum,  i;
  for (i = 7; i > 0; i--) {
            fifo[i] = fifo[i- 1];
}
    fifo[0] = in0;
    sum= fifo[0];
      for (i= 1; i< 8; i++) {
            sum += fifo[i];
        }
        out0_v= sum / 8;
        out0 = out0_v;}
```

What does this program do?

C for SW vs. HW

```
int in0;
int out0;
int fifo[8] = {0, 0, 0, 0, 0, 0, 0, 0};
 main(){
int  out0_v, sum,  i;
  for (i = 7; i > 0; i--) {
          fifo[i] = fifo[i- 1];
}
    fifo[0] = in0;
    sum= fifo[0];
      for (i= 1; i< 8; i++) {
          sum += fifo[i];
       }
      out0_v= sum / 8;
      out0 = out0_v;}
```

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

Combining C for SW vs. HW HLS in 1 file

```
#ifdef C
    int in0;
    int out0;
     int fifo[8] = {0, 0, 0, 0, 0, 0, 0, 0};
main(){
#else
    in  ter(0:8)  in0;
    out ter(0:8)  out0;
    var(0:8)  fifo[8] = {0, 0, 0, 0, 0, 0, 0,
0};
process ave(){
#endif
```

```
int  out0_v, sum,  i;
for (i = 7; i > 0; i--) {
         fifo[i] = fifo[i- 1];}
fifo[0] = in0;
   sum= fifo[0];
    for (i= 1; i< 8; i++) {
         sum += fifo[i];
      }
      out0_v= sum / 8;
      out0 = out0_v;
}
```

When compile or synthesize:

```
%g++ -DC ave8.c –o ave8.exe
```

Synthesis directives (pragmas) Example

```
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 ave(){
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

HLS Example - Scheduling

- How many number of FUs are needed to fully unroll the loops ?

```
@FCNT{
            NAME      add8u
            LIMIT     4
#           COMMENT
}
@FCNT{
            NAME      add12u
            LIMIT     3
#           COMMENT
}
```

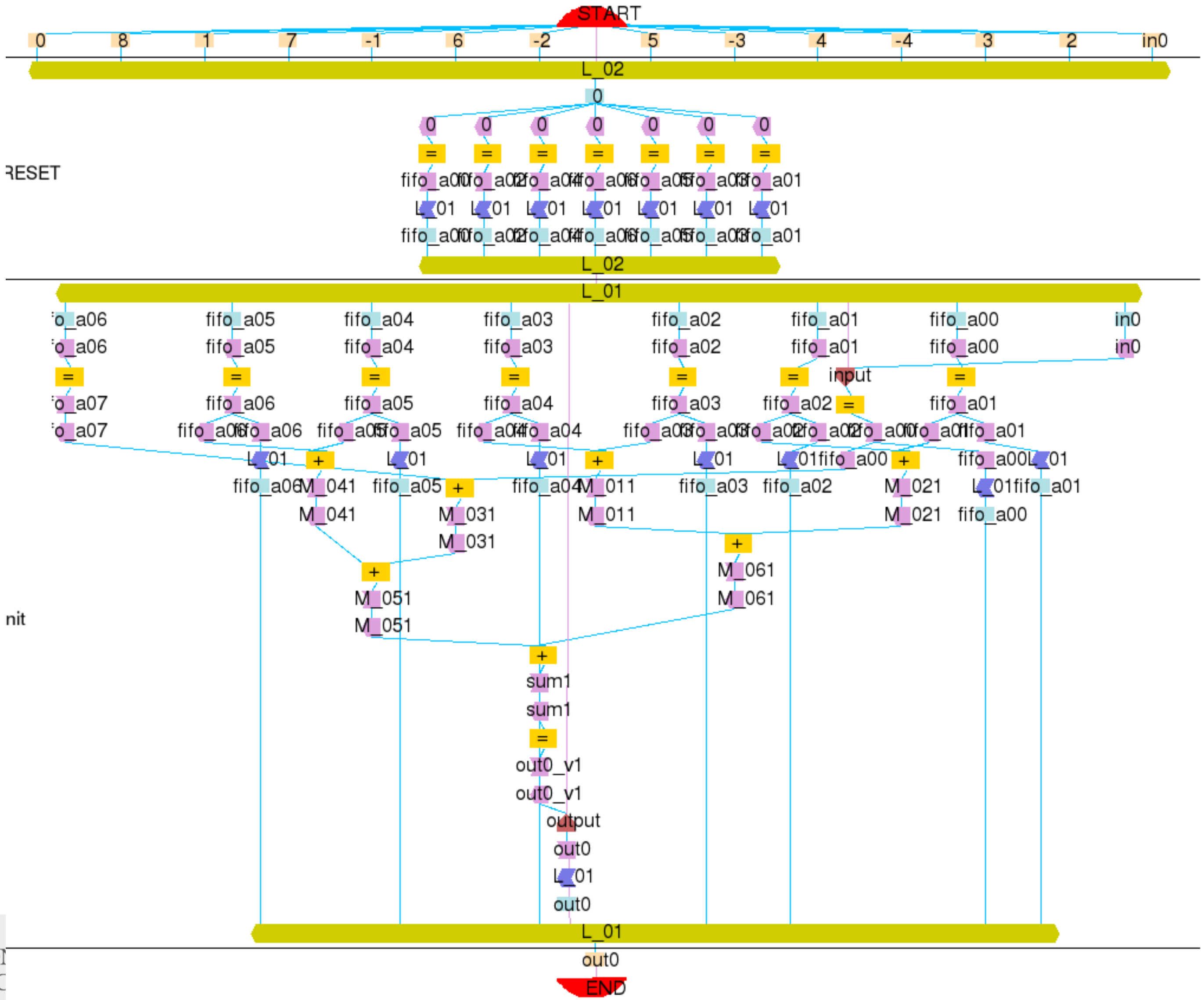
HLS Example - Scheduling

- Schedule the code manually given:
  - the following constraint and delay files
  - Target frequency of 100 MHz (delay = 10ns) 1000 x 1/100ns [unit]

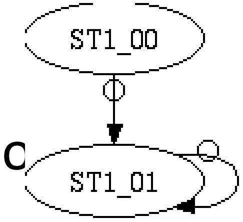
```
@FCNT{
          NAME     add8u
          LIMIT    4
}
@FCNT{
          NAME     add12u
          LIMIT    3
}
```

```
DELAY_UNIT 1/100ns
@FLIB{
          NAME     add8u
          DELAY    52
          AREA     312
}
@FCNT{
          NAME     add12u
          DELAY     61
          AREA      507
}
```

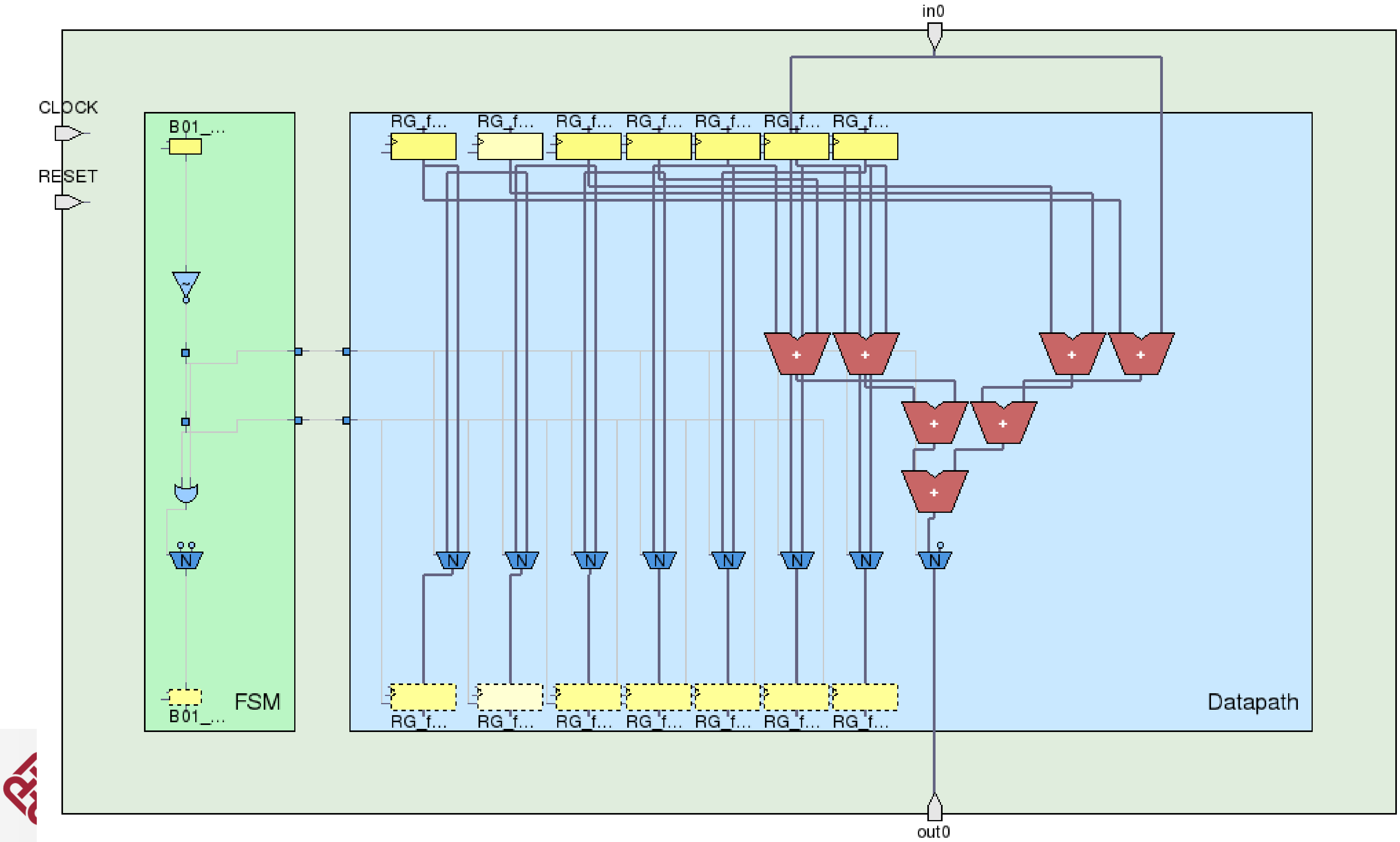
HLS - Scheduling



HLS Example- Binding

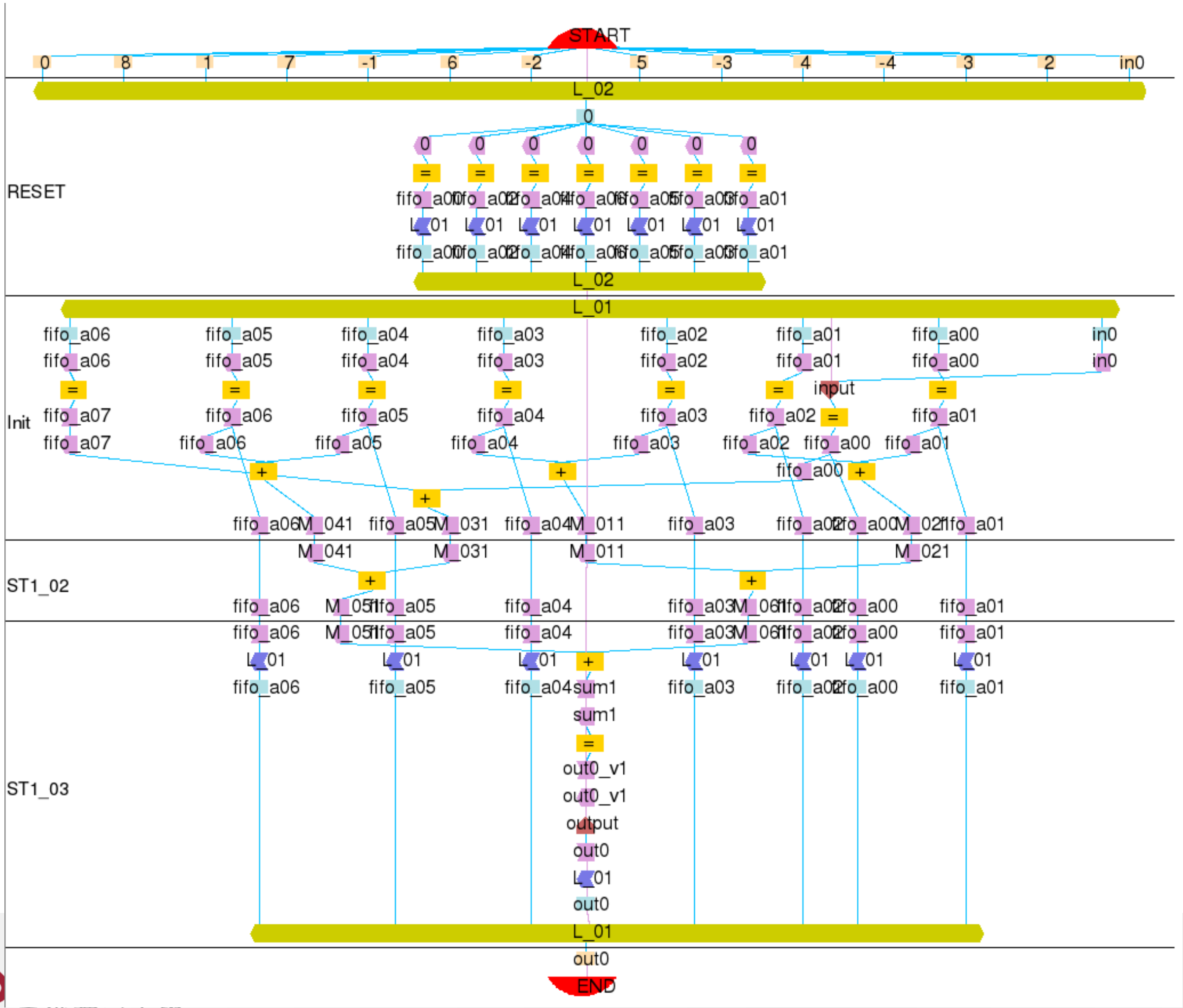


- Bind the scheduled CDFG and report the total number of states c the FSM controller

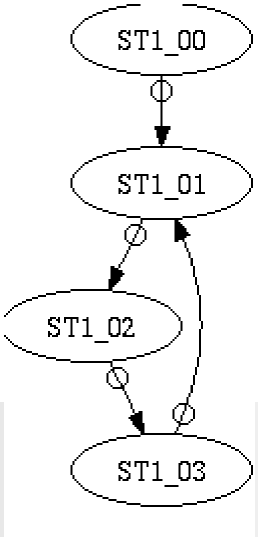


HLS - Example

- What would happen if the target frequency increased to 1 GHz (1ns) given the same constraints ?

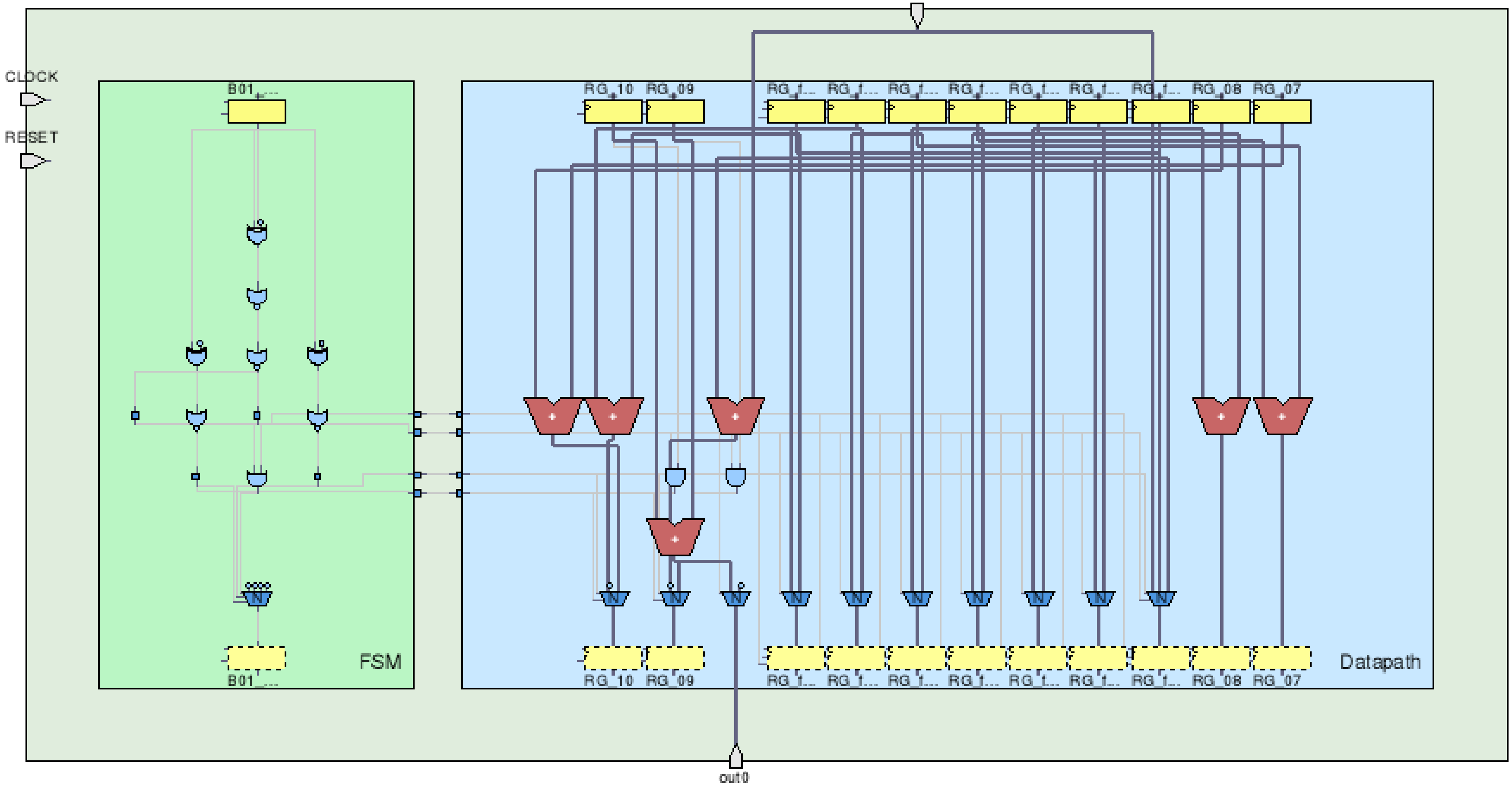


- FUs operations cannot be scheduled together in 1 clock cycle ➔ Need 3 cycles (states to execute)

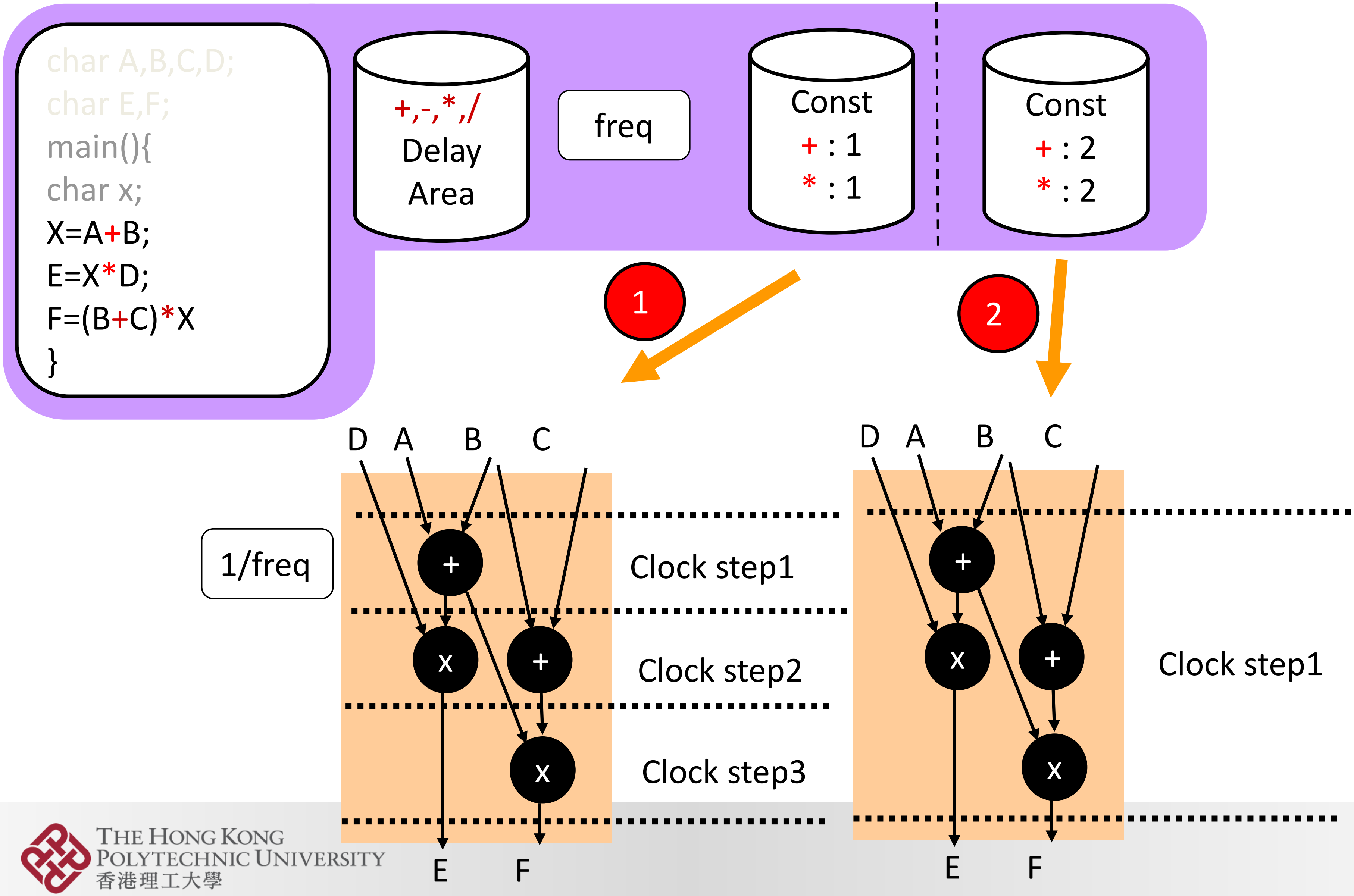


HLS - Binding

- FSM needed to steer data through 3 states
- Less FUs needed because cannot execute all operations on the same stage



Benefits of HLS: (1) Automatic Alternative Architecture Generation



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

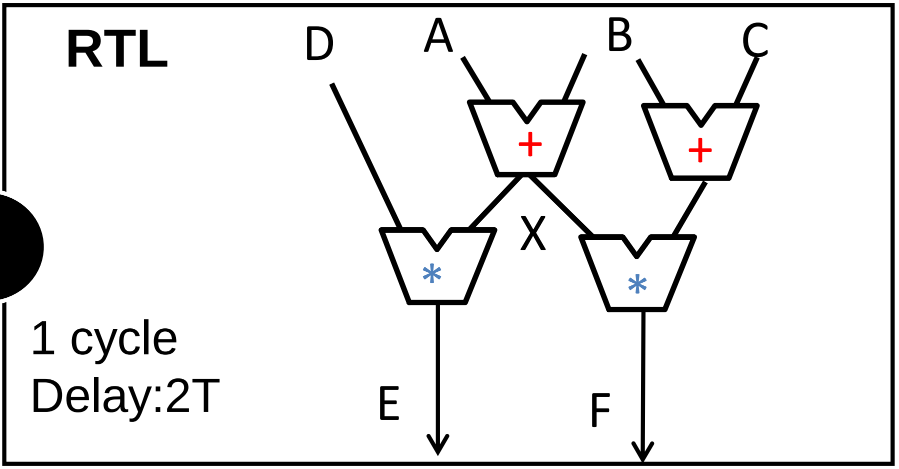
Benefits of HLS: (1) Automatic Alternative Architecture Generation

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

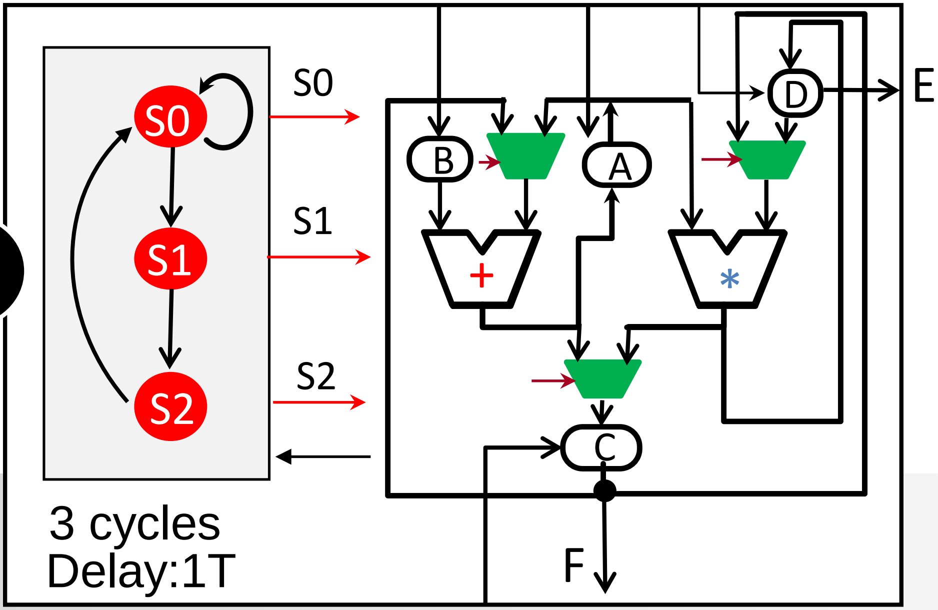
1
+ : 2
* : 2



RTL

1 cycle
Delay:2T

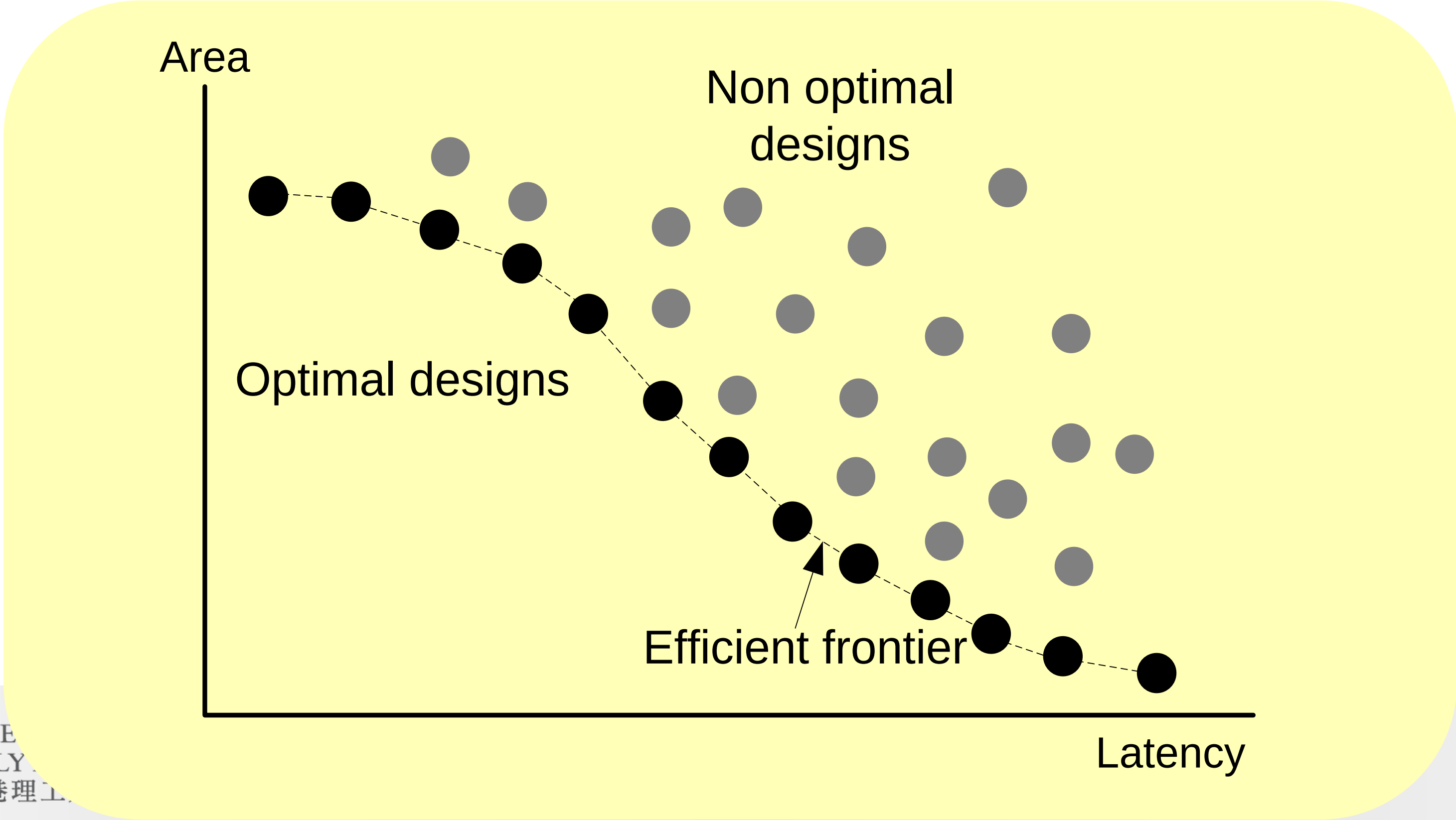
2
+ : 1
* : 1



3 cycles
Delay:1T

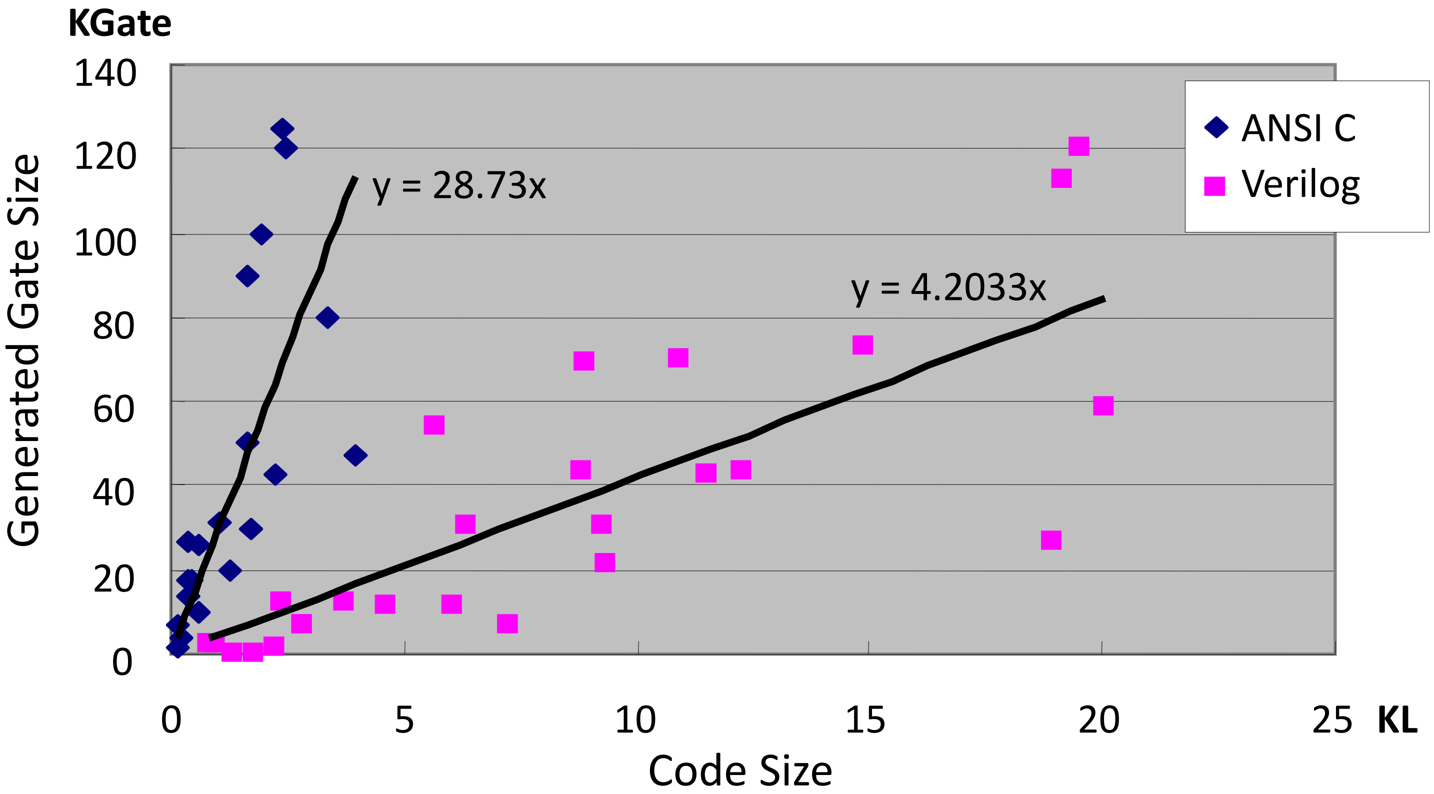
Benefits of HLS: (2) Architectural Design Space Exploration

- Design Space
  - Set of all feasible designs
- Objectives
  - Performance (latency, throughput , max frequency)
  - Area
  - Power



Benefits of HLS: (3) Increase Productivity

- Less details needed ➔ Faster to design and verify
- Less bugs
- Easier to maintain source code
- Easier to read



RTL (Verilog)	4 gates / line
Behavioral C	28 gates / line

➔ **7x more productive**

Commercial HLS Tool

- Cadence – CtoSilicon (C,C++, SystemC)
- Forte – Cynthesizer (SystemC) → Hast just been acquired by Cadence (Feb 2014)
- Mentor Graphics (Calypto) – CatapultC (C++, SystemC)
- NEC – CyberWorkBench (C, SystemC)
- Synopsys – Synphony (C, SystemC)
- Xilinx – VivadoHLS (C,C++, SystemC)

HLS Adoption Problems

- Input language
  - ANSI C? (subsets), SystemC?
- New design methodology. Needs time to be adopted ➔ Still not being taught at each school
- Current RTL designers need to adopt it, but they don't 'trust' the tool
- QoR compared to hand-coded RTL

Input Language - ANSI-C

Pros:

- Large user base
- Lots of legacy code
- Easy to learn

- Cons:
  - Does not have HW specific constructs. E.g. custom bitwidth data types, parallel statements ➔ each vendor developed its own subset
  - ANSI-C subsets bind users to the EDA vendor. High switching cost
  - ANSI-C subsets cannot be compiled with ‘gcc’ (need a dedicated compiler)

Input Language - SystemC

- Open Source C++ class library for HW description
- IEEE standard (IEEE 1666)
- Freely available at www.accellera.org
- Simulation is fast
- Can develop and co-simulate HW and SW
- Promote interoperability between tools
- Detail SystemC information available at: http://videos.accellera.org/tlm20andsubset/index.html

Why was SystemC needed?

- In ANSI C/C++ you can not:
  - **Express Concurrency**: HW system operate by nature in parallel
  - **Custom Data types**: e.g. specify any bitwidth, fixed-point data types,
  - **Model communications**: Signals, protocols
  - Notion of time: Clock cycles

Synthesizable SystemC Subset

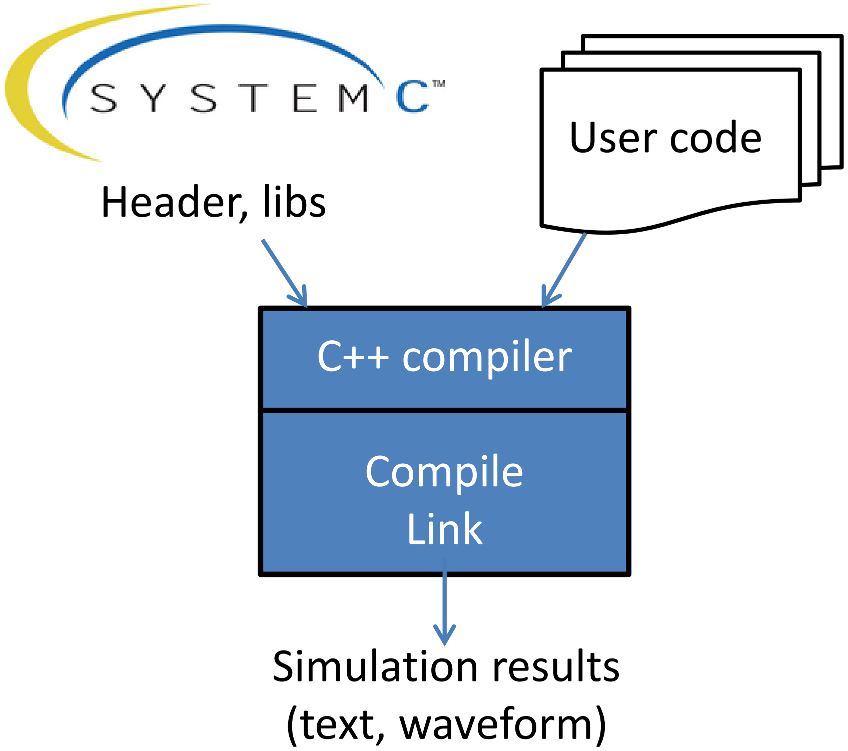
- SystemC parts that can be synthesized (converted to RTL)
- Unified subset that all HLS vendors accept (users can change between tool vendors and SystemC code will still be synthesized into RTL)
- Synthesizable subset draft 1.3 available online
- DARClab has released a synthesizable SystemC Benchmark suite called S2CBench (www.s2cbench.org)

SystemC Example – Half Adder

```
#include “systemc.h”
SC_MODULE(half_adder) {
  sc_in<bool>a, b;
  sc_out<bool>sum, carry;

  void proc_half_adder();
  SC_CTOR(half_adder) {
    SC_METHOD (proc_half_adder);
    sensitive << a << b;
  }
};

void half_adder::proc_half_adder()
   {
  sum = a ^ b;
  carry = a & b;
}
```



Compilation example with g++:

g++ -o [output name] [file name] –I/systemc_path/include –L/systemc_path/lib –lsystemc

Conclusions

- High Level Synthesis
- Benefits
- Main steps:
  - Allocation
  - Scheduling
  - Binding
- HLS in practice
- HLS optimizations
  - Global synthesis options vs. local attributes
- HLS input language
- SystemC