Schedule

- 1. Introduction
- 2. System Modeling Language: System C *
- 3. HW/SW Cosimulation *
- 4. C-based Design *
- 5. Data-flow Model and SW Synthesis
- 6. HW and Interface Synthesis
  (Midterm)
- 7. Models of Computation
- 8. Model based Design of Embedded SW
- 9. Design Space Exploration
  (Final Exam)
  (Term Project)

codesign environment

References

- **SDF**
  - [13] E.A.Lee and D.G.Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” IEEE Transactions on Computers, January, 1987.
  - E.A.Lee and D.G.Messerschmitt, “Synchronous Data Flow”, IEEE Proceedings, September, 1987.
- **Ptolemy Classic Approach**
  - [14] José Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck, “Software Synthesis for DSP Using Ptolemy,”Journal on VLSI Signal Processing, vol. 9, no. 1, pp. 7-21, January, 1995.
- **CSDF**
  - [18] G. Bilsen, et. al., "Cyclo-Static Dataflow," IEEE TSP, 44(2), February 1996.

- **SPDF**
  - [15] Chanik Park, et. al., "Extended Synchronous Dataflow for Efficient DSP System Prototyping," Design Automation for Embedded Systems, Kluwer Academic Publishers, pp.295-322,March 2002.
- **FRDF**
  - [16] Hyunok Oh and Soonhoi Ha, "Fractional rate dataflow model for efficient code synthesis", Journal of Vlsi Signal Processing Systems for Signal Image and Video Techno Vol. 37 pp 41-51 May. 2004
- **Software Synthesis**
  - [17] Hyunok Oh and Soonhoi Ha, "Memory-optimized Software Synthesis from Dataflow Program Graphs with Large Size Data Samples", EURASIP Journal on Applied Signal Processing Vol. 2003 pp 514-529 May 2003.

codesign environment

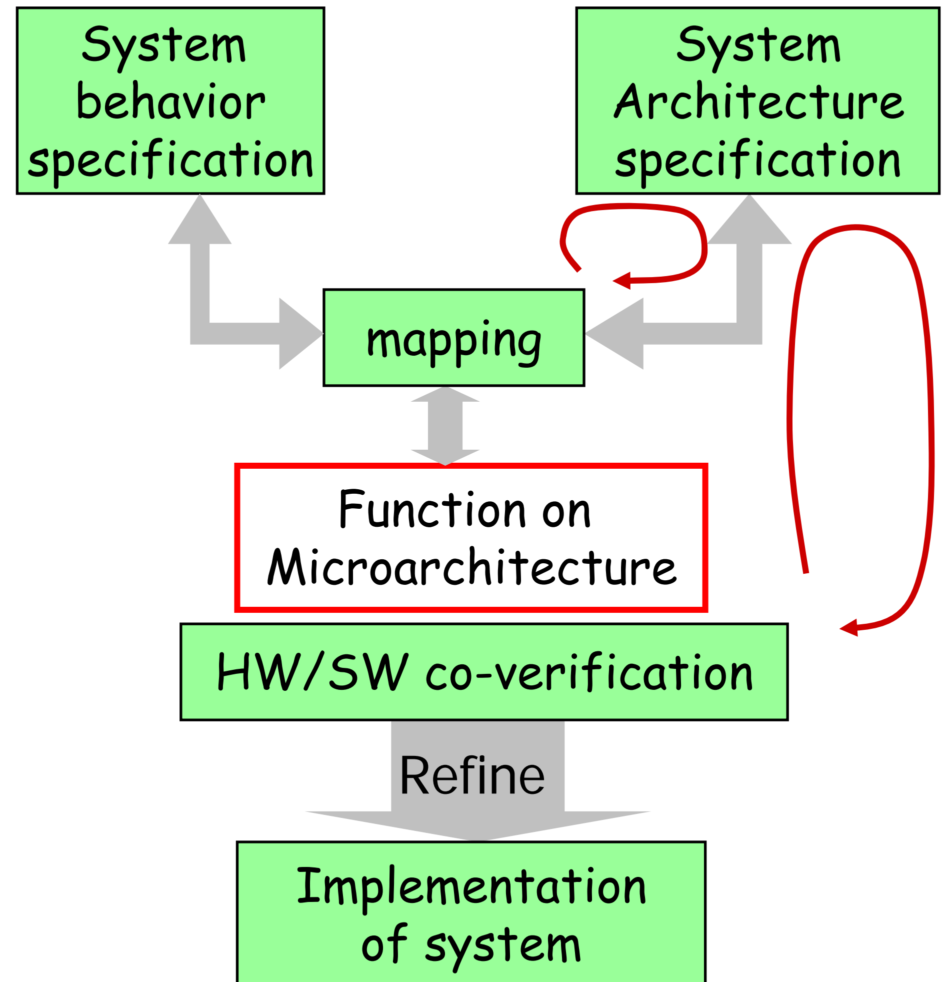
Outline

- Introduction: Model based design
- SDF (Synchronous Data Flow) Model
- Software Synthesis from SDF Model
- Translation of Reference C code to SDF Model
- SDF Extensions

codesign environment

Reminder: Y-Chart

- **Separate specification**
  - System behavior
  - System architecture
- **Mapping**
  - HW/SW partitioning
- **HW/SW co-verification**
- **HW/SW co-synthesis**



codesign environment

System Behavior Specification

- **Wish list**
  - User friendliness
  - Executable / simulatable
  - Implementation Independence
  - Design validation / error detection capability
  - Design Maintenance and collaboration
  - Well defined path to synthesis
- **Two approaches**
  - Language-based approach
  - Model-based approach

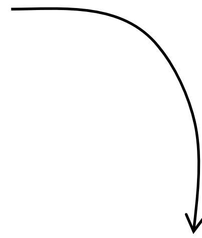
codesign environment

Language-based specification

- **Use popular programming language**
  - SW: C, C++, Java
    - Imperative Language: not easy to exploit concurrency
  - HW: HDL (VHDL, Verilog)
- **Simulation based verification**
  - Not easy to detect semantic error
- **Not easy to debug and maintain the code**
- **Not easy to grasp the global view of the application**
  - Code reuse is difficult
  - Not easy to cooperate with others
- **Useful only for simple design**
  - Well-known algorithm on a single core architecture

Model-based specification

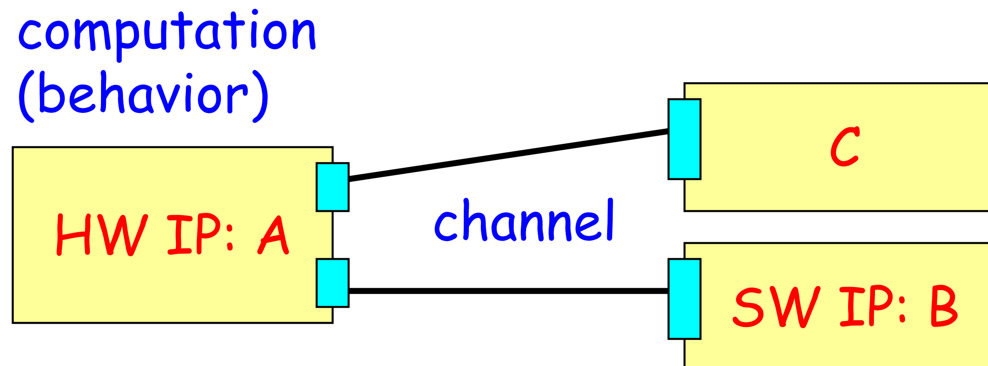
- **State-oriented models**
  - Finite-State Machines (FSM), statechart, Esterel
- **Object-oriented model**
  - UML
- **Actor-oriented models**
  - Dataflow model
  - Discrete-event model
  - SystemC model



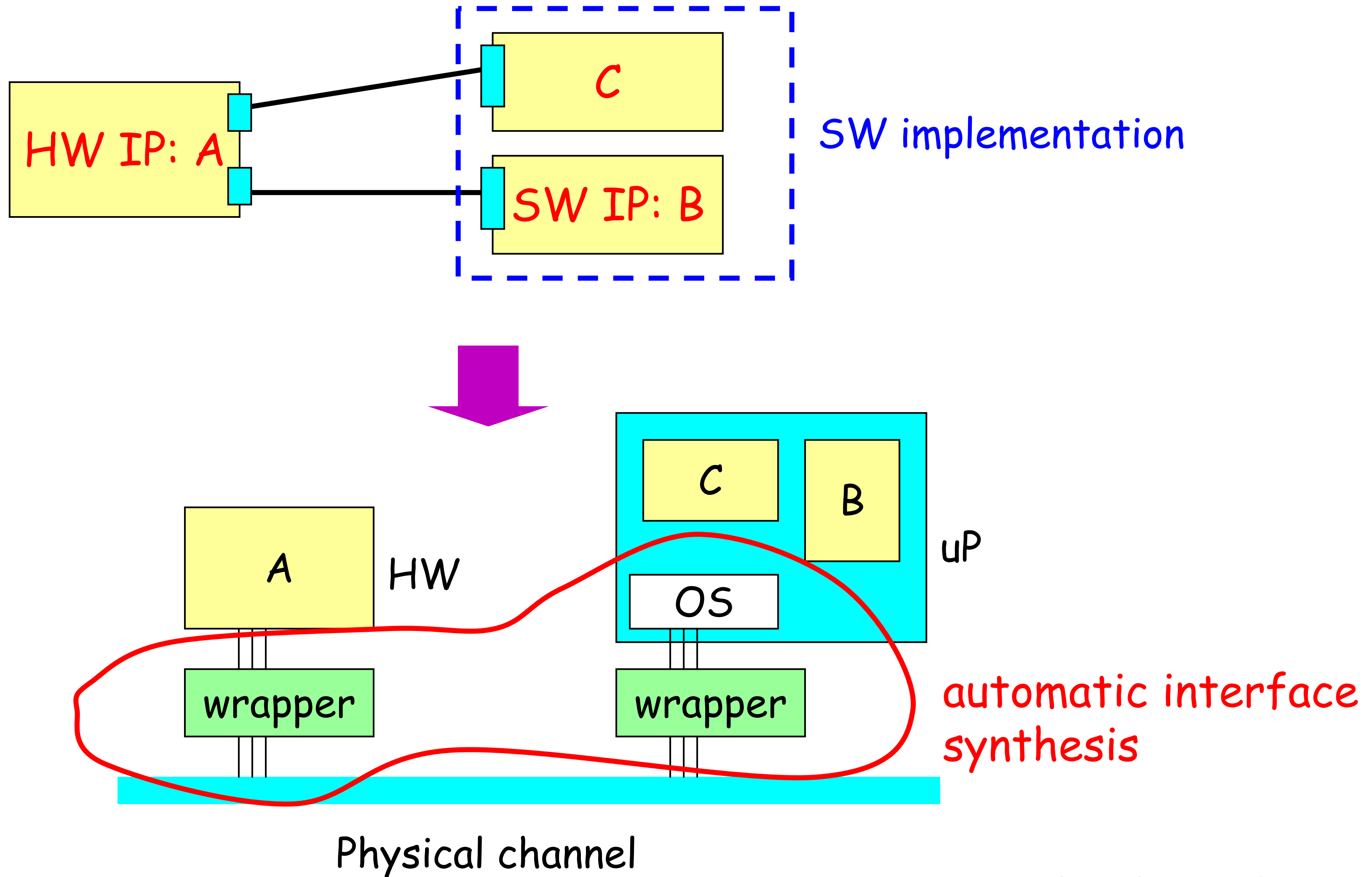
- **Concurrent process models**
  - CSP, Kahn process networks, Dataflow process network
- **Heterogeneous models**
  - Statemate, Ptolemy, Metropolis, PeaCE
- **Others**
  - SIMULINK

Actor-Model

- **Model the system behavior as a composition of active modules (actors)**
  - Express concurrency naturally
- **Separate computation and communication**
- **Implementation independent**
  - Can explore wide design space
- **Design reuse**
- **Easy to understand the system behavior**

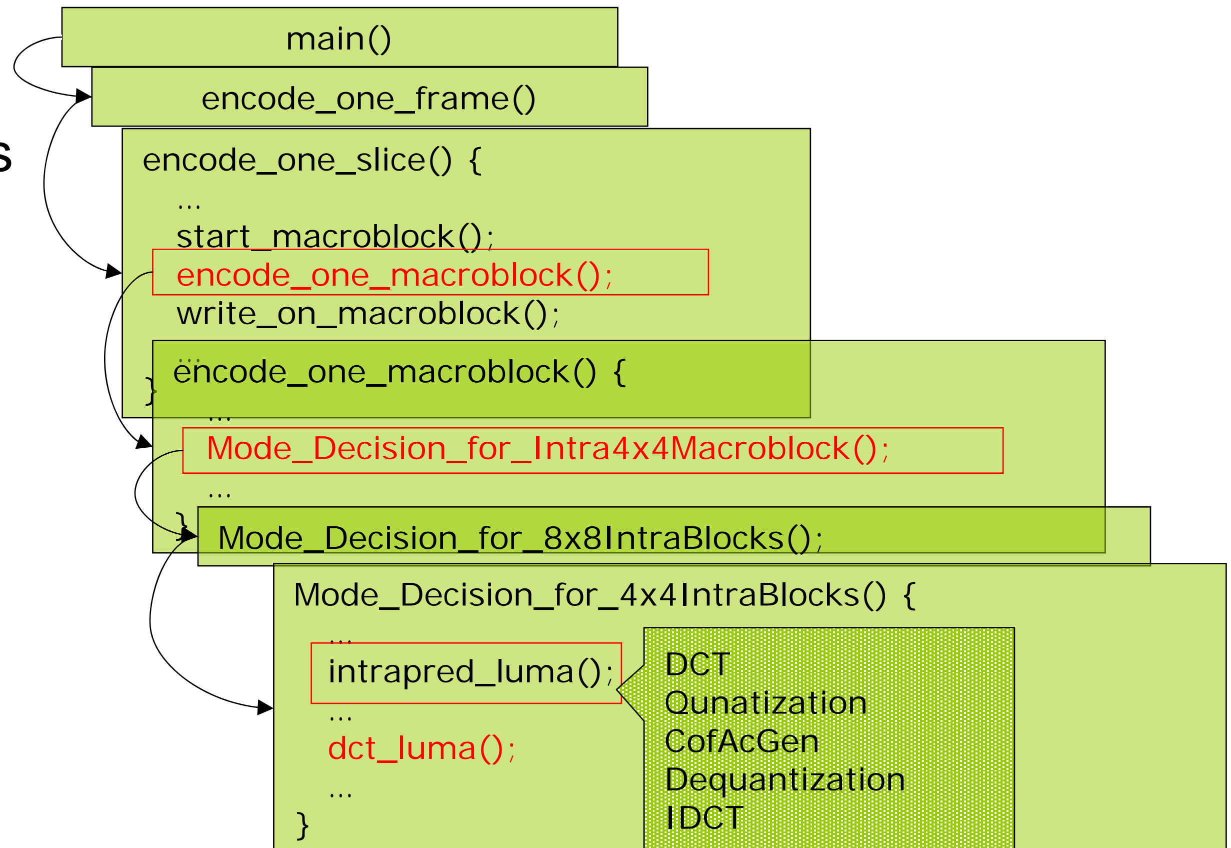


Remind: Model to Implementation



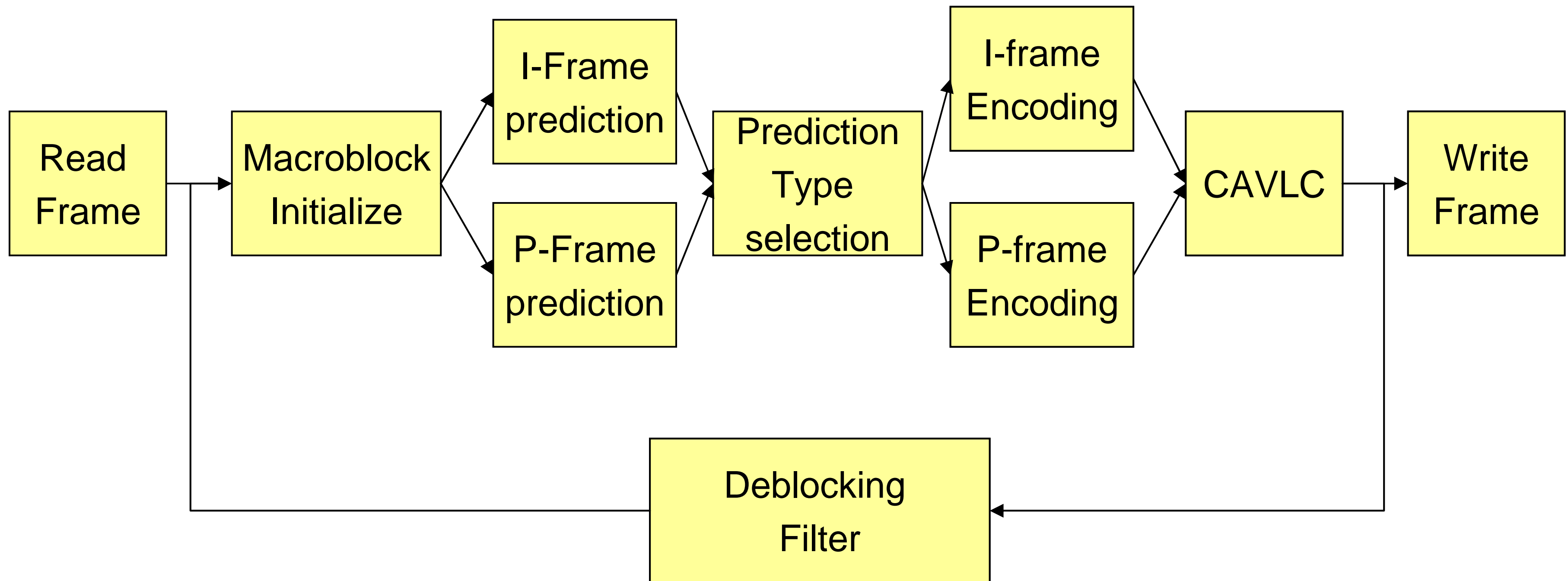
Comparison: H.264 encoder

- **Language-based specification**
  - Deep nested function calls
  - Shared variables



Comparison – cont'd

- **Model-based specification**

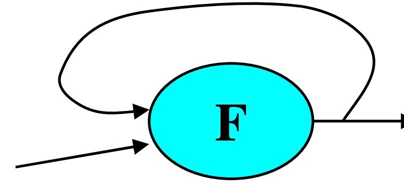


Much easier to understand!

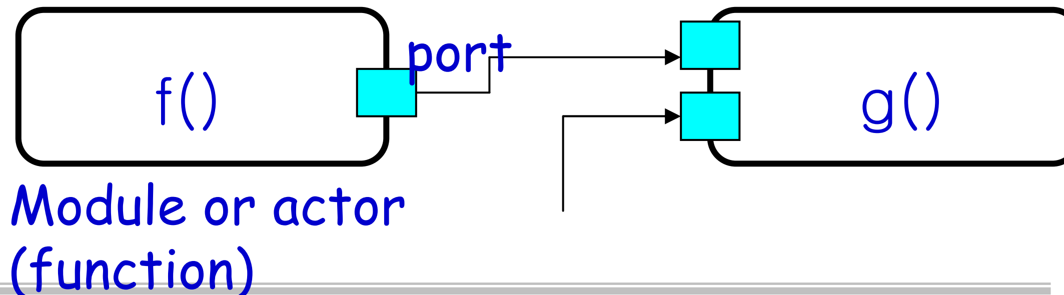
codesign environment

Model of Computation

What does it mean?



- **What is the firing condition of F block?**
  - What is the result?
  - The answers are different depending on the model of computation.
- **Formal models: Precise and unambiguous semantics**
  - Functional specification: f (input, output, state)
  - Well defined function composition
  - Properties and Constraints



codesign environment

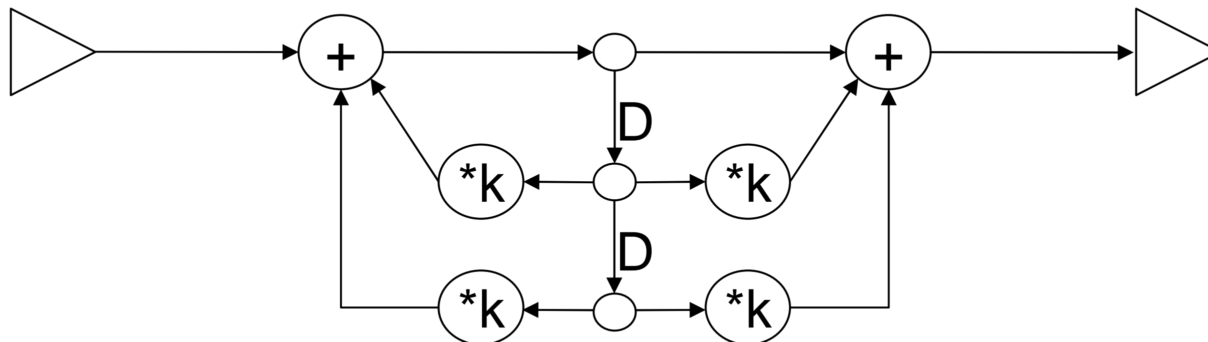
Outline

- Introduction: Model based design
- SDF (Synchronous Data Flow) Model
- Software Synthesis from SDF Model
- Translation of Reference C code to SDF Model
- SDF Extensions

codesign environment

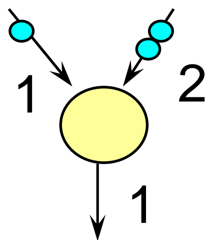
Dataflow Model

- **A directed graph**
  - A node represents computation (or function) and an arc represents data path
- **No shared variable between nodes: no side effect**
  - nodes communicate each other through arcs
- **Data-driven execution**
  - A node is fireable (can start execution) when the required number of data samples exist on the incoming arcs.
  - Express concurrency naturally
- **An example**

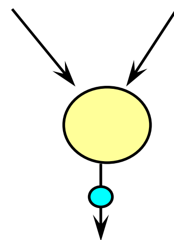


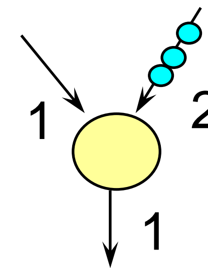
SDF (Synchronous Data Flow)

- **Non-uniform sample rates are allowed**
  - The number of samples to consume or produce on each arc can be greater than 1: we call this number as input or output sample rate.
  - Useful to describe DSP algorithms
- **A dataflow model with strict firing condition**
  - The sample rates are constant at run-time
- **We can determine the static schedule of the graph**
  - Performance and resource requirement can be predicted



enabled


fired

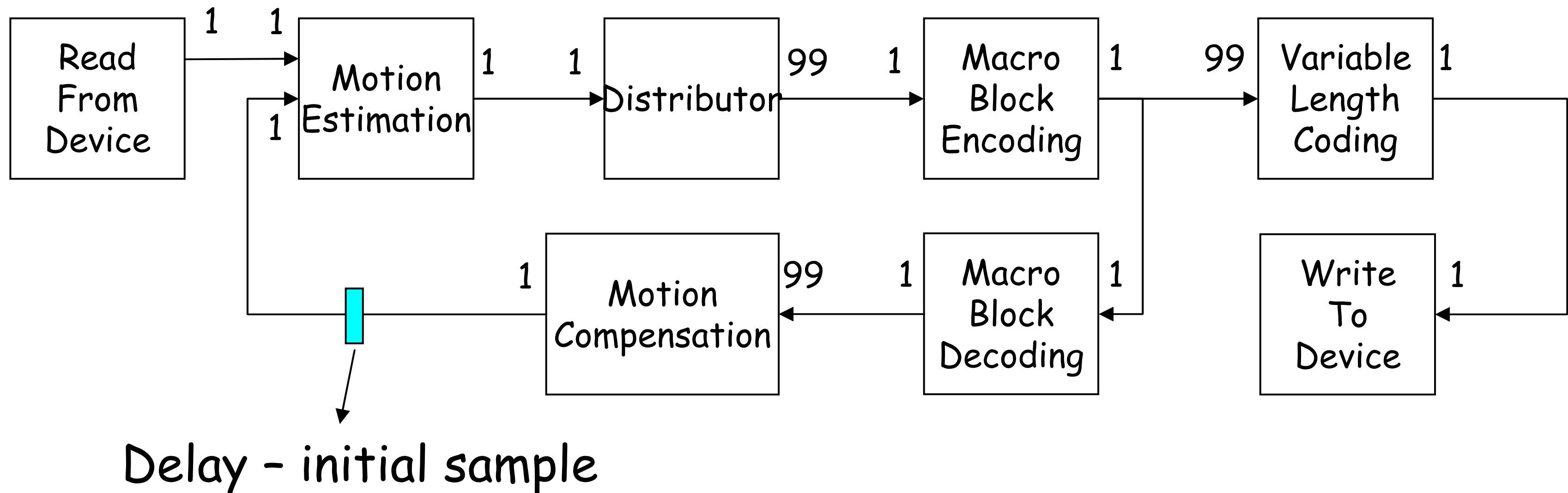


NOT enabled

codesign environment

Example: H.263 Encoder

- **SDF specification**



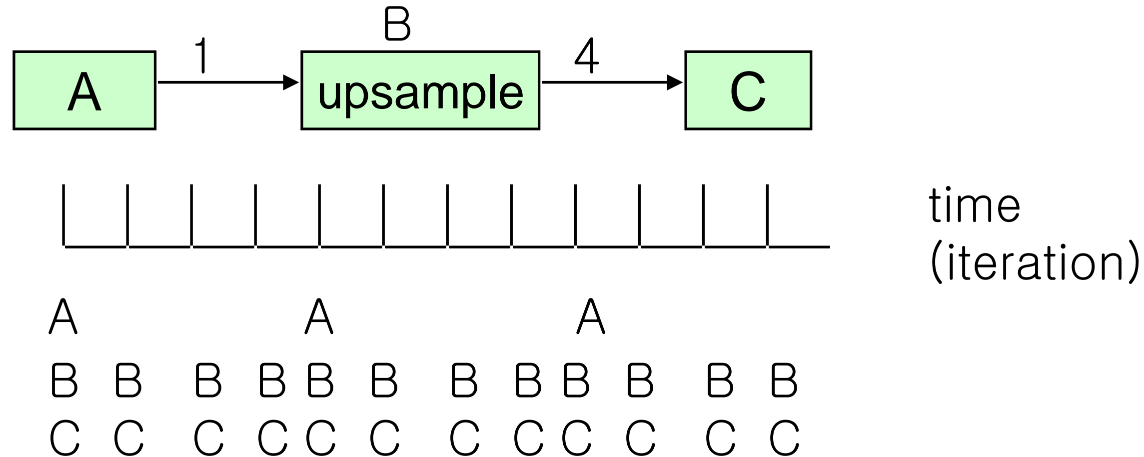
Homogeneous v.s. Synchronous

- **example: (10-point) FFT block**

- homogeneous data-flow: consume 1 sample per arc
  - 10 firings of FFT block to perform 10-point FFT
  - do nothing with first 9 samples inside the FFT block
  - hidden pipelining to produce 1 sample at a time
- synchronous data-flow: consumes 10 samples at once and produce 10 samples at once
  - 1 firing of FFT block
  - natural representation

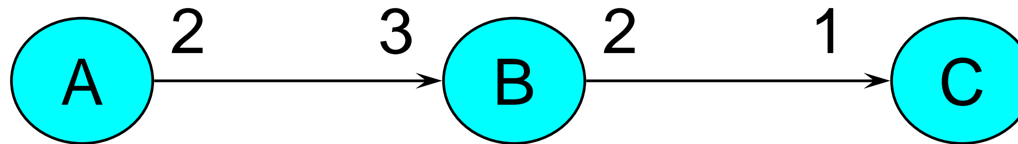
Two Implementations of Multi-rate

- **Multiple clocking (DSP Station)**



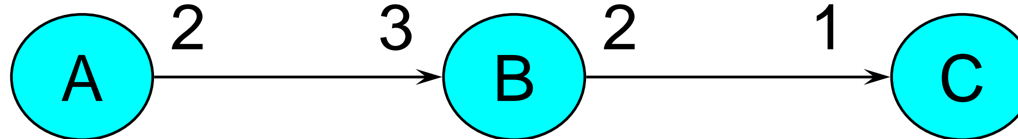
  - dynamic scheduling of A : one firing of B produces 1 sample repeat (A-B-C-skip-B-C-skip-B-C-skip-B-C)
  - buffer size between B and C is 1.
- **Multiple Samples (Ptolemy/PeaCE)**
  - produce 4 samples at once A-B-4(C): buffer size on BC is 4.

Static Scheduling



- **Ratio of node repetition counts should be maintained**
  - Repetition counts of A, B: r(A), r(B)
  - A produces (prod(A) = 2) samples and B consumes (cons(B)=3) samples per execution
  - Balance Equation: r(A) prod(A) = r(B) cons(B)
  - r(A) : r(B) = cons(B) : prod(A) = 3:2
  - Then, r(B) : r(C) = ?
    r(A):r(B):r(C)=?
- **iteration period: the minimum cycle that satisfies the ratio of node repetition counts**

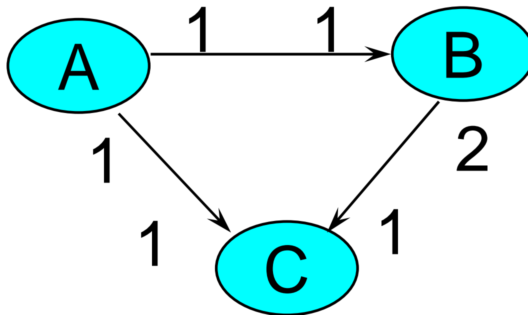
SDF Scheduling

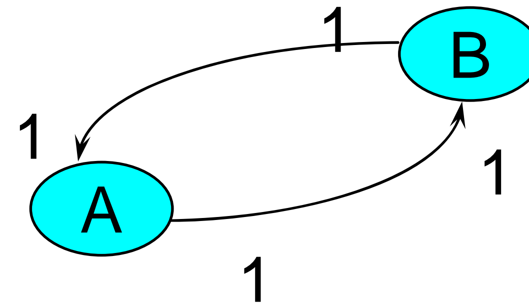


- **repetition counts**
  - A: 3 B: 2 C: 4
- **Valid schedules are not unique because the graph describes the partial order (true dependency) between blocks**
  - (ex) AABCCABCC – schedule for minimum buffer size
  - (3A)(2(B(2C))) – schedule with loop structure
- **What is the best schedule?**
  - Depends on the design objectives
  - (informal) programming describes just one schedule – no quarantee of optimality !!

Syntax check

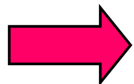
- **Sample rate inconsistency**
  - Some arc accumulates tokens without bound
- **Deadlock**
  - Graph can not start its execution





SDF Limitations

- **Expression Capability**
  - SDF model can not express control structures (dynamic constructs) such as conditional execution and data dependent iteration
- **SDF model does not allow shared memory (global states) between blocks due to side effect.**
  - why? memory update order may vary depending on the schedule.
- **SDF model does not allow pointer operation, but copies structured data as a token.**



Model extension: SPDF (Synchronous Piggybacked Data Flow)

codesign environment

Outline

- Introduction: Model based design
- SDF (Synchronous Data Flow) Model
- Software Synthesis from SDF Model
- Translation of Reference C code to SDF Model
- SDF Extensions

codesign environment

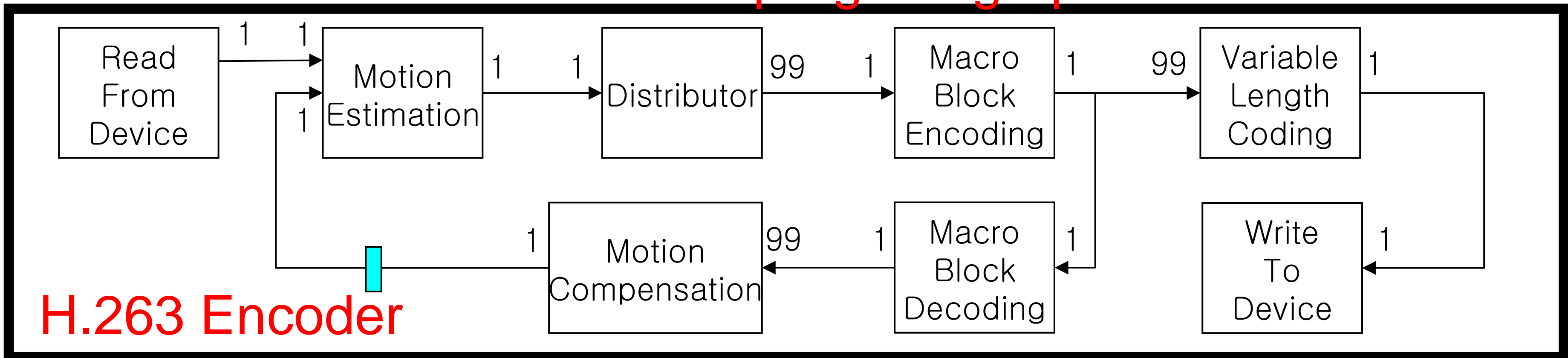
Software Synthesis

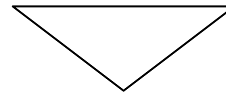
- **New trend of software development : model-based design**
  - growing complexity, fast design turn-around time,
- **Automatic code generation from data flow graph**
  - SDF semantics should be preserved - “refinement”
  - The kernel code of a block is already optimized in the library.
  - Determine the schedule and coding style.
  - Codes are generated according to the scheduled sequence
- **Fundamental Question**
  - Can we achieve the similar code quality as manually optimized code in terms of performance and memory requirement?

Code Synthesis from Dataflow Graph

codesign environment

Dataflow program graph

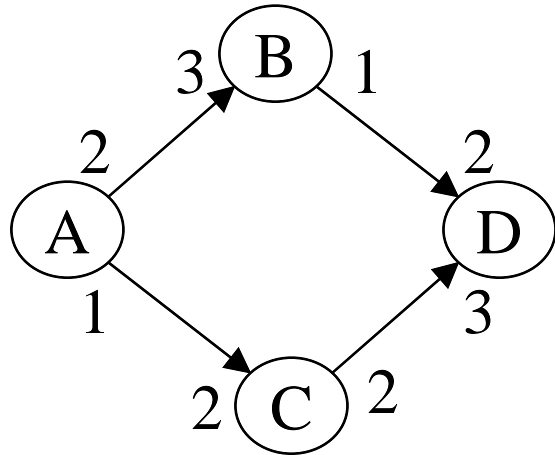




Synthesized code

```
main() {
/* local buffers for msg type */
void * Y_0[1] = {MsgGlobalBuffer+25344};
void * U_1[2] = {MsgGlobalBuffer+88704,MsgGlobalBuffer+76032};
void * V_2[2] = {MsgGlobalBuffer+95040,MsgGlobalBuffer+82368};
…
```

Software Synthesis Example



Possible Schedules :
= AABCABACDABABCD
= (6A)(4B)(3C)(2D)
= (2(3A2B))(3C)(2D)

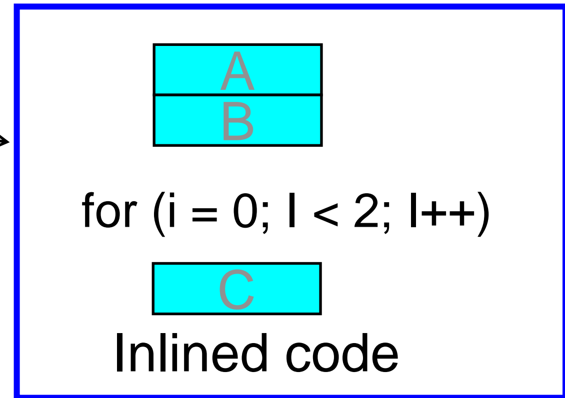
Single Appearance Schedule (SAS)

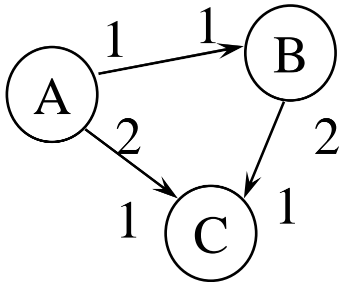
```
main(){
        for(i=0;i<6;i++){A}
        for(i=0;i<4;i++){B}
        for(i=0;i<3;i++){C}
        for(i=0;i<2;i++){D}
}
```

```
main(){
        for(i=0;i<2;i++){
                for(j=0;j<3;j++){A}
                for(j=0;j<2;j++){B}
        }
        for(i=0;i<3;i++){C}
        for(i=0;i<2;i++){D}
}
```

Coding Style

- **Inlined code**
- **Function calls**
- **Switch-case**



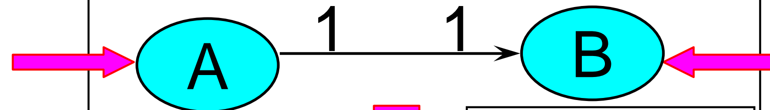



```
for() {
  switch(i) {
    case 1: A; break;
    case 2: B; break;
    case 3-4: C; break;
  }
  I++;
}
```

```
A(); B();
for (i = 0; I < 2; I++) {
  C();
}
A() { … } /* function body */
B() { … }
C() { … }
```

Example

```
defstar {
  name { TestSrc }
  domain { cgc }
  version { @(#) CGCTestSrc.pl 1.0 }
  author { Taewook Oh }
  location { CGC library }

  output { name { out } type { int } }
  defstate {
    name { mystate}
    type { int }
   default { 3 }
   desc { amumalena ssuseyum }
  }

  ccinclude { "Message.h" }

  initCode {
    addInclude("<stdio.h>");
    addGlobal(defstarsymbols);
  }

  go { addCode(block); }

  codeblock(defstarsymbols) {
    int $starsymbol(value);
  }

  codeblock(block) {
    $starsymbol(value)++;
    $ref(out) = $starsymbol(value)*$val(mystate);
    $ref(mystate)++;
  }
}
```



Run Count is 10 (sLC_3 loops 10 times)

```
#include <stdio.h>

int value_1;

int main(int argc, char *argv[]) {
  int mystate_2;
  int out_0;

  mainInit:
  mystate_2 = 3;
  out_0 = 0;
  { int sLC_3;
    for (sLC_3 = 0;sLC_3 < 10; sLC_3++) {
    { /*TestSrcl0 (class CGCTestSrc) */
      value_1++;
      out_0 = value_1 * 3;
      mystate_2++;
    }
    { /*TestDst2 (class CGCTestDst) */
      printf("%d\n", out_0);
    }
    }
  }

  return 1;
}
```

```
defstar {
  name { TestDst }
  domain { cgc }
  version { @(#) CGCTestDst.pl 1.0 }
  author { Taewook Oh }
  location { CGC library }

  output { name { in } type { int } }

  ccinclude { "Message.h" }

  initCode {
    addInclude("<stdio.h>");
  }

  go { addCode(block); }

  codeblock(block) {
    printf("%d\n", $ref(in));
  }
}
```

Inside PeaCE Code Generation

codesign environment

- **How to define code sections?**
- **Block definition**

/* user: sha Date: Mon, Oct. 20 1997 Universe: fm */

```
#include <math.h>                              include
int value_0;                                   globalDecls
void vset() { value_0 = 1; }                   procedures
int main() {
int output_0;  int file_2;                     mainDecls
output_0 = 0; file_2 = open(…);                mainInit
{ int k; for (k = 0; k < 100; k++) {
        { /* block code */ …. }                mainLoop
}
close(file_2); return 1;                       mainClose
}
```

codesign environment

Code Streams of CGCTarget

```
/* user: sha   Date: Mon, Oct. 20 1997    Universe: fm */        headerComment();
#include <math.h>                include
int value_0;                     globalDecls
void vset() { value_0 = 1; }     procedures
int main() {             // (const char*) funcName;
int output_0;  int file_2;       mainDecls
                                                        commInit
output_0 = 0; file_2 = open(…);  mainInit
{ int k; for (k = 0; k < 100; k++) {      // beginIteration()     wormInit
    { /* star code */ …. }       mainLoop          // CGCTarget::mainLoopCode()
}                                         // endIteration()       wormOut
close(file_2); return 1;         mainClose
}
```

CGCTarget::frameCode()

Block Code Generation

```
CGFoo.pl
…
codeblock(test) {
// this is a test
}
go {
    addCode(test);
}
```

ptlang →

```
CGFoo.h:
  class CGFoo {… static CodeBlock test; … }
CGFoo.cc:
  CodeBlock CGFoo::test("//this is a test");
  void CGFoo:: go() { addCode(test); }
```

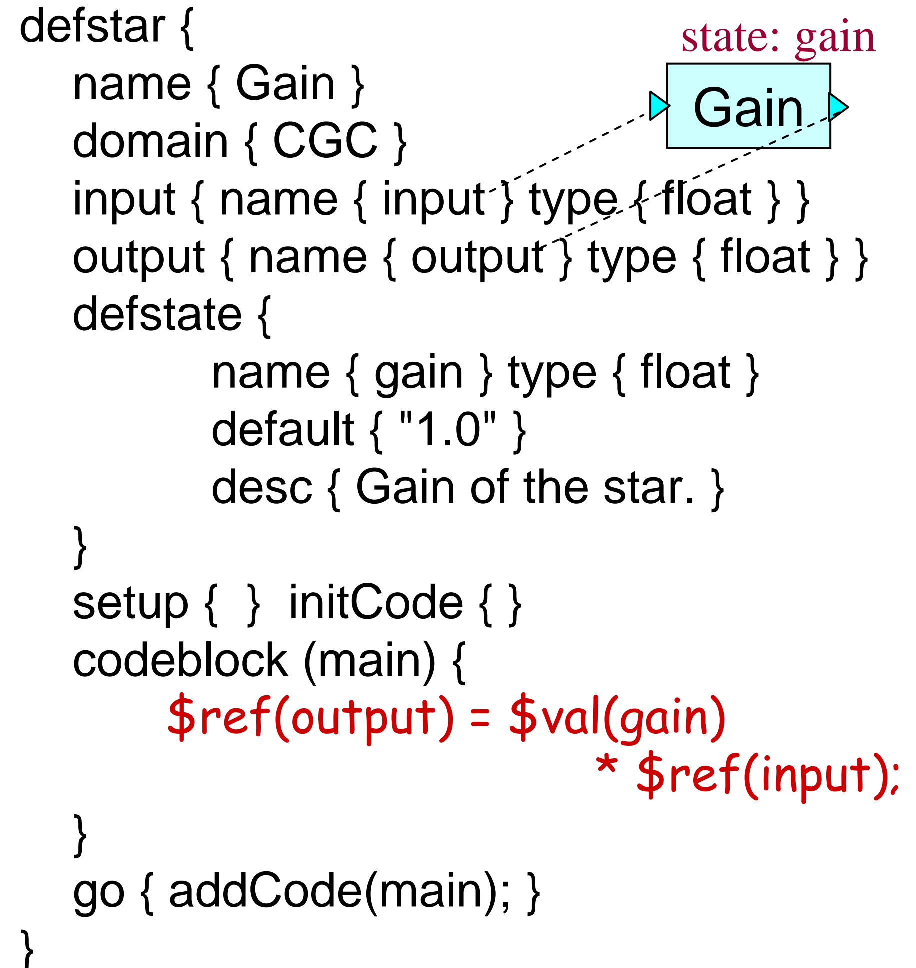
Put the "test" to myCode code stream of CGCTarget

Code insertion to the CodeStreams

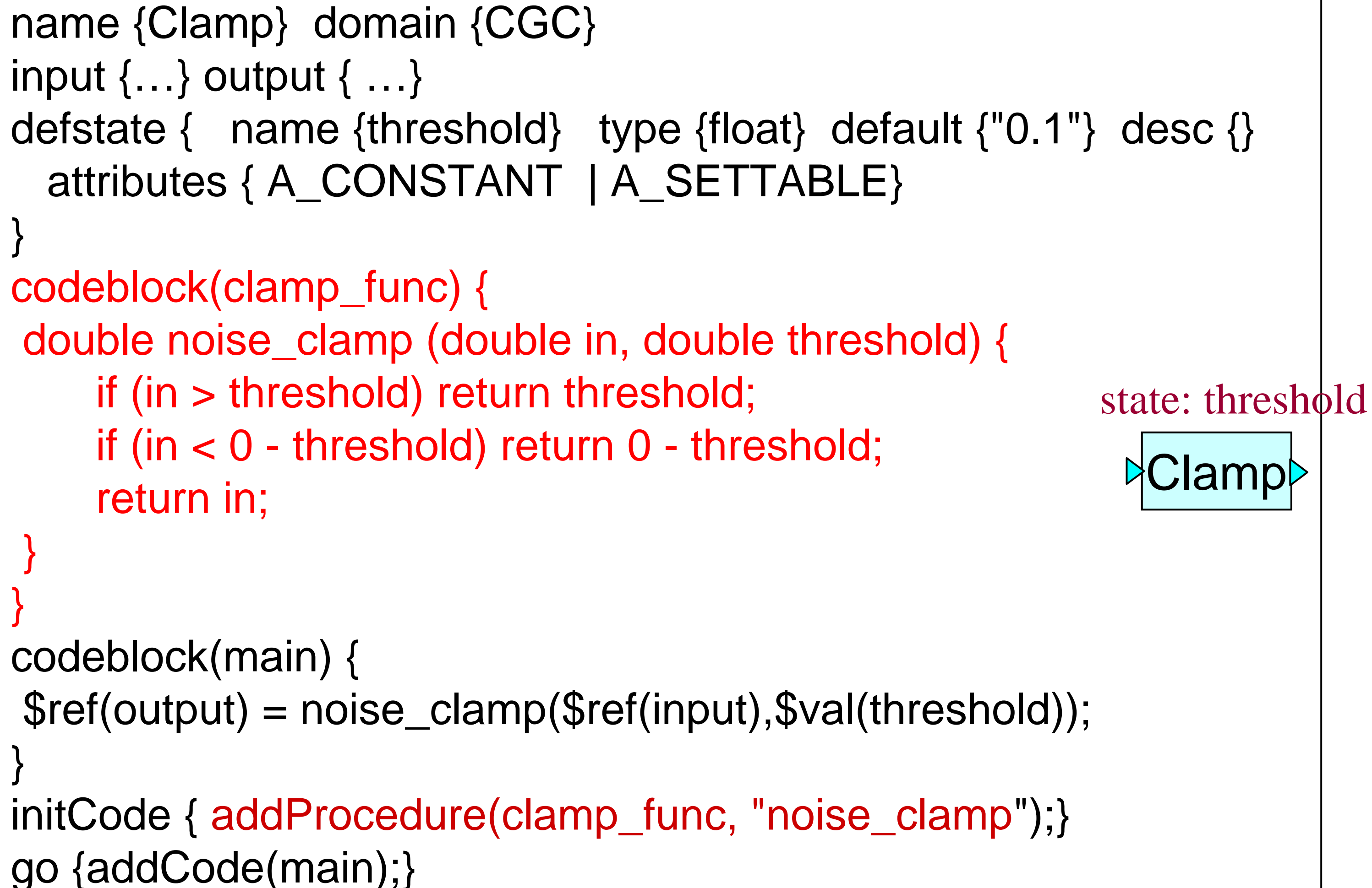
- addCode() - myCode
- addGlobal() – globalDecles
- addProcedure() – procedures
- addInclude() – include, addMainInit() – mainInit
- addDeclaration() - mainDecles

Block Definition

- **Block definition template (.pl)**
  - name
  - domain
  - input
  - output
  - state
  - setup
    - before schedule
  - initCode (or begin)
    - after schedule
  - go
    - generate code
  - codeblock
    - code to be generated



Function Style Code Generation



```
name {Clamp}  domain {CGC}
input {…} output { …}
defstate {   name {threshold}   type {float}  default {"0.1"}  desc {}
   attributes { A_CONSTANT  | A_SETTABLE}
}
codeblock(clamp_func) {
 double noise_clamp (double in, double threshold) {
       if (in > threshold) return threshold;
       if (in < 0 - threshold) return 0 - threshold;
       return in;
 }
}
codeblock(main) {
 $ref(output) = noise_clamp($ref(input),$val(threshold));
}
initCode { addProcedure(clamp_func, "noise_clamp");}
go {addCode(main);}
```

Block(Star) Macros

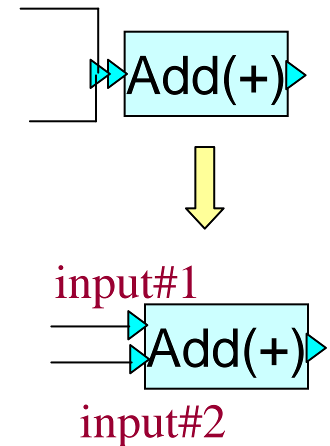
- **Need**
  - name variables
    - portholes
    - states
    - symbols
  - dynamic value(or size) assignment
- **Macros**
  - $ref(t), $ref(t,ix)
    - create a port or state variable with a unique name
  - $val(t)
    - substitute the value of the state
  - $starSymbol(t)
    - create any variable with a unique name

```
codeblock (declaration) {
   unsigned char $starSymbol(buf)
          [$val(blockSize)];}

codeblock (convert) {
{ int k;
  for (k = 0; k < $val(blockSize); k++)
    $ref(output,I) = $starSymbol
      (buf)[ $val(blockSize)-1-k]]; }
```

Block with a Multi-port

- **Multi-port**
  - The number of ports are determined by the number of connections

```
defstar {
        name {Add} domain {CGC}
        inmulti { name {input} type {float} }
        output { name {output} type {float}}
        go {
                StringList out = "\t$ref(output) = ";
                for (int i = 1; i <= input.numberPorts(); i++) {
                        out << "$ref(input#" << i << ")";
                        if (i < input.numberPorts())
                          out << " + ";
                        else
                          out << ";\n";
                }
                addCode(out);
        }
}
```



Complex Type Data

codesign environment

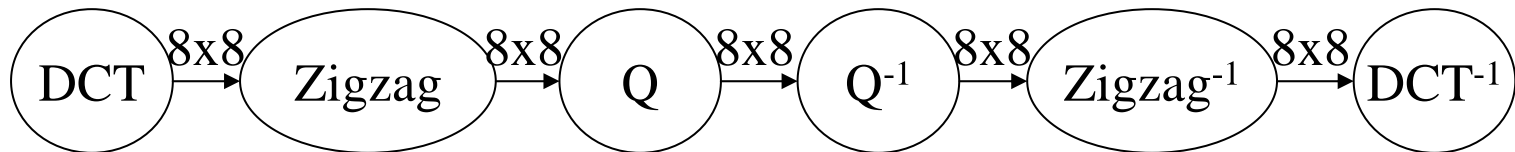
- **typedef struct complex_data { double real; double imag;} complex;**

```
defstar {
    name {AddCx}    domain {CGC}
    inmulti { name {input} type {complex} }
    output { name {output} type {complex} }
    go {
            addCode(startOp);
            for (int i=2;i<=input.numberPorts();i++)
                    addCode(doOp(i));
    }
    codeblock(startOp) {
            $ref(output).real = $ref(input#1).real;
            $ref(output).imag = $ref(input#1).imag;
    }
    codeblock(doOp,"int i") {
            $ref(output).real += $ref(input#@i).real;
            $ref(output).imag += $ref(input#@i).imag;
    }
}
```

codeblock with argument (points to `codeblock(doOp,"int i") {`)

Need of structure data type

- **Primitive data type**
  - int, float
- **Predefined data type**
  - complex
- **Demand for structure data type**
  - array data
  - packetized data: networked multimedia applications
  - bit array



Struct-type Data Specification

- **We define message type**
  - cf) int, float, complex, …
  - Example
    - *input { name { Y } type { message }}*
- **Define message name and size**
  - compile-time type checking

```
setup {
        Y.setMessageName(”struct Frame");
        Y.setMessageSize(64* sizeof(int));
}
```

Block(Star) code for message data

```
codeblock(FrameDef) {
        struct Frame { int pixel[8][8]; };
}
initCode {
        addGlobal(FrameDef, "Frame");   // message definition
}
codeblock(block) {
        int x;
        x = $ref(Y).data[0][0];
        $ref(Y).data[0][1] = x+3;
}
go {
        addCode(block);
}
```

Use of Predefined C Function

- **Assume there is a function foo(x,y) is defined elsewhere (foo.c). How can we use the function in PeaCE?**

Method 1: Make a new star CGCFoo.pl

```
addInclude("foo.c")
addCode("$ref(out) = f($ref(in1), $ref(in2));");
```

Method 2: Use a CGCGeneric.pl and define the following states

```
functionName = "foo"
inputDecles = "in1" "double" "1" "in2" "double" "1"
outputDecles = "out" "double" "1"
fileName = "foo.c"
```

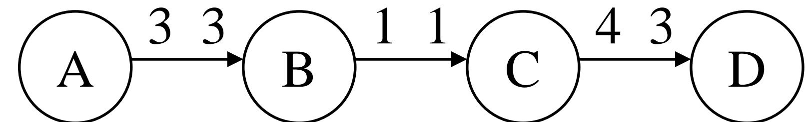
codesign environment

Memory Requirement

- **Code memory: depends on coding style**
  - Inlined-style, Switch-style
    - SAS (Single Appearance Schedule) is preferred.
    - In case of no-SAS, define a function in the body.
    - Good for fine granule node
  - Code sharing is helpful if the block context is smaller than the code size
- **Data memory: depends on schedule & buffer sharing**

Buffer Memory Requirement

- **<example>**
  - < Schedule 1 > 3(ABC)4(D)
    - minimum code size
    - buffer size: 3 + 1 + 12 = 16



  - < Schedule 2 > (3A)(3B)(3C)(4D)
    - minimum code size (inlined style)
    - buffer size: 9 + 3 + 12 = 24
    - buffer sharing: 3 + max (9,12) = 15
  - < Schedule 3 > 3(ABCD)D
    - code overhead
    - buffer size: 3 + 1 + 6 = 10

codesign environment

Previous Efforts

- **Single Appearance Schedule (SAS): APGAN,RPMC**
  - [by Bhattacharyya et. al.] in Ptolemy Group
  - SAS guarantees the minimum code size (without code sharing)
  - APGAN,RPMC : heuristics to find data minimized SAS schedule
- **ILP formulation for data memory minimization**
  - [by Ritz et. al.] in Meyr Group
  - flat single appearance schedule + sharing of data buffer
- **Rate optimal compile time schedule**
  - [by Govindarajan et. al.] in Gao Group
  - tried to minimize the buffer requirement using linear programming
- **An algorithm to compute the smallest data buffer size**
  - [by Ade et. al.] in GRAPE group

Buffer Memory Lower Bound

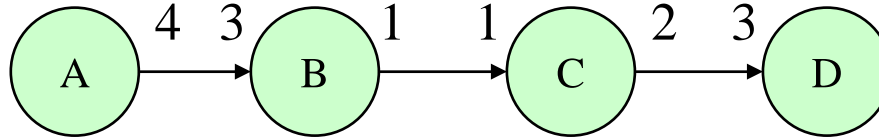
- **For single appearance schedule,**
  - a = produced(e), b = consumed(e), c = gcd{a,b}, d = delay (e)

$$BMLB(e) = \begin{cases} (\eta(e) + d) & \text{if } d < \eta(e) \\ d & \text{if } d \geq \eta(e) \end{cases}, \quad \text{where} \quad \eta(e) = \frac{ab}{c}$$

- **For any schedule**

$$LB(e) = \begin{cases} a + b - c + (d \bmod c) & \text{if } d < a + b - c \\ d & \text{otherwise} \end{cases}$$

Optimal Schedule for Chained Graph



- **Find an optimal factoring transformation**
- **No buffer sharing is considered**
- **Dynamic programming: $O(n^3)$**
  - *b[i,j] = min({(b[i,k] + b[k+1,j] + C[k] | (I <= k < j)})*
  - (3A)(4B): 12, (BC): 1, (3C)(2D): 6
  - {A,B,C}: ((3A)(4B))(4C): 12+4=16, (3A)(4(B)(C)): 12+1 = 13
    {B,C,D}: (3(B)(C))(2D): 1+ 6 = 7, (3B)((3C)(2D)): 3+6 = 9
  - {A,B,C,D}: 9A4{B,C,D} = (9A)((3(B)(C))(2D)): 36+7 = 43
    3{A,B}4{C,D} = (3(3A)(4B))(4(3C)(2D)): 12 + 6 + 12 = 30
    3{A,B,C}8D = (3(3A)(4(B)(C)))(8D): 13+24 = 37

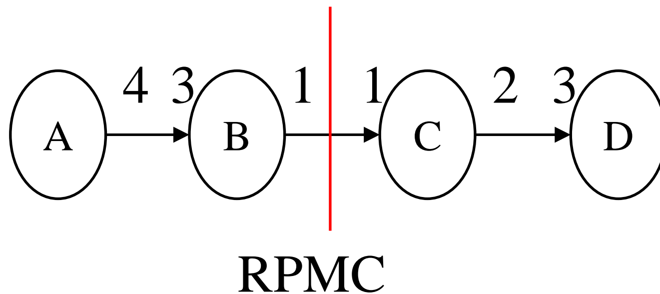
codesign environment

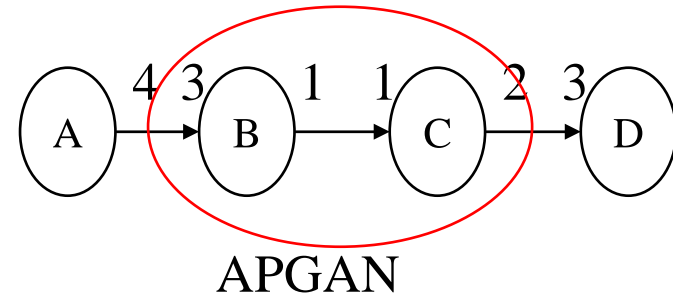
Extensions

- **Chain-structured graph with delays**
- **Well-ordered graph: partial order is a total order**
  - topological sort gives a single list to traverse.
- **Acyclic SDF graph**
  - the number of topological sort is exponential.
  - It is proved that the problem of constructing buffer-optimal single appearance schedules for acyclic graphs with delays is NP complete
    - reduced to vertex cover problem
  - Without delays, open problem.

Heuristic: RPMC and APGAN

- **Buffer minimization problem is NP hard for general graphs -> heuristics**
- **RPMC: Recursive Partitioning by Minimum Cuts**
  - find the cut where the minimum amount of data is transferred.
- **APGAN: Acyclic Pairwise Grouping of Adjacent Nodes**
  - a pair of adjacent nodes that has the largest *gcd* value of repetition counts.

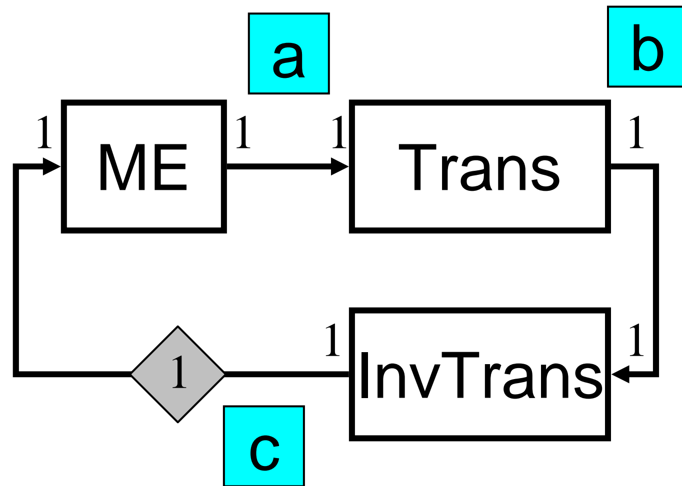


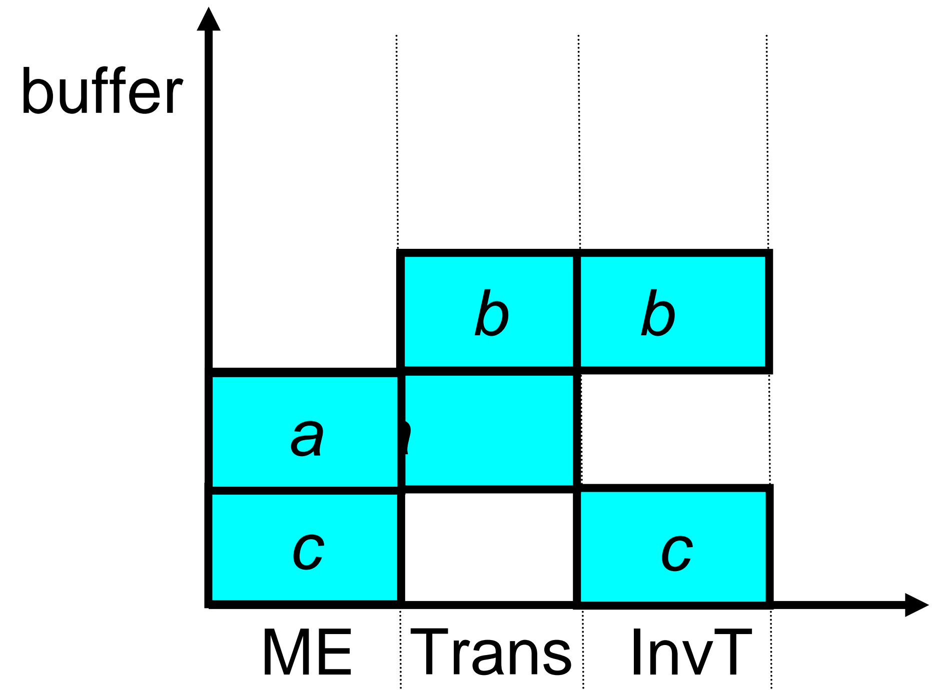


SDF Schedulers in PeaCE

- **AcyLoopScheduler (Acyloop)**
  - APGAN + RPMC heuristics
  - Applicable to acyclic graphs
  - Default scheduler
- **LoopScheduler (Loop)**
  - Compilcated looping heuristic for cyclic graphs
- **ClustScheduler (Clust)**
  - Simple looping heuristic
- **Default Scheduler**
  - Try to reduce the buffer memory requirement
  - Old default scheduler – not recommended for multirate graphs

Motivation: Buffer Sharing



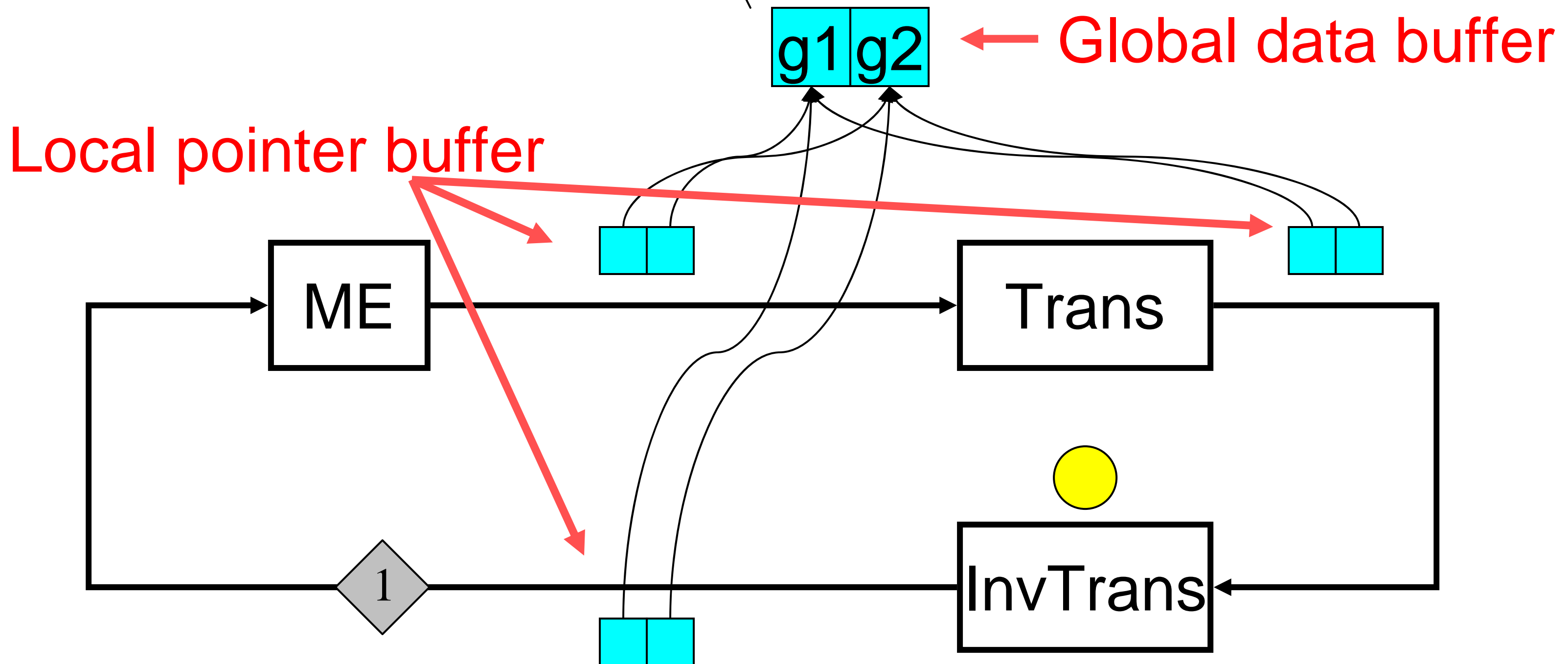


3 buffers

= 3x(176x144) = 76KB

Buffer lifetime chart

Our Approach

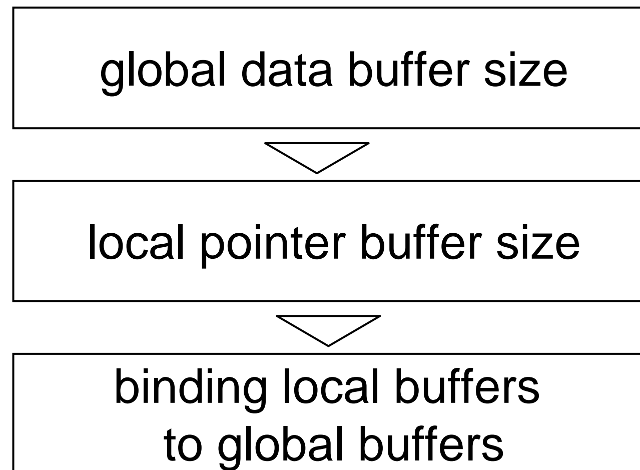


2 global buffers + 6 local buffers = 2x(176x144)+6x4 = 51KB

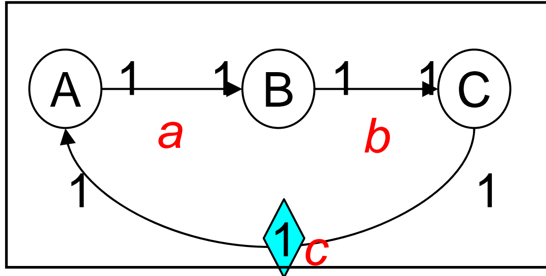
Buffer Sharing Problem

- **Determine both the local buffer sizes and the global buffer sizes**
  - for the objective of minimizing the sum of them
  - given program graph
  - sample lifetime chart
  - given schedule

- **Proposed Heuristic**



Subproblem 1 : Global Data Buffer Minimization



take out a sample lifetime with earliest start time

samples

s(c,2)

s(b,1)

s(a,1)

s(c,1)

A B C

"sample life-time chart"

Interval Scheduling

global buffer

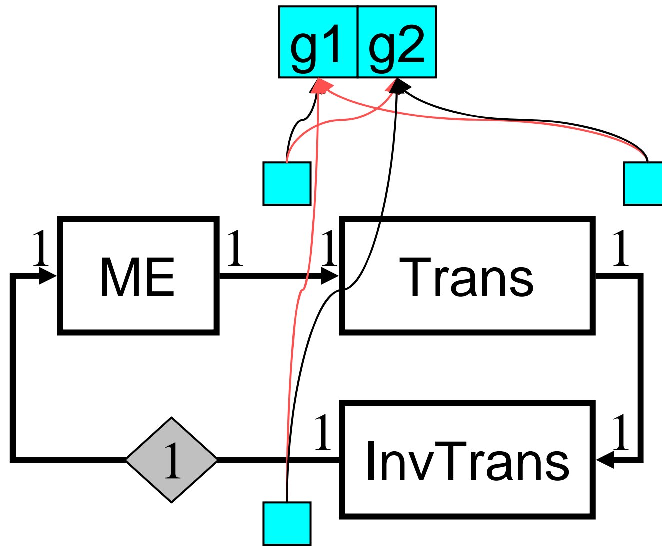
g(1) s(a,1) s(c,2)

g(0) s(c,1) s(b,1)

A B C

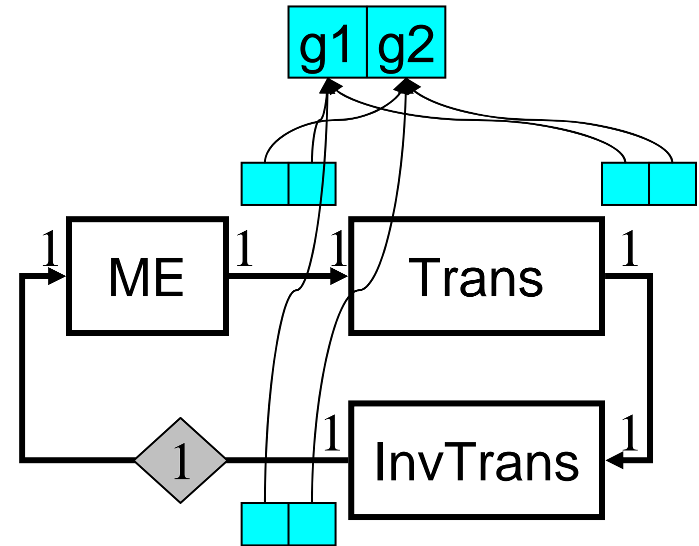
Subproblem 2: Local Buffer Size Determination

Dynamic binding



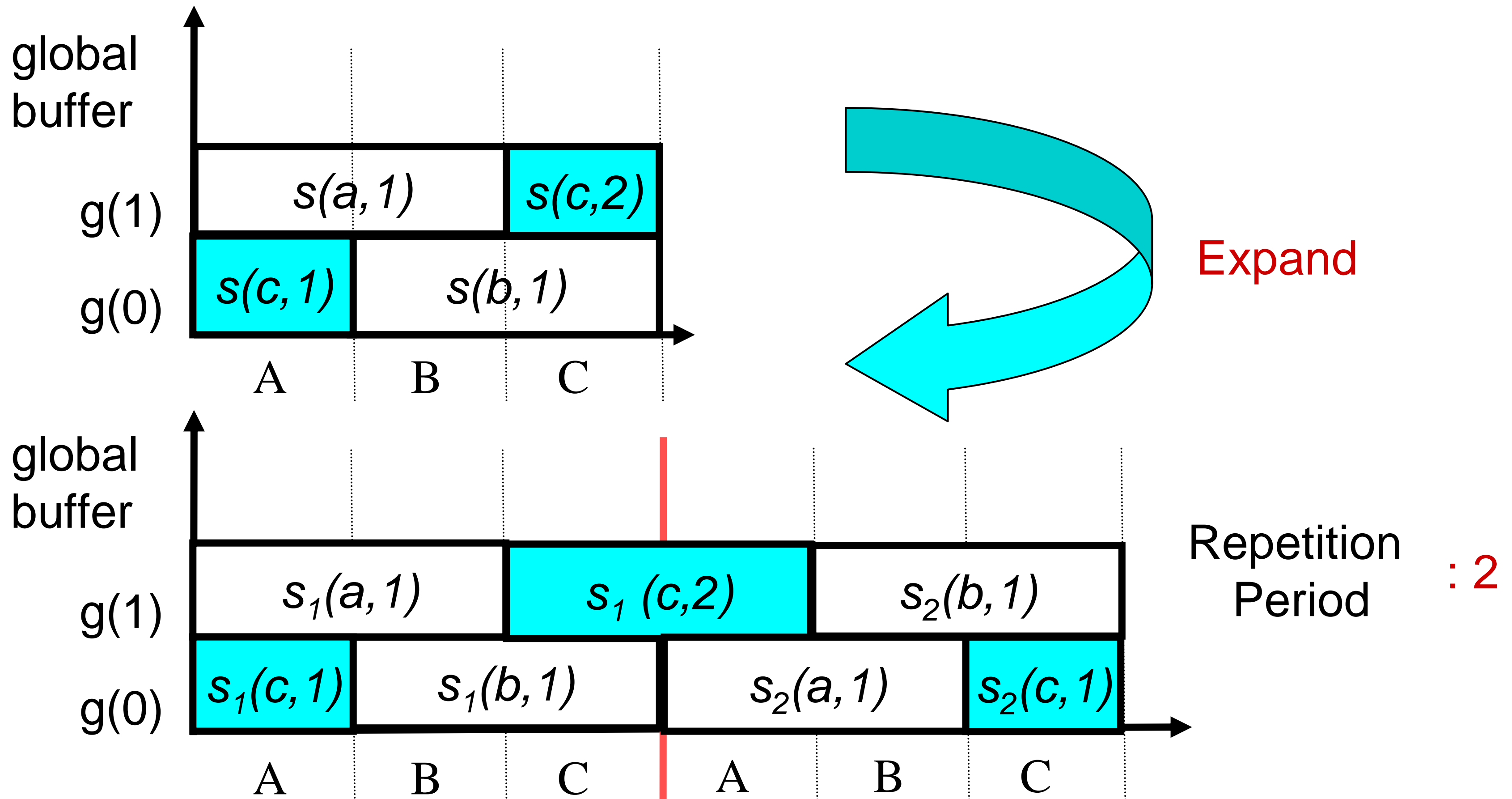
Local buffer size = the maximum number of live samples at any time

Static binding



Local buffer size = ?

Subproblem 3: Repetition Period



Code Generation with Static Binding

```
struct Frame g[2];
main()
{
  struct G *a[2]={g+1,g}, *b[2]={g,g+1},  *c[2]={g,g+1};
  int in_A=0, out_A=0, in_B=0, out_B=0, in_C=0,
out_C=1;

  for(int i=0;i<max_iteration;i++) {
    {  // A's codes. Use c[in_A] and a[out_A].
       in_A = (in_A+1)%2; out_A = (out_A+1)%2; }
    {  // B's codes. Use a[in_B] and b[out_B].
       in_B = (in_B+1)%2; out_B = (out_B+1)%2; }
    {  // C's codes. Use b[in_C] and c[out_C].
       in_C = (in_C+1)%2; out_C = (out_C+1)%2; }}}
```

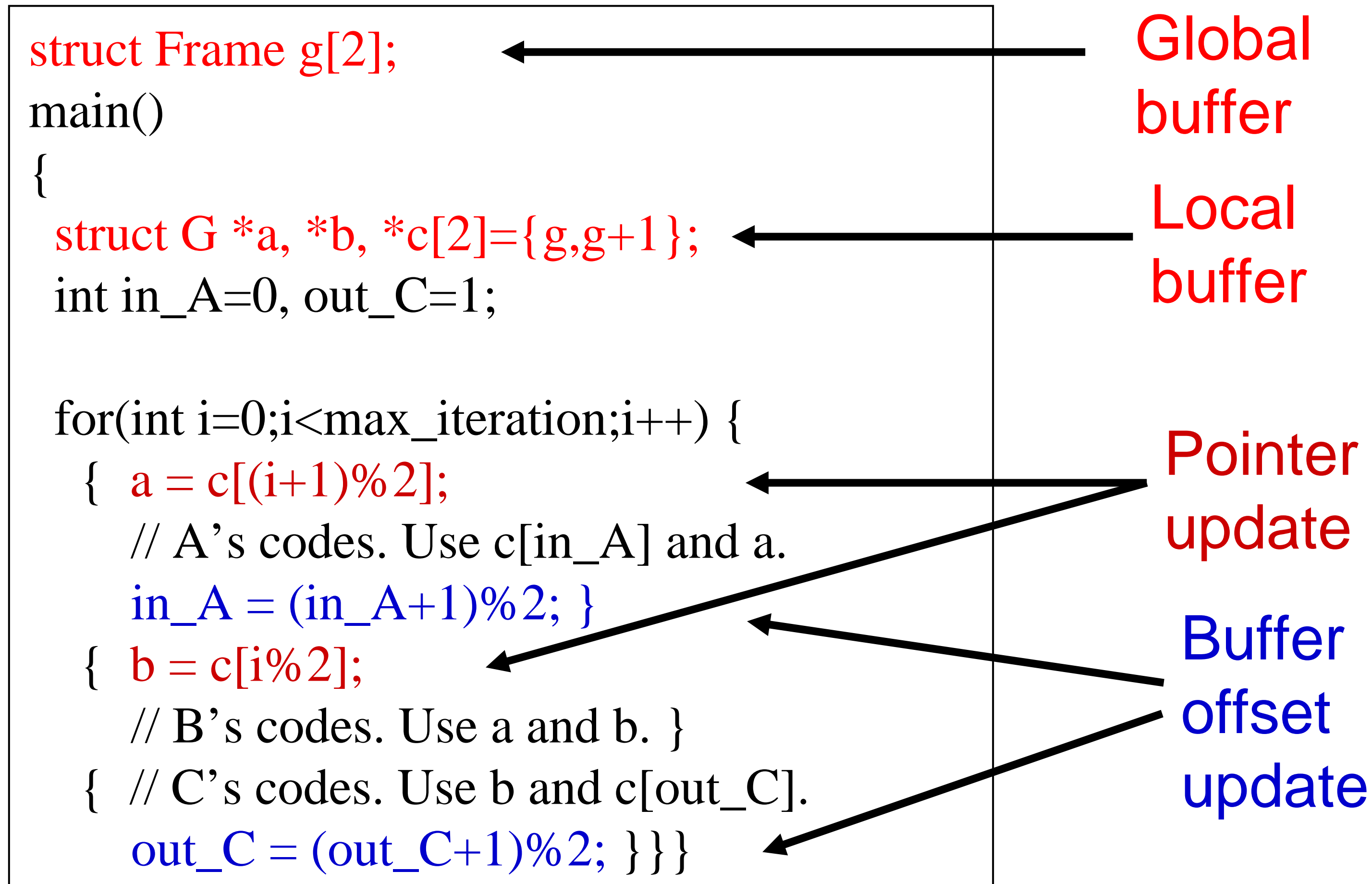
Global buffer

Local buffer

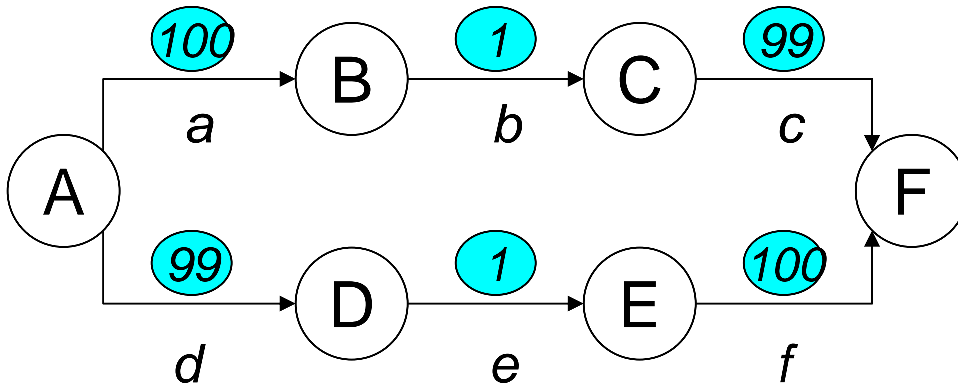
Buffer offset update

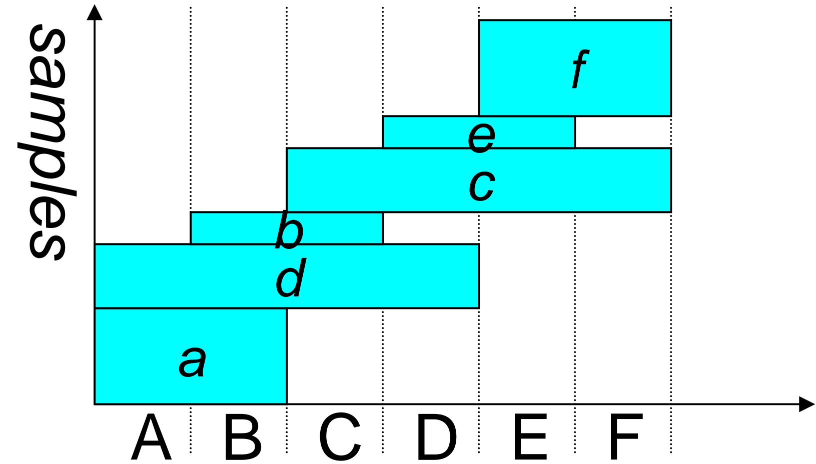
Code Generation with Dynamic Binding

codesign environment

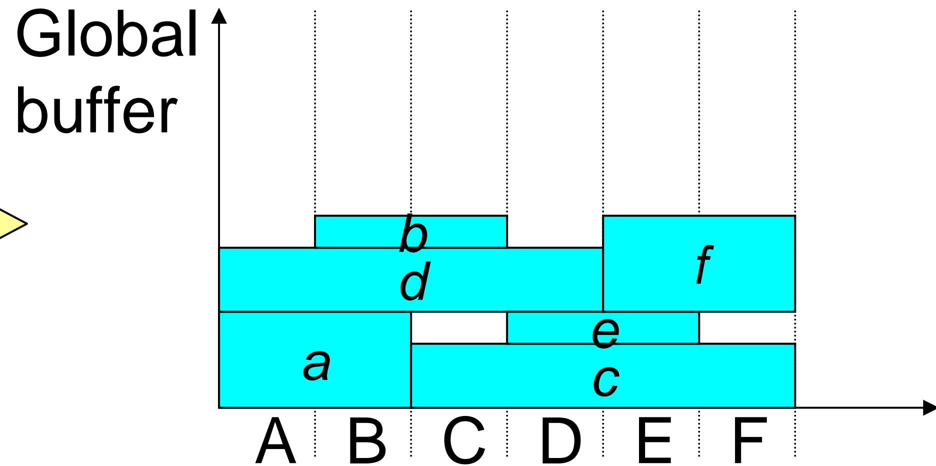


Different Size Buffer Sharing





Lowest Offset first
Earliest Start-time first



codesign environment

Experiments of Buffer Sharing

Example	JPEG	MP3	Simplified H.263	H.263
# of samples	6	336	3	1804
No sharing	1536B	36KB	111KB	659KB
Sharing of same size	512B	23KB	111KB	510KB
Sharing w/o separation	512B	11KB	111KB	510KB
Sharing with separation	-	-	74KB	396KB

codesign environment

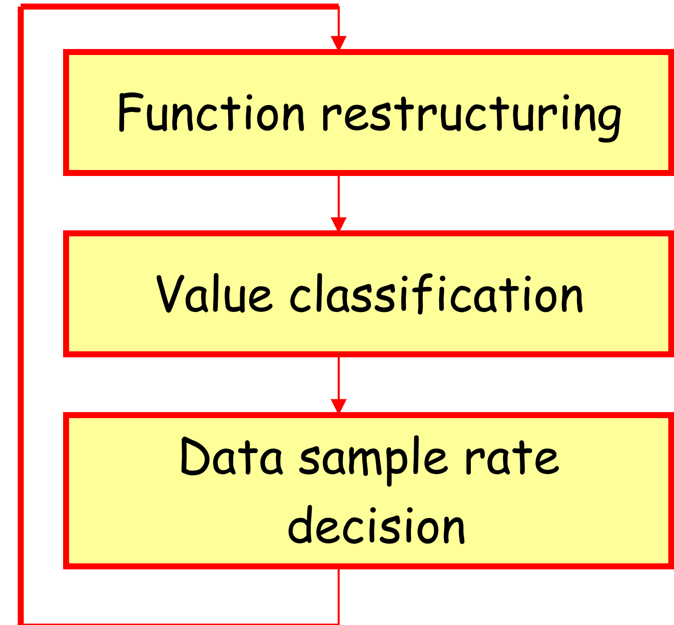
Outline

- Introduction: Model based design
- SDF (Synchronous Data Flow) Model
- Software Synthesis from SDF Model
- Translation of Reference C code to SDF Model
- SDF Extensions

Reference C Code Translation

Reference C code

```
int shared_A;

main() {
   double a[16], b[2];
   int c;
   code_A; // some code here accessing shared_A
   c = func_B (a, b);
   code_C; // some code here
}

int func_B (double x, double* y) {
    code_D; // some code accessing shared_A, and y
    return func_E(x);  // function body is composed
                       // with code_E
}
```

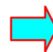


Code transformation procedure

Function Restructuring

- **Identify functions to be made as functional blocks (nodes of an SDF graph).**
- **Move them to the top level in the call graph.**

```
int shared_A;

main() {
   double a[16], b[2];
   int c;
   code_A; // some code here writing a, b, shared_A
   c = func_B (a, b);
   code_C; // some code accessing c
}

int func_B (double *x, double* y) {
    code_D; // some code accessing shared_A, and y
    return func_E(x);  // function body is composed
                       // with code_E
}
```



```
int shared_A;

void func_A(double*a, double* b) { ... }
void func_D(double* b) { ... }
int func_E(double *a) { ...}
void func_C(int c) { ... }

main() {
   double a[16], b[2];
   int c;
   func_A(a,b); // composed with code_A that uses
        shared_A
   func_D(b); // composed with code_D that accessing
                  // shared_A, and b
   c = func_E(a); // composed with code_E accessing a
   func_C(c); // composed with code+C some code here
}
```

Variable Classification

- **Determine the lifetime of all variables**
- **Remove communication through global variables as much as possible**

```
int shared_A;

void func_A() { ... }
void func_D(double* b) { ... }
int func_E(double *a) { ...}
void func_C(int c) { ... }

main() {
   double a[16], b[2];
   int c;
   func_A(); // composed with code_A that shared_A
   func_D(b); // composed with code_D that accessing
                 // shared_A, and b
   c = func_E(a); // composed with code_E accessing a
   func_C(c); // composed with code+C some code here
}
```

```
// Now, there is no more global variable left

void func_A(double* a, double* b, int* s) { ... }
void func_D(double* b, int* s) { ... }
int func_E(double *a) { ...}
void func_C(int c) { ... }

main() {
   int shared_A;
   double a[16], b[2];
   int c;
   func_A(a, b, *shared_A); // pass shared_A as an
      parameter
   func_D(b, *shared_A); // add shared_A to argument list
   c = func_E(a); // composed with code_E accessing a
   func_C(c); // composed with code+C some code here
}
```

Data Sample Rate Decision

codesign environment

- **Determine the sampling rate of functional blocks within a period, after computing the number of input and output samples per invocation**

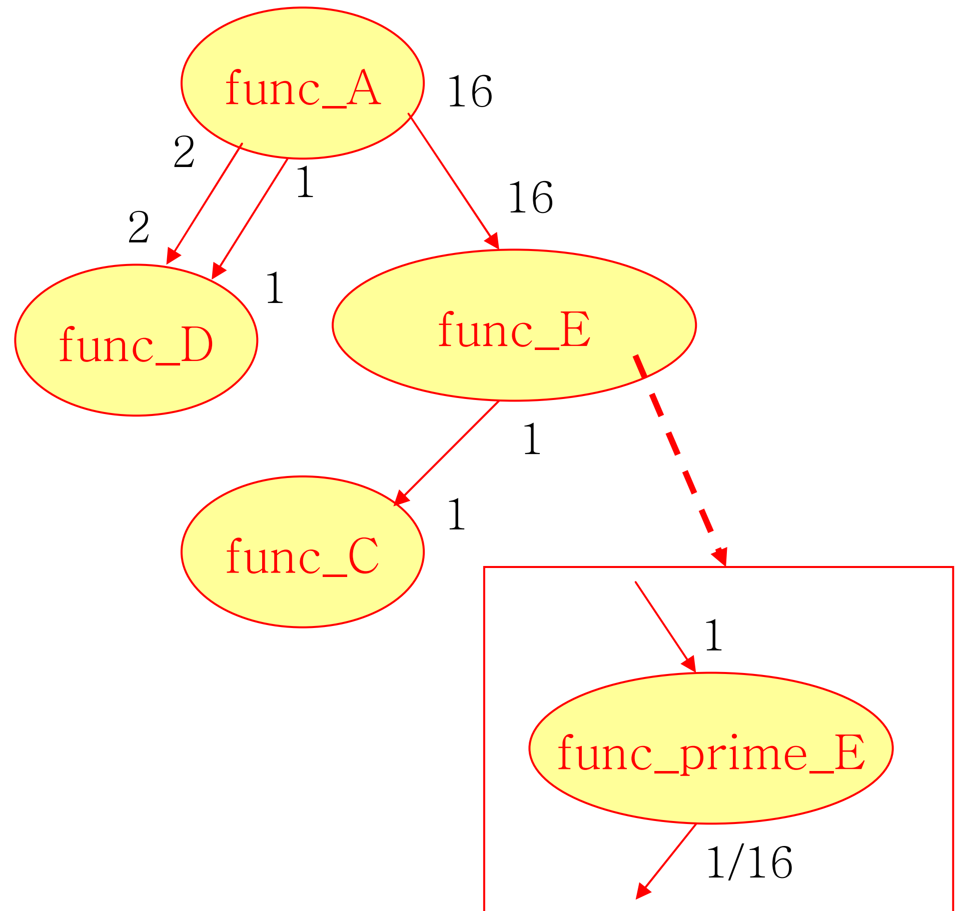
```
void func_A(double* a, double* b, int* s) { ... }
void func_D(double* b, int* s) { ... }
int func_E(double *a) { ...}
void func_C(int c) { ... }

main() {
   int shared_A;
   double a[16], b[2];
   int c;
   func_A(a,b,*shared_A); // pass shared_A as an
       parameter
   func_D(b, *shared_A); // add shared_A to argument list
   c = func_E(a); // composed with code_E accessing a
   func_C(c); // composed with code+C some code here
}
```

or

```
void func_A(double* a, double* b, int* s) { ... }
void func_D(double* b, int* s) { ... }
int func_E_prime(double a) { ...}
void func_C(int c) { ... }

main() {
   int shared_A;
   double a[16], b[2];
   int c, i;

   func_A(*shared_A); // pass shared_A as an parameter
   func_D(b, *shared_A); // add shared_A to argument list
   for (i = 0 ; i < 16 ; i++)
      c = func_E_prime(a[i]);
   func_C(c); // composed with code+C some code here
}
```

Resulting SDF graph

```
int shared_A;

main() {
   double a[16], b[2];
   int c;
   code_A; // some code here writing a, b, shared_A
   c = func_B (a, b);
   code_C; // some code accessing c
}

int func_B (double *x, double* y) {
    code_D; // some code accessing shared_A, and y
    return func_E(x);  // function body is composed
                       // with code_E
}
```



codesign environment

Outline

- **Introduction: Model based design**
- **SDF (Synchronous Data Flow) Model**
- **Software Synthesis from SDF Model**
- **Translation of Reference C code to SDF Model**
- **SDF Extensions**
  - CSDF
  - FRDF
  - SPDF

codesign environment

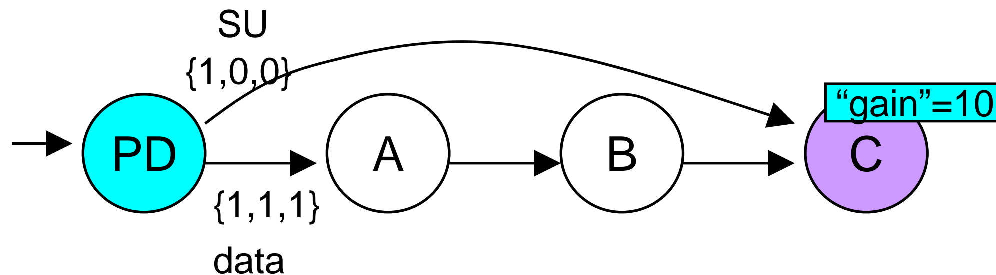
SDF Limitations

- **Expression Capability**
  - SDF model can not express control structures (dynamic constructs) such as conditional execution and data dependent iteration
- **SDF model does not allow shared memory (global states) between blocks due to side effect.**
  - why? memory update order may vary depending on the schedule.
- **SDF model does not allow pointer operation, but copies structured data as a token.**

SDF Extensions - CSDF

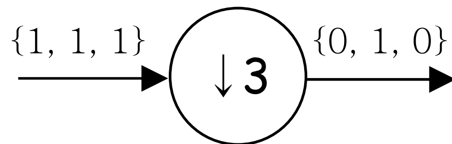
- **Cyclo Static Data Flow**
  - Allow periodic sample rate change
- **For each node A in a CSDF graph**
  - Basic period $\tau(A) \in \{1, 2, \ldots\}$
    - Can specify the phases of operation
    - Input sampling rate: $\tau(A)$-tuple $\{C_{e,1}, C_{e,2}, \ldots, C_{e,\tau(A)}\}$
    - Output sampling rate: $\tau(A)$-tuple $\{P_{e,1}, P_{e,2}, \ldots, P_{e,\tau(A)}\}$

- **Example**

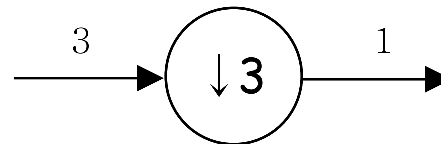


Translation: CSDF to SDF

- **SDF is a special case of CSDF: the period of all nodes are 1**
- **CSDF node A can be translated to SDF node A'**
  - For each output arc e'
    - *delay(e')* = *delay(e)*
    - *production(e')* = $\sum_{i=1}^{\tau(A)} P_{e,i}$
  - For each input arc e',
    - *consumption(e')* = $\sum_{i=1}^{\tau(A)} C_{e,i}$
- **Example: Downsampler**



(a) CSDF version

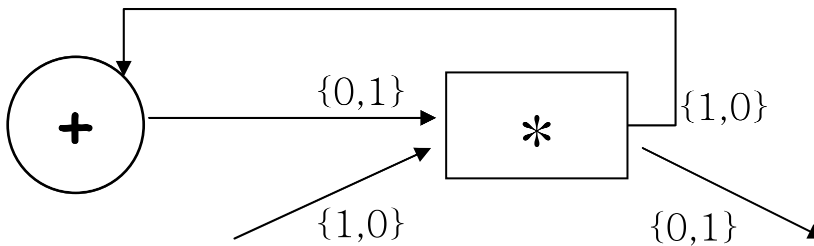


(b) SDF version

codesign environment

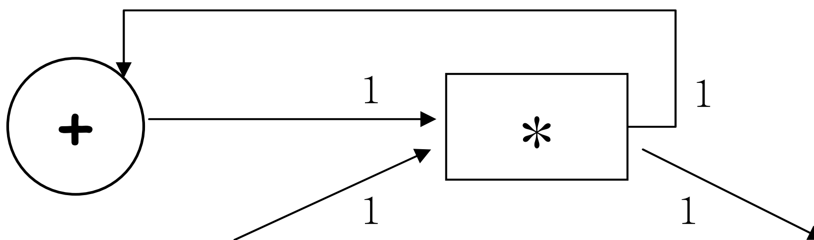
Various Communication Patterns

- **Even possible to form a deadlock-free feedback-loop without any delay on the arc**



(a) CSDF version

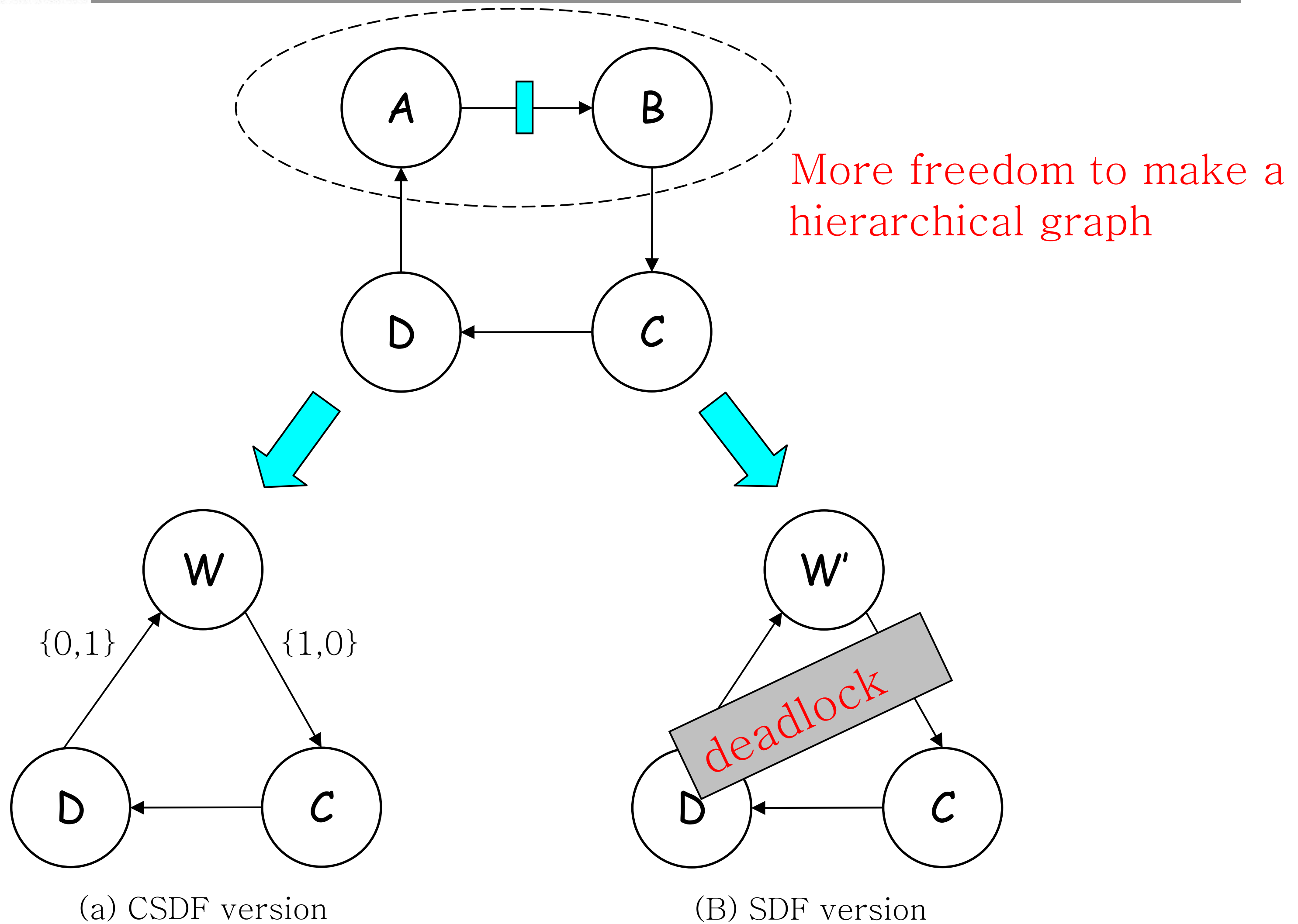
CSDF cycle
=> schedulable



(b) SDF version

delay-free SDF cycle
=> deadlock

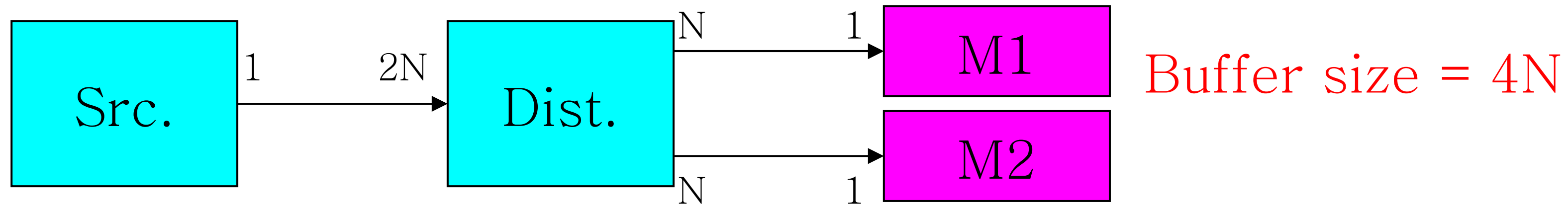
Hierarchical Graph Formation



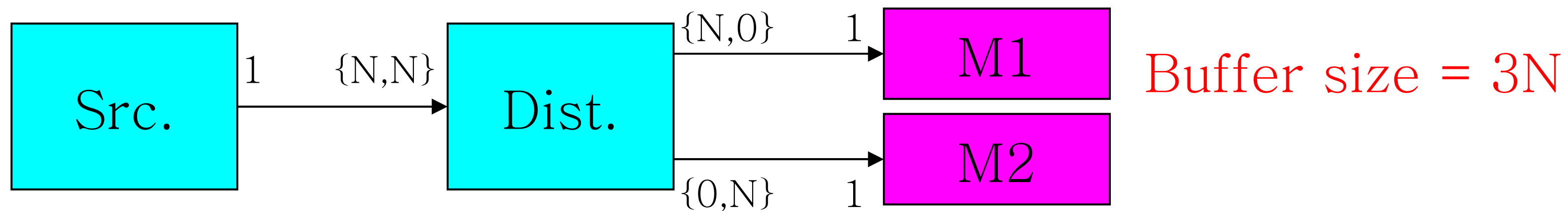
(a) CSDF version

(B) SDF version

Buffer Size Reduction



(a) SDF version

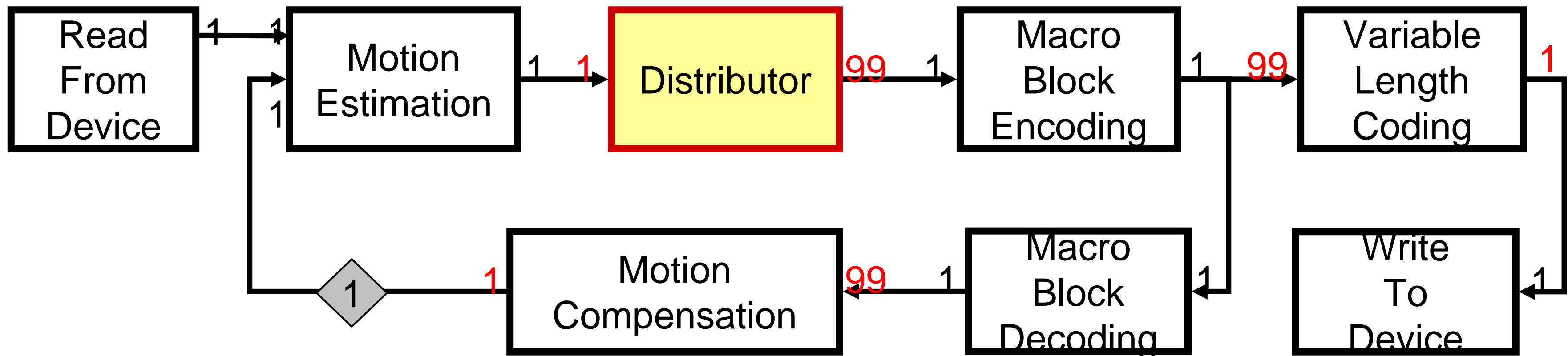


(b) CSDF version

SDF Extensions: FRDF

codesign environment

- **Motivation – H.263 encoder in SDF**
  - More memory requirement than reference C code
  - Separate buffers for frame data and macroblock data



	Reference Code (TMN 2.0)	SDF
Buffer size	361KB	686KB

Schedule : (ME,D)99(EN..)

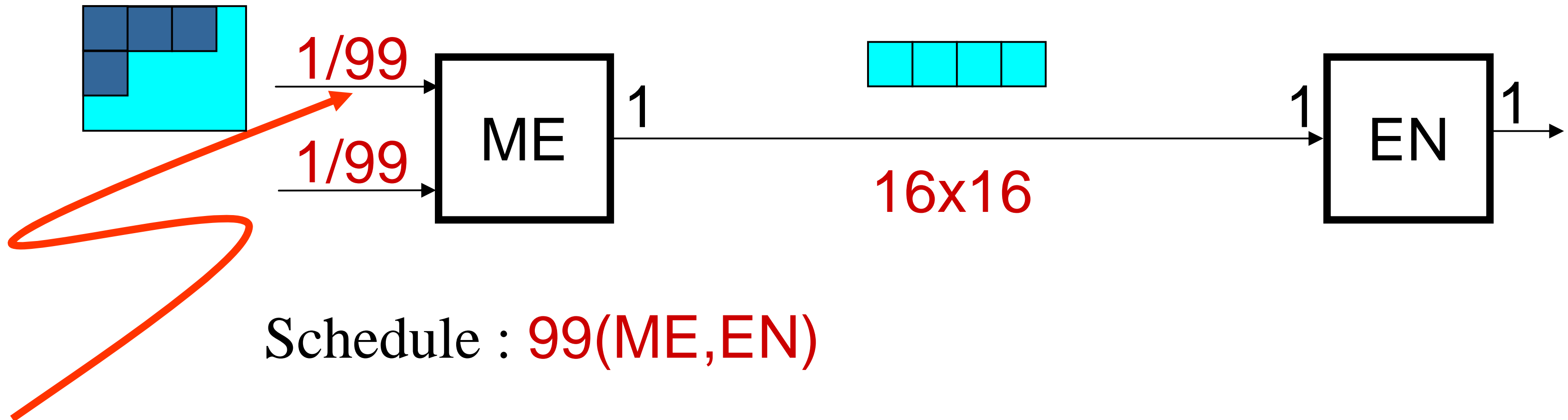
codesign environment

Observation

- **ME(motion estimation) node in hand-optimized code**
  - need not produce the frame-size output at once
  - Generates output samples at the unit of macro block for short latency and minimizing buffer memory

```
for(i=0; i<99; i++) {
  MotionEstimation(motion_vector,macroblock, currFrame,prevFrame);
  DCT(macroblock);
  Quantization(macroblock);
}
```

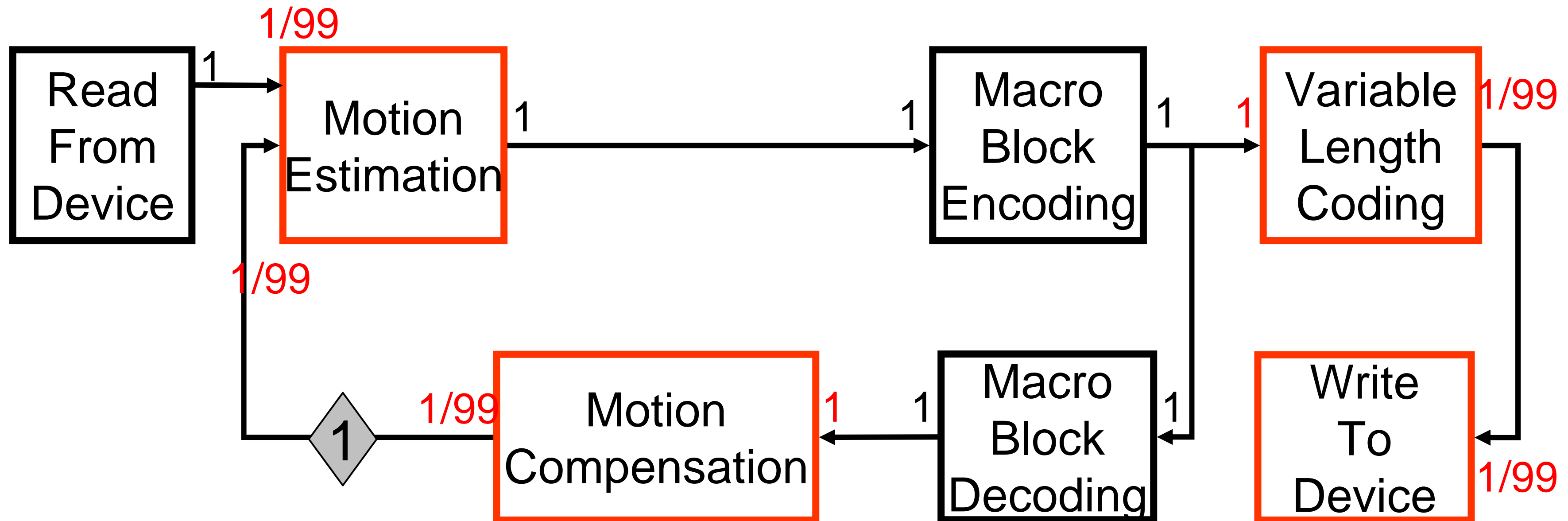
Fractional Rate Dataflow



The macro block is 1/99 of the video frame in its size.

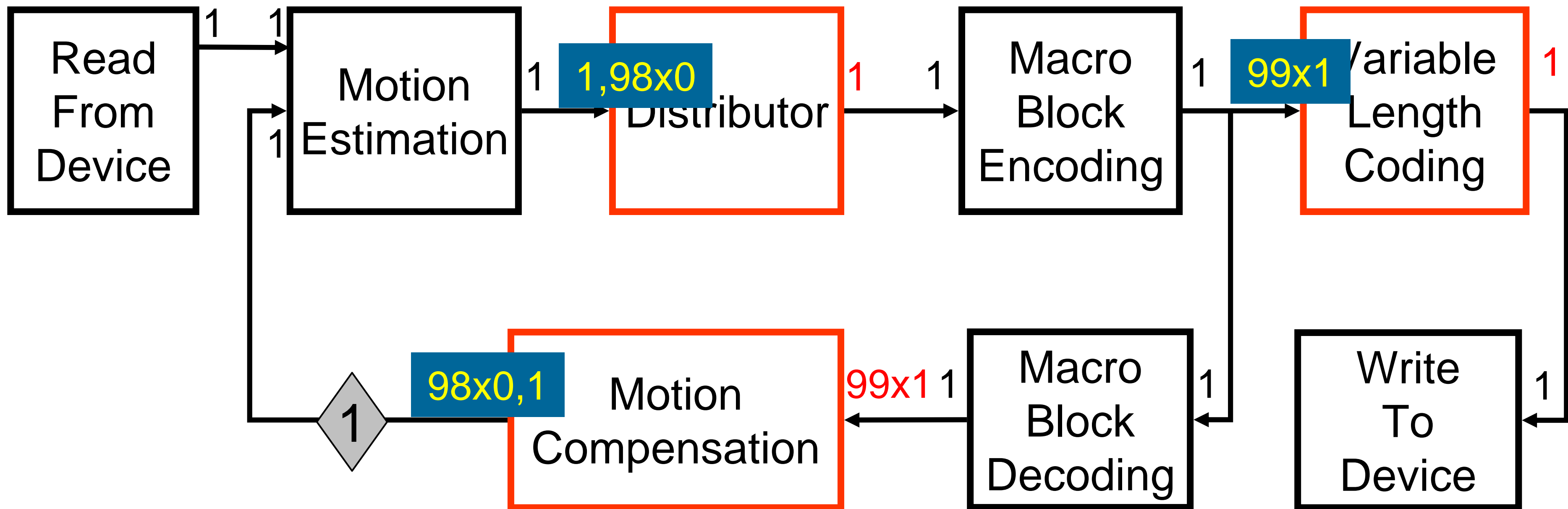
For each execution of the ME node, it consumes a macro-block from the input frame, and produces a macro-block output.

H.263 Encoder in FRDF



	Reference Code	SDF	FRDF
Buffer size	361KB	686KB	225KB
# frame size variables	5	8	3

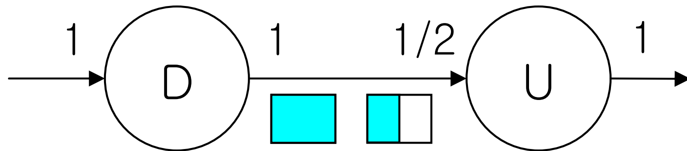
H.263 Encoder in CSDF



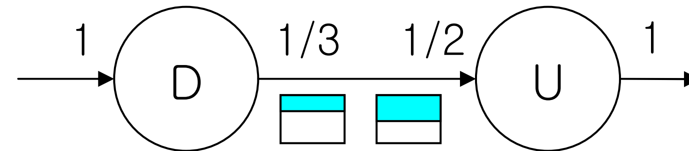
Buffer size : 291KB

Interpretation of Fractional Rate

- **Composite Type**



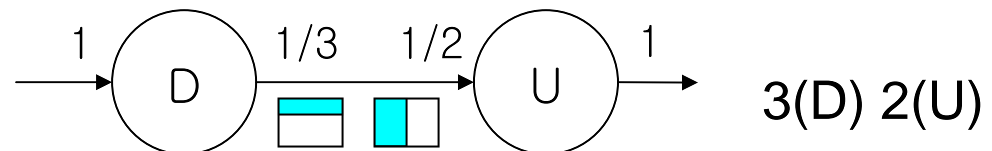
output rate is integer



output and input data have same access sequences

DDUDU is a valid schedule(SDF schedule)

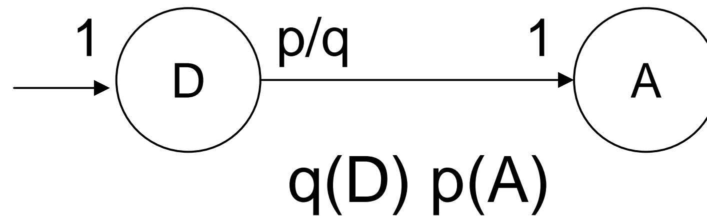
- **The consumer and the producer should have the same interpretation on the fraction. Otherwise, it is regarded as atomic type.**



3(D) 2(U)

Interpretation for Atomic Type

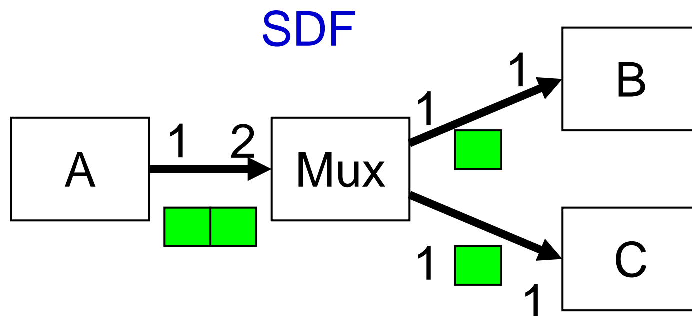
- **Statistical interpretation**



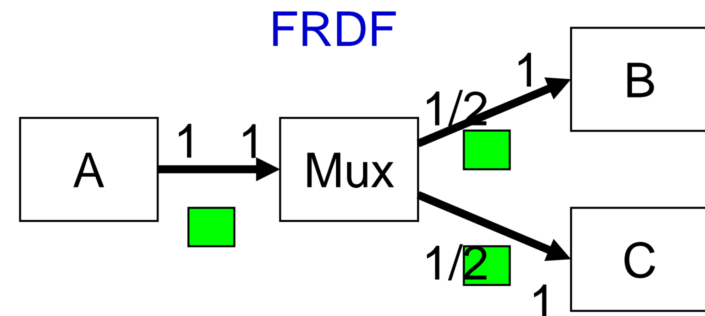
q(D) p(A)

  - p/q = p samples per q executions

FRDF requires less buffer space!



2(A) (Mux,B,C)



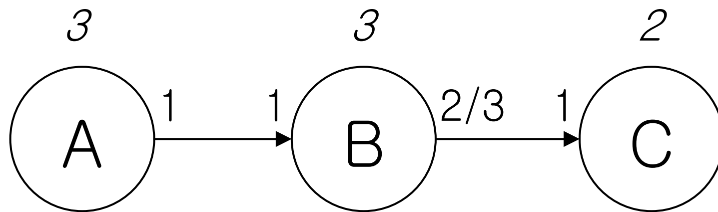
2(A,Mux) (B,C)

Transformation from SDF to FRDF

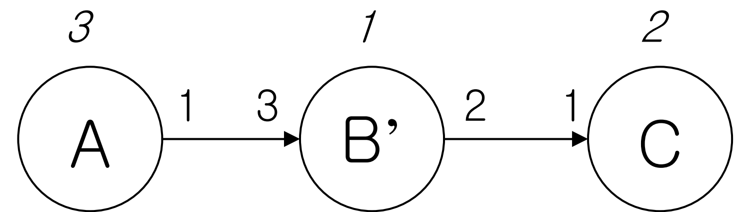
- **Equivalence Relationship between SDF and FRDF**

$$\frac{p_i}{q_i} \text{ (FRDF)} \rightarrow \frac{p_i}{q_i} \times Q \text{ (SDF)}$$

Where Q is the I/O period = LCM(repetition periods of all ports)



FRDF
3(A B) 2(C)

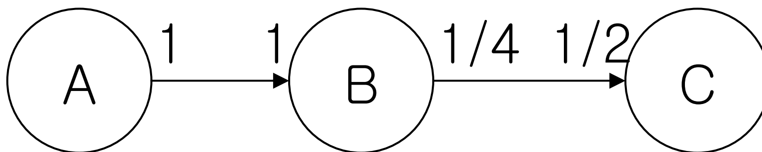


SDF
3(A) B' 2(C)

Firing Condition

- **An FR-SDF (fractional rate extension of SDF) is consistent if the equivalent SDF graph is consistent.**

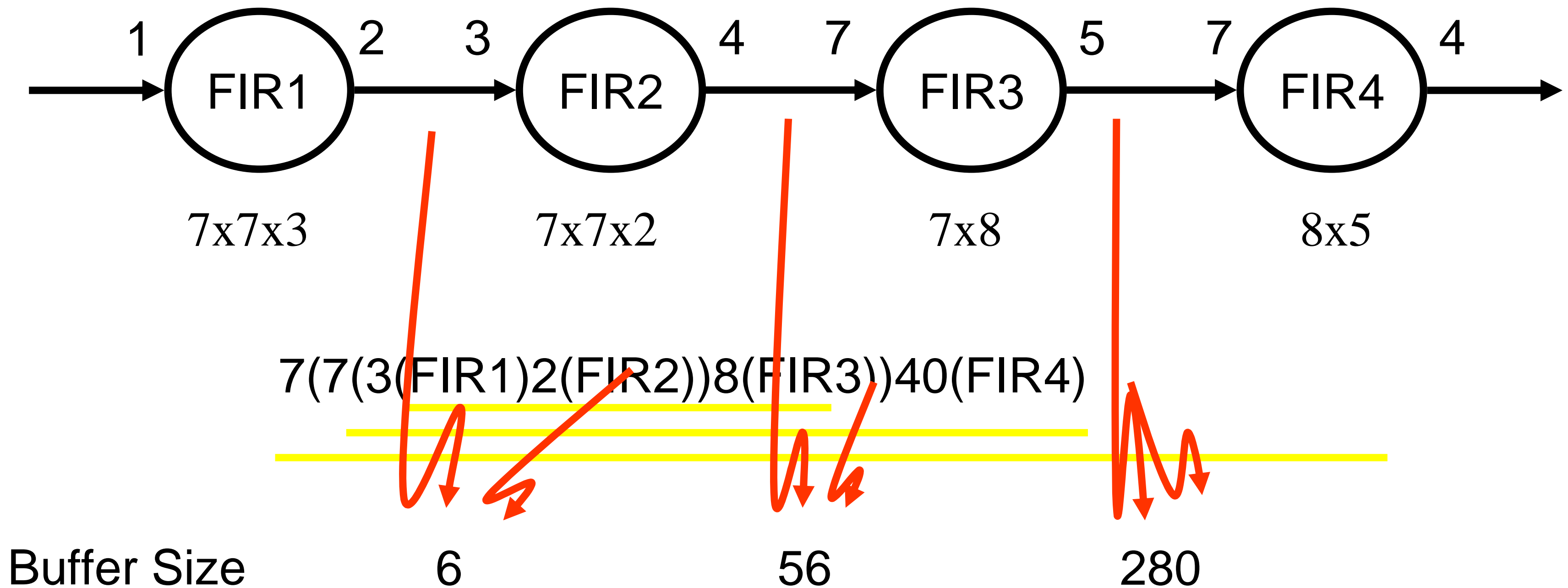
- **Firing Condition of FRDF node**
  - If data-type is composite, there must be at least as large fraction of samples stored as the fractional input rate
  - Otherwise, there must be at least as many samples stored as the numerator value of the fractional sample rate



atomic : 4(AB)2(C)

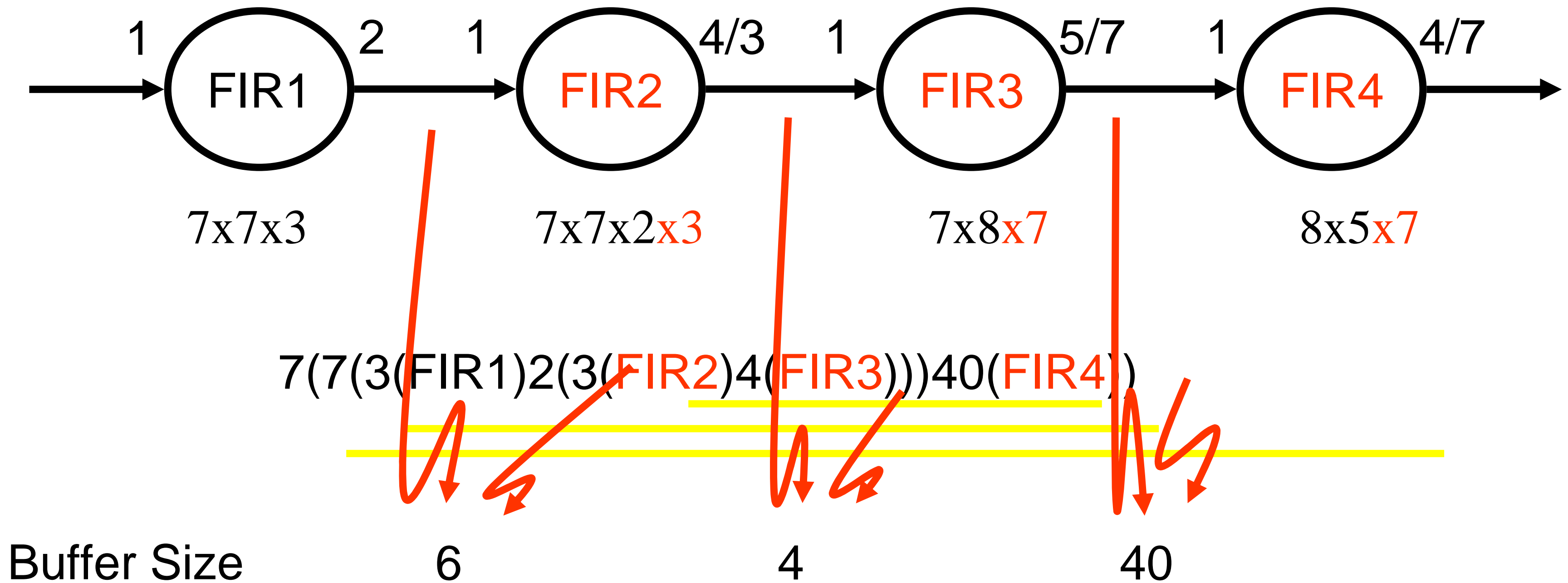
composite: 2(2(AB)C)

CD2DAT Example



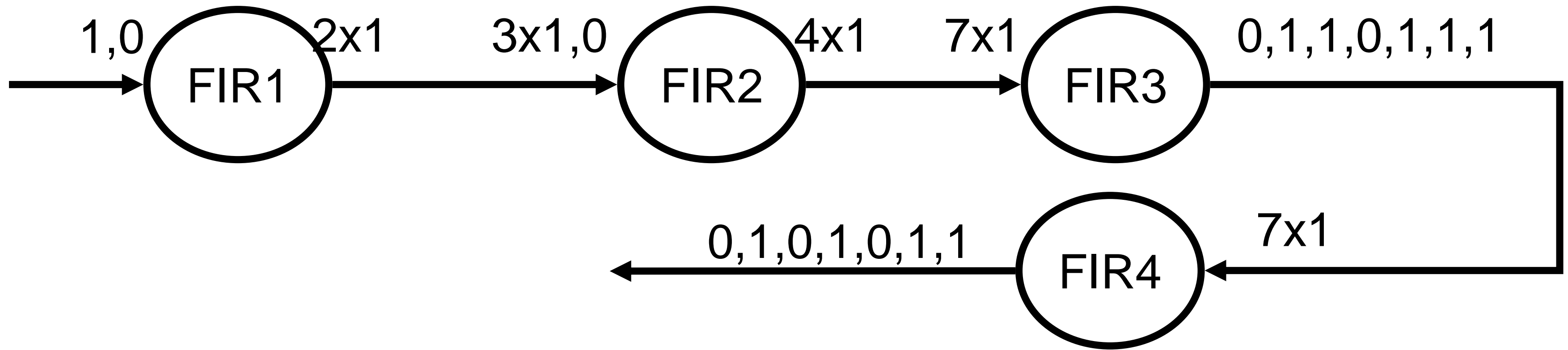
codesign environment

CD2DAT Example



CD2DAT Example

CSDF



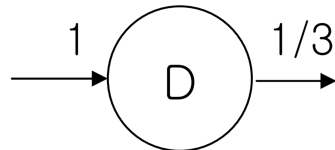
Buffer Size	6	4	40

FRDF Implementation

- setSDFParams(int,int,int)
- setAtomicType(int)
- setAccessSequence(const char*)
- $phase(port)

setSDFParams

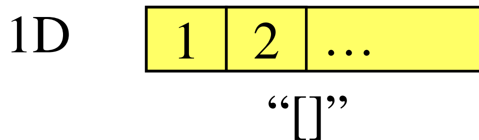
- **Sample Rate Specification**
  - name.setSDFParams(m,p, divider)
    - name.setSDFParams(m/divider, p/divider);

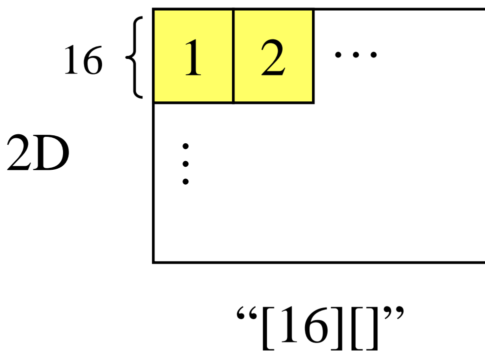


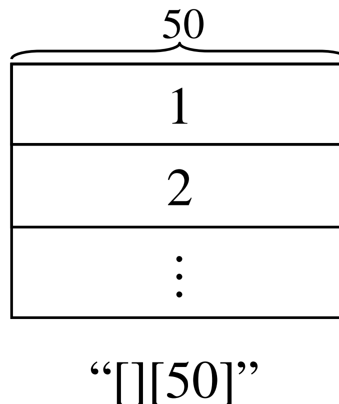

```
input.setSDFParams(1,0);
output.setSDFParams(1,0,3);
```

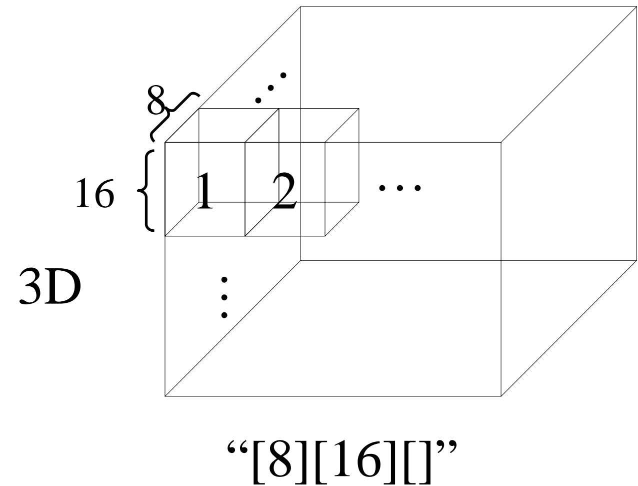
setAccessSequence()

- **The port.setAccessSequence() represents the patterns of accessing a sample.**
- **Rule**
  - Given array a[z][y][x]
  - Access block a[8][16][16] → setAccessSequence(“[8][16][]”)

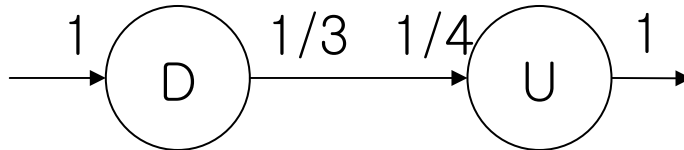








Example



For block D

```
output.setSDFParams(1,0, 3);
output.setAcessSequence("[]");
```

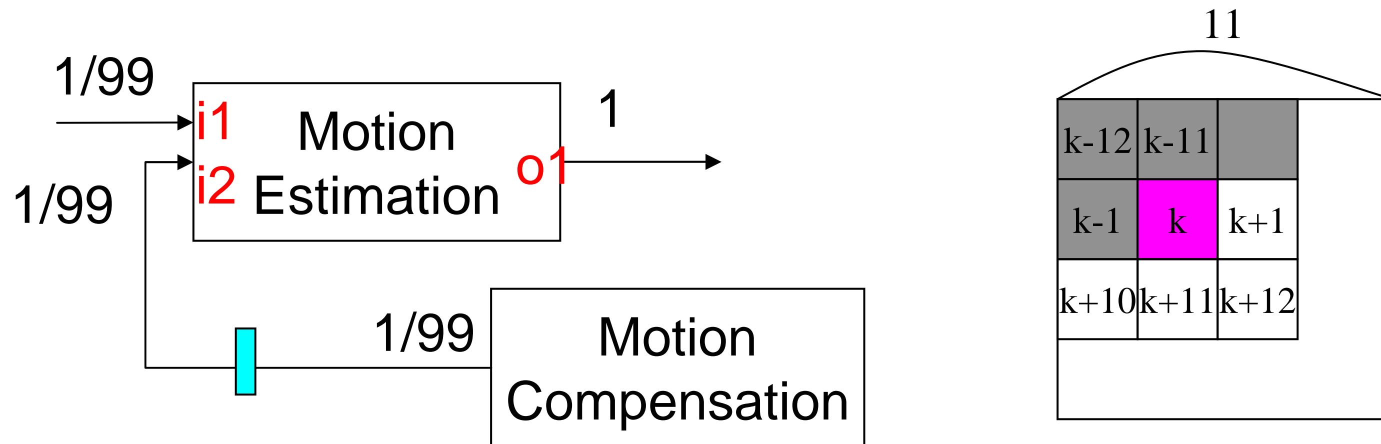
For block U

```
output.setSDFParams(1,0, 4);
output.setAcessSequence("[]");
```

setAtomicType

- **If** *port.setAtomicType(TRUE)* **then the port generates a sample of atomic type; otherwise it does a sample of composite type.**
- **Default**
  - Primitive type : atomic type
  - Message type : composite type

Example



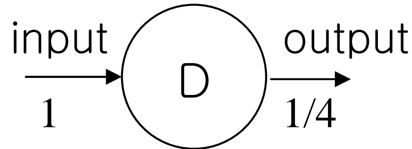

```
i1.setSDFParams(1,0, 99);
i1.setAccessSequence("[16][]");
i1.setAtomicType(FALSE);
```
← Default value when i1 consumes message type samples

```
i2.setSDFParams(1,0, 99);
i2.setAtomicType(TRUE);
```
← i2 accesses parts of a sample from k-12 to k+12 when it consumes k-th part of a sample.

```
o1.setSDFParams(1,0);
```

$phase(port) Macro

- **$phase(port) indicates the state of port**

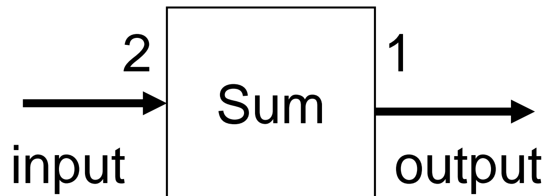


$phase(output) has 4 values (0 to 3).
$phase(output) increases every execution of node D.
When it becomes 4, it is reset to 0.

D1 : $phase(output) = 0
D2 : $phase(output) = 1
D3 : $phase(output) = 2
D4 : $phase(output) = 3
D5 : $phase(output) = 0
D6 : $phase(output) = 1
D7 : $phase(output) = 2
…

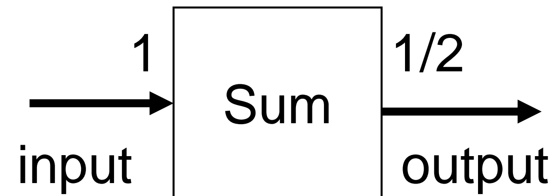
FRDF Node for Atomic Type

SDF-style Sum



output = input[0]+input[1];

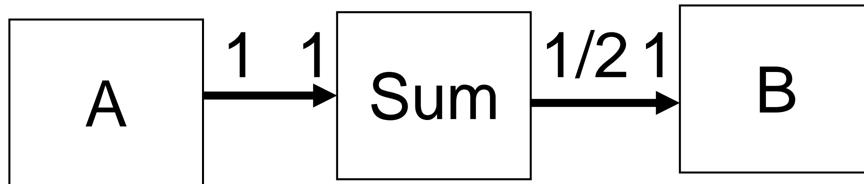
FRDF-style Sum




```
if($phase(output)==0)
  output = input;
else
  output += input;
```

$phase(output) : internal variable to count node execution. ({0,1})

Code Generation from FRDF

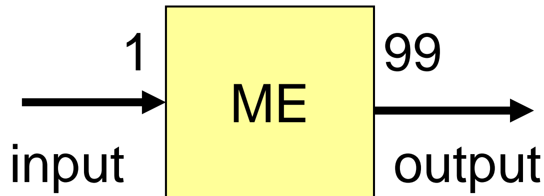


2(A,Sum) (B)

```
main()
{
  while(1) {
    for(i=0;i<2;i++) {
      {A}
      {if(Sum_phase_output==0)
          Sum_output = A_output;
        else
          Sum_output += A_output;
        Sum_output =
          (Sum_output+1)%2;
       }
    }
    {B}
}}
```

FRDF Node for Composite Type

SDF-style Motion Estimation




```
for(i=0; i<99; i++) {
  x = (16*i)%144;
  y = 16*((16*i)/144);
  motionEstimation(input,x,y,
     output[i]);
}
```

FRDF-style Motion Estimation

```
x = (16*$phase(input))%144;
y = 16*((16*$phase(input))/144);
motionEstimation(input,x,y,
    output);
```

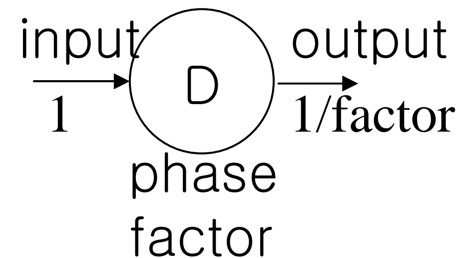
$phase(input) = i

codesign environment

Star Examples

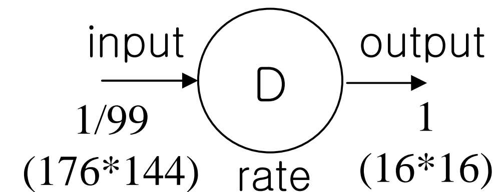
CGCFRDownSample.pl

```
setup {
        input.setSDFParams(1,0);
        output.setSDFParams(1,0,int(factor));
 }

 codeblock (sendsample) {
 if($phase(output) == $val(phase)) {
        $ref(output)=$ref(input);
 }
 }
 go {
        addCode(sendsample);
 }
```



CGCMsgDeCom.pl : Message decomposer

```
setup {
        input.setSDFParams(1,0,99);
        output.setSDFParams(1,0);
        input.setAccessSequence("[16][]");
}
codeblock (block) {
        int xIn, yIn, xOut,yOut;

        yIn = 176*144*(int)($phase(input)/(16*16));
        xIn = 16*($phase(input)%(176/16));

        for(yOut=0; yOut<16; yOut++) {
            for(xOut=0; xOut<16; xOut++) {
                $ref(output).data[yOut*16+xOut] =
                        $ref(input).data[yIn+yOut*176+xIn+xOut];
            }
        }
}
```



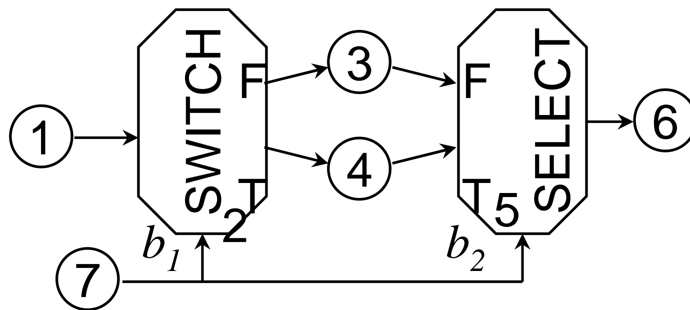
codesign environment

FRDF Wrap-up

- **Fractional number of samples can be produced or consumed**
- **Reduce latency and memory size**
- **Well match with multimedia applications**
  - Reduce the buffer size by 70% compared with SDF
- **Future plan**
  - Buffer-optimal schedule

Boolean-controlled Data Flow

- **Keeps static property with data dependent operations**
  - Strong, week consistency

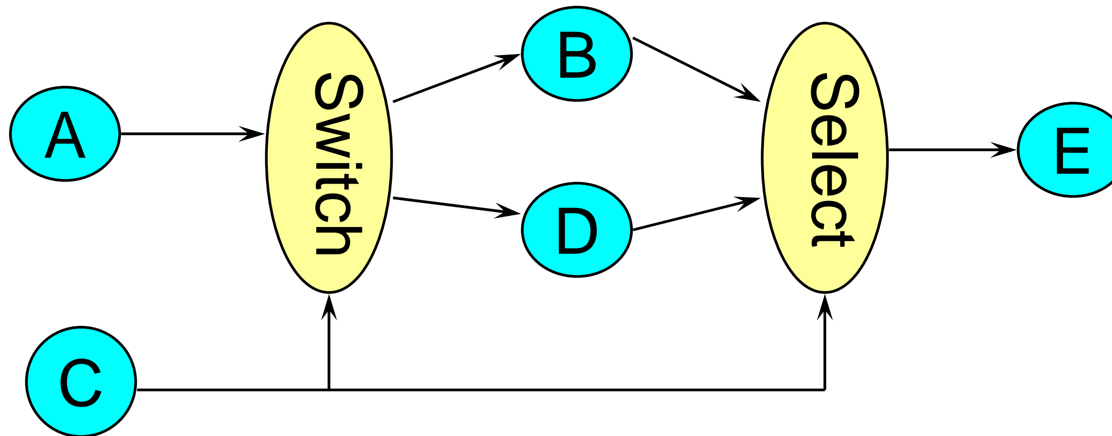


$$\Gamma(\boldsymbol{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & (1-p_1) & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & (p_2-1) & 0 & 0 \\ 0 & p_1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

$$1,\ 7,\ 4\ (if\ b1 = TRUE),\quad 6$$
$$3\ (if\ b1 = FALSE)$$

BDF Scheduling

- **Add if-then-else structure to SDF model - Turing equivalent**

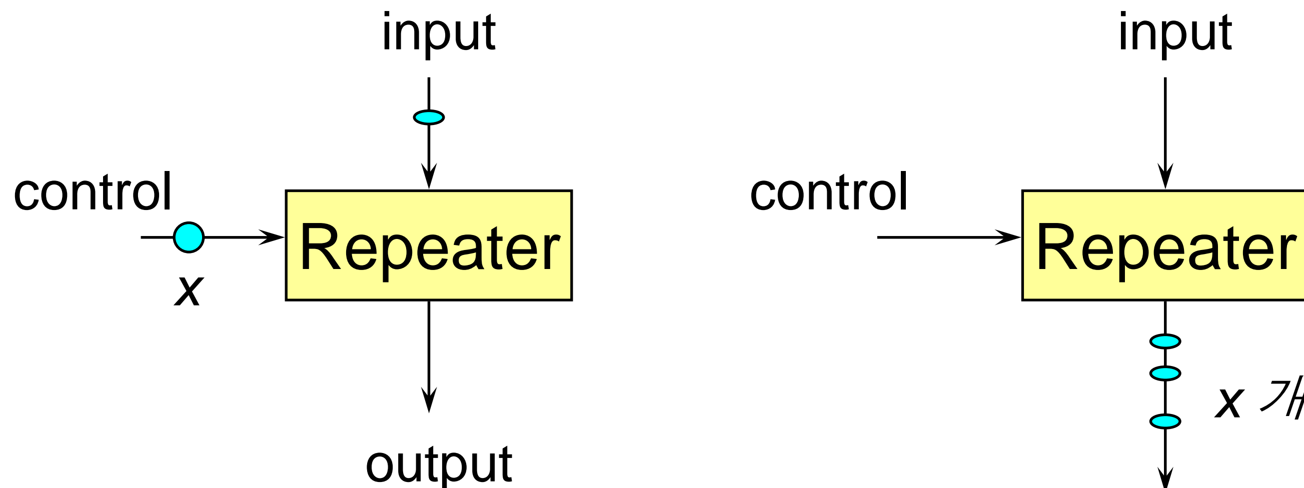


- **Compile-time scheduling (Quasi-static scheduling)**

A - C - { is C is true, D, else, B } - E

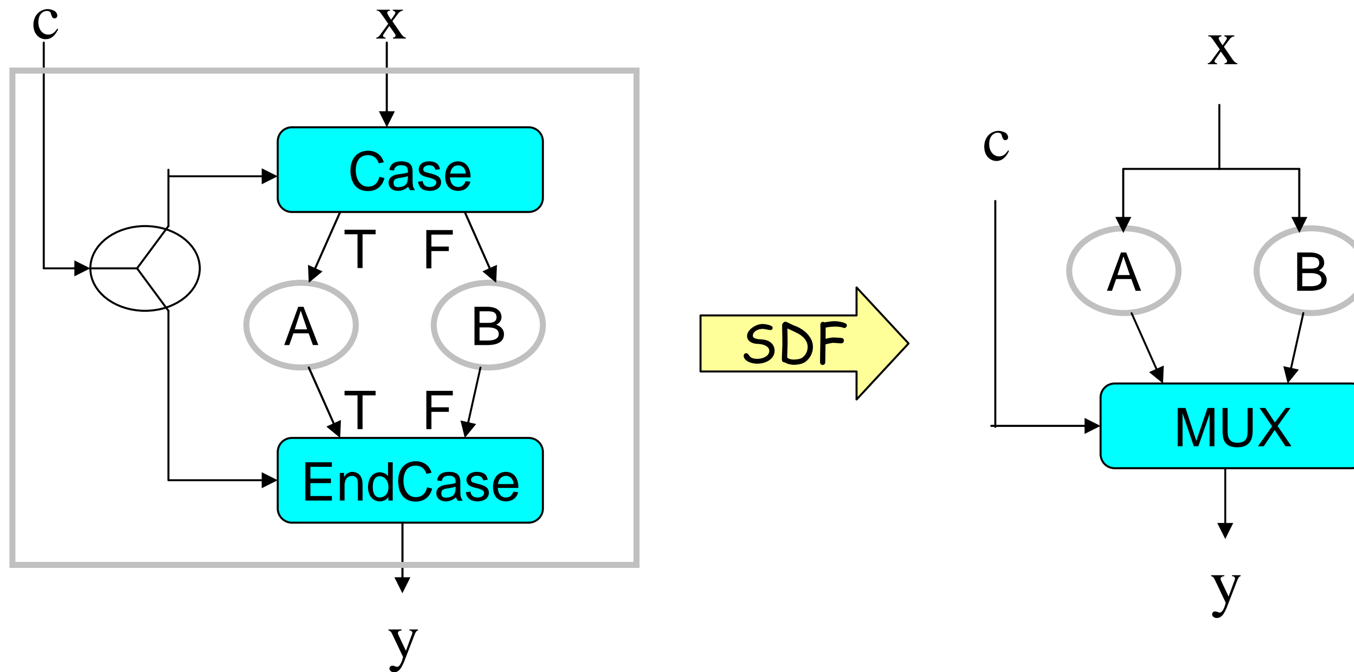
Dynamic Dataflow (DDF) Model

- **Allow more general dynamic structures**
  - data-dependent iteration, case, recursion, ...
- **Equivalent to Kahn process networks: blocking read**
- **Runtime scheduler is needed.**



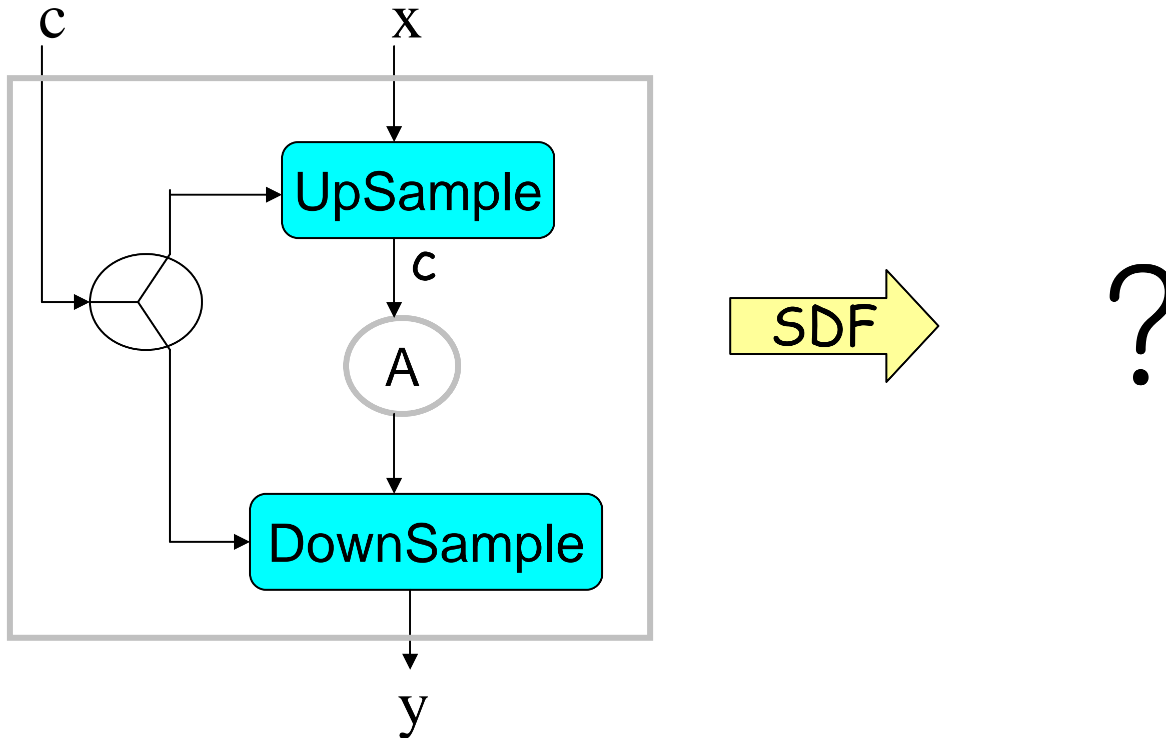
If-then-else construct

- **Dynamic data flow: block firing condition is determined at run-time**



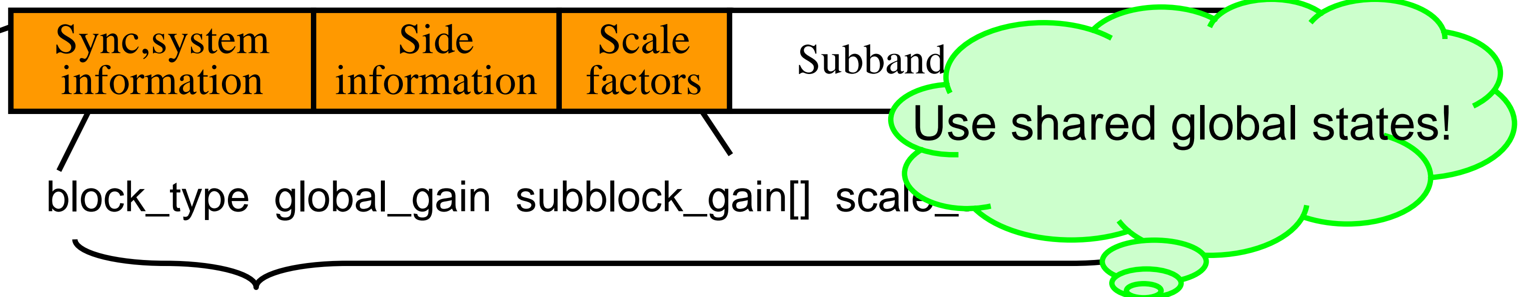
Data Dependent Iteration

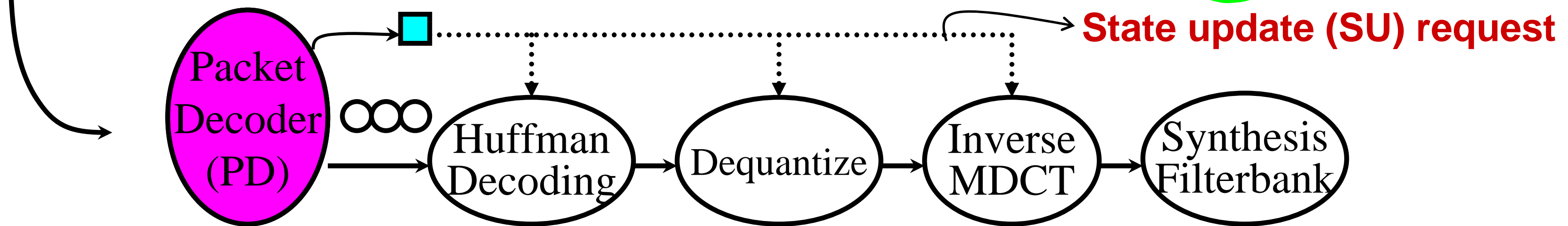
- **Dynamic dataflow: sample rate changes at run-time**



Need of Global States

- **arises in many multimedia applications.**
  - MP3 decoder, OpenGL machine



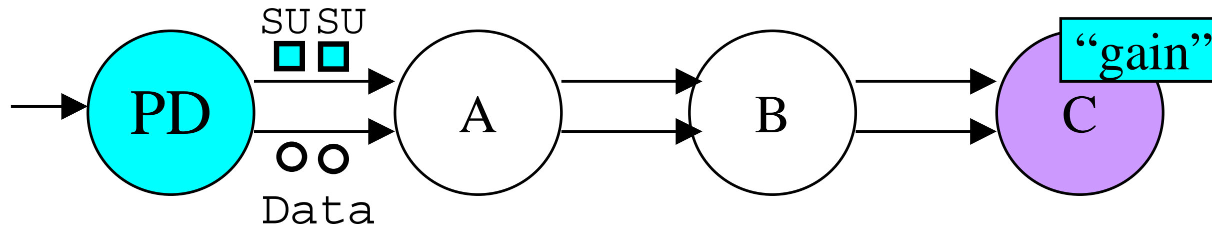


MPEG-1 decoder structure

codesign environment

Ad-hoc Approach (1)

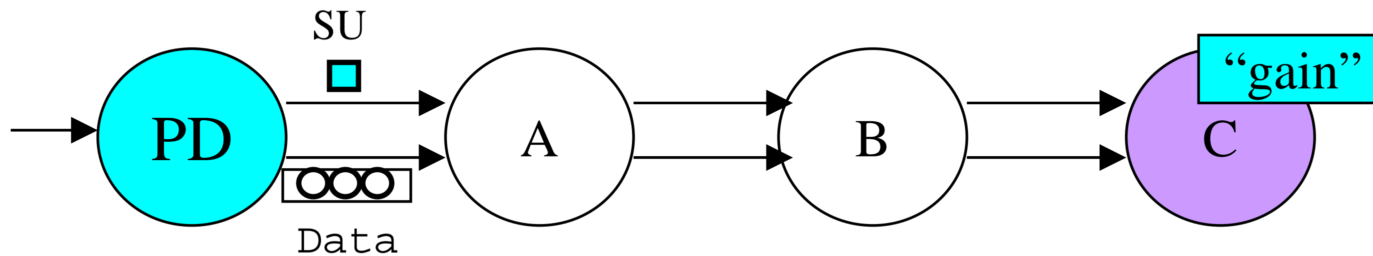
- **To duplicate a state update (SU) request for each data sample**



  - Redundant copy problem
  - Block reuse problem

Ad-hoc Approach (2)

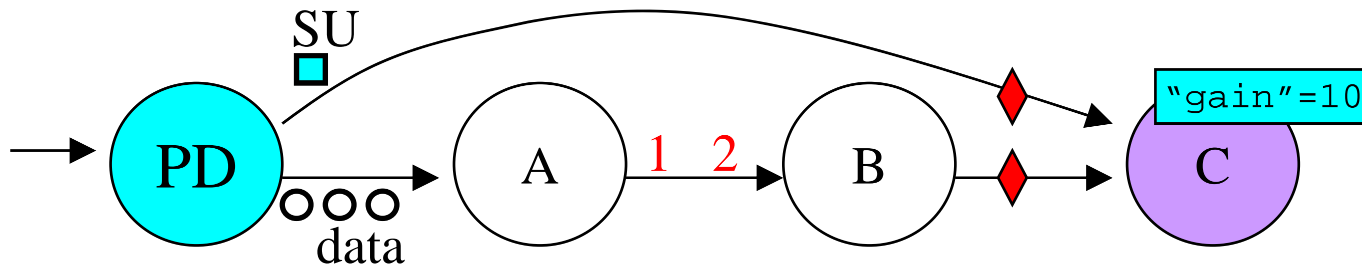
- **To group data samples as a unit to remove redundant copy**



- Large buffer problem
- Block reuse problem

Ad-hoc Approach (3)

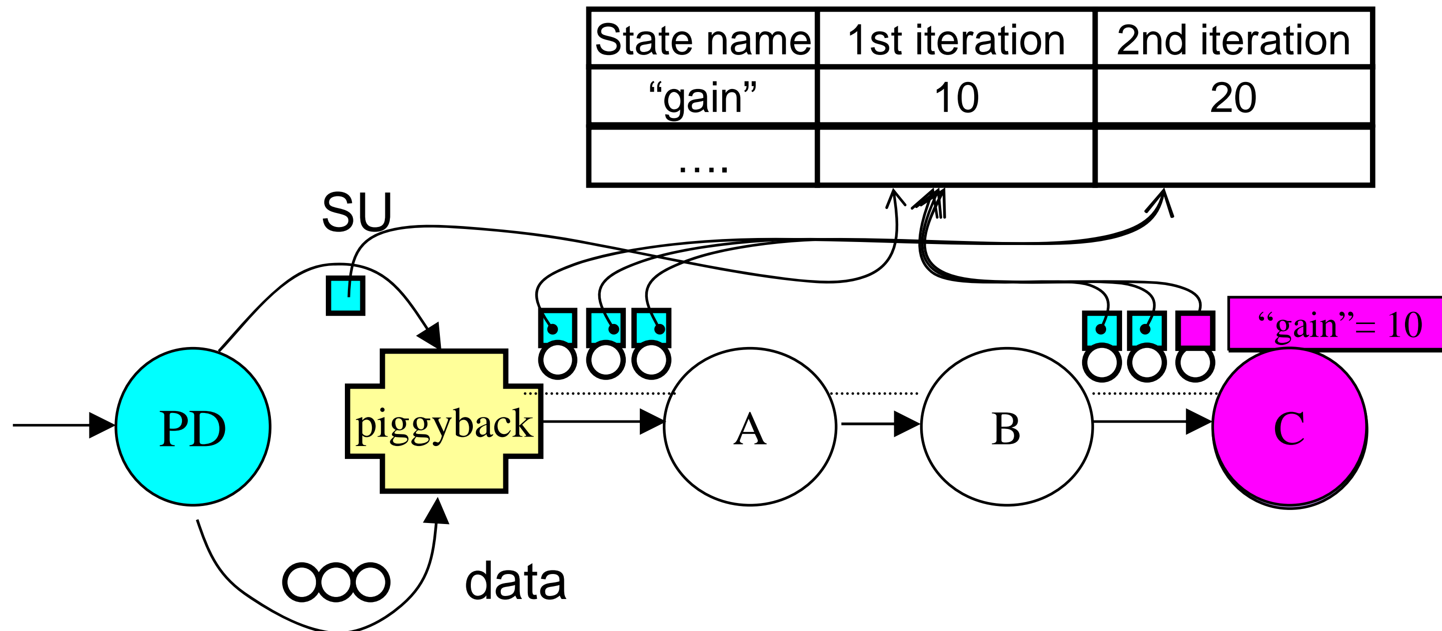
- **To bypass the intermediate nodes**



- Synchronization problem: initial sample, multirate
- Redundant copy problem
- Still have the block reuse problem for node C
- Visibility problem in a hierarchical graph

Synchronous Piggybacked Dataflow

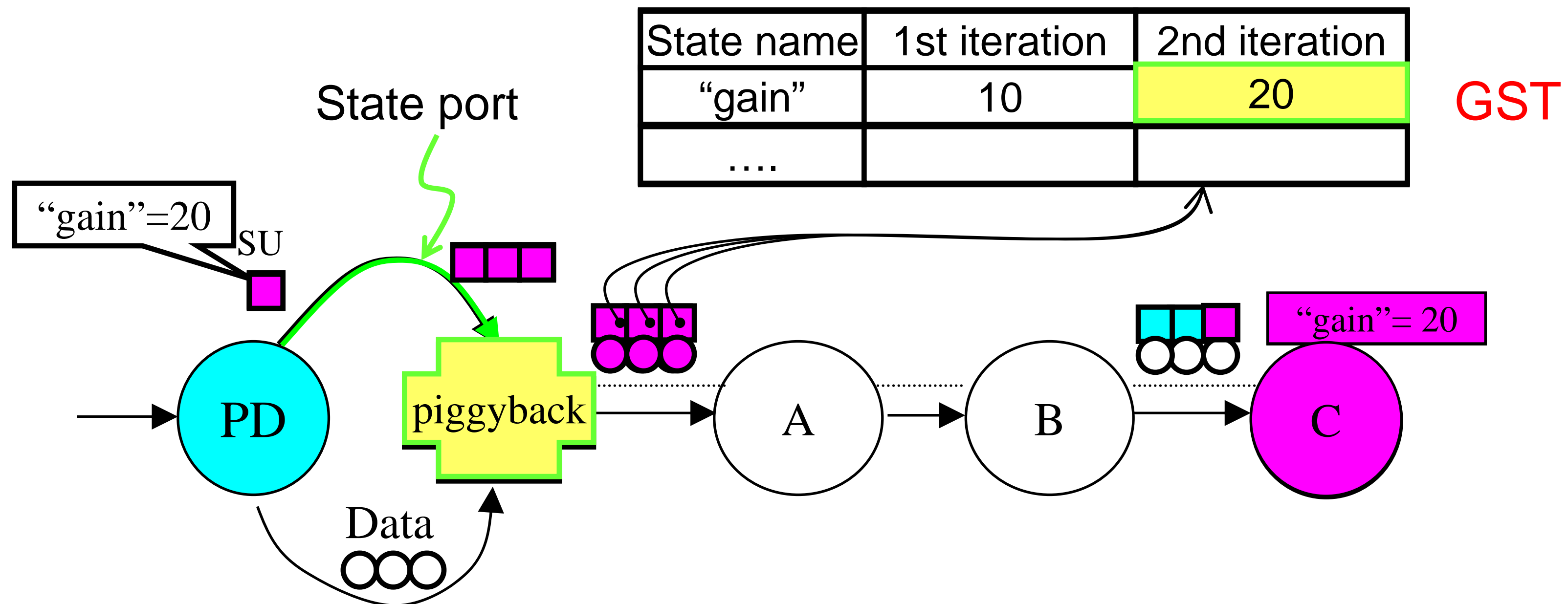
- global structure for global states with limited access
- each data sample is piggybacked with an SU request
- a block updates its internal parameters depending on applicability of SU request



State name	1st iteration	2nd iteration
“gain”	10	20
….		

Extensions to SDF

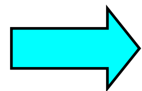
- **Global State Table (GST)**
  - maintain the outstanding values of global states
- **Piggybacking Block (PB)**
  - models the coupling of data samples and the pointers to the GST
  - single source of state update



State name	1st iteration	2nd iteration
“gain”	10	20
….		

SPDF Solution

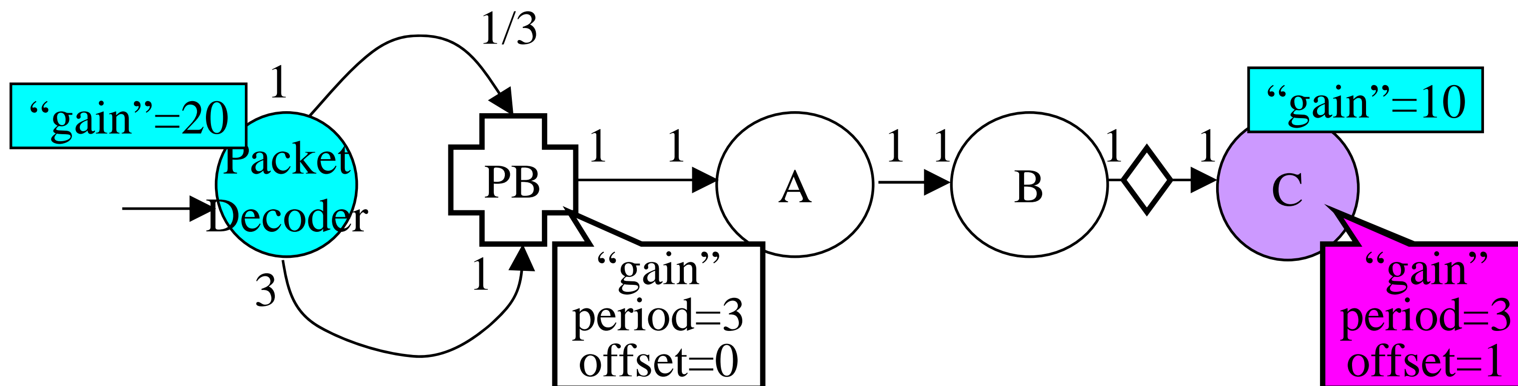
- **Block reusability**
- **Synchronization**
- **Buffer size from grouping**
- **Redundant copy problem remains to be unsolved!**
- **Overhead of extracting GST-related information from data sample!**



We use “Scheduler Assistance”

Scheduler-Assisted Code Synthesis

- **Period**
  - the repetition period of updating the states in terms of node's execution.
- **Offset**
  - the starting position of data samples to which the state update is coupled.



codesign environment

Scheduling Procedure

- **Examine Piggyback star**
- **Identify stars with global state**
- **Starting from each piggyback star**
  - 1) propagate { period, offset } pair to the target star
  - 2) construct data structure for global states
  - 3) attach preamble for the target star

PD,SC,3(**PB**,A,B,(**SU**,C))

Scheduling result

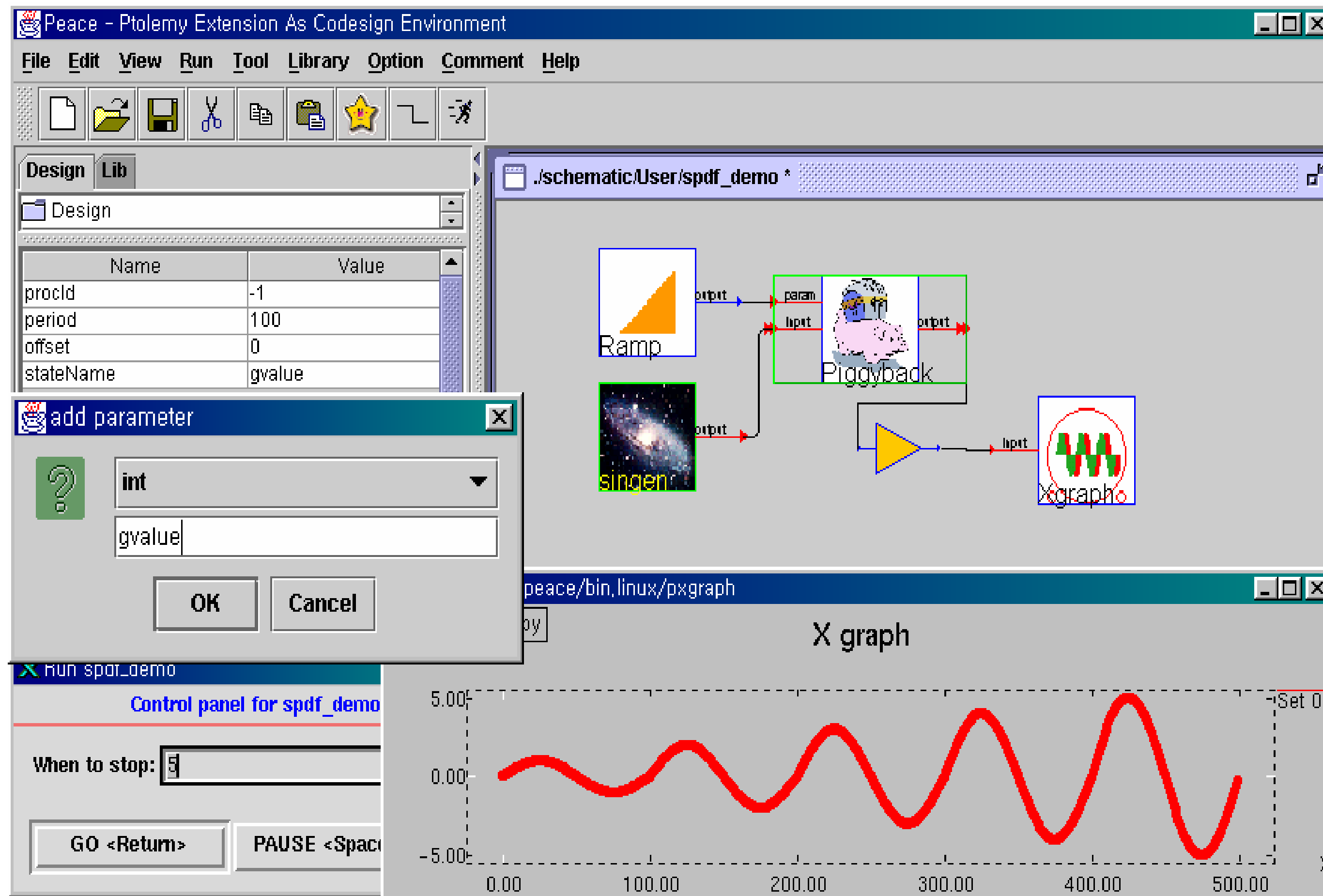
```
If (count==offset){
  new_value = read_state_port();
  update_GST("gain",new_value);
}
data_out[]=data_in[];
if (++count >= period) count=0;
```

PB code

```
{ // SU request code
if (count==offset)
  gain=read_GST("gain");
if (++count >= period) count=0;
}
```

SU code

Experimental Results



codesign environment

Dynamic Construct with SPDF

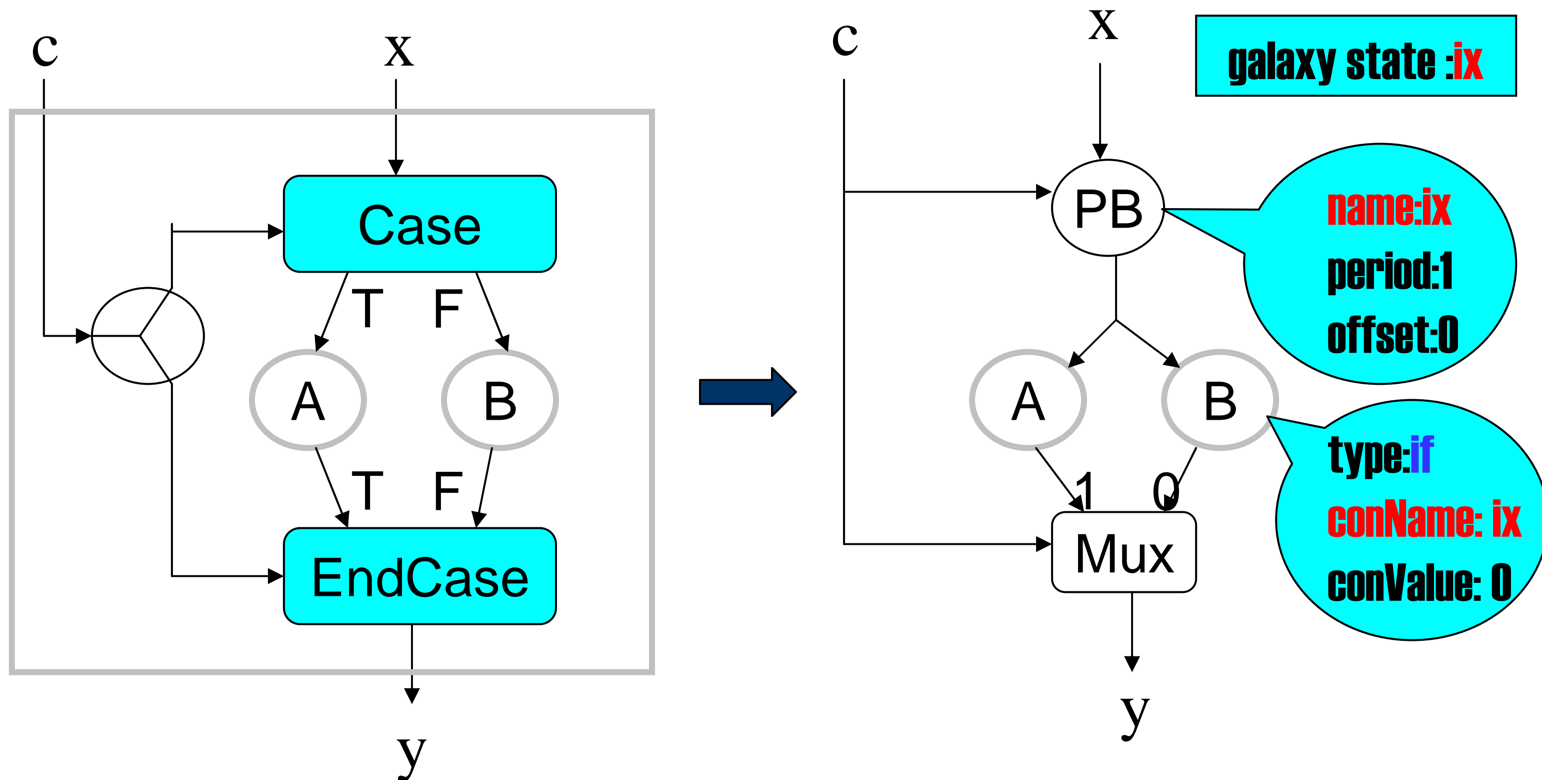
- **How to represent dynamic constructs graphically?**
  - if-then-else construct
  - data-dependent iteration: for, do-while

- **Wish lists**
  - Slight extension of SDF model
    - Preserve static analysis capability of the SDF model
  - Ease of use
  - Efficient implementation

codesign environment

SPDF – If construct

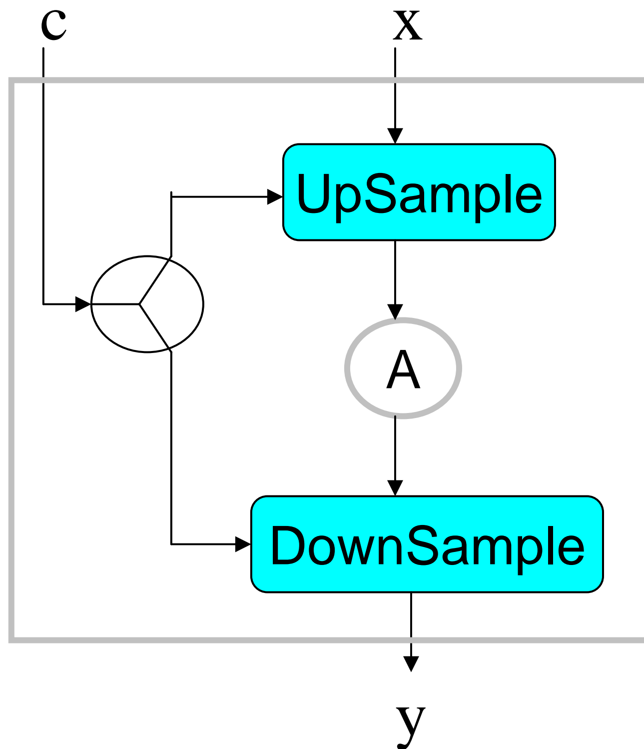
- Piggybacking the control variable
- Special galaxy definition for the body of DC

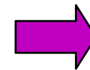


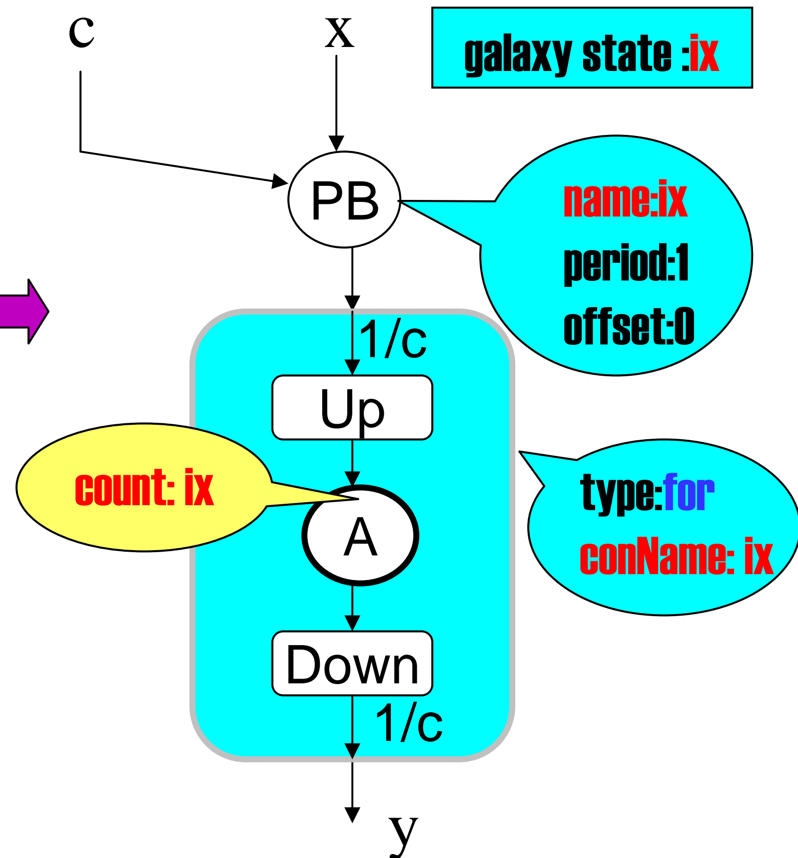
codesign environment

For construct

- **Dynamic FRDF star at the boundary**







codesign environment

SPDF Wrap-up

- **We extended SDF model so that it can allow global states without any side-effects.**
- **We implemented efficient code synthesis with the scheduler's assistance.**
- **Can enlarge the domain of application where dataflow representation can be used for rapid system prototyping.**

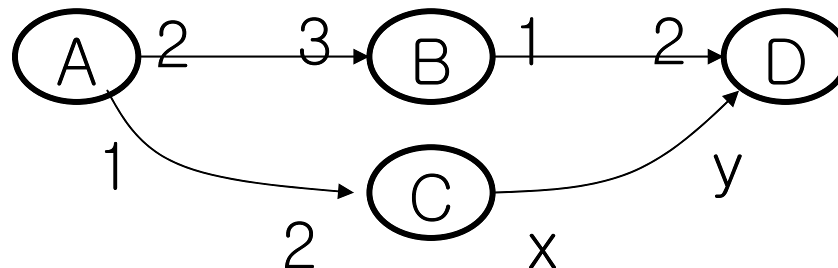
PeaCE codesign environment

Summary

- **Dataflow Model is a formal model of computation that has been successfully used for DSP algorithm specification in particular.**
- **SDF has a nice feature of static analyzability that determine the static schedule and resource requirement**
- **But SDF has some weaknesses to be used for general purpose**
  - Restricted expression capability
  - No shared memory
- **To overcome these weaknesses, novel extended models have been proposed: CSDF, FRDF, SPDF.**
- **Refinement from SDF to SW/HW implementation guarantees the correctness of the implementation by “correct-by-construction” principle.**

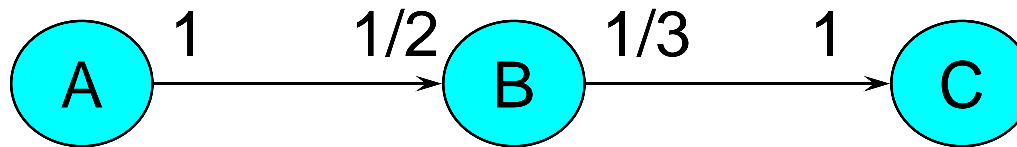
Questions

- **1. Explain why it is better to use SDF model than C-code when exploring the design space.**
- **2. Answer the followings. (numbers on arcs indicate the number of samples consumed/produces per node execution).**
  - (a) Find the smallest integers of x, y to make the SDF graph consistent.
  - (b) With the answers of (a), compute the repetition counts of all nodes in a single iteration.
  - (c) Find an optimal schedule with minimum buffer requirement and the required buffer size of all arcs.



Continue…

- **3. What is the "Correct-by-construction" principle? Can you guess how to keep this principle when refining the SDF graph to H/W?**
- **4. Answer the following questions for the FRDF graph below.**



  - (a) Find the repetition counts of all nodes.
  - (b) Find an optimal schedule with minimum buffer requirement.
  - (c) Draw a CSDF graph with the same buffer requirement and the same schedule.
- **5. Why can't we use shared memory in an SDF graph between nodes? How does the SPDF model solve this problem?**