VLSI System Design
Part V : High-Level Synthesis(1)

Lecturer : Tsuyoshi Isshiki

Dept. Information and Communications Engineering,
Tokyo Institute of Technology

isshiki@ict.e.titech.ac.jp

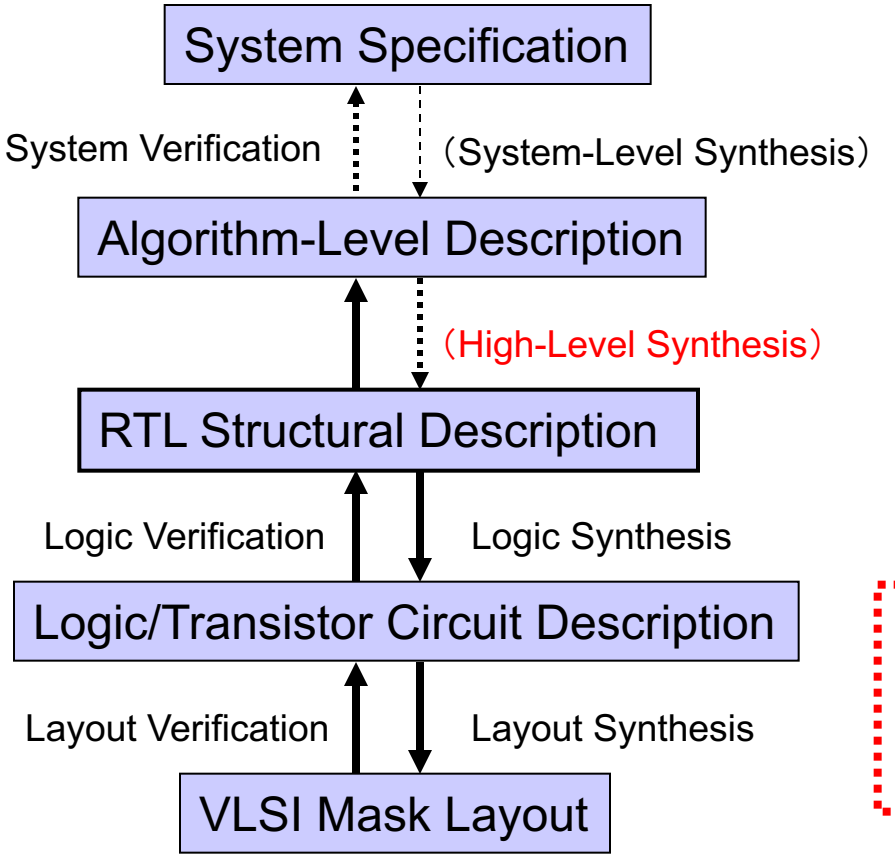
High-Level Synthesis and Its Design Environment

- High-level synthesis converts the design described at algorithm-level to RTL

→ Needs a rich module library consisting of high-level circuit blocks (adders, multipliers, ALUs, decoders, RAM, ROM, etc.) as well as high quality standard cell library

→ Numerous circuit implementation techniques for arithmetic logic are captured into these libraries, and high-level synthesis provides a path for utilizing these design resource efficiently and intelligently.

CAD Technology in VLSI Design



Synthesis tools : transformation of a design description into a more detailed form of description (logic synthesis, layout synthesis)

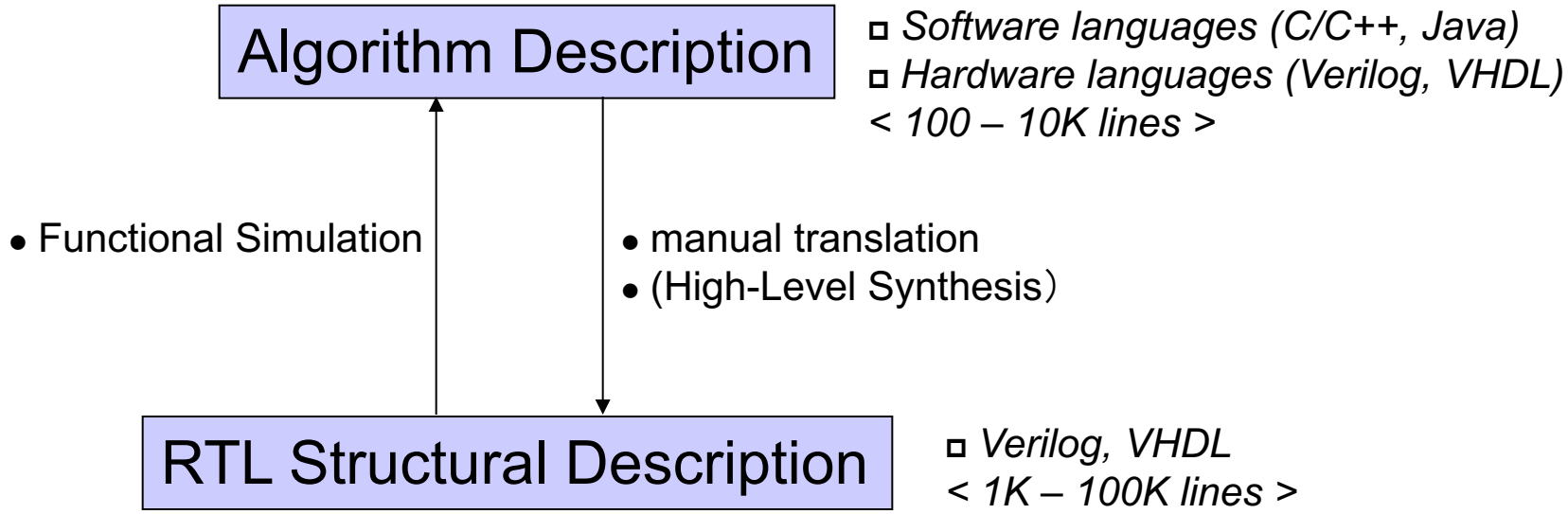
Verification tools : checking the correctness of the description (simulators, symbolic verification)

– ***Logic synthesis*** and ***layout synthesis*** tools have matured enough to be used by most designers

– ***High-level synthesis*** tools started to appear in real design cases (but many designers still prefer RTL as their design entry)

– ***System-level synthesis*** tools do not yet exist. (currently an active research area)

High-Level Synthesis/Verification



- ✓ Architecture description
  - ➢ Module (CPU, memory, register, functional unit, IO interface)
  - ➢ Bus architecture
- ✓ Module description (functional/structural)
  - ➢ Combinational/sequential circuit description

Register-Transfer Level Description and Synthesis

- *Register-transfer level description* specifies the sequence of events at each clock cycle at each circuit block
  - Design description at *architecture level*
  - Synthesis parameters (constraints/cost function) : clock period, circuit size, power
  - ✓ Advantage : full control of architecture specification and cycle-accurate behavior specification
    - → Exploit various forms of process parallelization techniques
    - → Accurate system level simulation (with external hardware devices) is possible
    - → Synthesis tools (logic synthesis, technology mapping, cell library generator) are mature
  - ✓ Disadvantage : time consuming
    - → Concurrent behavior of RTL can be hard understand (Bugs are easily introduced, but hard to keep track)
    - → RTL verification through simulation is time-consuming
    - → Requires experience in hardware design

Behavioral Level Description and High-Level Synthesis

- *Behavioral level description* specifies the sequence of computations (such as in software programs)
  - Design description at *algorithm level*
  - Synthesis parameters (constraints/cost function) : # functional units, # registers, # control steps
  - ✓ Advantage :
    - → Explore design space more efficiently (by changing synthesis parameters, trying different algorithms)
    - → Less time consuming (less number of codes, easy to debug, fast simulation, does not require much hardware design experience)
  - ✓ Disadvantage : architecture determined by the (somewhat inmature) synthesis tool
    - → May produce less efficient architecture compared to manual architecture design (current high-level synthesis tools usually produce Very-Long Instruction Word (VLIW) architecture only)

Background of High-Level Synthesis

- Processor technology
  - Instruction-level parallelism
  - Very-Long Instruction Word (VLIW) architecture
    - ✓ Issues multiple operations on multiple functional units
- Digital signal processing
  - Input description : signal-flow graph
    - ✓ Edge : signal
    - ✓ Vertex : operator (add/subtract, multiply)
    - ✓ Behavior : repetitive process
  - Hardware compilation
    - ✓ Direct mapping : allocate hardware to each operator
    - ✓ Resource shared mapping : allocate hardware to multiple operators using VLIW architecture

Applications for High-Level Synthesis

- Digital signal processing
  - ✓ Filtering : audio, image, data transmission (wired/wireless)
  - ✓ Transformation : DCT, FFT, wavelet, etc.
  - ✓ Codec : audio/video compression, decompression
- Graphics Engine
- Communication
  - ✓ Network switching
  - ✓ Protocol
  - ✓ Terminal, modem
- Embedded controllers
  - ✓ Motors
  - ✓ Sensors

High-Level Synthesis Flow

A) Design capture (HDLs, C/C++, signal-flow graph, etc)

B) Compilation to internal representation
- Data-flow graph (DFG)
- Control-flow graph (CFG)
- Control-data-flow graph (CDFG)

C) Resource allocation
- Specify available functional units

D) Operation scheduling
- Assign each operation to control steps

E) Resource binding
- Assign each data to registers
- Assign each operation to functional units

Design Example : IIR Filter (1)

- Infinite Impulse Response (IIR) filter

$$w(t) = x(t) + b1 \cdot w(t-1) + b2 \cdot w(t-2)$$
$$y(t) = a0 \cdot w(t) + a1 \cdot w(t-1) + a2 \cdot w(t-2)$$

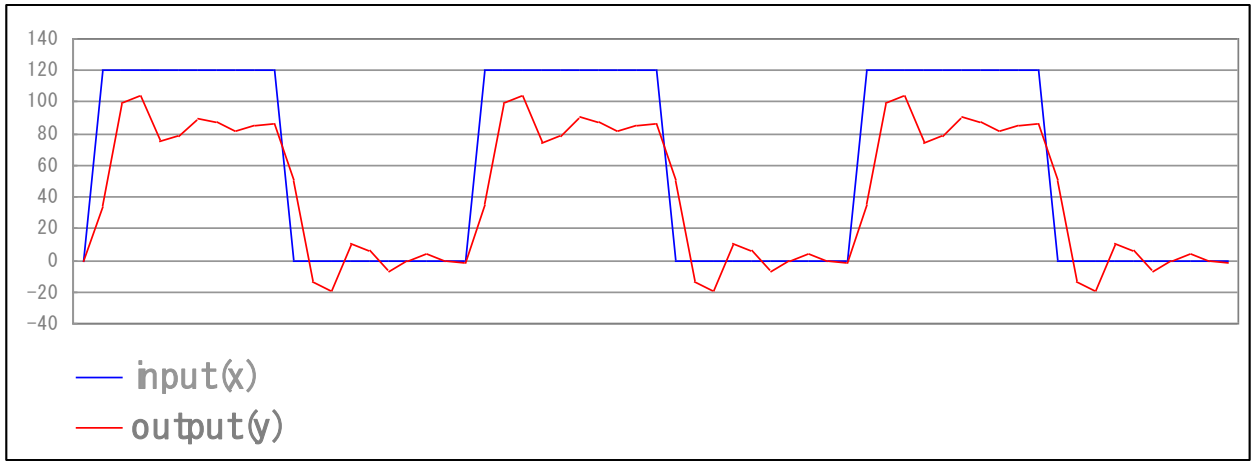
difference equations for IIR filter

a0, a1, a2, b1, b2 : filter coefficients

x(t) : input signal at time t

y(t) : output signal at time t

w(t), w(t – 1), w(t – 2) : internal signals (states)



low-pass filter

a0 = 0.2958
a1 = 0.5876
a2 = 0.2958
b1 = – 0.2138
b2 = – 0.4518

Design Capture : Signal-Flow Graph

- Infinite Impulse Response (IIR) filter

$$w(t) = x(t) + b1 \cdot w(t-1) + b2 \cdot w(t-2)$$

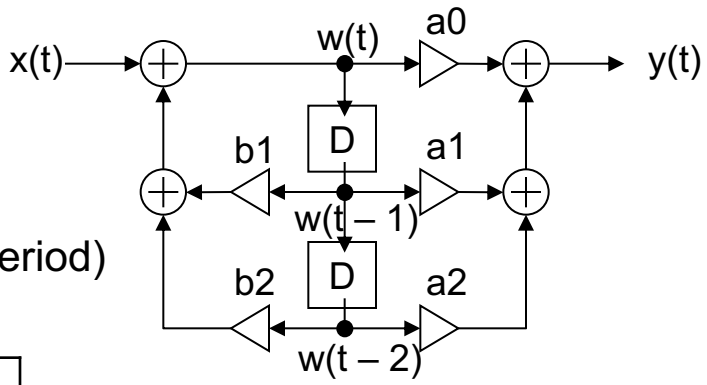
$$y(t) = a0 \cdot w(t) + a1 \cdot w(t-1) + a2 \cdot w(t-2)$$

a0 = 0.2958
a1 = 0.5876
a2 = 0.2958
b1 = – 0.2138
b2 = – 0.4518

⊕ : addition

◁ : multiplication (by constant)

D : sampling delay
(signal is delayed by 1 sampling period)



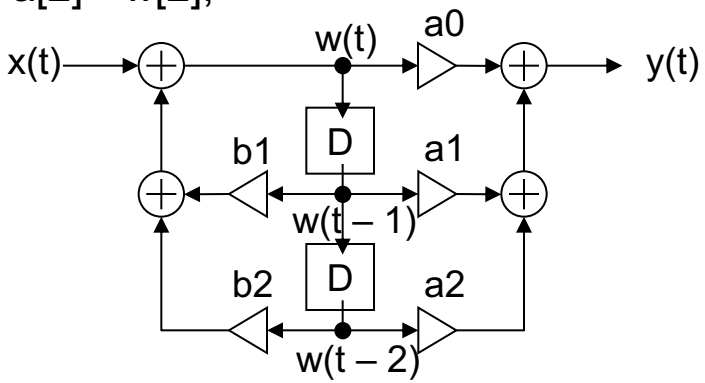
signal flow graph

t	0	1	2	3	4	5
w(t)	w(0)	w(1)	w(2)	w(3)	w(4)	w(5)
w(t–1)	–	w(0)	w(1)	w(2)	w(3)	w(4)

sampling period

Design Capture : C Language

```
void IIR()
{
        extern const float a[ ], b[ ]; // filter coefficients
        int x, y; // input and output signals
        int w[3] = {0, 0, 0}; // internal signals (states)
        extern int enable; // assume "enable" is controlled by different program thread
        while (enable != 0){
                w[2] = w[1];
                w[1] = w[0];
                x = GetInput ();
                w[0] = x + b[1] * w[1] + b[2] * w[2];
                y = a[0] * w[0] + a[1] * w[1] + a[2] * w[2];
                PutOutput (y) ;
                }
        }
}
```

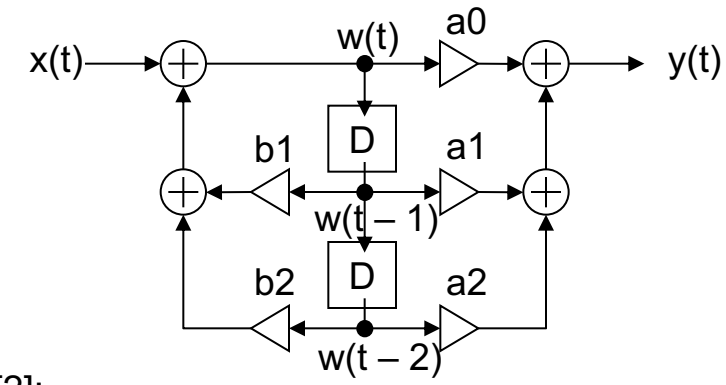


signal flow graph

Design Capture : Verilog

Signed integer (2's complement) cannot be directly implemented in Verilog

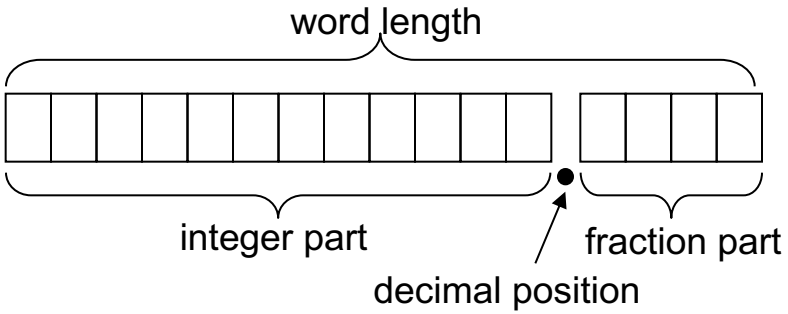
```
module IIR(x, y, xready, yready);
input [15:0] x; // port signals can only be unsigned
input xready;
output [15:0] y; // unsigned : needs to be reinterpreted to signed number externally
output yready;
integer w[0:2]; // signed integer
reg yready;
parameter a0 = 0.2958, a1 = 0.5876, a2 = 0.2958;
parameter b1 = – 0.2138, b2 = – 0.4518;
initial begin
        w[0] = 0; w[1] = 0; w[2] = 0; yready = 0;
end
always@(xready) begin
        if(xready == 1) begin
                w[2] = w[1];
                w[1] = w[0];
                w[0] = x + b1 * w[1] + b2 * w[2];
                y = a0 * w[0] + a1 * w[1] + a2 * w[2];
                yready = 1;
        end
        wait(xready == 0) yready = 0;
end
endmodule
```



signal flow graph

Limitations in Current Design Capture Environment (1)

- Numerical accuracy problem
  - ✓ Digital signal processing theory : based on real numbers
  - ✓ Digital implementation : fixed-point numbers
    - ➢ Quantization noise, overflow
    - ➢ *Major effort on algorithm design is to determine the optimal word length and the decimal position for each data (these decisions directly affect the hardware complexity)*
    - ➢ Need to explicitly describe bit-shift operations to implement the required word lengths and decimal positions (BUT decimal positions are implicit)
  - ✓ *State-of-the-art Digital Signal Processors (DSPs) have floating-point units because of these complications*



word lengths and decimal positions may vary for different data

Limitations in Current Design Capture Environment (2)

- Limitation in behavioral description
  - ✓ Software language (C, Java) :
    - ➢ Concurrent behavior
    - ➢ Interfacing with external hardware device
  - ✓ HDL (Verilog, VHDL) :
    - ➢ Difficult to describe the behavior based on event-triggered processes
    - ➢ May need to introduce signals for the sake of event-triggering which are not relevant to the actual behavior.
  - ✓ Signal-flow graph :
    - ➢ Cannot express control-flows (if-else, while, ...)
  - ✓ There are other behavioral description languages
    - ➢ Recent efforts based on C++ : SystemC, SpecC
- *Lack of an adequate behavioral description language is one of the major obstacle for the high-level synthesis to gain its popularity*

High-Level Synthesis Flow

A) Design capture (HDLs, C/C++, signal-flow graph, etc)

B) Compilation to internal representation
- Data-flow graph (DFG)
- Control-flow graph (CFG)
- Control-data-flow graph (CDFG)

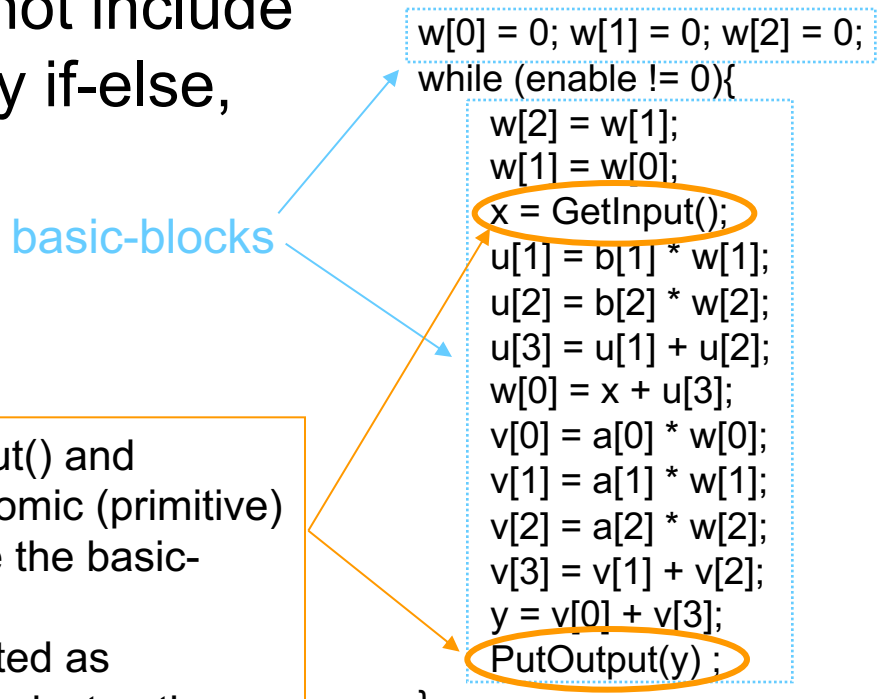
C) Resource allocation
- Specify available functional units

D) Operation scheduling
- Assign each operation to control steps

E) Resource binding
- Assign each data to registers
- Assign each operation to functional units

Control-Data-Flow Graph Generation (1)

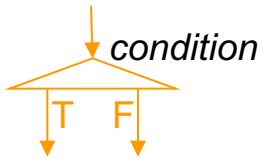
- *Basic-block* : a sequence of instructions which do not include jumps (such caused by if-else, for/while loop)



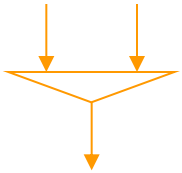
Here, IO function calls GetInput() and PutOutput(y) are treated as atomic (primitive) operations and included inside the basic-block.
Usually, function calls are treated as independent basic-blocks since instruction sequence jumps inside those functions.

Control-Data-Flow Graph Generation (2)

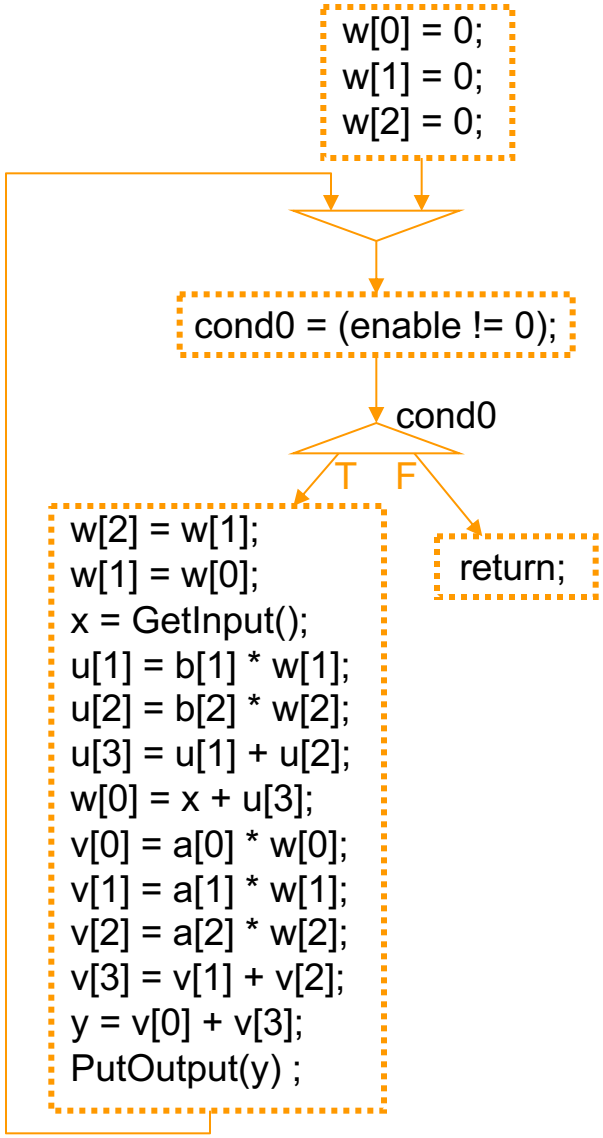
- *Control-flow graph* : specifies the control-flow of basic-block executions
  - Vertex : basic-block, branch, join
  - Edge : control-flow



branch node
(control fork node)

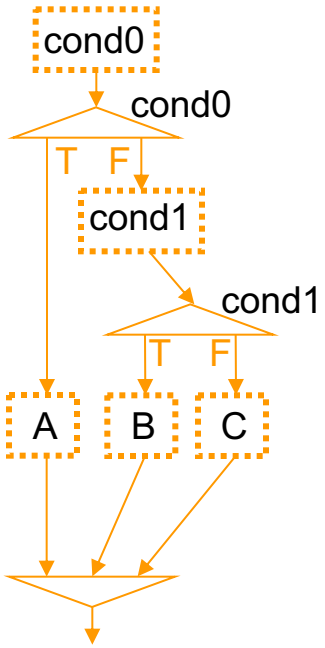


control join node

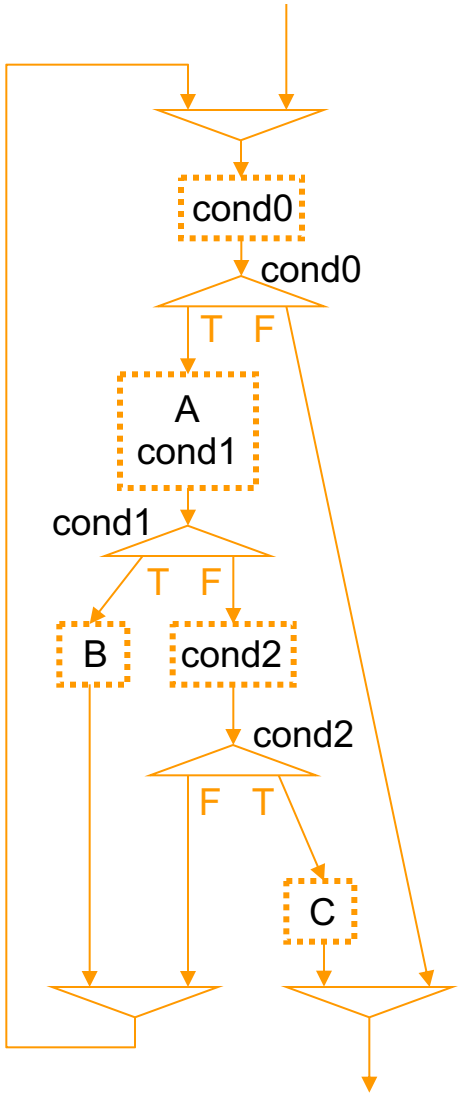


Control-Flow Graph Examples

```
if (cond0) A;
else if (cond1) B;
else C;
```

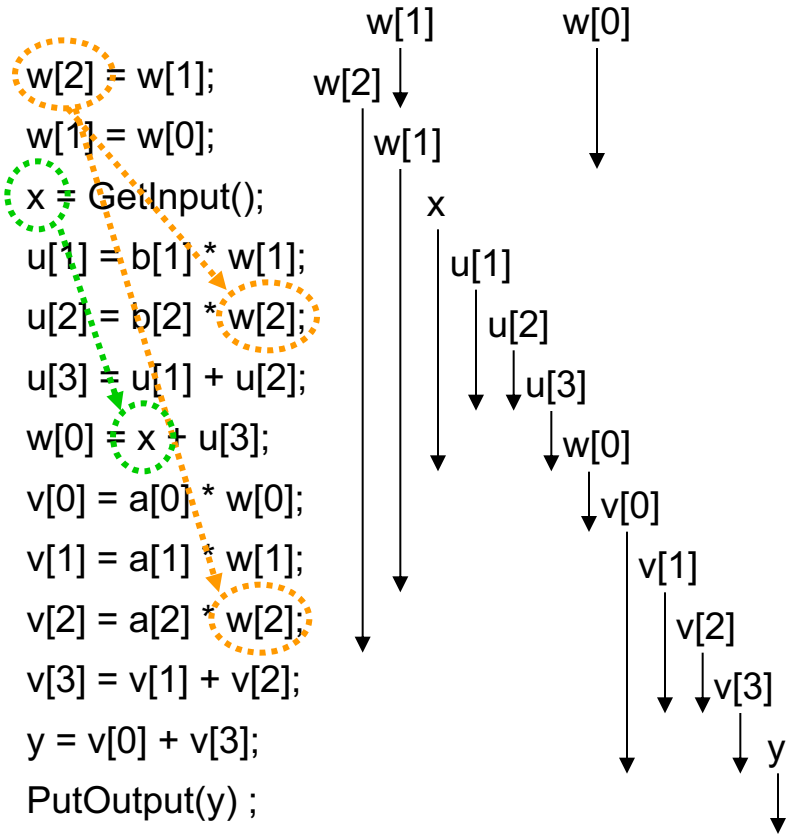



```
while (cond0){
        A;
        if(cond1){
                B;
                continue;
        }
        if(cond2){
                C;
                break;
        }
}
```



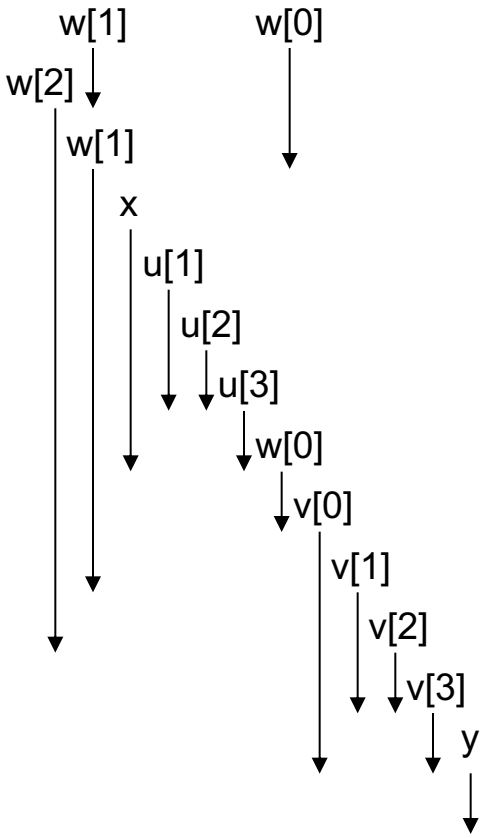
Control-Data-Flow Graph Generation (3)

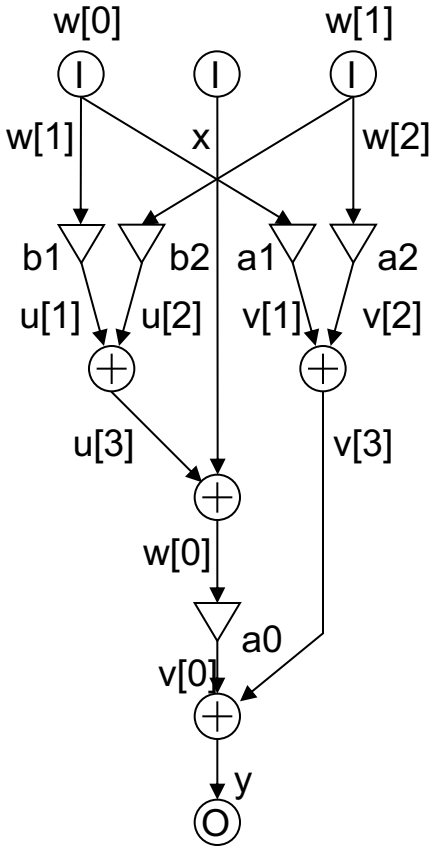
- *Data lifetime* within basic blocks:
  - Starts after the data assignment operation
  - Ends after the last operation using the data
- Here, the purpose of lifetime analysis is to distinguish disconnected lifetimes having the same data name (w[1] and w[0] have disconnected lifetimes)
- Data-flow arc generates from the data assignment operation and terminates to each operation using the data within its lifetime



Control-Data-Flow Graph Generation (4)

```
w[2] = w[1];
w[1] = w[0];
x = GetInput();
u[1] = b[1] * w[1];
u[2] = b[2] * w[2];
u[3] = u[1] + u[2];
w[0] = x + u[3];
v[0] = a[0] * w[0];
v[1] = a[1] * w[1];
v[2] = a[2] * w[2];
v[3] = v[1] + v[2];
y = v[0] + v[3];
PutOutput(y) ;
```



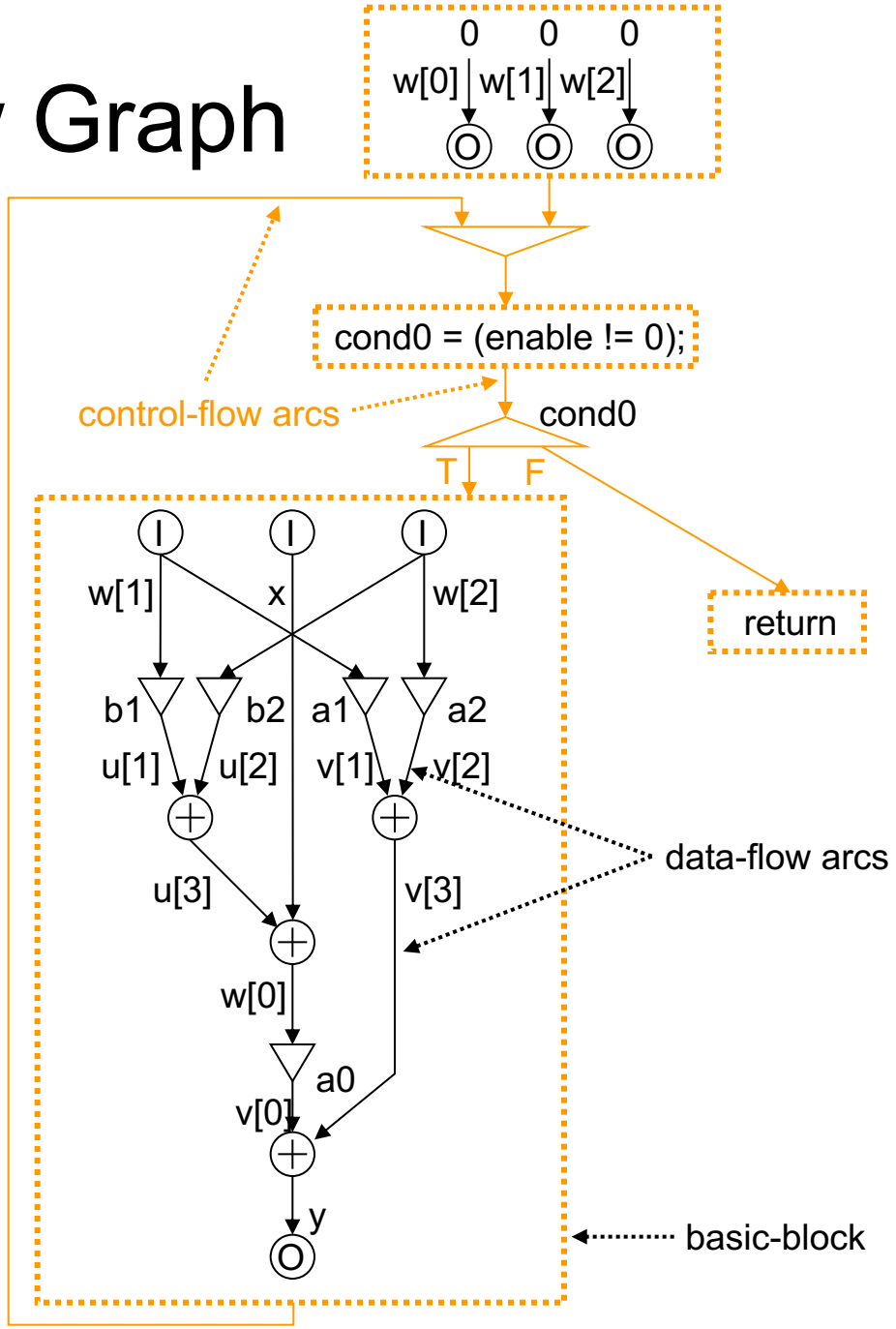


- Data lifetime without its assignment operation within the basic-block is connected to input node

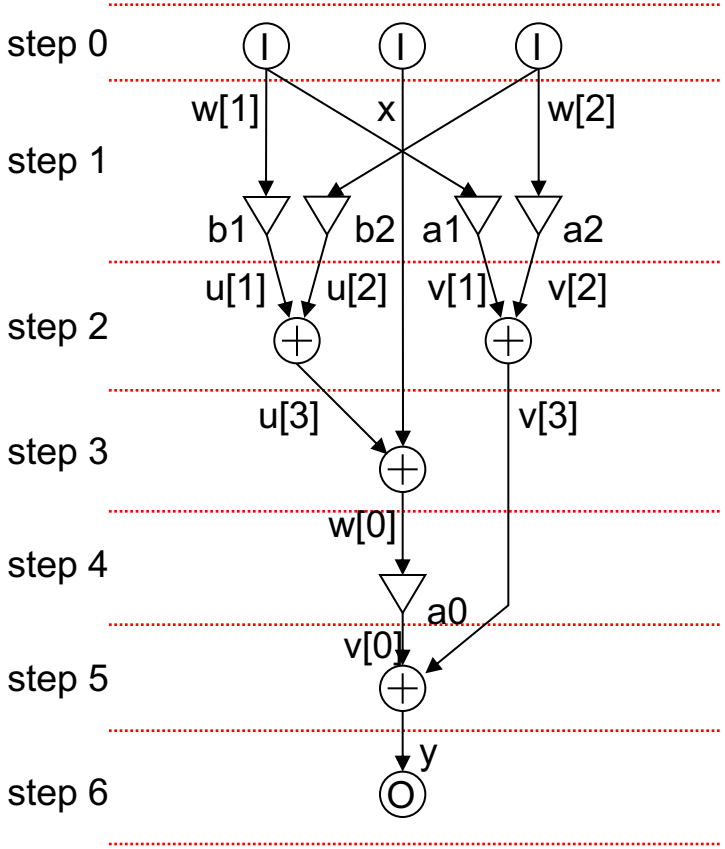
Control-Data-Flow Graph

- *Control-flow graph* (directed graph) : specifies the control-flow of basic-block executions
- *Data-flow graph* (directed acyclic graph) : specifies data dependencies among operations within the basic-block

```
w[0] = 0; w[1] = 0; w[2] = 0;
while (enable != 0){
        w[2] = w[1];
        w[1] = w[0];
        x = GetInput();
        u[1] = b[1] * w[1];
        u[2] = b[2] * w[2];
        u[3] = u[1] + u[2];
        w[0] = x + u[3];
        v[0] = a[0] * w[0];
        v[1] = a[1] * w[1];
        v[2] = a[2] * w[2];
        v[3] = v[1] + v[2];
        y = v[0] + v[3];
        PutOutput(y) ;
}
```



General Behavioral Model of Control-Data-Flow Graph (1)

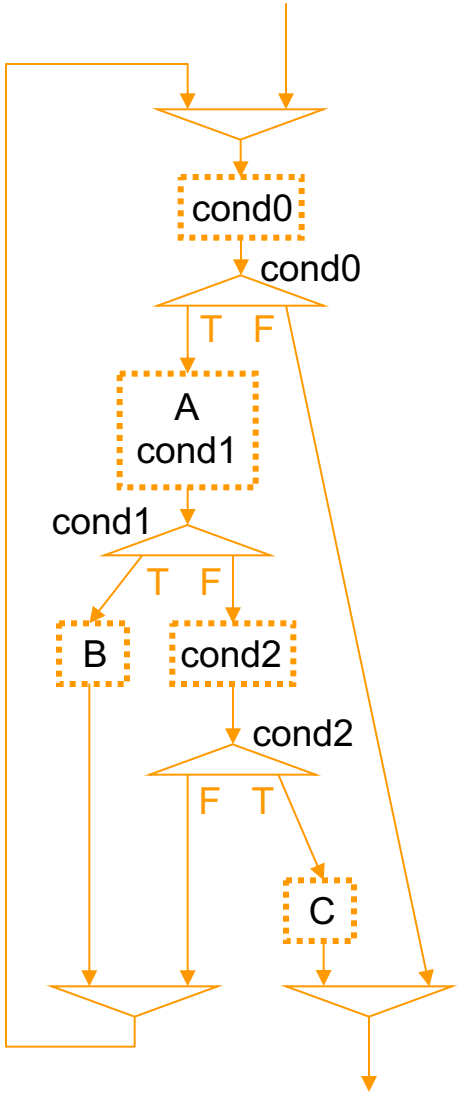


As-Soon-As-Possible (ASAP) Scheduling

- Data dependencies specified by the data-flow arcs determines the order of executions among operations :
  - Arc $op_i \rightarrow op_j$ indicates that op_j cannot execute until op_i is executed
  - An operation can execute if all the input data have already been computed
- Within the basic-block, operations can be executed in parallel as long as the execution order do not violate the data dependencies.
- Multiple basic-blocks cannot be executed in parallel (control-flows needs to be evaluated sequentially)

→ Parallelism is limited within the basic-block (Instruction-Level Parallelism)

General Behavioral Model of Control-Data-Flow Graph (2)



- Multiple basic-blocks cannot be executed in parallel (control-flows needs to be evaluated sequentially)

→ Parallelism is limited within the basic-block (Instruction-Level Parallelism)

→ Fundamental performance bottleneck for general high-level synthesis

→ Parallel compiler techniques such as speculative execution and loop unrolling can make the basic-blocks larger by moving operations across basic-block boundaries (code-motion)