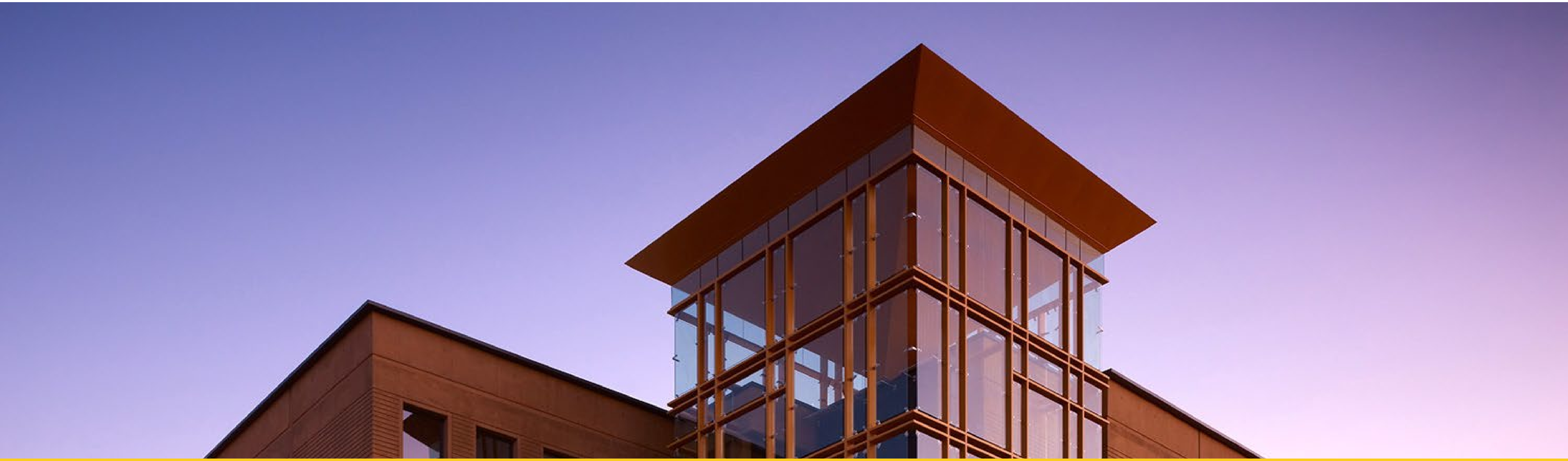
Lecture 5:
High-Level Synthesis

Sitao Huang

sitaoh@uci.edu

February 1, 2022

Slide courtesy of Prof. Deming Chen, UIUC, ECE 527: SoC Design



Logistics

- **Homework 1** released, due: ***January 31***, 11:59 PM on Canvas
- Homework 2 expected to release later this week, due February 7

- **Midterm**: **February 10** (Thursday), 8:00-9:20 AM (in class)
- Midterm review session / Q&A: February 8 (Tuesday)

- Project proposal due: February 14
    - Options: (a) literature review paper or (b) compiler + accelerator project

- In-person ***(hybrid)*** instruction starting next week (week 5)! Classroom: SSTR 101
- First in-person class: ***February 1***

High-Level Synthesis

- **High-level synthesis (HLS)**, also referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavior synthesis
- HLS takes an abstract behavioral specification of a digital system, and finds a register-transfer level structure that realizes the given behavior

INPUT:

- A high-level, algorithmic description
    - Control structures (if/else, loop, subroutines)
    - Concurrent and sequential semantics
    - Abstract data types
    - Logical and arithmetic operators
- A set of constraints
    - Speed, power, area, interconnect style
    - A library of pre-specified components

OUTPUT:

- A register-transfer level description for further synthesis and optimization

Reference: Register-Transfer Level (RTL), Abstract Data Type (ADT)

High-Level Synthesis

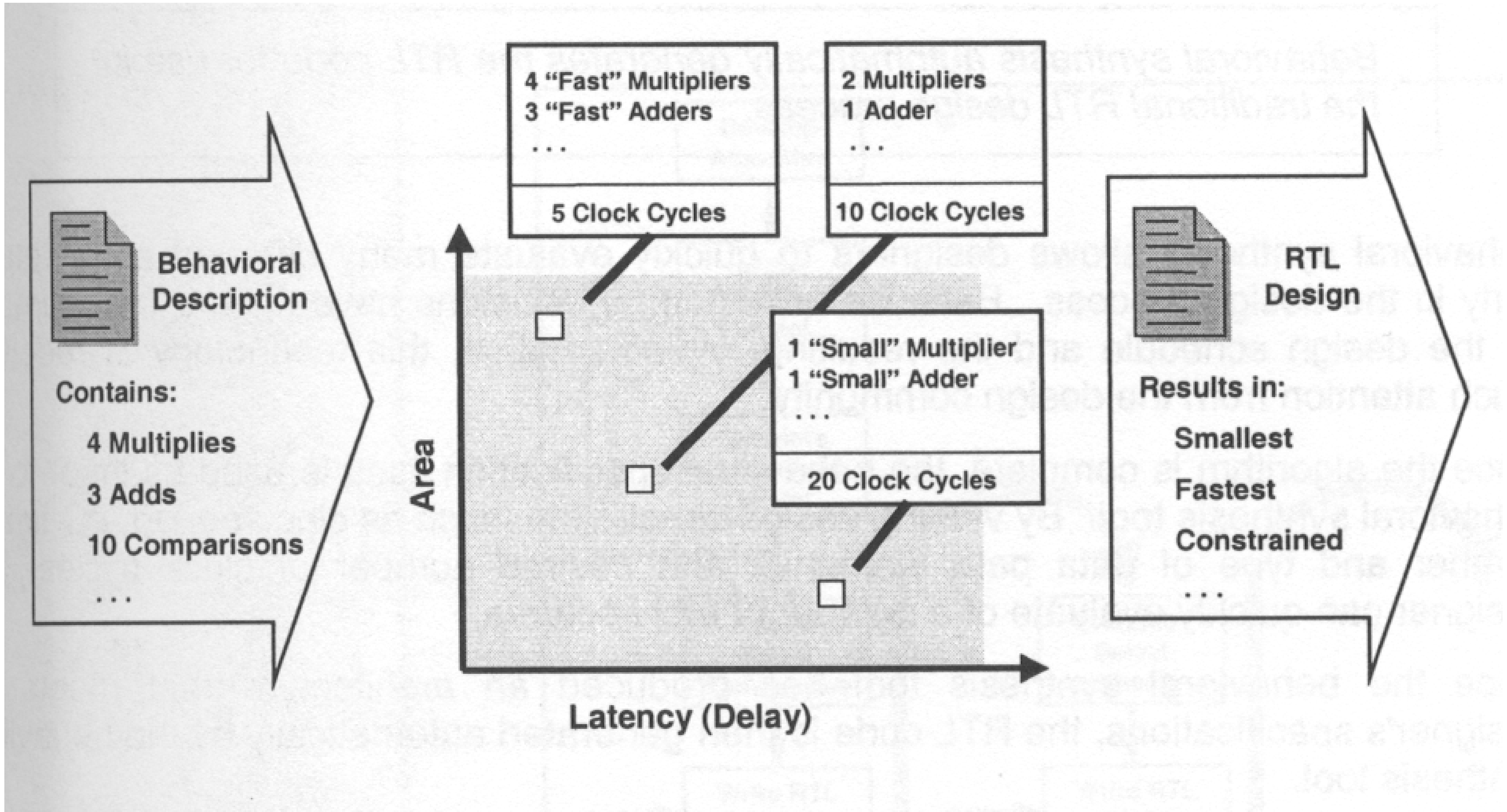
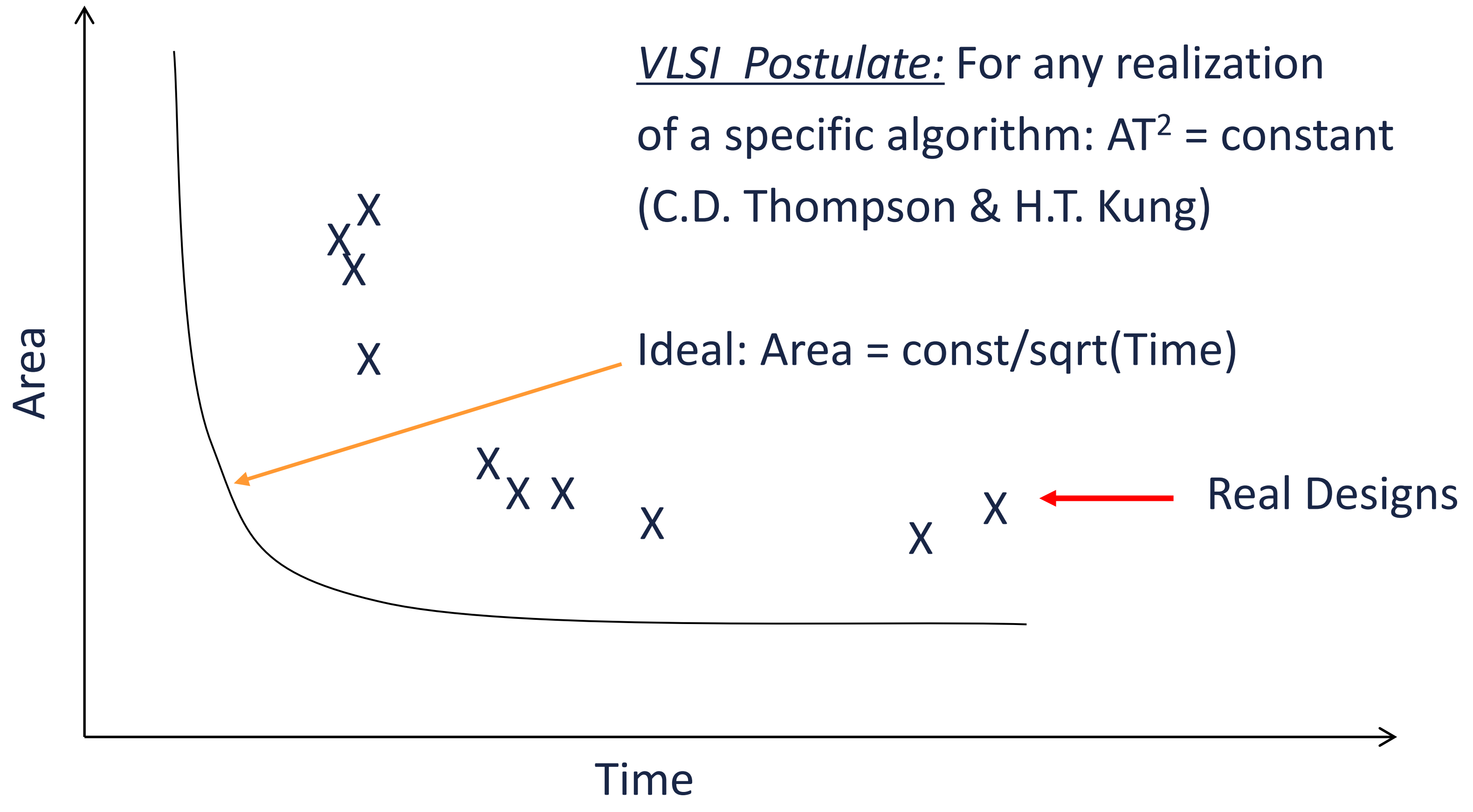


Figure 2-5: Architectural alternatives from a behavioral description

John P Elliott, "Understanding Behavioral Synthesis: A Practical Guide to High-Level Design", 1999.

Design Space Exploration

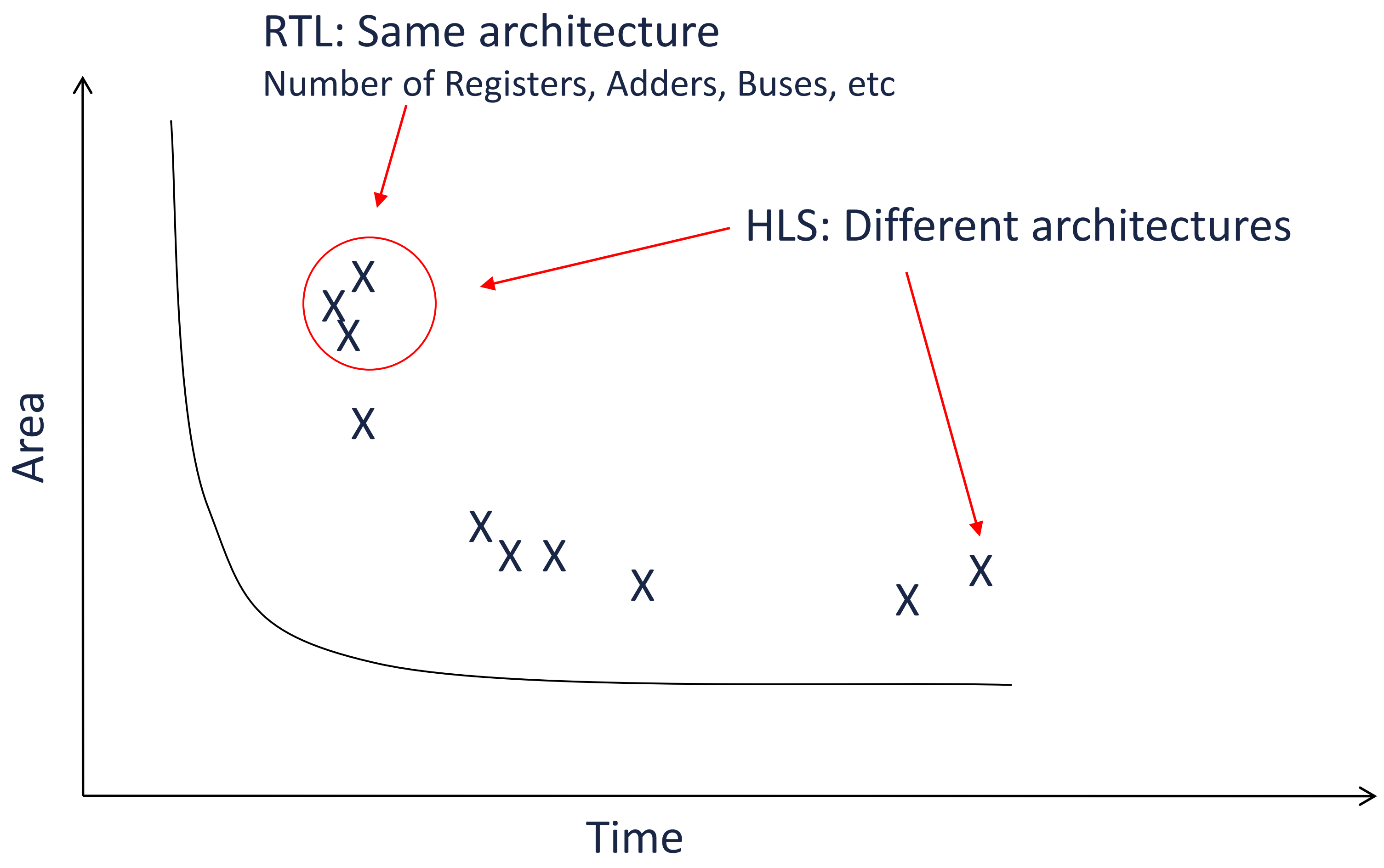


VLSI Postulate: For any realization of a specific algorithm: AT^2 = constant (C.D. Thompson & H.T. Kung)

Ideal: Area = const/sqrt(Time)

Add the Power tradeoff and this becomes a 3-dimensional graph

Design Space Exploration: RTL vs. Behavioral Synthesis



High-Level Synthesis Process

Resource Allocation:

- Allocating resources (library components) to each of the operations, buses, muxes, and registers for storage

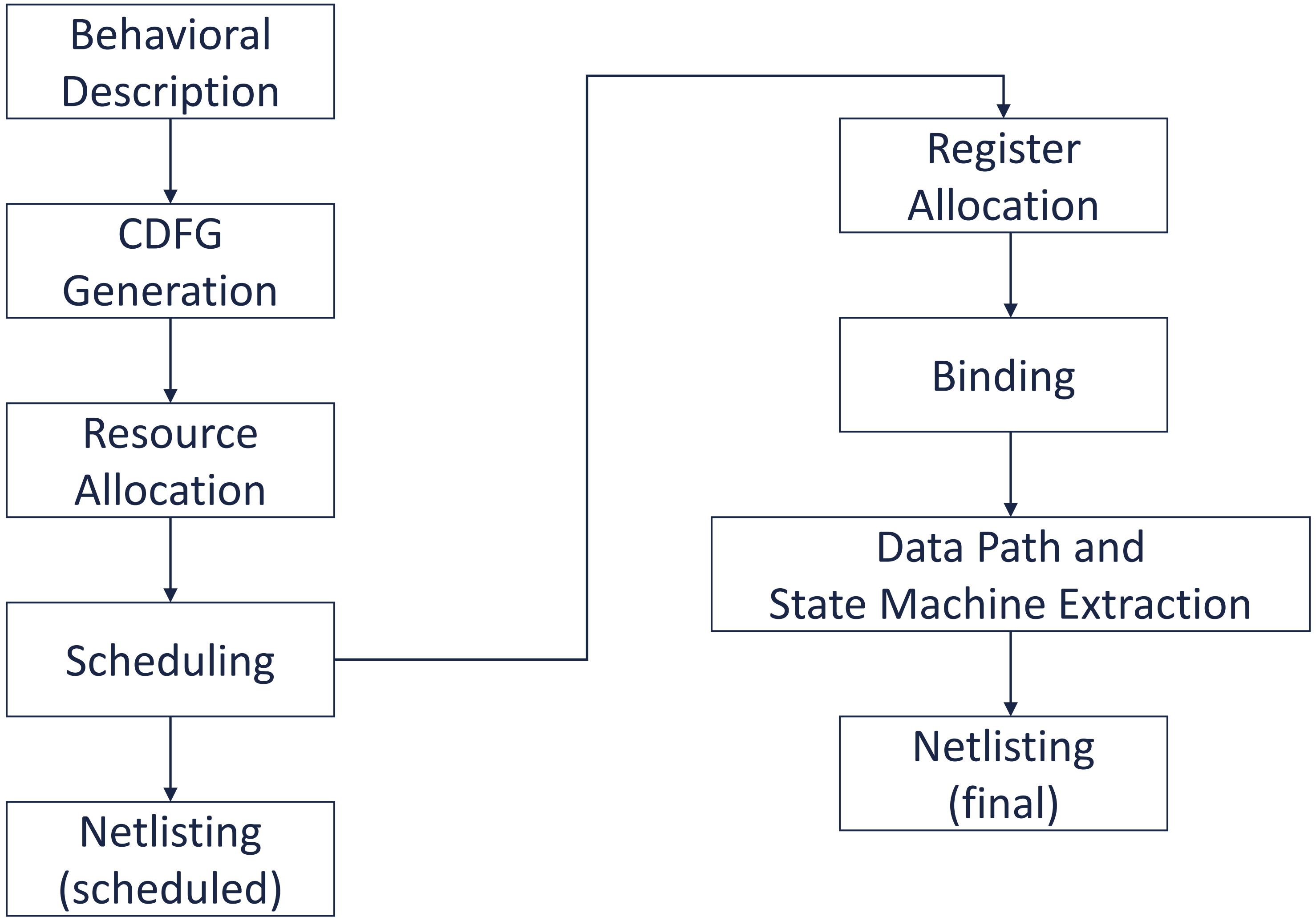
Scheduling:

- Scheduling the operations in the CDFG to minimize area, time and/or power

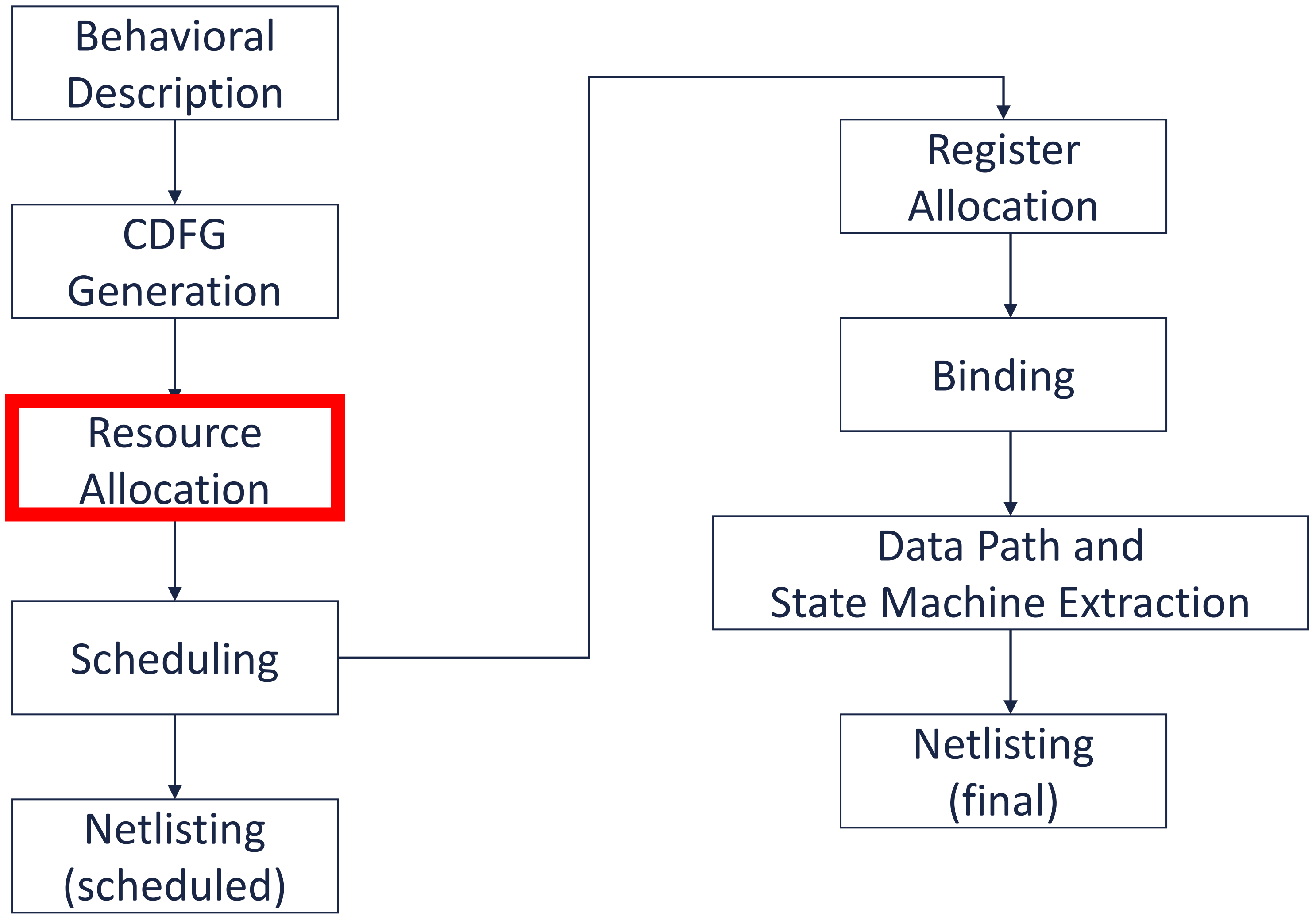
Binding:

- Determining the time of use of each component
- e.g., which registers to use and when

High-Level Synthesis Steps



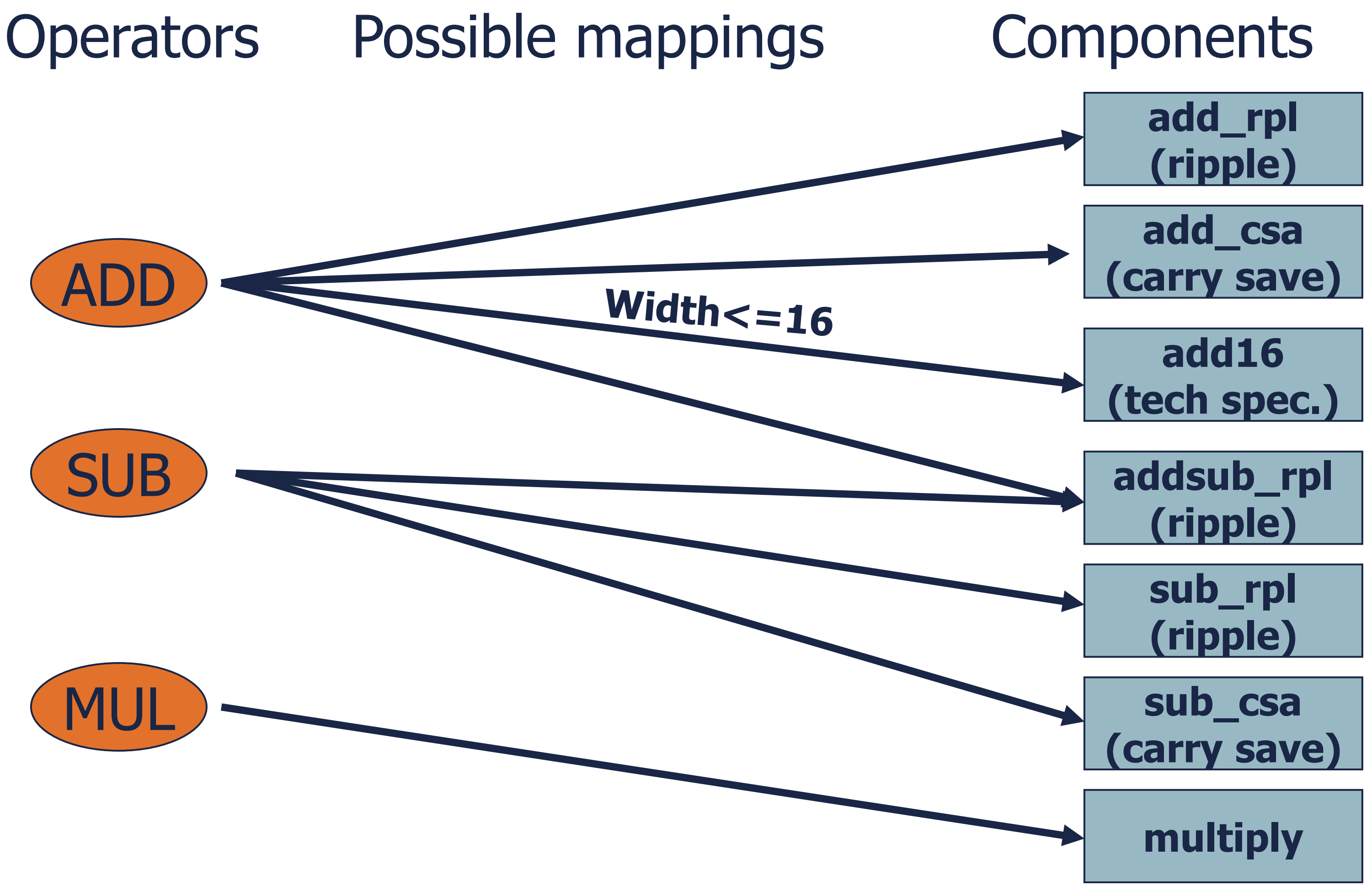
High-Level Synthesis Steps



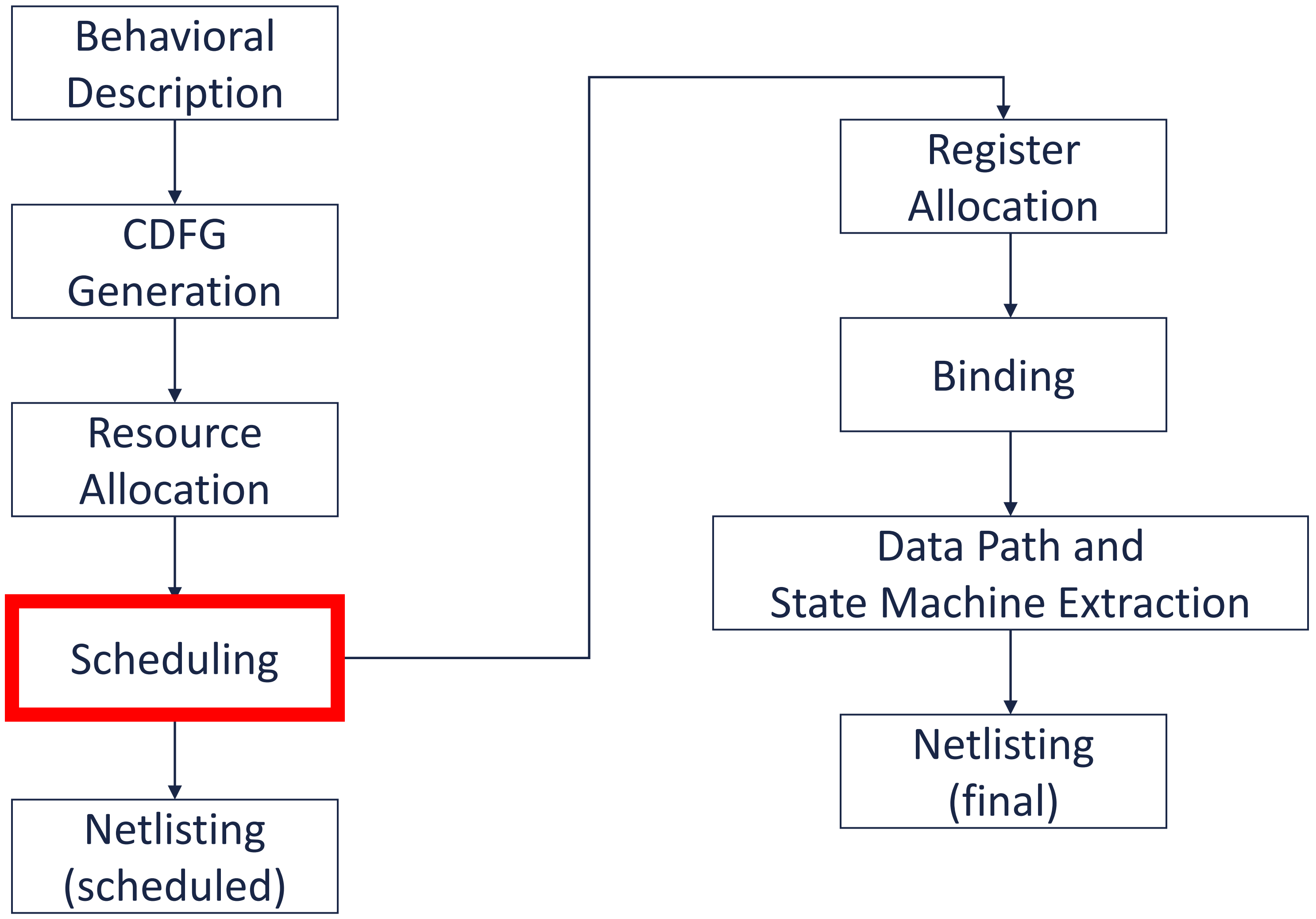
HLS Step: Resource Allocation

- Deciding how many and which kinds of resources will be used in a given implementation
- This has a major impact on final design
  - Number of operation units (multiple adders?) set the maximum parallelism that the architecture can provide
  - Reuse of overloaded operators (e.g., an adder/subtractor unit) provides smallest designs
  - Choice of buses or muxes provides parallelism vs. size
  - Choice of registers, multi-ported register files or RAM also limits parallelism in data movement

Example: Mapping of Operators to Components



High-Level Synthesis Steps



HLS Step: Scheduling

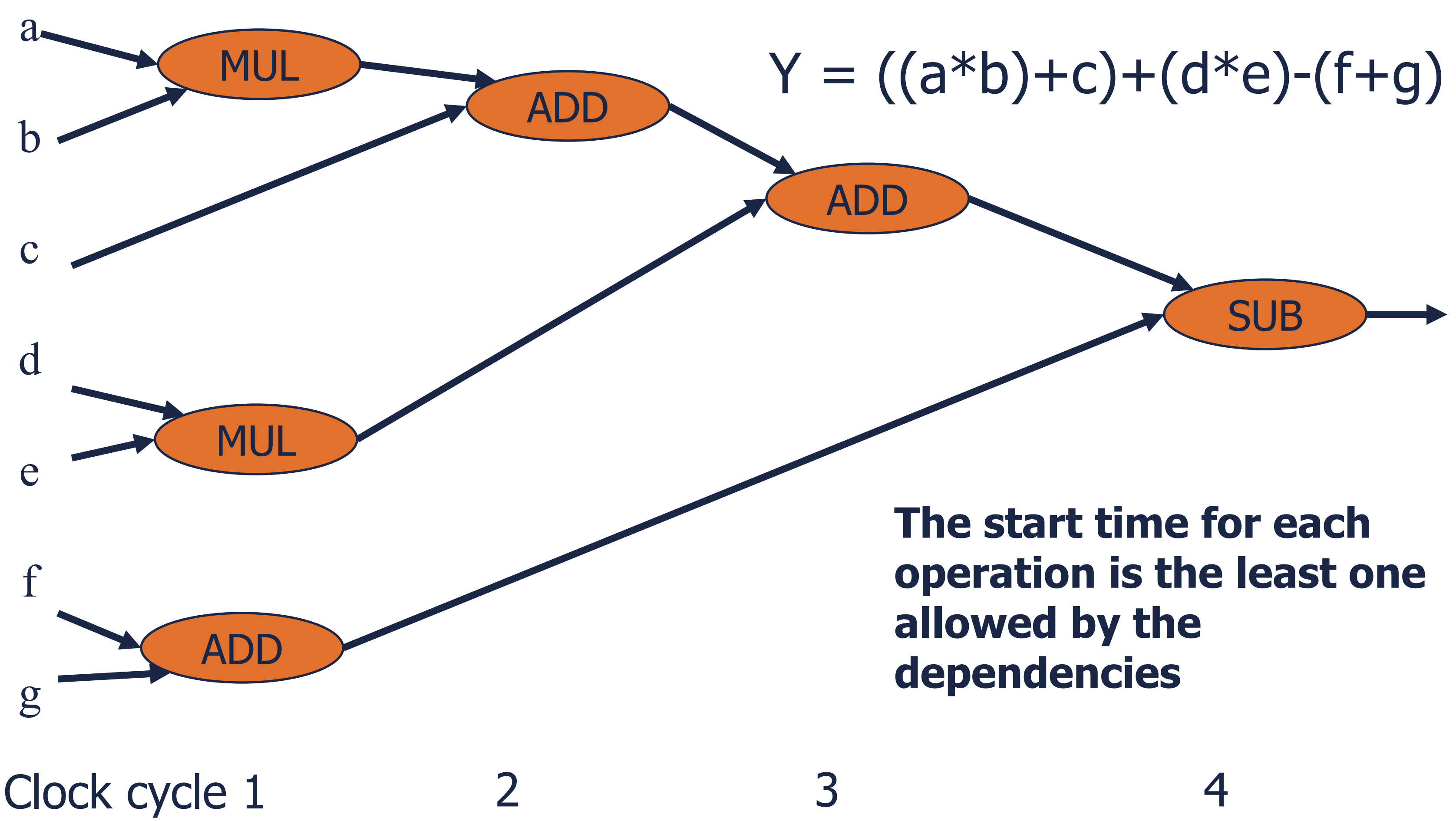
Scheduling:

- Mapping of operations to time slots (cycles)

Basic Algorithms:

- ASAP: As Soon As Possible
- ALAP: As Late As Possible
- List Scheduling

ASAP Schedule (unconstrained)



Y = ((a*b)+c)+(d*e)-(f+g)

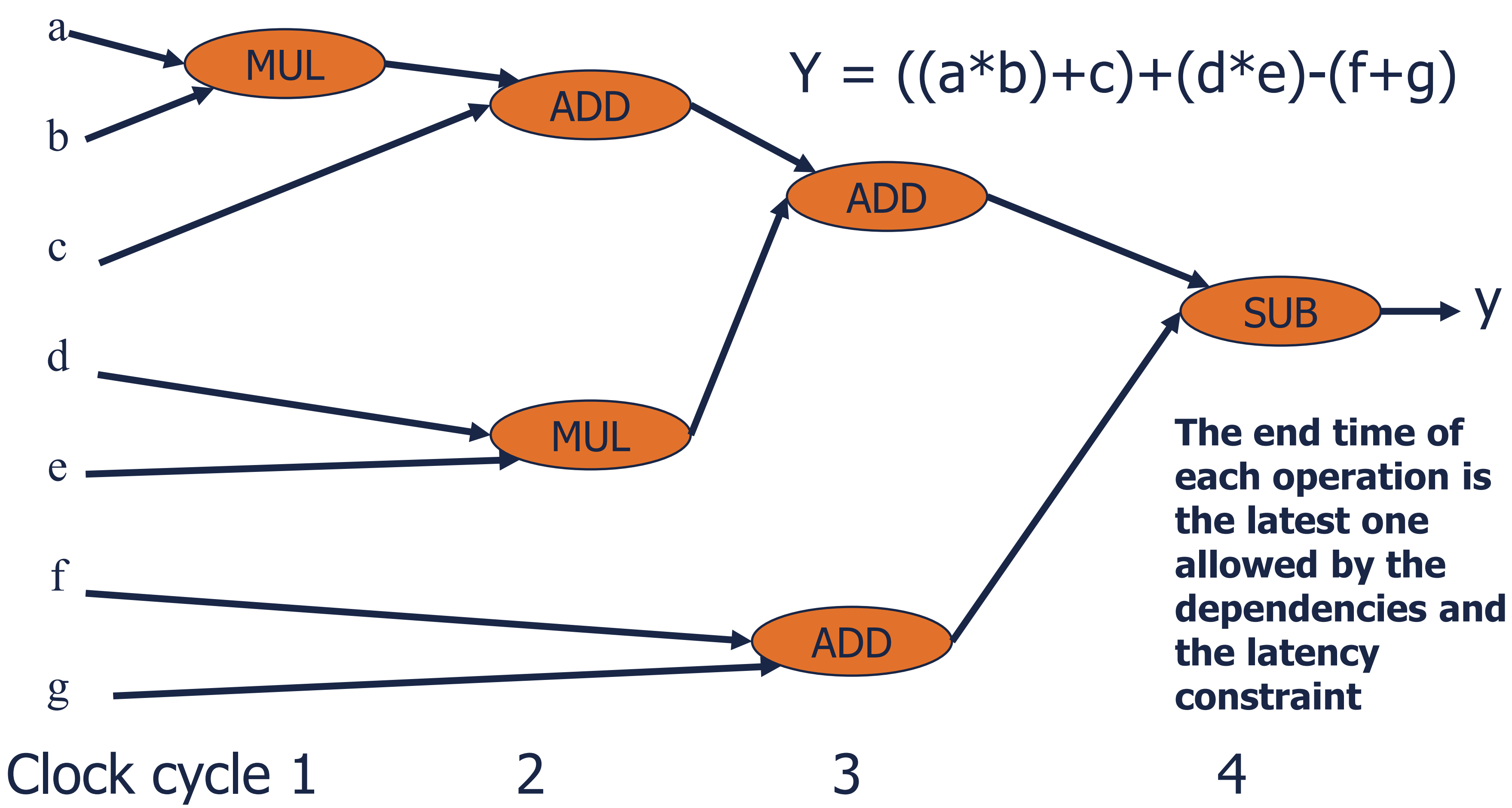
The start time for each operation is the least one allowed by the dependencies

Can we do better than this in terms of the DFG itself? What is the resource usage?

ASAP Algorithm

```
ASAP ($G(V, E)$) {
    Schedule all the nodes driven only by PIs to cycle 1,
      for all such $v_i$ nodes, $t_i$ (staring time) = 1;
    Repeat {
        Select a vertex $v_i$ whose predecessors are all scheduled;
        Schedule $v_i$ by setting $t_i$ = MAX($t_j + d_j$); $(v_j, v_i) \in E$
    }
    Until all the nodes are scheduled;
    Return the schedule in a vector;
}
```

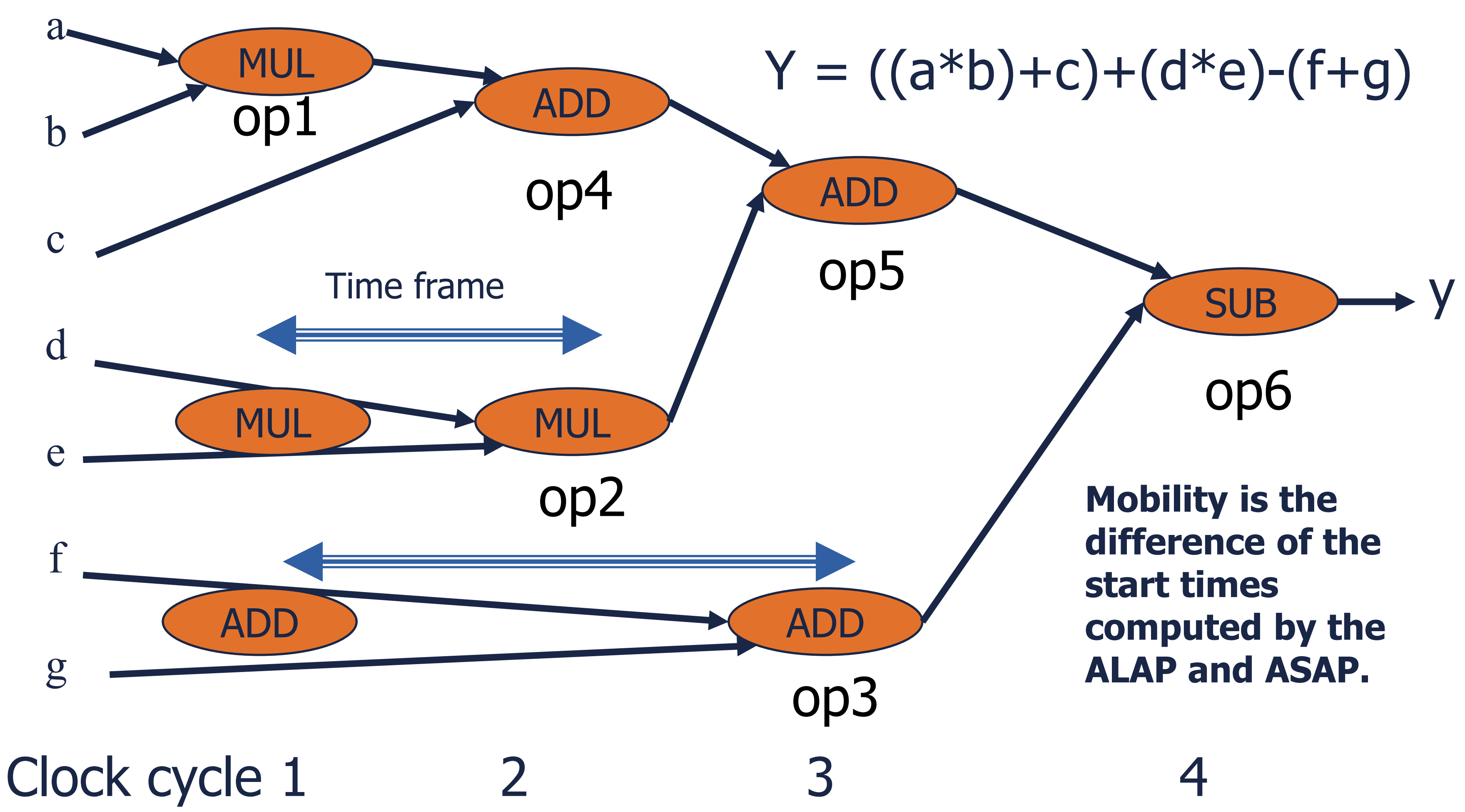
ALAP Schedule



$Y = ((a*b)+c)+(d*e)-(f+g)$

The end time of each operation is the latest one allowed by the dependencies and the latency constraint

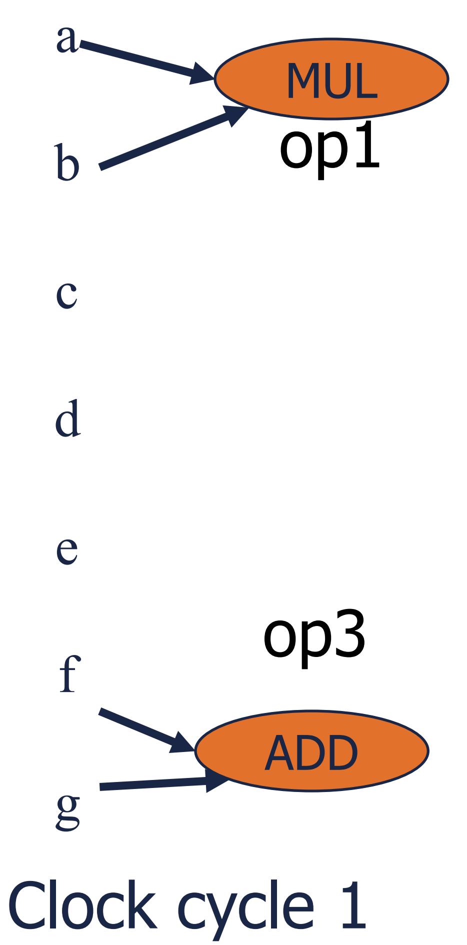
Mobility (or Slack)



Y = ((a*b)+c)+(d*e)-(f+g)

Mobility is the difference of the start times computed by the ALAP and ASAP.

List Scheduling (1)



Schedule
op1 and op3

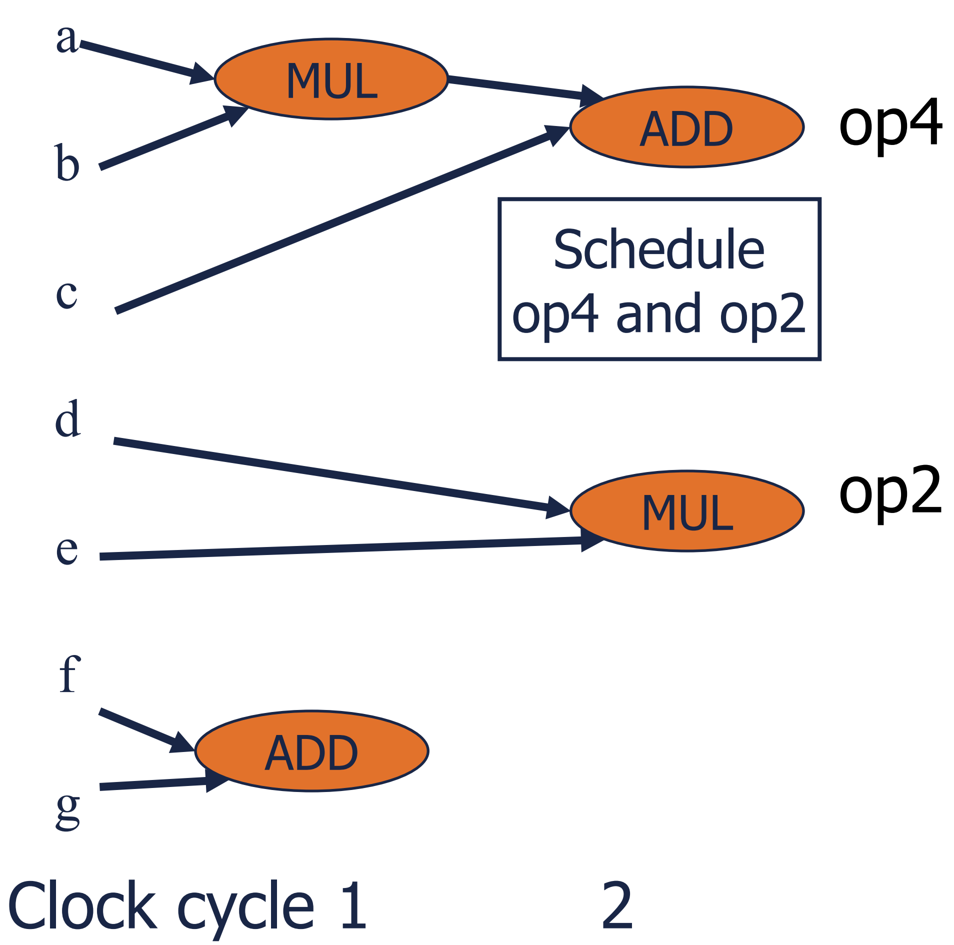
2 3 4

Prioritized Ready List
op1(mul)0 op2(mul)1
op3(add)0

- Priority based on mobility (other metrics possible)
- Resource constraints: one adder, one multiplier
- Schedule ready nodes

A popular scheduling algorithm

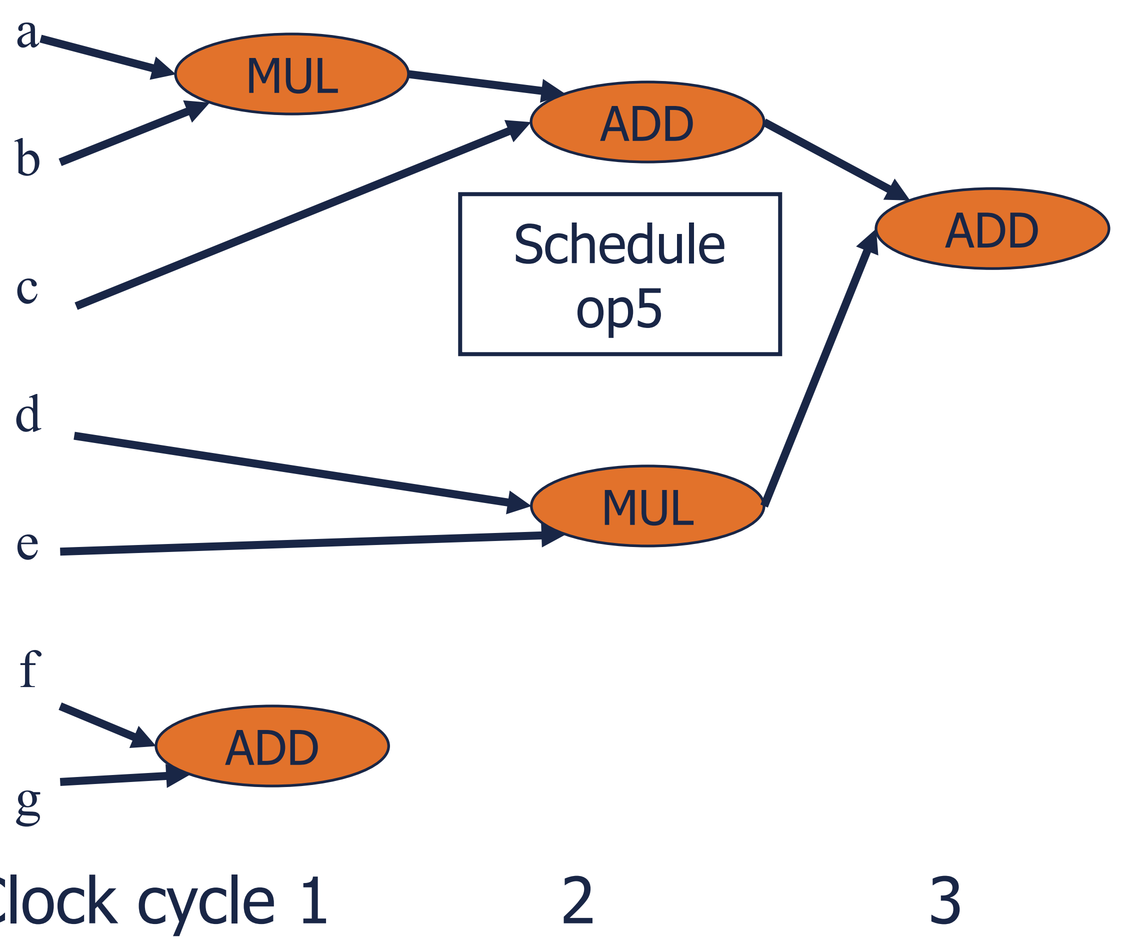
List Scheduling (2)



Schedule op4 and op2

Prioritized Ready List
op2(mul)0
op4(add)0

List Scheduling (3)

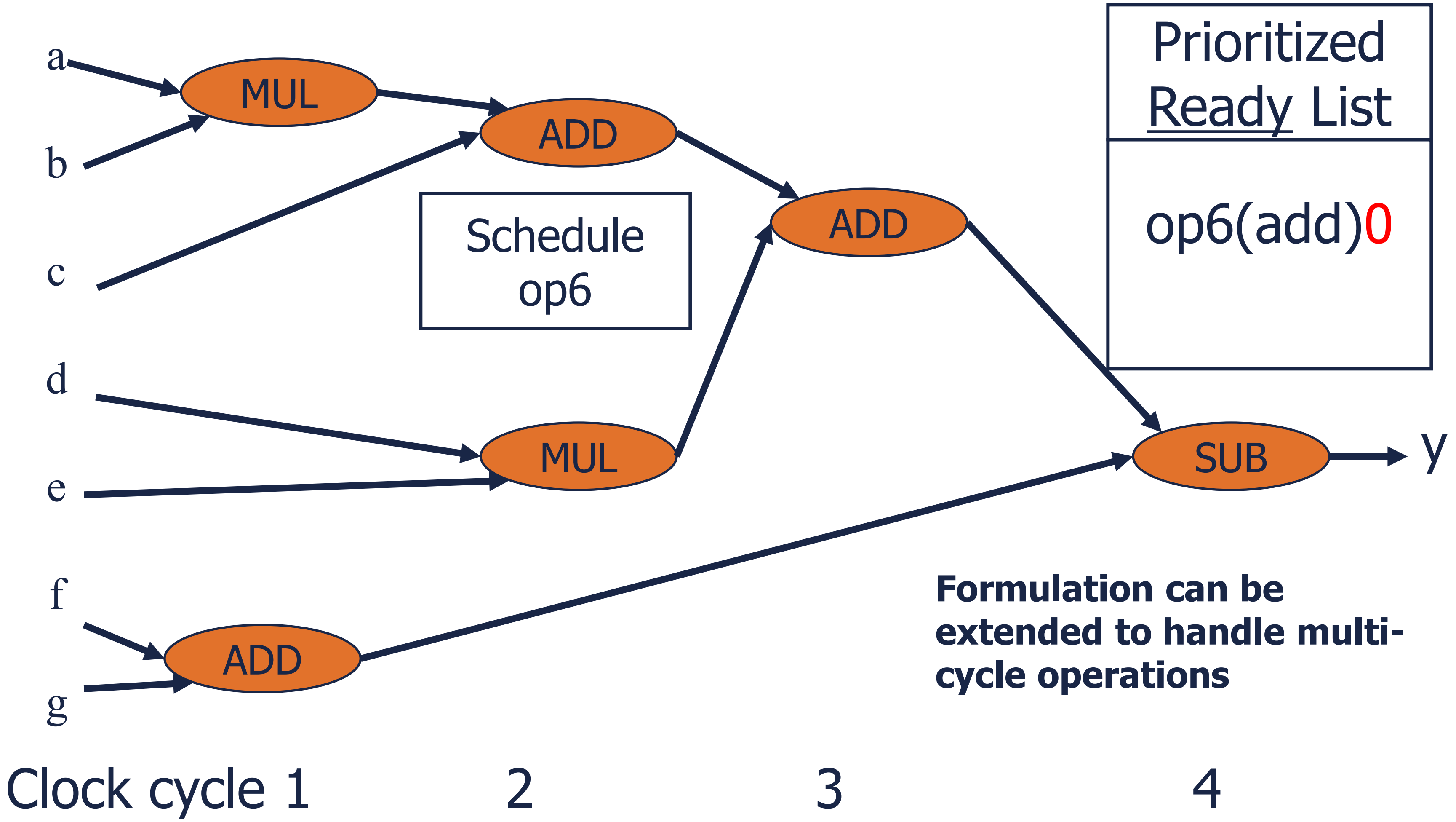


Schedule op5

Prioritized Ready List

op5(add)0

List Scheduling (4)

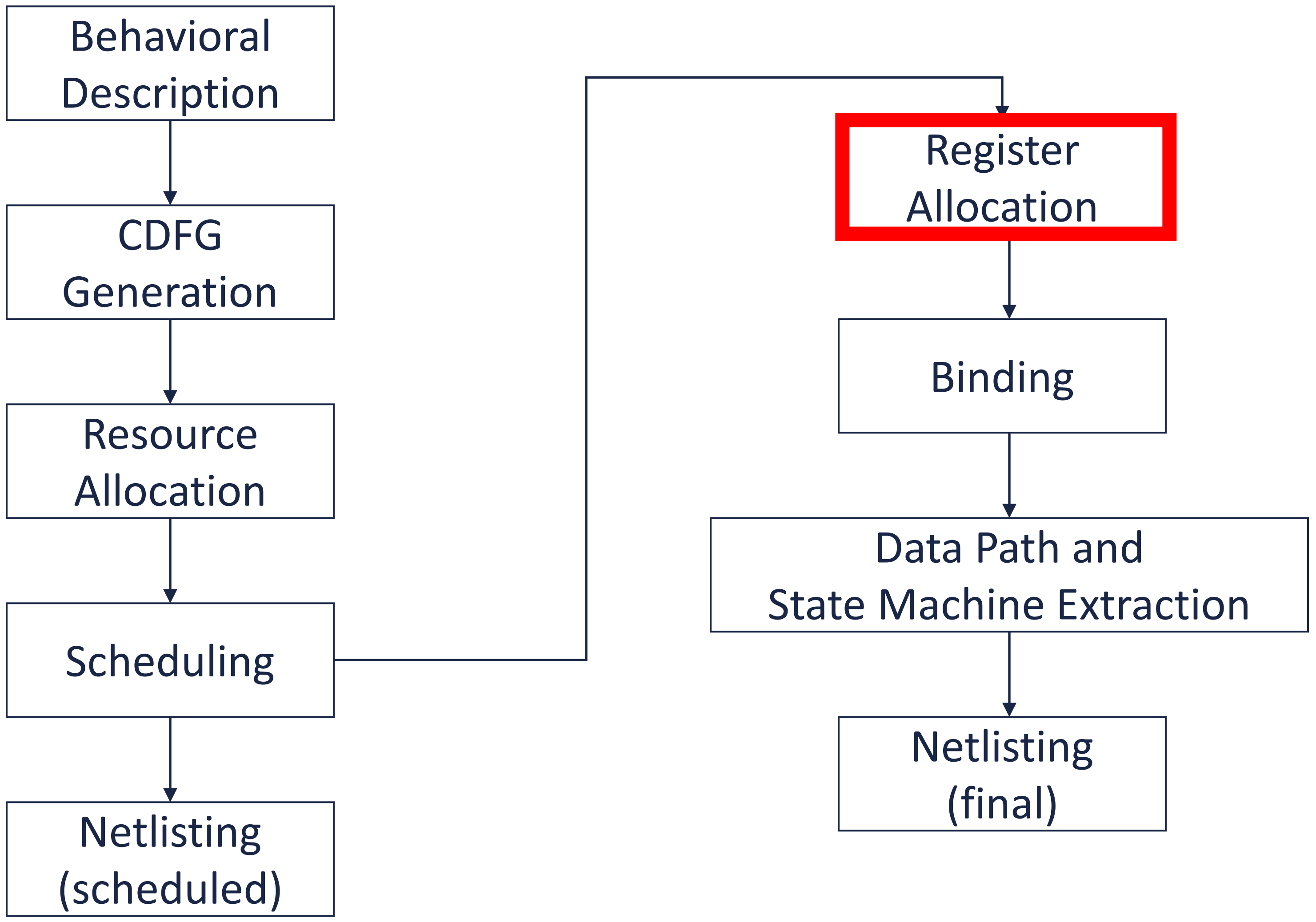


Formulation can be extended to handle multi-cycle operations

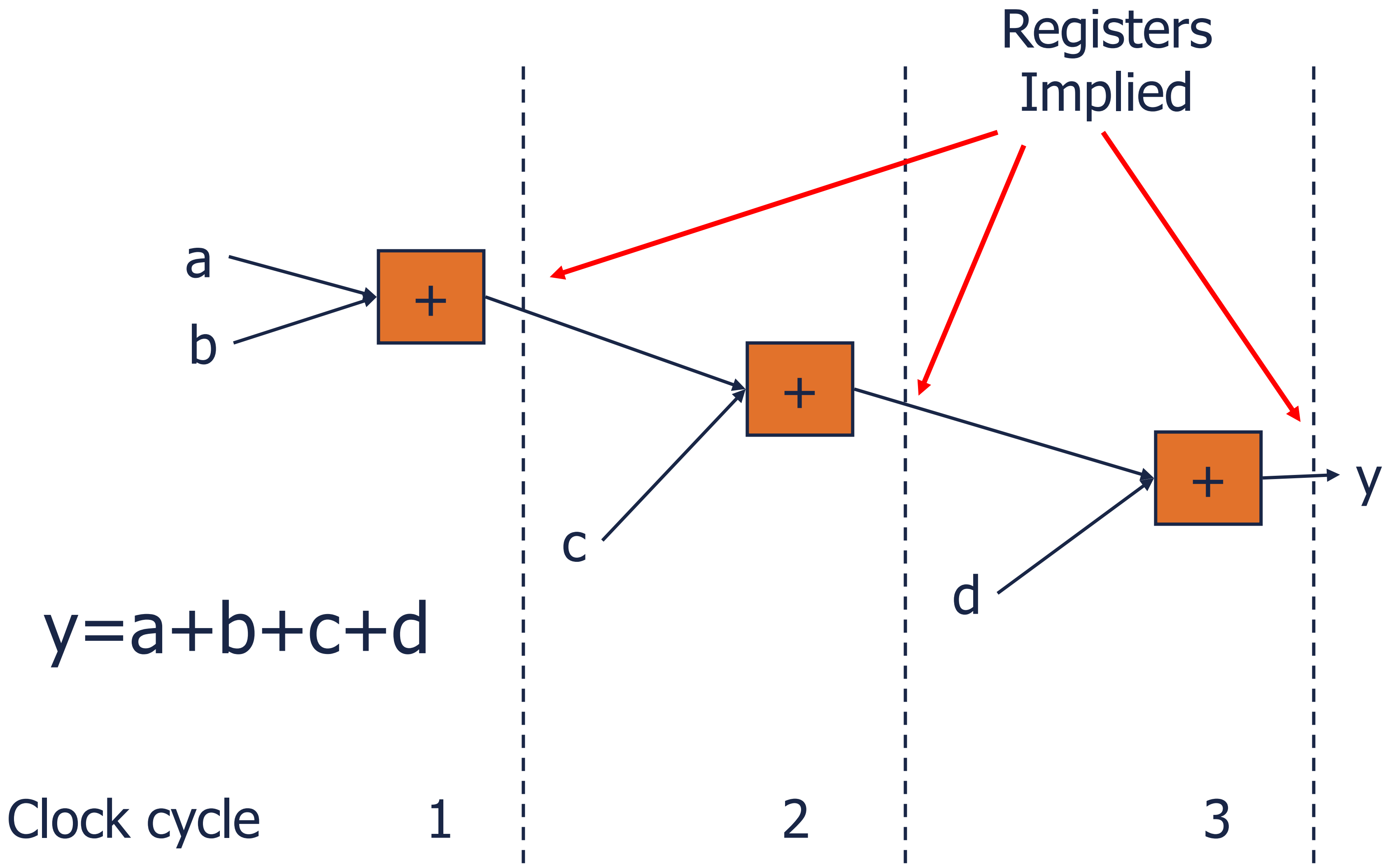
List Scheduling Algorithm

```
LIST_L(G(V, E), a) {
    $l$ = 1;
    repeat {
        for each resource type $k = 1,2,…,n_{res}$ {
            Determine candidate operations $U_{l,k}$;
            Determine unfinished operations $T_{l,k}$;
            Select $S_k \subseteq U_{l,k}$ vertices, such that $|S_k| + |T_{l,k}| \leq a_k$;
            Schedule the $S_k$ operations at step $l$ by setting
                    $t_i = l$ for all $i : v_i \in S_k$;
        }
        $l = l + 1$;
    }
    until (all nodes are scheduled);
    return (t);
}
```

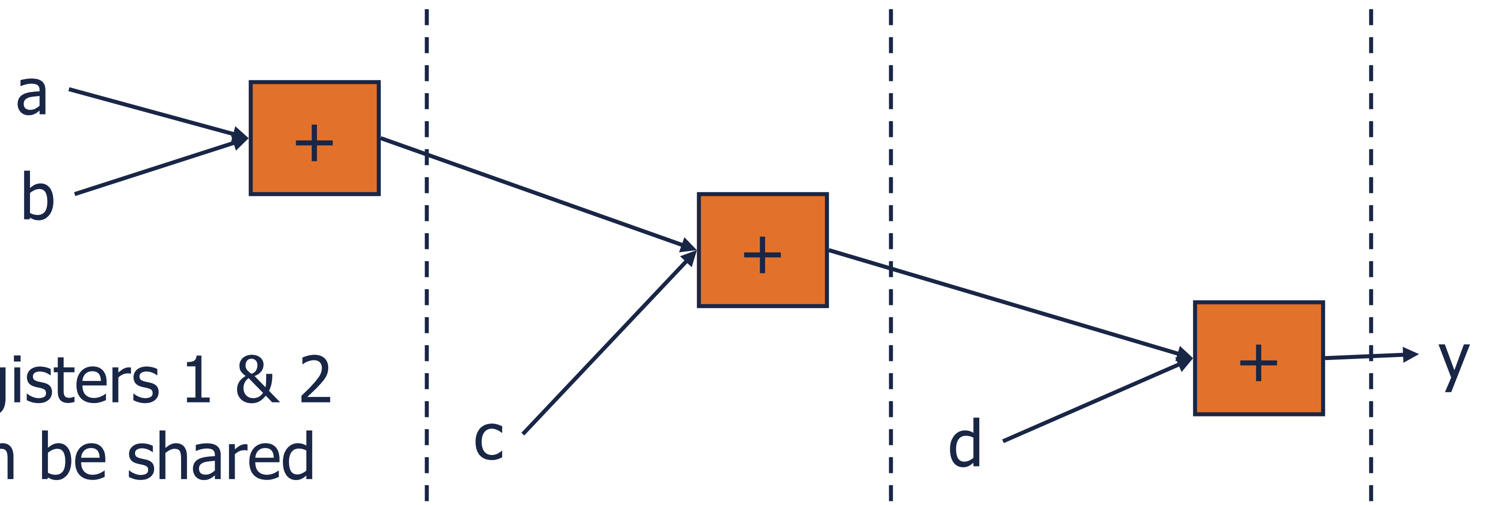
High-Level Synthesis Steps



Register Allocation



Lifetime Analysis



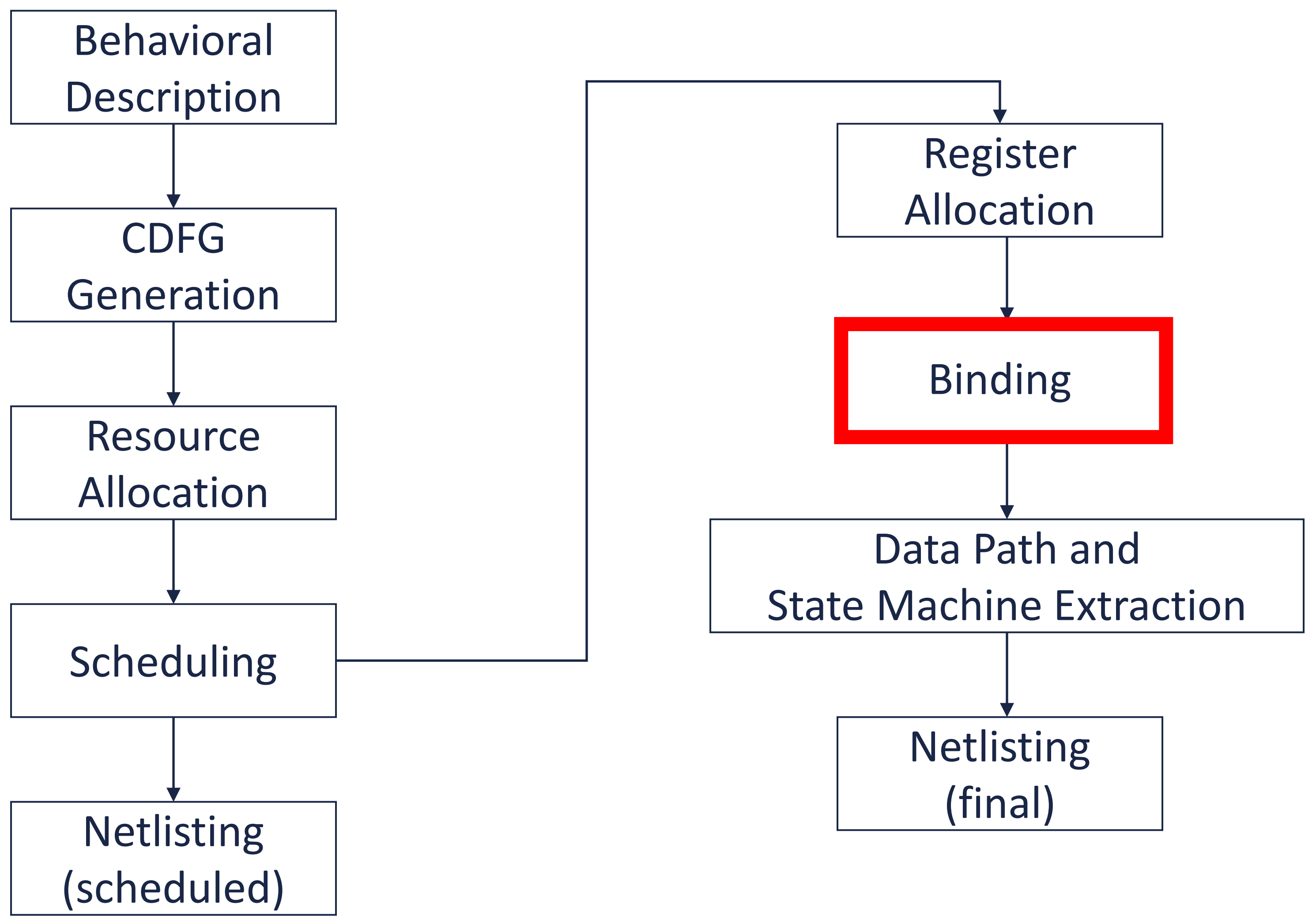
Registers 1 & 2 can be shared

R1		▨		
R2			▨	
R3	▨	▨	▨	▨

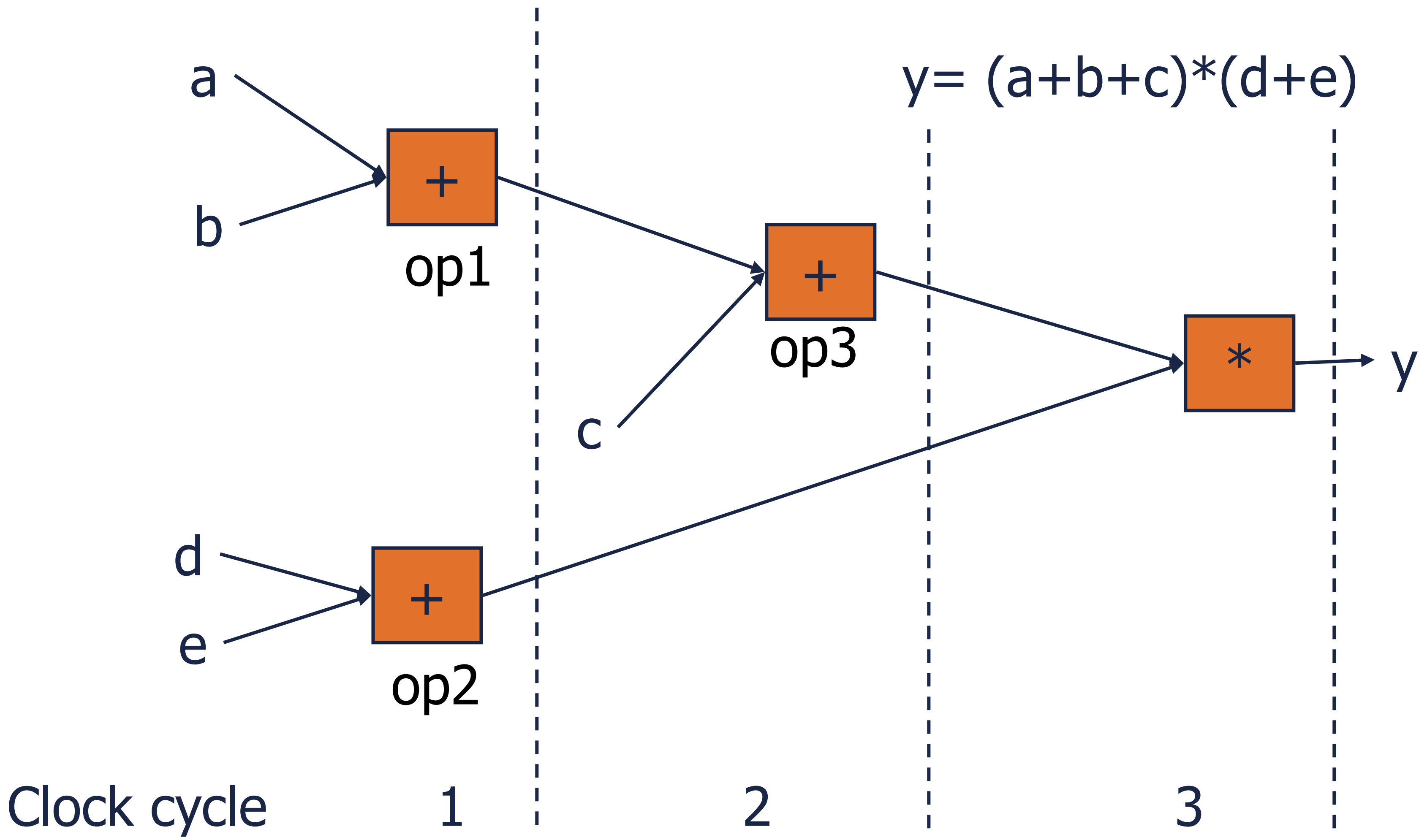
Clock cycle 1 2 3

(R3 needed for output latch)

High-Level Synthesis Steps



Binding



Binding Choices

Binding 1

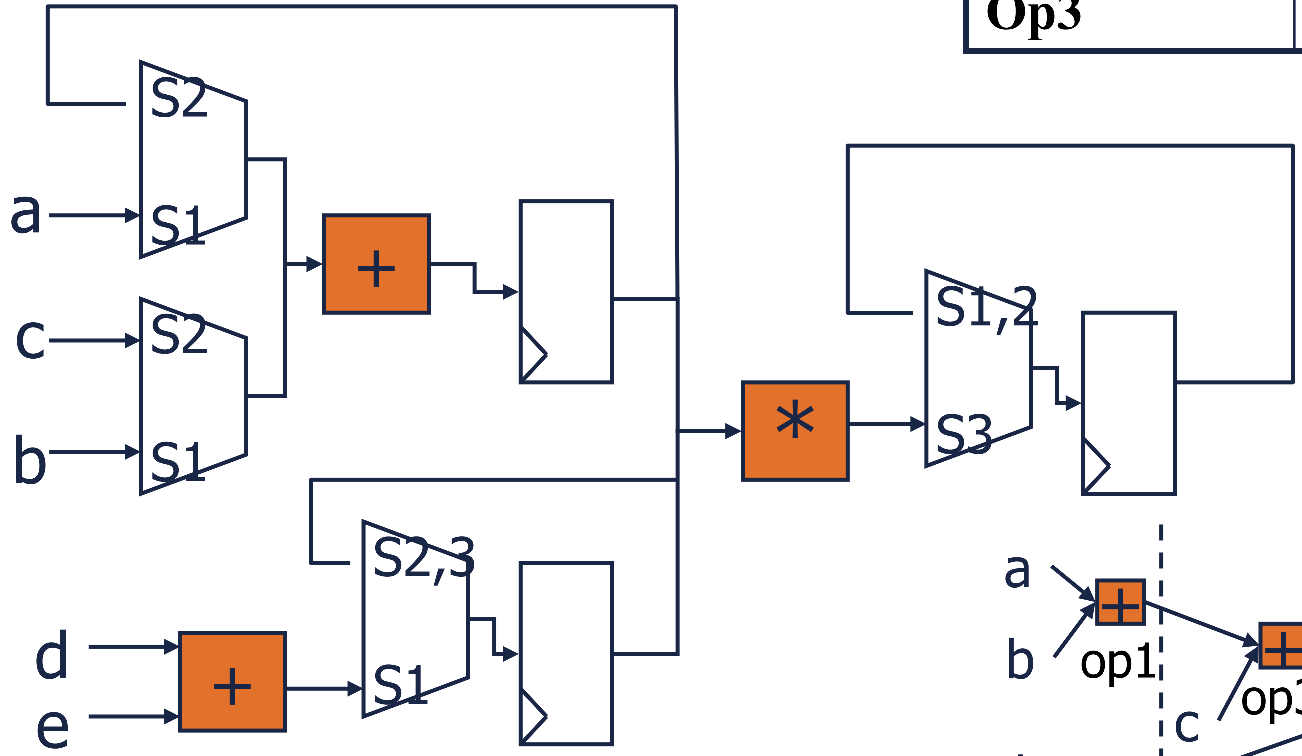
Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add1

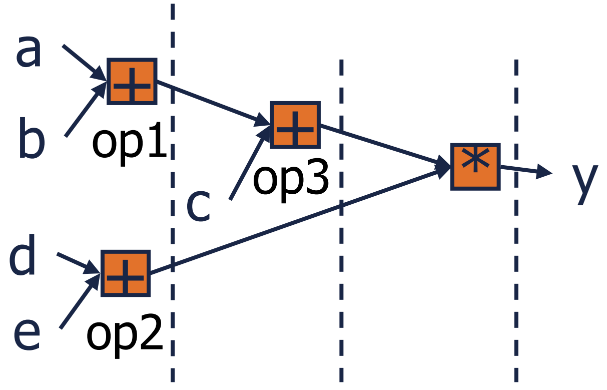
Binding 2

Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add2

Binding 1 Results

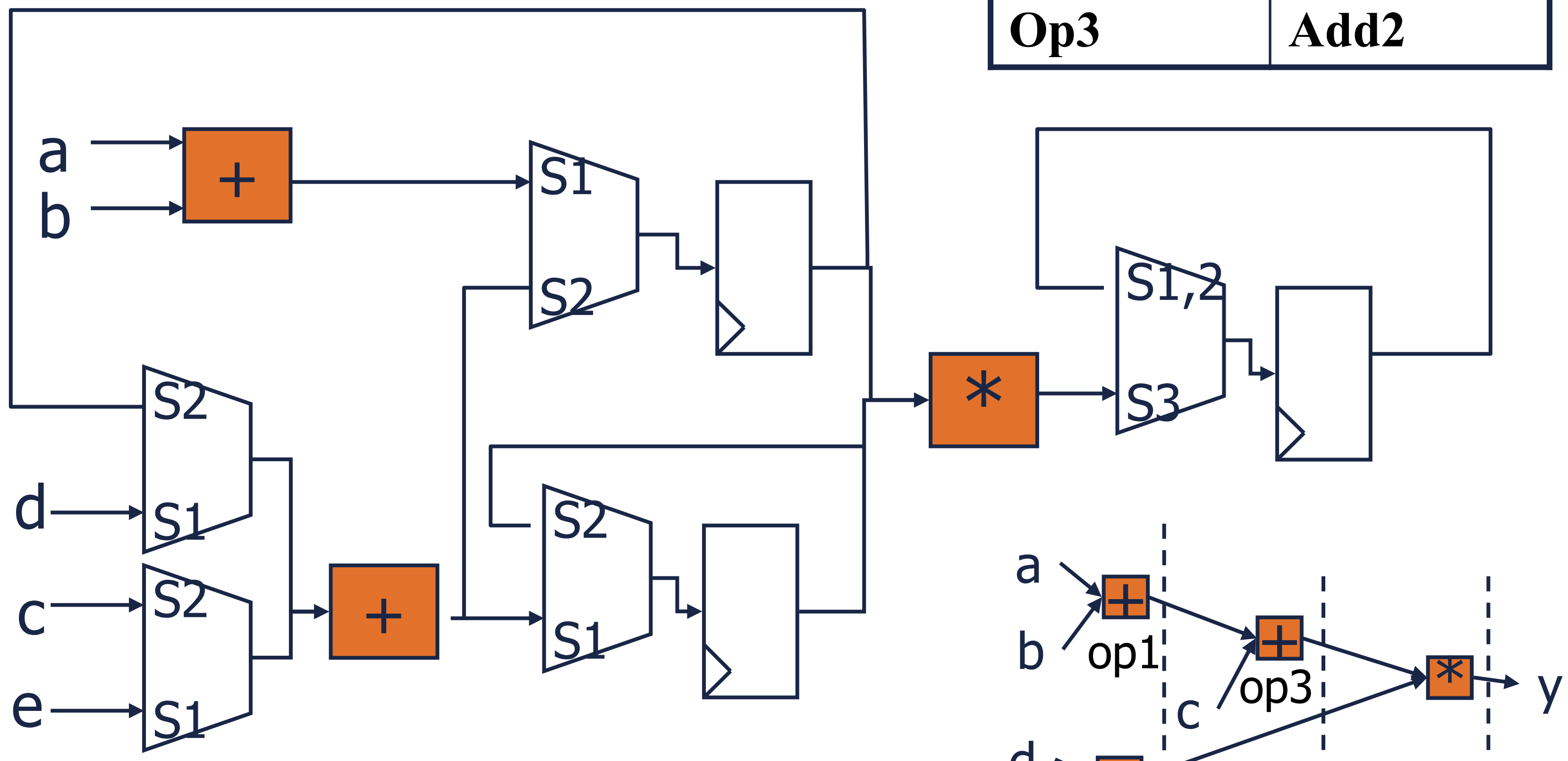
Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add1

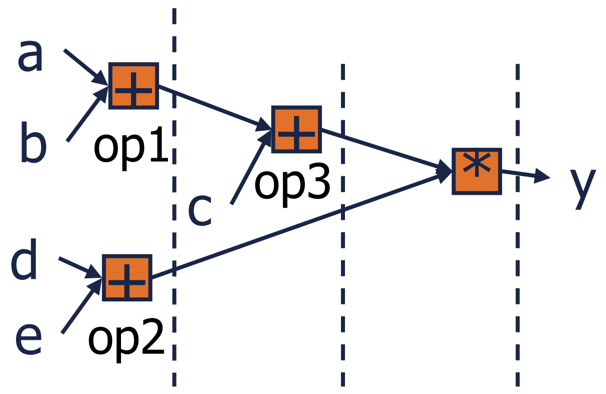




Binding 2 Results

Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add2





Linear Programming

- Objective function: a linear function to be maximized or minimized, e.g.,
  - Maximize $c_1 x_1 + c_2 x_2$
- Constraints: e.g.,
  - $a_{11} x_1 + a_{12} x_2 \leq b_1$
  - $a_{21} x_1 + a_{22} x_2 \leq b_2$
  - $a_{31} x_1 + a_{32} x_2 \leq b_3$
- Non-negative variables, e.g.,
  - $x_1 \geq 0$
  - $x_2 \geq 0$
- The problem is usually expressed in *matrix form*, and then becomes:
  - Maximize $\boldsymbol{c}^T \boldsymbol{x}$
  - Subject to $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$, $\boldsymbol{x} \geq \boldsymbol{0}$

Integer Linear Programming (ILP)

- If the variables are all required to be integers, then the problem is called an integer programming (IP) or integer linear programming (ILP) problem.
- In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in the worst case NP-hard.
- 0-1 integer programming is the special case of integer programming where variables are required to be 0 or 1 (rather than arbitrary integers). This method is also classified as NP-hard, and in fact the decision version was one of Karp's 21 NP-complete problems.

ILP-Based Scheduling

- Use binary decision variables
  - $X = \{x_{il};\ i = 1, \ldots, n;\ l = 1, \ldots, L\}$
  - $x_{il} \in \{0, 1\}$
- First the start time of each operation is unique

$$\sum_{l} x_{il} = 1,\ i = 1, 2, \ldots, n$$

The start time of any operation v_i :

$$t_i = \sum_{l} l \cdot x_{il}$$

Synthesis and optimization of digital circuits, Giovanni De Micheli, McGraw-Hill, 1994

ILP-Based Scheduling: Constraints

- Given the CDFG represented by G(V, E)   *d_j : latency of operation v_j*

  $t_i \geq t_j + d_j \;\; \forall\, i, j : (v_j, v_i) \in E$ implies

  $$\sum_{l} l \cdot x_{il} \;\geq\; \sum_{l} l \cdot x_{jl} + d_j \quad i, j = 1, 2\ldots, n : (v_j, v_i) \in E$$

- Resource bounds must be met at every schedule time step.

  $$\sum_{i:T(v_i)=k} \;\; \sum_{m=l-d_i+1}^{l} x_{im} \;\leq a_k,\; k = 1, 2\ldots, n_{res},\; l = 1, 2,\ldots, L$$

ILP-Based Scheduling: Formulation

- Denote *t* the vector whose entries are the start times
  - Minimize $c^T t$ such that

$$\sum_{l} x_{il} = 1, \; i = 1, 2, \ldots, n$$

$$\sum_{l} l \cdot x_{il} - \sum_{l} l \cdot x_{jl} - d_j \geq 0, \;\; i, j = 1, 2 \ldots, n : (v_j, v_i) \in E$$

$$\sum_{i:T(v_i)=k} \; \sum_{m=l-d_i+1}^{l} x_{im} \leq a_k, \; k = 1, 2 \ldots, n_{res}, \; l = 1, 2, \ldots, L$$

$$x_{il} \in \{0, 1\}, \; i = 1, 2, \ldots, n, \; l = 1, 2, \ldots, L$$

ILP-Based Scheduling: Vector *c*

- Minimize $c^T t$

  The start time of any operation v_i :

  $$t_i = \sum_l l \cdot x_{il}$$

- $c = [0, \ldots, 0, 1]^T$
  - Minimize the latency of the schedule
- $c = [1, \ldots, 1, 1]^T$
  - Finding the earliest start times of all operations

What designs fit HLS well?

- Data path, uni-directional data streaming pipeline
- Flexible in latency requirement, loose timing relationship between interfaces
- Computational kernels
  - E.g., image/video processing/arithmetic computation
  - Dataflow design is better than control flow design
  - HLS tool may provide IPs for arithmetic and other typical computational operations
- No inter-iteration dependencies
  - Preferably no dependencies between two loop iterations
  - Advanced HLS tool can handle dependencies to certain extent
- Independent kernels and module level designs
  - Not top level/system level designs entirely for HLS

Challenges for HLS

- May not be a good fit for very high speed design
  - Designers can push until the last FF and create deep pipelines with RTL
- Cycle accurate design, strong timing relationship on interfaces
- Design with feedback loops and strong timing requirement
  - E.g., accumulators whose current result affects next cycle accumulation immediately
- Control-intensive logic -- but is getting better
- HLS may need designer's manual intervention in order to generate hierarchical designs with high quality
- Coarse-grain pipelining
  - Handling multiple kernels at the top level can be a challenge
  - Advanced HLS tools can work adequately in this area (an example to follow)
- Complicated protocol handling
- Analog/mixed signal portions of the design

Where are these challenges from?

- HLS works with a design entry at a higher abstraction level
  - C/C++ has no idea of “time”
  - It might be do able to capture some timing concept, but the efficiency drops greatly, and we lose the benefit not worth it.
- RTL, as well as SystemC, has “time” built in
  - Fits design with accurate timing requirement best

Coding with HLS

- Write code with sub functions corresponding to modules
  - With hardware implementation concepts in mind
  - Think about the inter-function communication scheme
- Generate hardware for each function using HLS, leave the arguments as interface/ports
- Generate the glue logic/communication and top-level design
  - Integrate the hardware block generated from HLS into a system (e.g., a virtual platform)
  - If the hardware is hierarchical, to achieve better QoR, manual work may be required

Software vs. Hardware Compilation

C Construct	Example	Software Compilation	High-level Synthesis
Function	void go() { … }	groups of instructions	hardware module
Function args and return	int det(int a[16][16]);	pushed onto call stack	input and output ports
Function call	go();	call instructions	submodule
Operations	prod = a.x * b.x + a.y * b.y;	computation instructions	functional units
Local variables	unsigned index; int sum;	architectural registers	physical registers
Arrays	int A[16][16];	stack, heap, static memory	memory blocks
Control flow	for (i = 0; i < 16; i++) { … }	branch instructions	state machines
source code	**all of the above**	**machine instructions**	**hardware modules**

C/C++ Code Synthesizability for HLS

- In HLS, your code is the hardware specification

Unsynthesizable C Construct	Example	High-level Synthesis?	Synthesizable alternative
Variable memory allocation	A = malloc(rows * cols * sizeof(A[0][0]));	resizable RAM?	int A[max_rows] [max_cols];
Recursion	int bin_search(Array array) { … bin_search(array); … }	infinitely nested modules?	use iterative form of binary search
Indirect calls /function pointers	typedef void Callback(); void go(Callback callback) { … callback(); … }	dynamically defined submodule?	inline or create “go” function for each callback
Operating system calls	file = fopen(filename, “r”);	I/O? Interrupt?	use input ports for streaming in data

C/C++ Code Synthesizability for HLS (cont'd)

Construct	Example	Synthesizable?
Debug output	printf("sum: %x\n", sum);	Can be translated to unsynthesizable RTL debug functions.
Time functions	clock_gettime(CLOCK_REALTIME, &cur_time);	Not synthesizable. HLS tools often have profiling support built-in.
C++ memory allocation	Node * node = new Node();	Equivalent to malloc; not synthesizable.

C++ template library	std::vector<int> a;
	…
	std::sort(a.begin(), a.end());
		Uses dynamic memory allocation and recursion; not synthesizable.

Floating point operations	float a, b, c; … float d = a * b + c;	Supported by some HLS engines. Can be expensive, especially on FPGAs.

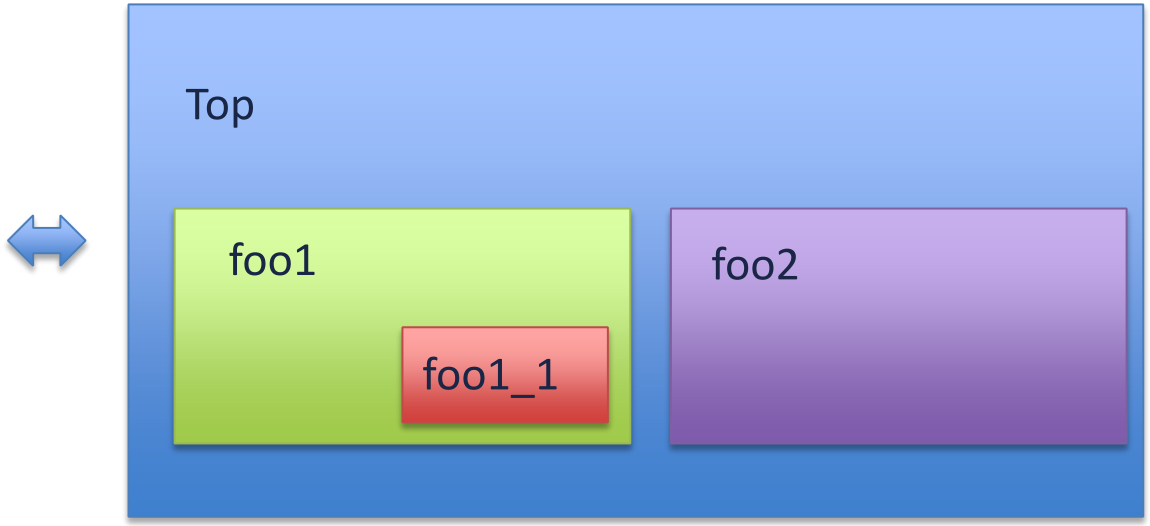
Variables

- Customize bit-widths (arbitrary bit-widths) instead of standard C data types
  - E.g., use 24-bit width instead of 32 bits if 24 bits precision is enough
  - Most HLS tools support arbitrary bit-width data types
    - E.g., `ap_int`, `ap_fixed` data types in Vivado HLS
  - Improves efficiency, performance, and saves area

Functions

- C functions are synthesized to RTL modules
  - By default, sub functions become sub modules
  - Usually, every function call is an instance of the RTL module
  - Inline function is an exception – it flattens the design

```
void top(int a, int b) {
        foo1(a);
        foo2(b);
}
void foo1(int a) {
        foo1_1(a);
}
```

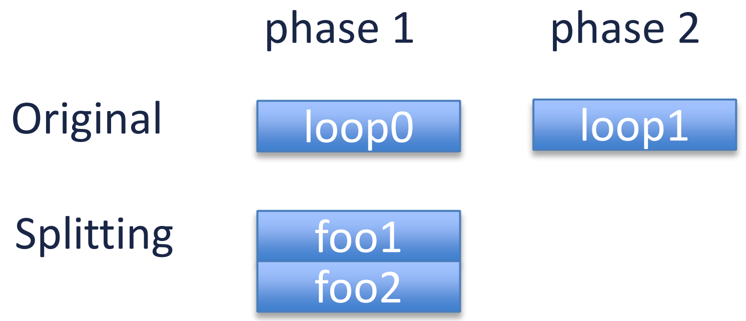


Functions: Splitting

- Use subfunctions to improve parallelism
  - Use function to maintain hierarchy
  - Most tools support dataflow/block-level pipelining and parallelism
  - May not be feasible due to dependencies

```
void top (int a[100], int b[100]) {
        for(i=0; i<100; i++) {
                a[i] = i;
        }
        for(j = 0; j < 100; j++) {
                z += b[j];
        }
}
```

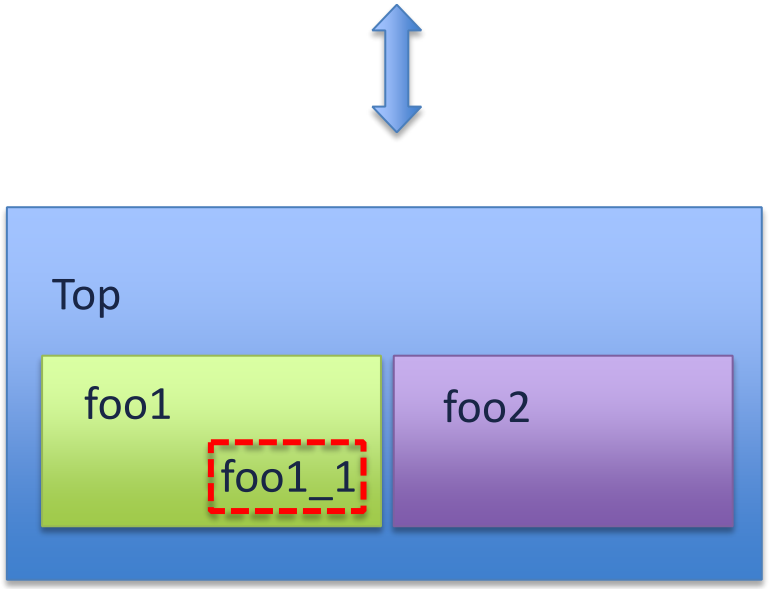
```
void foo1 (int a[100]) {
        for(i=0; i<100; i++) {
                a[i] = i;
        }
}
void foo2 (int b[100]) {
        for(j = 0; j < 100; j++) {
                z += b[j];
         }
}
void top (int a[100], int b[100]) {
        foo1(a);
        foo2(b);
}
```



Functions: Inlining

- Inline functions to flatten the hierarchy
- Pros:
  - Enable more optimization opportunities
    - Creates more objects to schedule and optimize
  - Enable resource sharing
  - Eliminates function call overhead
- Cons:
  - Module instance is effectively copied for each inlined call
  - Inlining of multiple function calls increases area cost

```
void top(){int a, int b){
        foo1(a);
        foo2(b);
}
void foo1(int a){
        foo1_1(a);
}
inline void foo1_1(int a){}
```



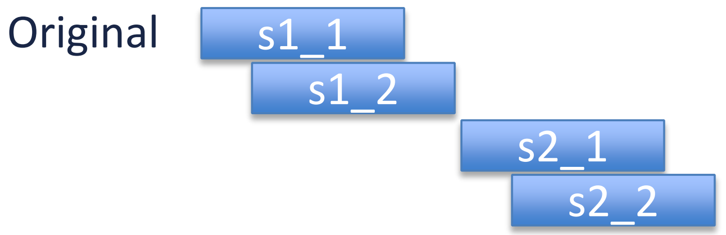
Inlining: Example

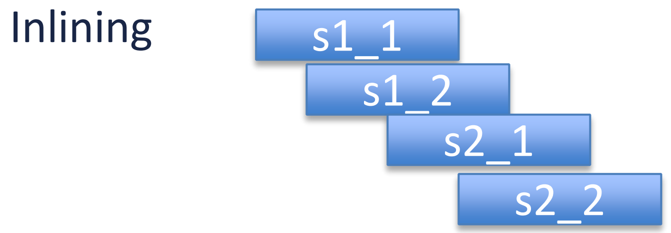
- Merge foo1 and foo2, and apply loop pipelining

```
int foo1(int a, int b, int c) {
        z0 = a * b;       //s1_1
        z1 = c * b;       //s1_2
        return (z0+z1);
}
int foo2(int c, int d, int a){
        z2 = c * d;       //s2_1
        z3 = d * a;       //s2_2
        return (z2+z3);
}
int top(int a, int b, int c, int d){
        y1 = foo1(a, b, c);
        y3 = foo2(c, d, a);
        return (y3+y1);
}
```

```
int top(int a, int b, int c, int d){
        z0 = a * b;              //s1_1
        z1 = c * b;              //s1_2
        z2 = c * d ;             //s2_1
        z3 = d * a;              //s2_2
        return (z0+z1+z2+z3);
}
```

Assume each multiplication takes multiple cycles, and it can be pipelined





Also, enable resource sharing, e.g. for the multiplier

Loops: Bounds

- Not recommend using variable loop bounds, because:
  - HLS tools may not be able to estimate the performance
  - Limits the optimization opportunities
    - E.g., cannot deliver complete loop unrolling, thus prevent the pipelining
    - Solutions: Set the bounds to the maximum value, and conditionally execute the loop body

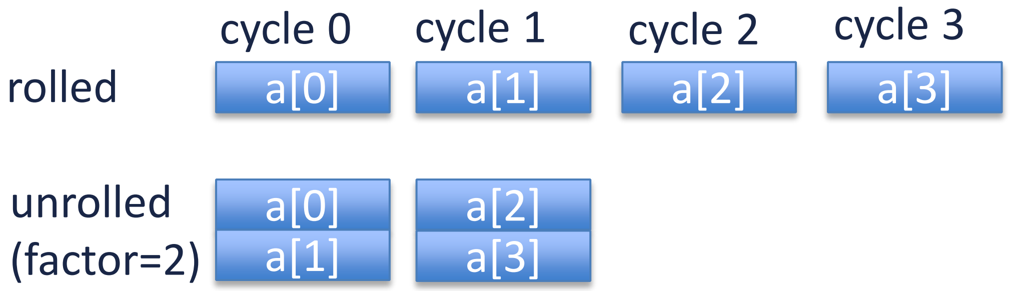
```
for (i=0; i < len; i++) {
        A[i] += b;
}
```

```
for (i=0; i < N-1; i++) {
        if(i < len)
                A[i] += b;
}
```

Loop Unrolling

- Most HLS tools support automatic loop unrolling (by using HLS unroll pragmas)
  - But have strict limits, e.g., constant loop bounds
- May need manual loop unrolling to
  - Improve pipeline opportunity
  - Improve parallelism
- Things to notice
  - Need to consider memory port limit when accessing array
  - Expensive hardware replication
  - Need to balance the unroll level

```
void top(){
     for(i=0; i<=N; i++){
          a[i] = i;
                    …
          }
}
```



Loop Fusion/Merging

- Merge several loops into one loop
- Some tools support loop fusion
  - Strict bound check of loops, e.g., identical loop bounds
- Manually loop fusion can be used to
  - Improve data locality
  - Shorten latency
  - Share resources

```
void top(){
        for(i=0; i<N; i++){
                a[i] = i;
        }
        for(i=1; i<N; i++){
                x+=a[i-1];
        }
}
```

```
void top(){
        for(i=0; i<N; i++){
                a[i] = i;
                if(i>=1)
                    x+=a[i-1];
        }
}
```

After fusion, loop bodies are optimized together

Loop Perfectization

- Convert loops to perfect loop
  - All statements are in the innermost loop
  - An example: Loop Sinking

```
for (i=0, i<20; i++){
        a  = 0;
        for(j=0; j<20; j++){
                a += A[j] + j;}
        B[i] = a * 20;}
```

```
for (i=0, i<20; i++){
        for(j=0; j<20; j++){
                if(j==0) a =0;
                a += A[j] + j;
                 if(j==19) B[i] = a*20; }}
```

HLS result (Use Vivado_HLS):

```
Loop_I:
        *Trip count: 20
        *Latency: 480
Loop_J:
        *Trip count: 20
        *Latency: 21
        *Pipeline II: 1
        *Pipeline depth: 2
```

```
Loop_I_Loop_J:
        *Trip count: 400
        *Latency: 401
        *Pipeline II: 1
        *Pipeline depth: 2
```

The original code can only pipeline Loop_J. After perfection, two loops can be flattened and pipelined.

Why Loop Perfectization?

- Form very regular loop structure
  - Some tools can only flatten the perfect loops
  - Easy for HLS scheduling and discovering optimization opportunities
  - Improves performance by eliminating the loop entering/exiting/cost
- Different ways of loop perfection
  - Loop sinking
  - Loop unrolling
  - Loop splitting

Loop Perfectization: Code Sinking

```
for (i=0, i<20; i++){
        a  = 0;
        for(j=0; j<100; j++){
                a += A[j] + j;
        }
        B[i] = a * 20;
}
```

```
for (i=0, i<20; i++){
        for(j=0; j<20; j++){
                if(j==0) a = 0;
                a += A[j] + j;
                if(j==19) B[i] = a*20;
         }
}
```

- Pros:
  - Easy to implement
  - No performance penalty for branch operations in pipeline
- Cons:
  - Expensive for hardware
    - sometimes the branches can be messy when code is complicated
  - Could sequentialize loop nest and affect performance
    - Change the data dependency

Loop Perfectization: Loop Unrolling

```
for (i=0; i<100; i++) {
        z = a*b[i];
        z+= c*d[i];
        for(j=0; j<4; j++) {
                z+=x[j] + j;
        }
}
```

```
for (i=0, i<100; i++) {
        z = a*b[i];
        z += c*d[i];
        z += x[0] + 0;
        z += x[1] + 1;
        z += x[2] + 2;
        z += x[3] + 3;
}
```

- Pros:
  - Enable pipelining at outer loop
  - Enable resource sharing
    - More statements to schedule in the same loop level
  - Higher performance, good for expensive operations
- Cons:
  - Not always feasible
  - Expensive hardware, best if the innermost loop is small

Loop Perfectization: Loop Splitting

```
for (i=0; i<100; i++){
        z = a*b[i];
        z += c*d[i];
        for(j=0; j<4; j++){
                z+=x[i] + j;
        }
}
```

```
for(i=0; i<100; i++) {
        z = a*b[i];
        z += c*d[i];
}
for(i=0, i<100; i++) {
     for(j=0; j<4; j++) {
        z += x[0] + 0;
        z+= x[1] + 1;
        z+= x[2] + 2;
        z+= x[3] + 3;
     }
}
```

- Pros:
  - Enable optimization per loop
  - Opportunity for dataflow streaming between two loops
- Cons:
  - Not always possible because of data dependency

Arrays

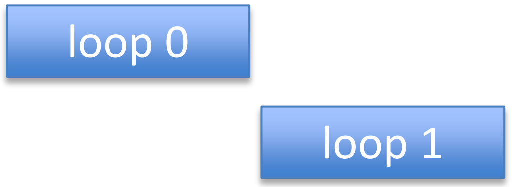
- By default, it is implemented as RAM/ROM
- Expensive and have limited ports
- Optimizations:
  - Array streaming
    - Replace the RAM with FIFO, if possible
    - Requires array access to be sequential
    - Requires the array access order to be the same
    - Transform the code to make array access sequential
  - Array partitioning
    - Eliminate the ports limit

Array Streaming: Example

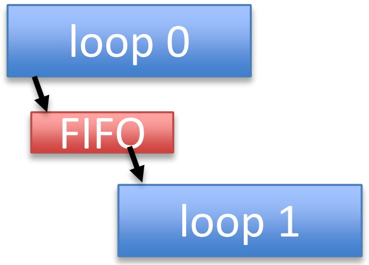
```
for (i=0; i<100; i++)
        for(j=0; j<100; j++){
                A[i][j] = a * b; //row
                ...
        }
}
for (i=0; i<100; i++)
        for(j=0; j<100; j++) {
                C[i][j] = A[j][i]; //col
        }
}
```

```
hls::stream<int> A;
for (i=0, i<100; i++){
        for(j = 0; j<100; j++)
                A.write(a*b);
}
for(i=0, i<100; i++){
        for(j = 0; j<100; j++){
                A.read(C[j][i]);
}}
```

- Original array A access order:
  - Loop1: row; Loop2: column
  - No dataflow pipeline
  - No streaming

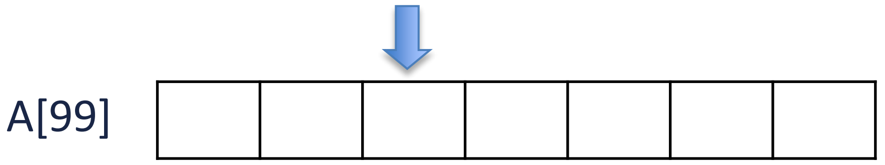


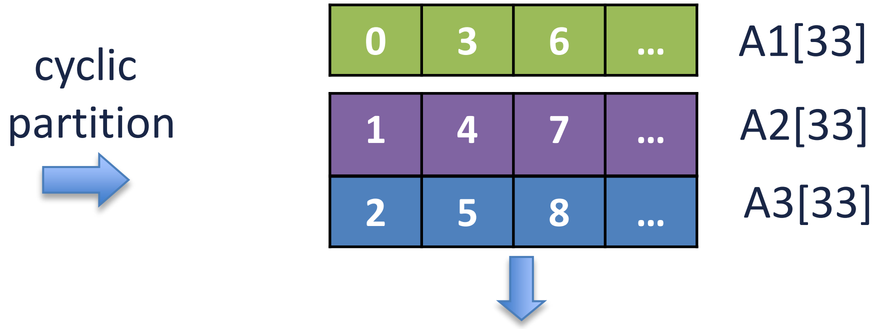
- New array A access order:
  – Loop1: row; Loop2: row
  – Dataflow pipeline
  – Streaming



Array Partitioning

```
for (i=0; i<99; i++) {
     A[i] = i;
     ...
}
```





```
for (i=0; i<99; i+=3) {
    A1[i] = i;
    A2[i] = i+1;
    A3[i] = i+2;
    ...
}
```

- To mitigate the # of ports limitation
- To improve parallelism
- Be careful:
  - sometimes array partition can be subtle
  - can easily introduce bugs

Some Tips for C-based HLS Design

- Understand the HLS tool behavior and the optimization goal
  - No unique coding style is “useful” or “good” for all optimizations
- Reuse data and minimize array access
  - Once the data has been read into a block or buffer and reused, it can improve the parallelism.
  - Array ports are limited. Array partitioning can be an alternative, but it’s not free.
- Loop transformation to enable parallelism
  - Pipelining/Unrolling/Dataflow
- Bit-width selection for efficient hardware
- The optimization goals for CPU and Hardware are different
  - E.g., branch mis-prediction
  - But many software code optimization techniques are useful for hardware design too

UCI EECS

EECS 221:
Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu