

# **Synthesis of Digital Systems**

## **Part 3: Behavioural Synthesis**

### **or**

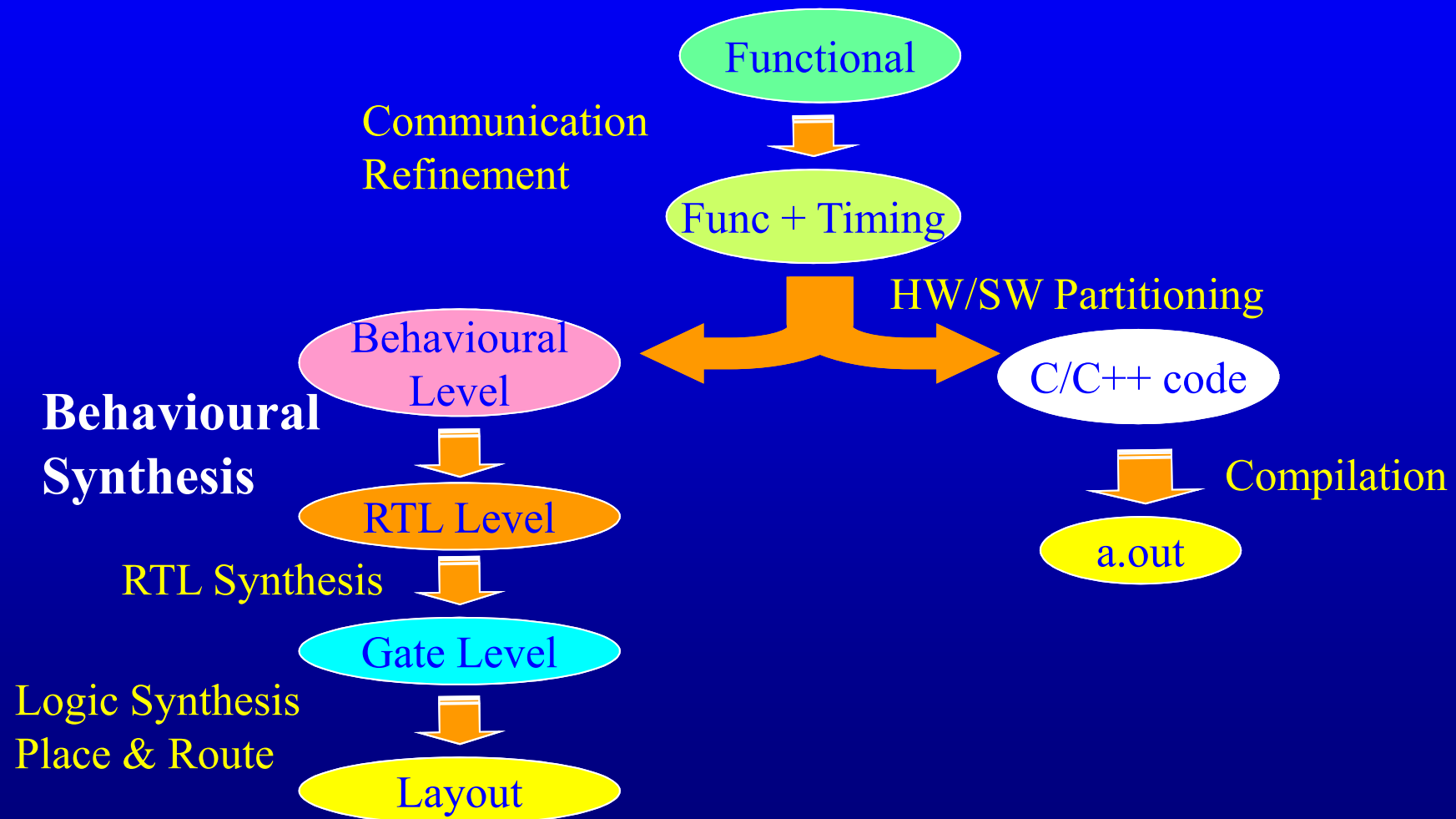
## **High-level Synthesis**

Preeti Ranjan Panda  
Department of Computer Science and Engineering  
Indian Institute of Technology Delhi

# Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

# Levels of Abstraction



# HLS: Inputs and Outputs

- Inputs
  - HDL processes
  - Component library
  - Constraints (resources/delays/power)
  - Clock period
- Output
  - FSM
  - Datapath

# Behavioural VHDL Process

- Abstract
- Only functionality is modelled
- No timing commitments

```
process (clk)
begin
  y <= (A + B) * C - (D + E) / F;
end;
```

# Refined Behavioural Process

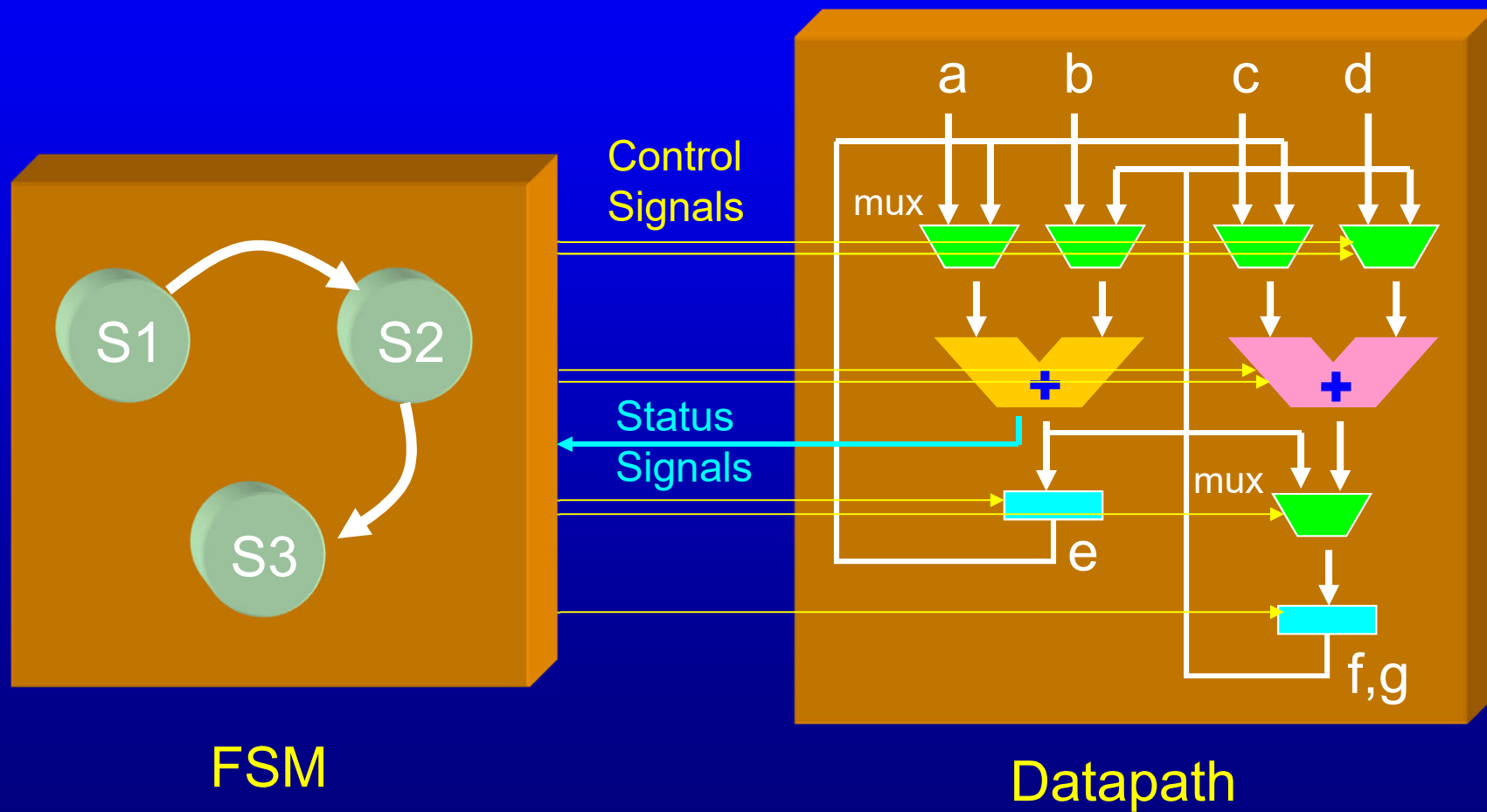
- Abstract behaviour partitioned into clock cycles

```
process (clk)
begin
  y <= (A + B) * C - (D + E) / F;
end;
```



```
process
begin
  t <= A + B;
  s <= D + E;
  wait until rising_edge (clk);
  p <= t * c;
  q <= s / F;
  wait until rising_edge (clk);
  y <= p - q;
  wait until rising_edge (clk);
end;
```

# HLS Output – FSM + Datapath



# Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges



# Resource Library

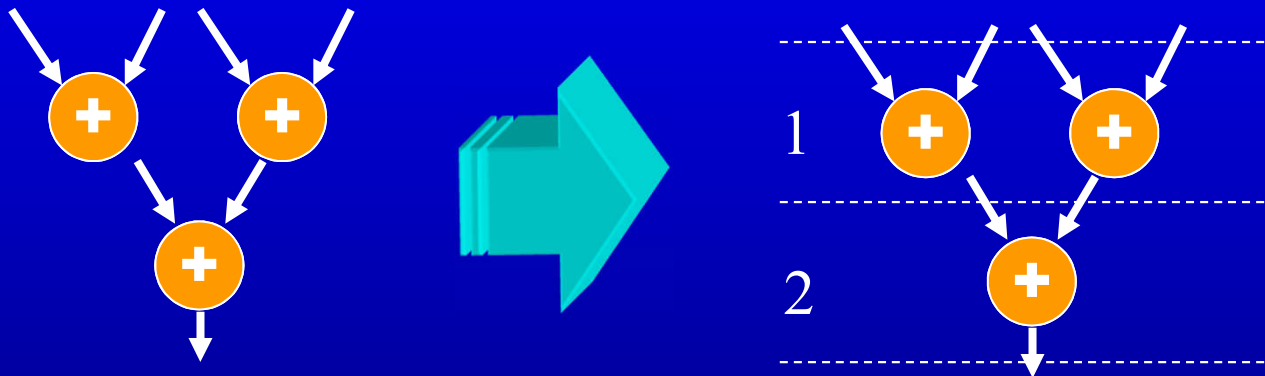
- **Function Units: Adders, Multipliers, Comparators, ALUs**
  - Number
  - Type: differing area/delay
- **Memory**
  - Number and size
  - Port number and types
- **Information**
  - Function
  - Area
  - Delay
  - Power Dissipation



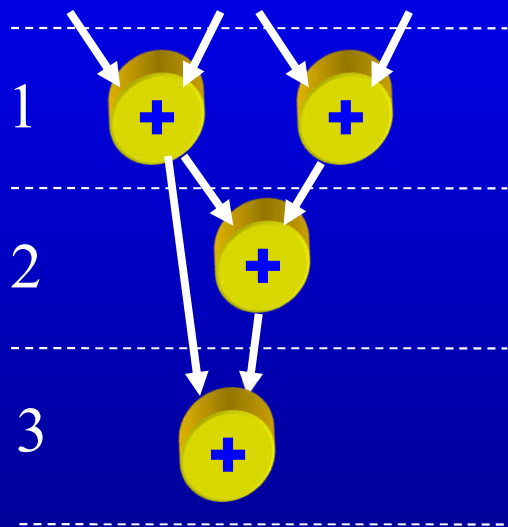
Resource  
Library

# Scheduling

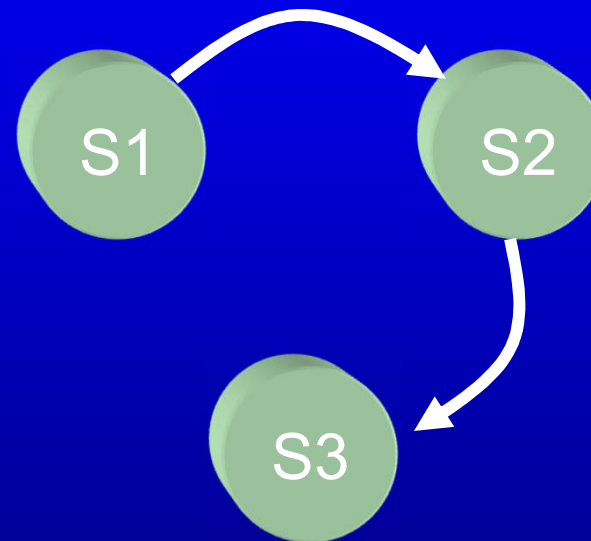
Mapping operations to clock cycles



# Schedule Leads to FSM



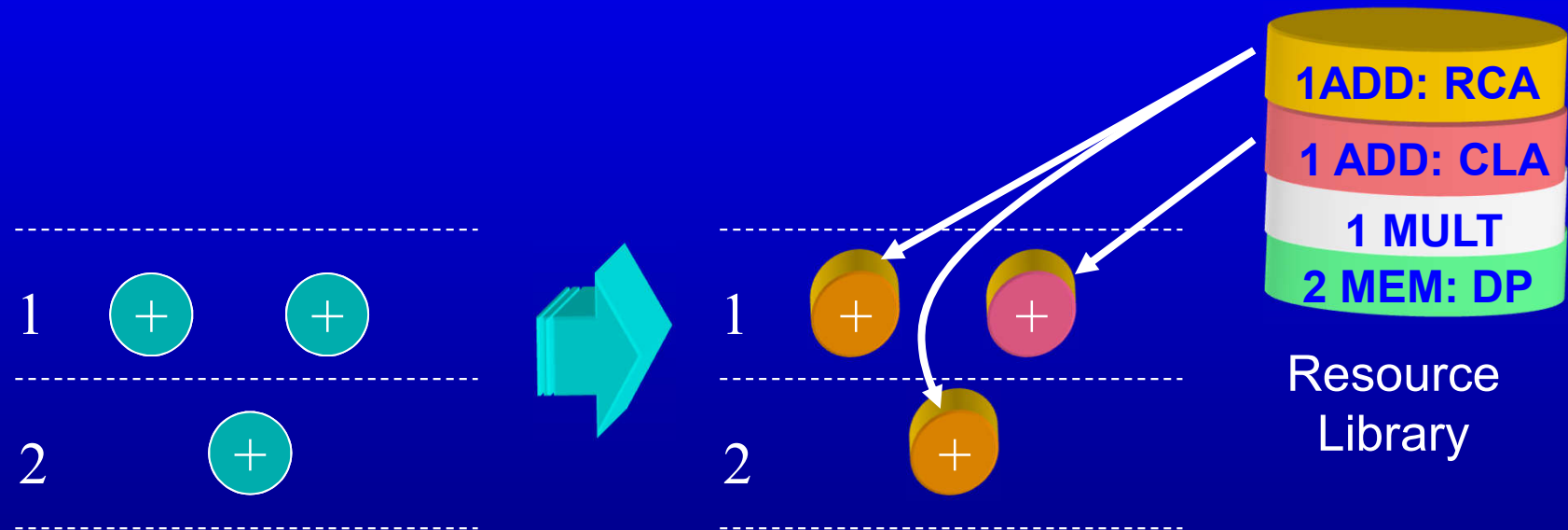
Schedule



FSM

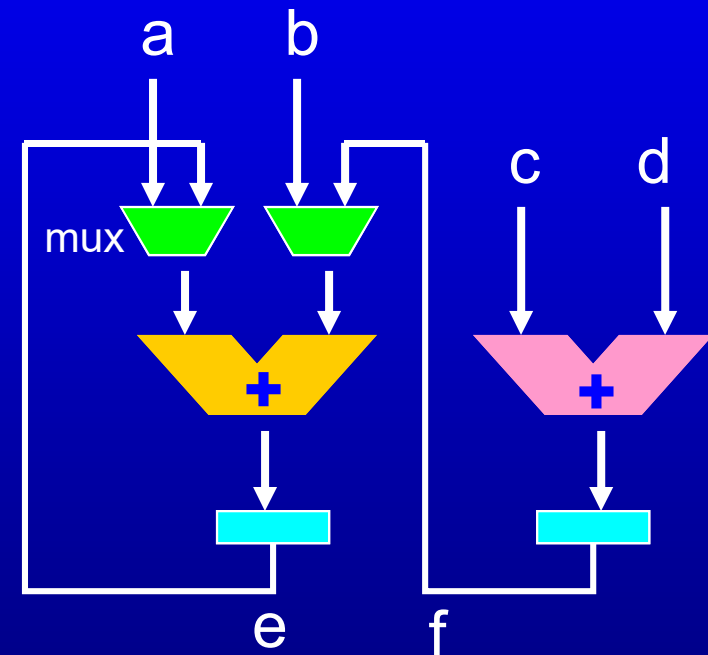
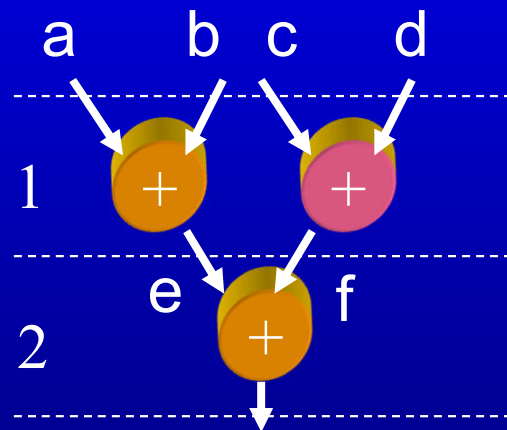
# Function Unit Binding

Mapping operators to Function Units



# FU Binding Leads to Datapath Architecture

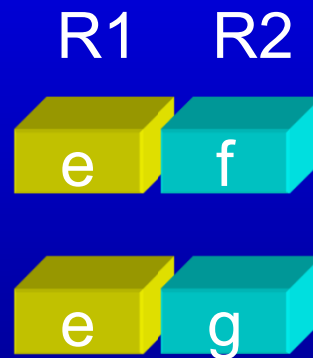
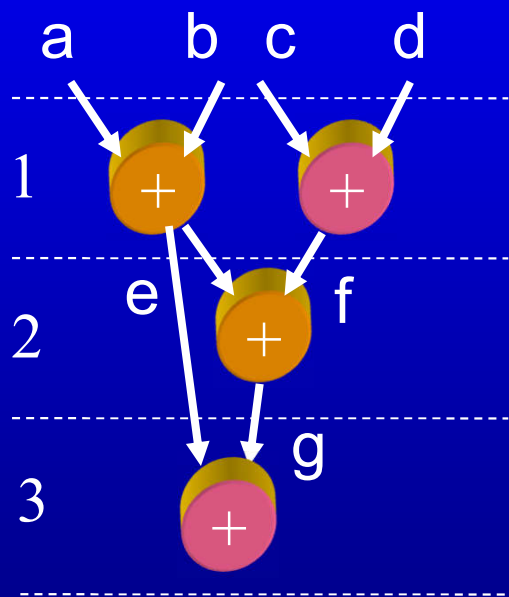
FU Binding determines FU + Input Mux Interconnections



**Datapath**

# Register Allocation

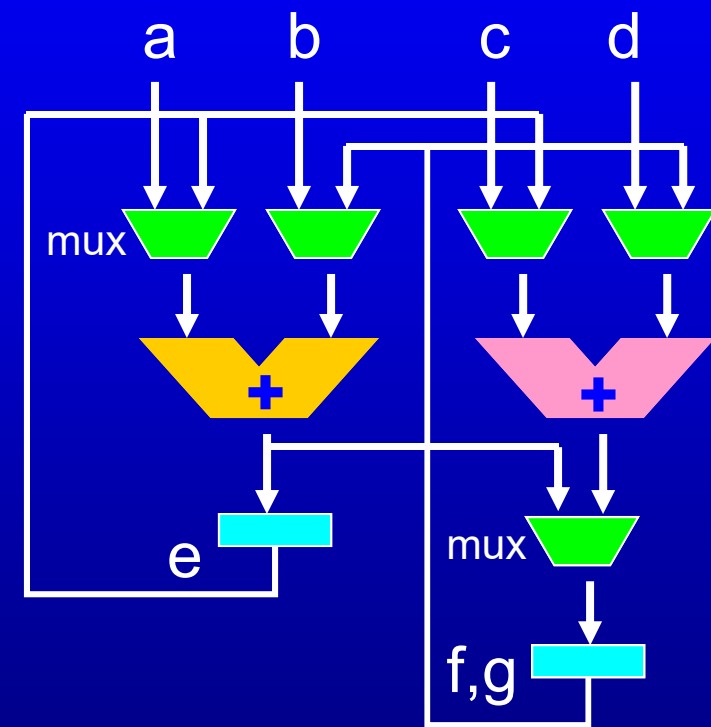
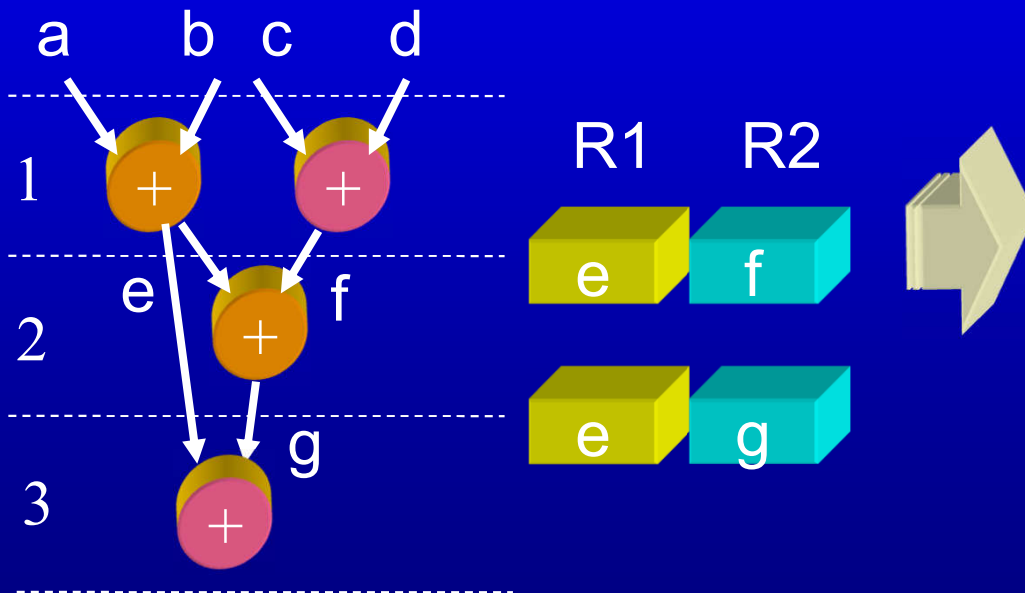
Assigning Variables to Registers



Variable persisting across clock boundary implies Register

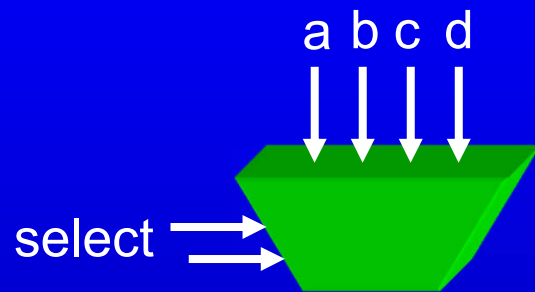
# FU Binding + Register Allocation Completes the Datapath

Register Allocation determines # Registers  
and Reg. Input Mux Interconnections

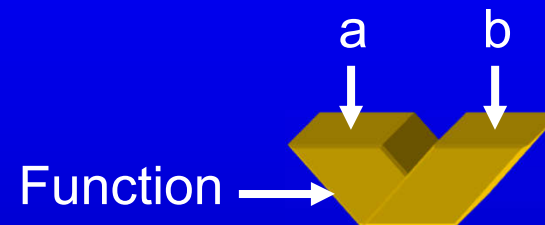


Datapath

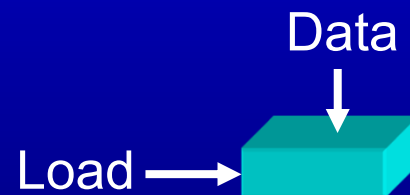
# Completing the HW



Mux Select Signals



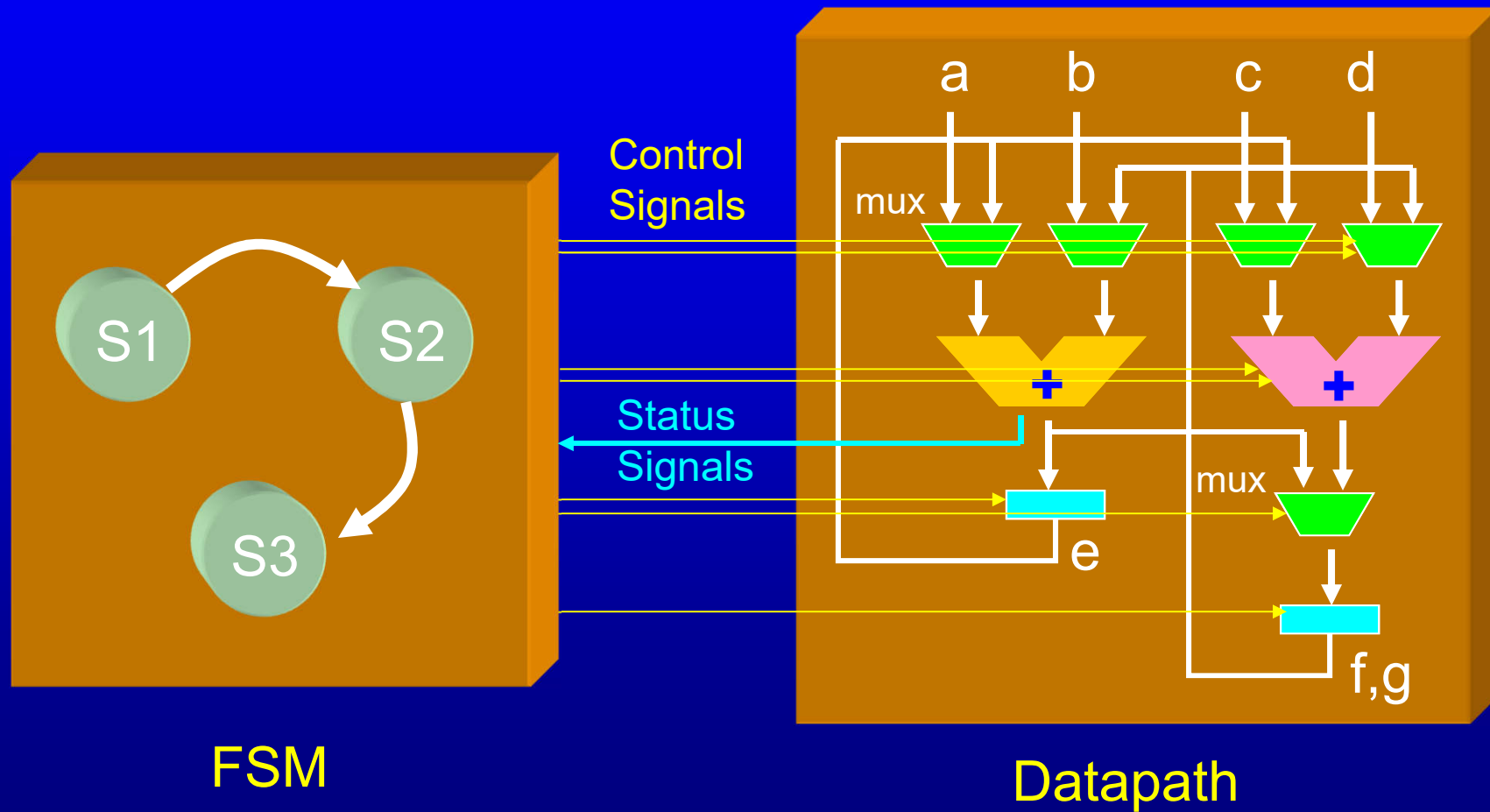
ALU Function Signals  
(Selecting from +, -, >, etc.)



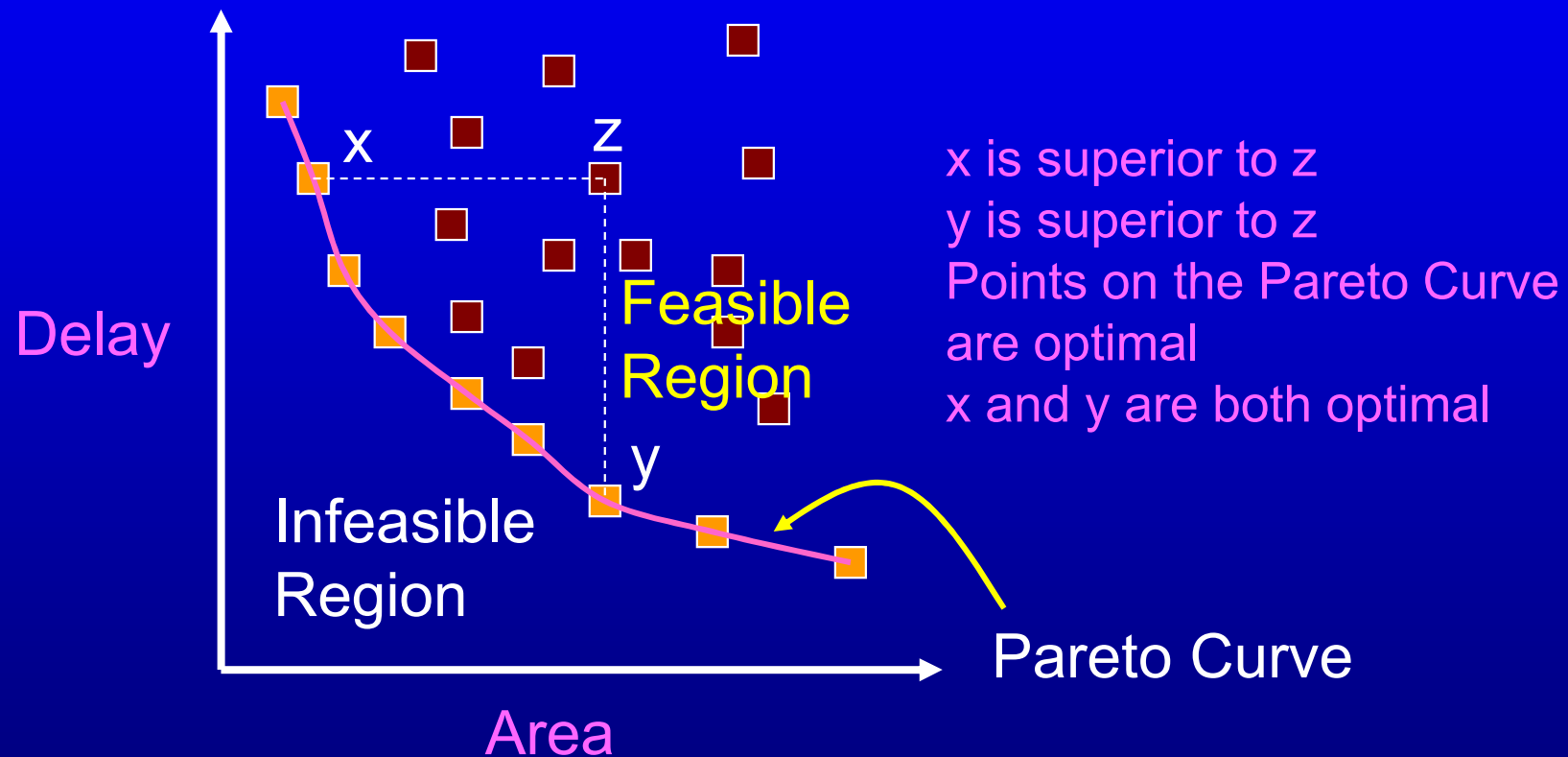
Register Load Signals



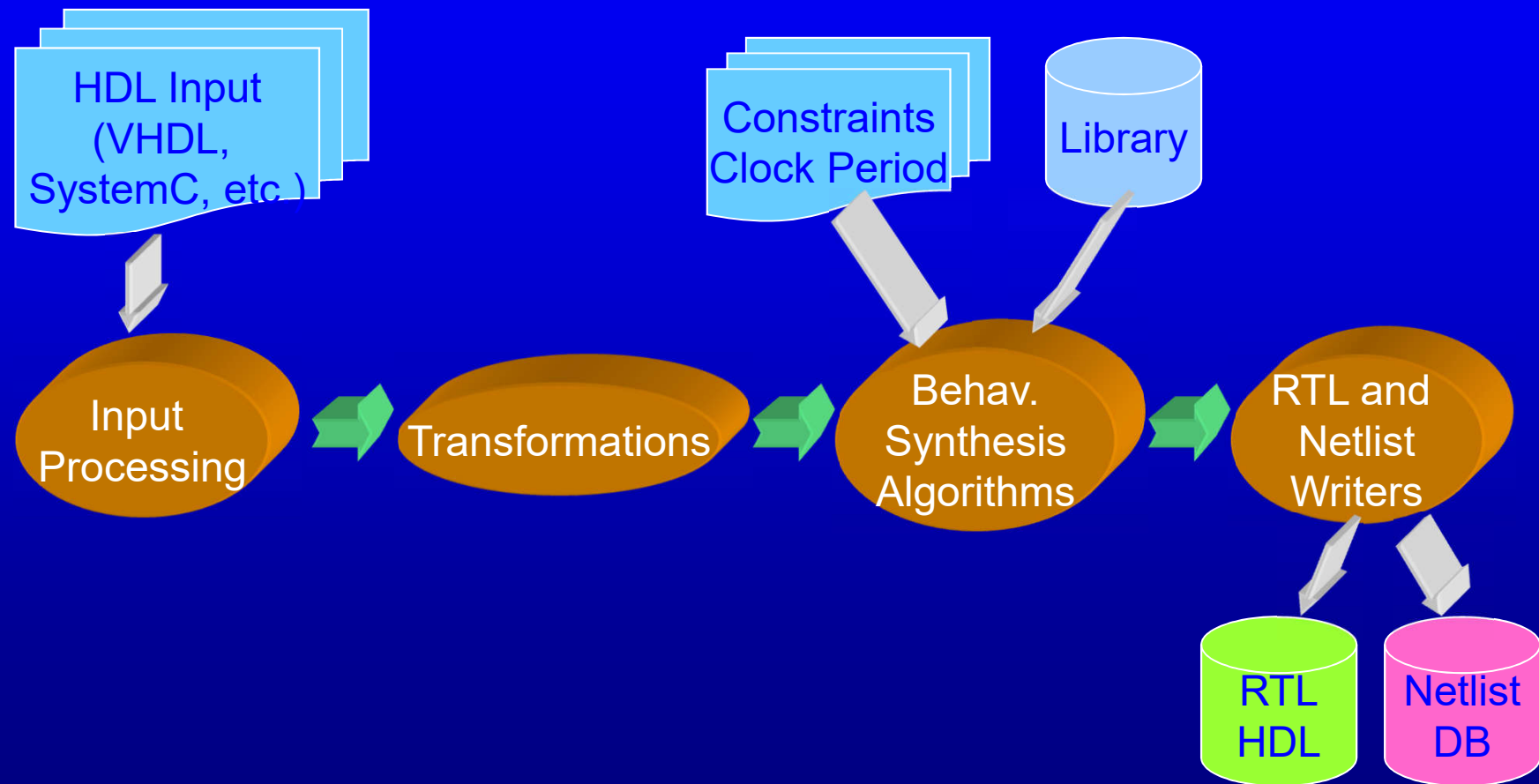
# Control Signals Generated by FSM



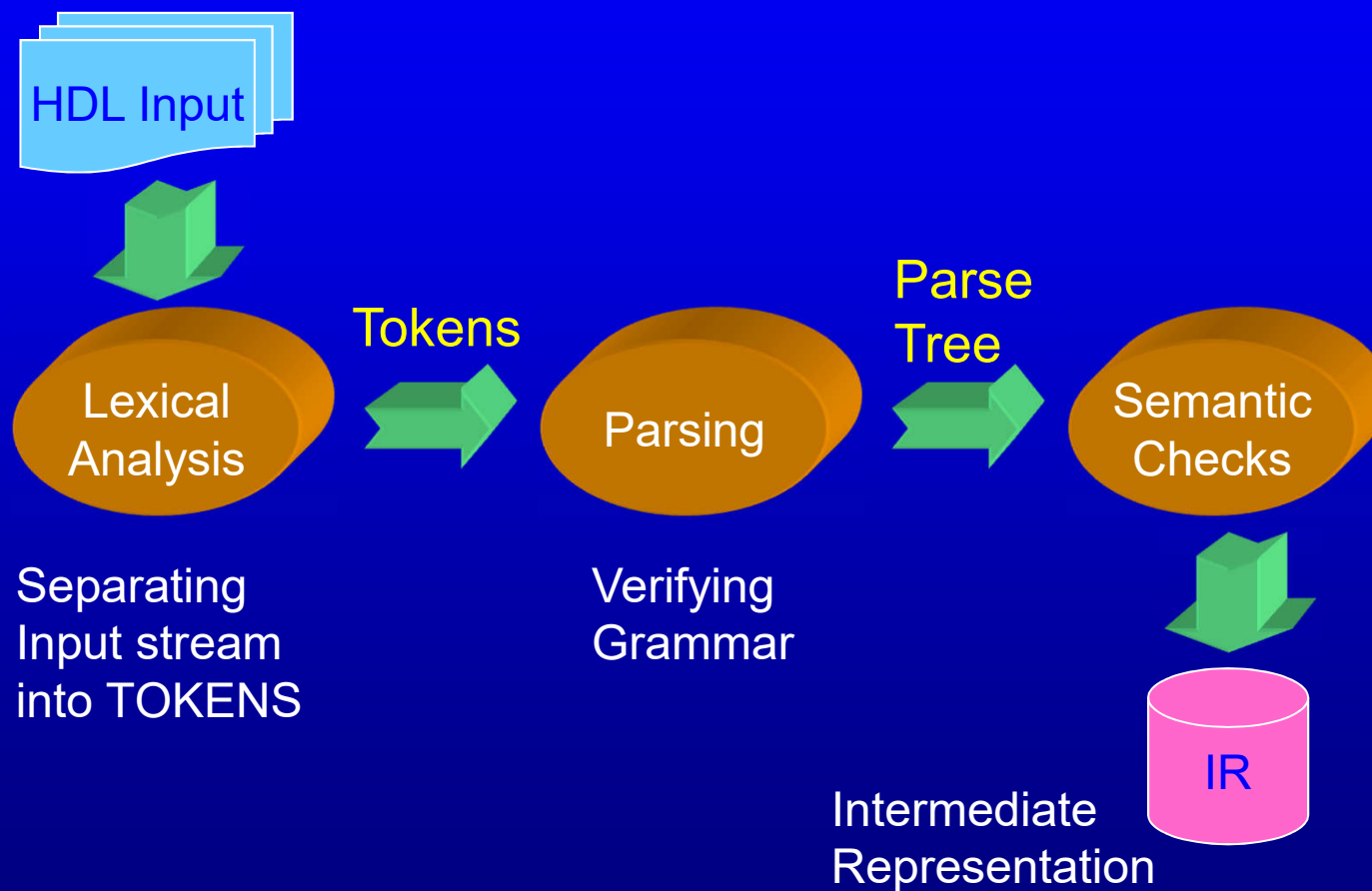
# Design Space



# Behavioural Synthesis Flow



# Front-end: Input Processing



# Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

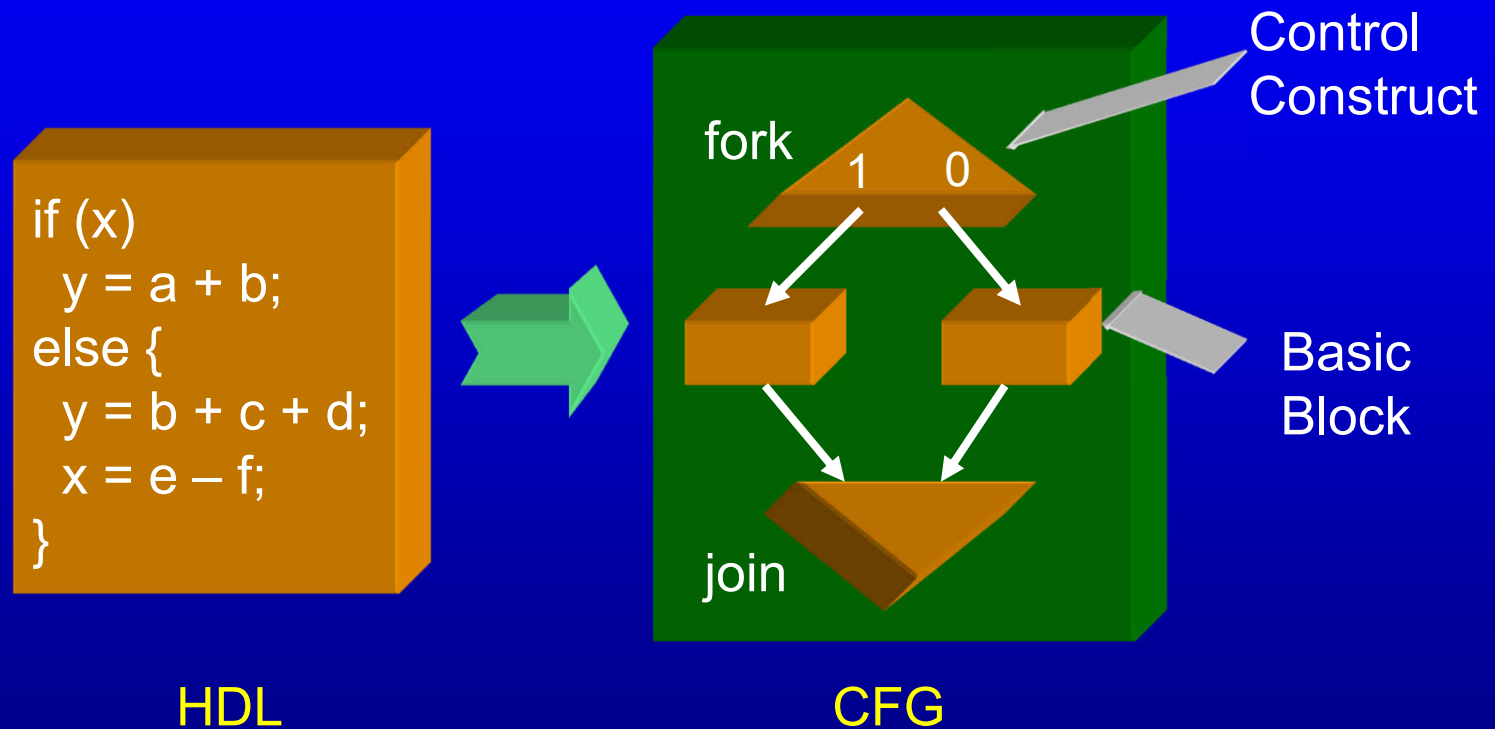
# Design Representation

- Language-independent
- Easy to add front-end for new language
- Easy to transform behaviours
  - traversal
  - insertion, deletion, and replacement
- Right representation different for different parts of the flow
  - Abstract Syntax Tree (AST) after parsing
  - Control Data Flow Graph during synthesis

# Control Data Flow Graph (CDFG)

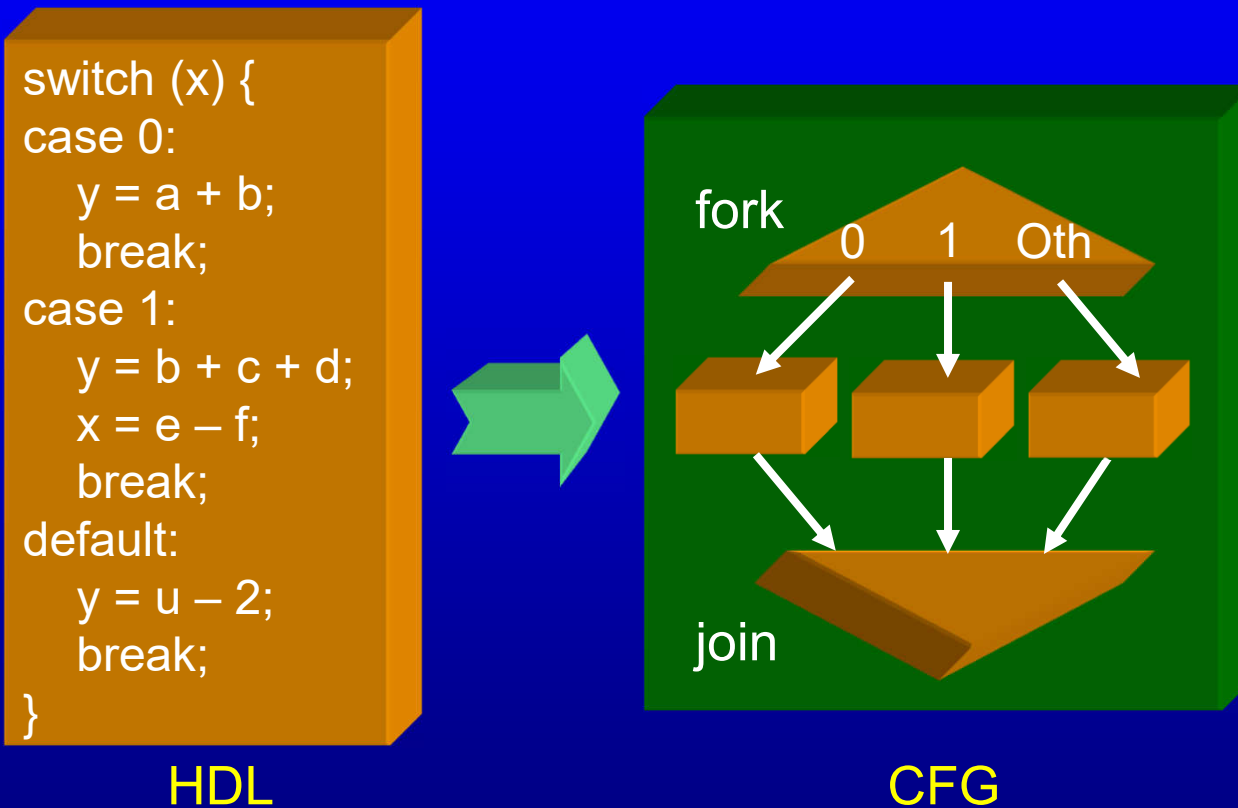
- Combination of:
  - Control Flow Graph (CFG)
    - Sequencing
    - Conditional Branching
    - Looping
  - Data Flow Graph (DFG)
    - Computation
    - Assignment

# CFG for IF Statement





# CFG for CASE Statement



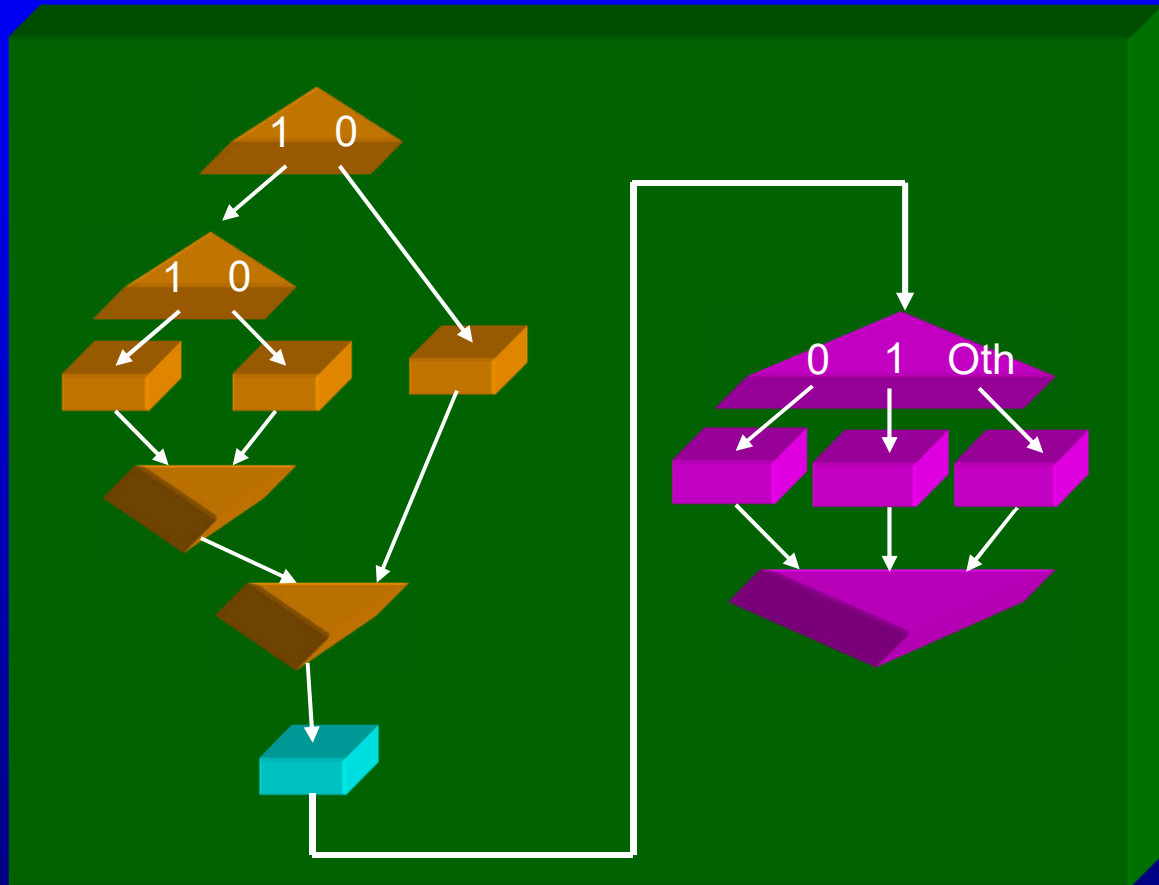
# CFG Composition

```
if (x)...
    if (y) ...
    else ...
else ...
```

```
x = y + 5;  
y = z;
```

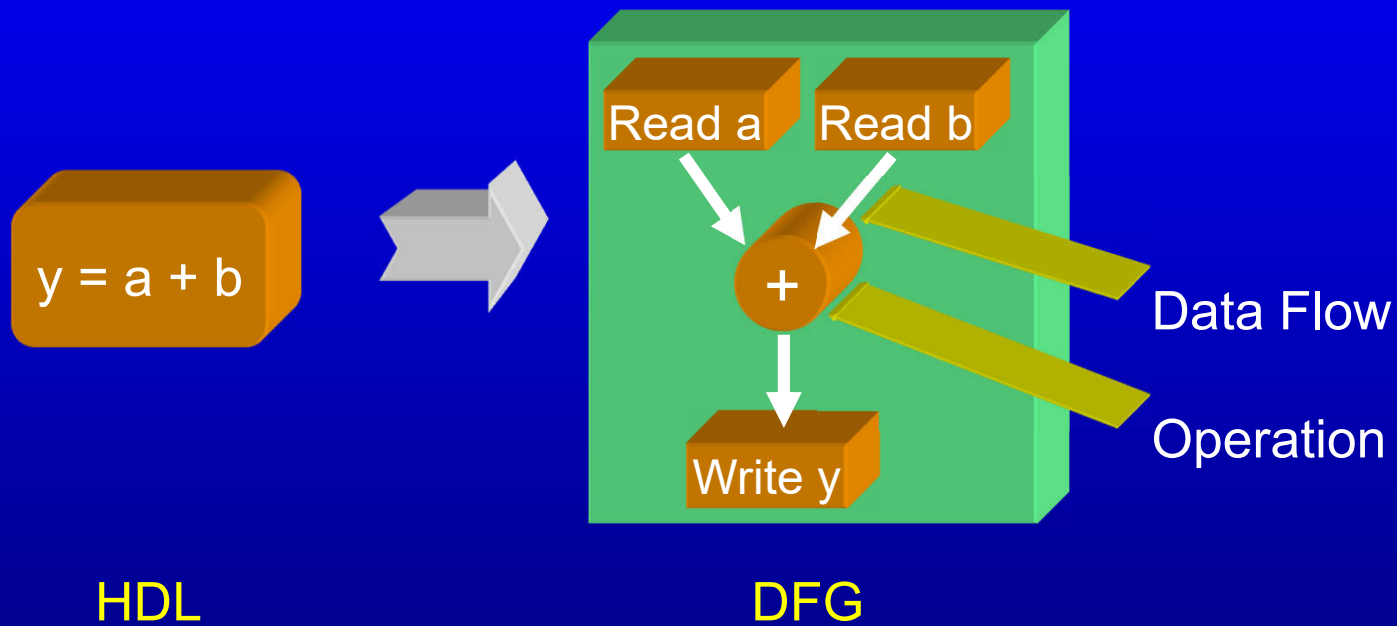
```
switch (x) {
  case 0:
  case 1:
  default:
}
```

# HDL



# CFG

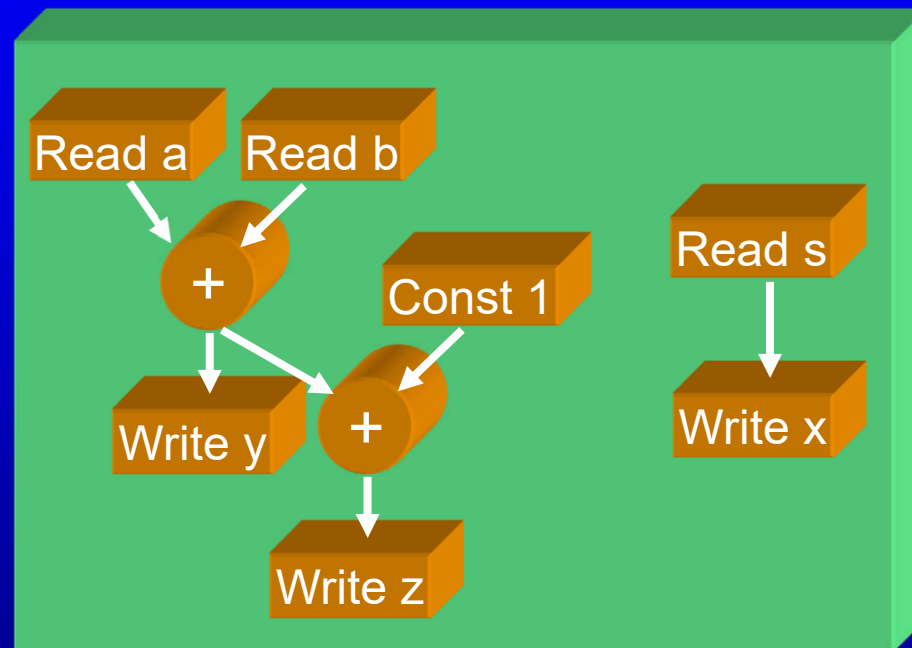
# DFG for Simple Operation



# DFG Composition

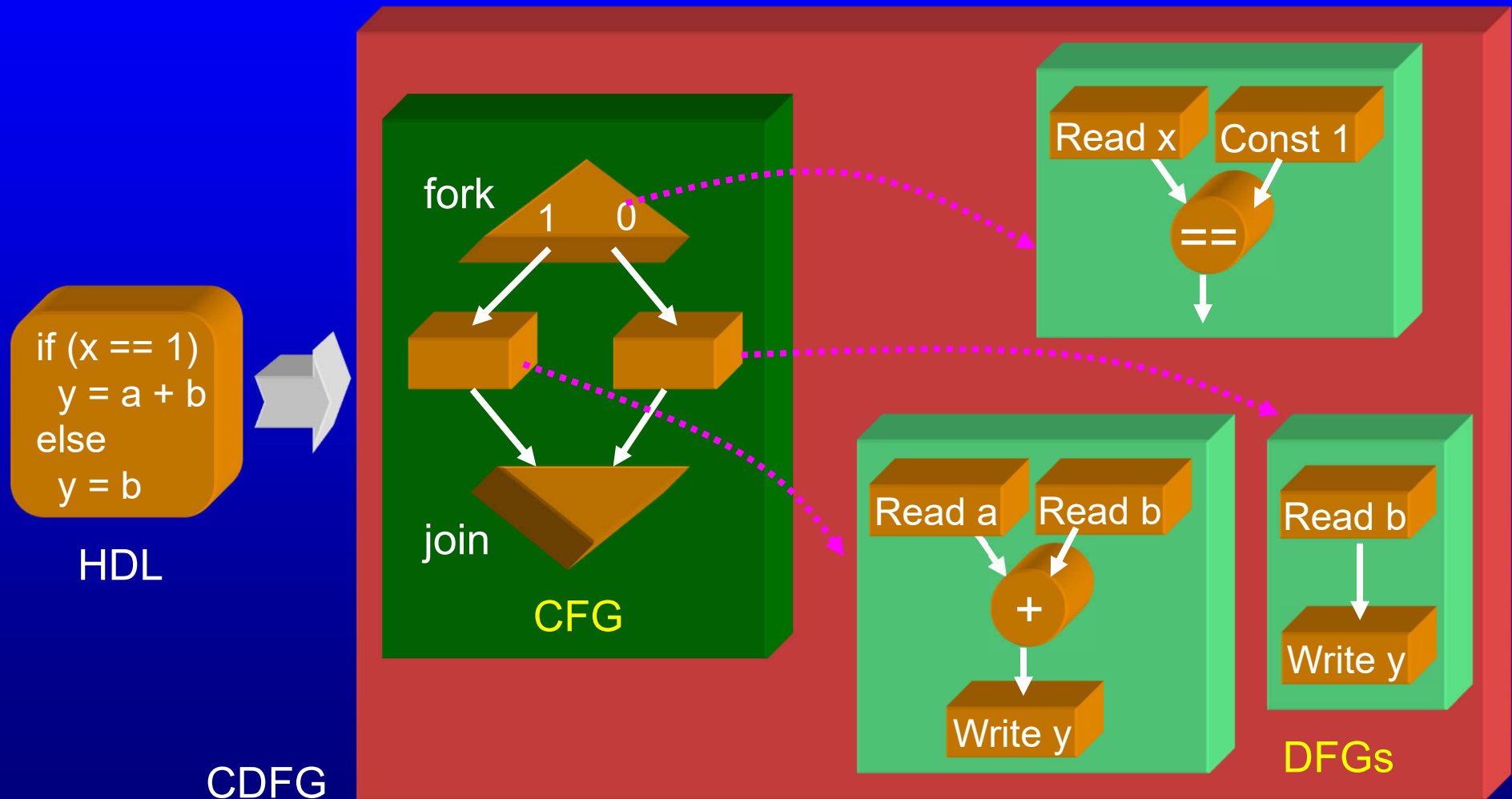
$y = a + b$   
 $z = y + 1$   
 $x = s$

HDL



DFG

# Control Data Flow Graph (CDFG)



# Contents

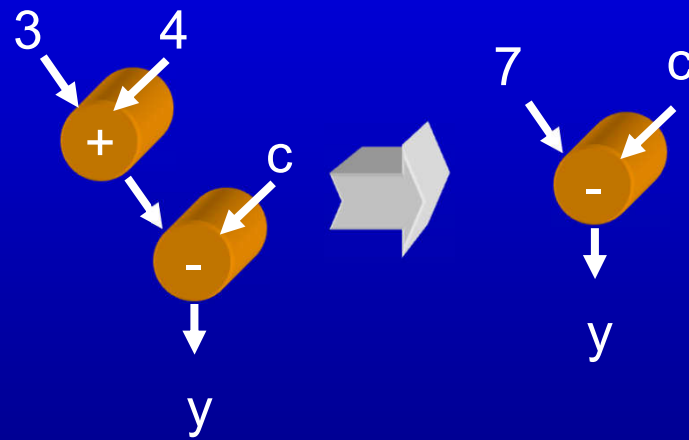
- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

# CDFG Transformations

- **Compiler Transformations: Mostly independent of Hardware**
  - strength reduction
  - common subexpression elimination
  - constant folding
  - dead code elimination
  - constant propagation
  - loop invariant code motion
  - loop unrolling
  - function inlining

# Compiler Transformations: Constant Folding

- Evaluating constant expressions leads to simplification

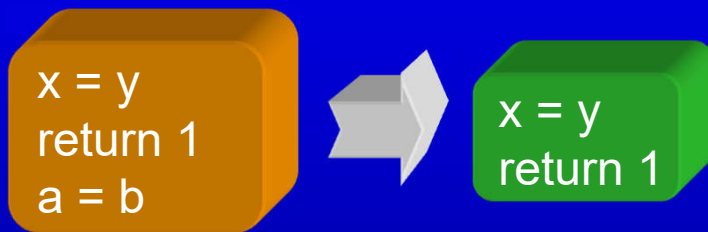


Eliminate  $(3 + 4)$  operation from CFG

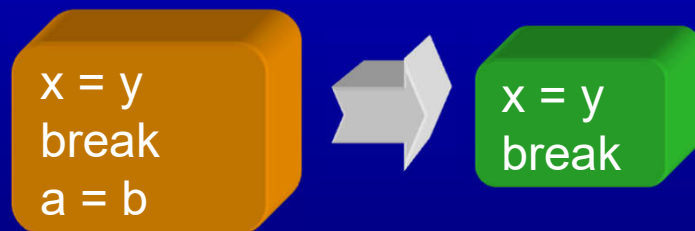


# Compiler Transformations: Dead Code Elimination

- Delete redundant parts of the CDFG



Eliminate assignment (a = b)



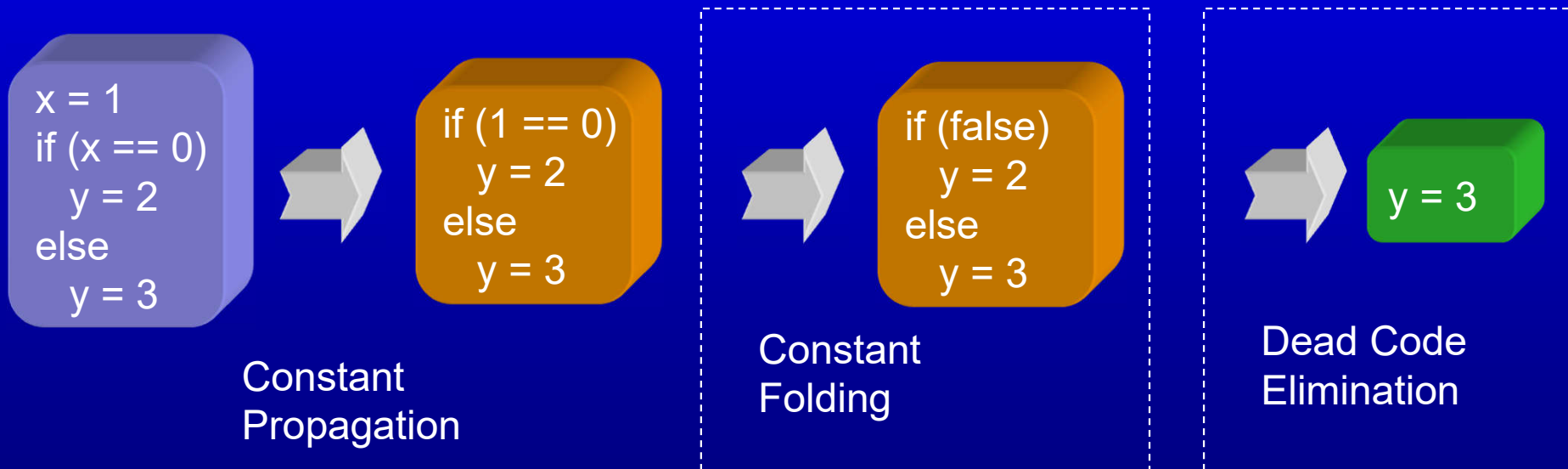
Eliminate assignment (a = b)



Eliminate THEN branch

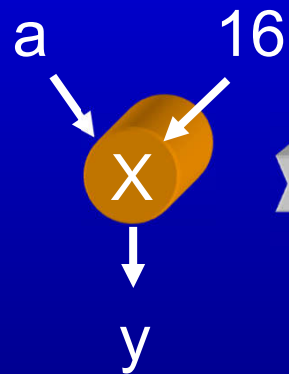
# Compiler Transformations: Constant Propagation

- Propagating constant value may lead to constant folding opportunity

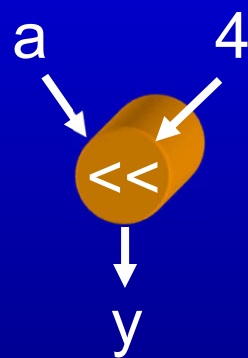


# Compiler Transformations: Strength Reduction

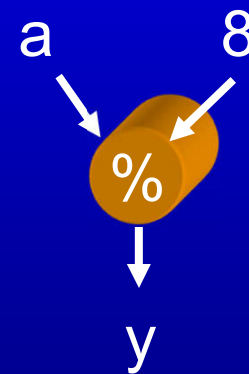
- Replacing expensive operation by cheaper operation



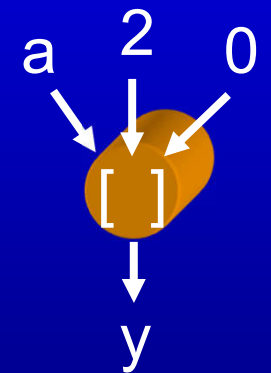
Multiplication  
by  $2^n$



Left Shift



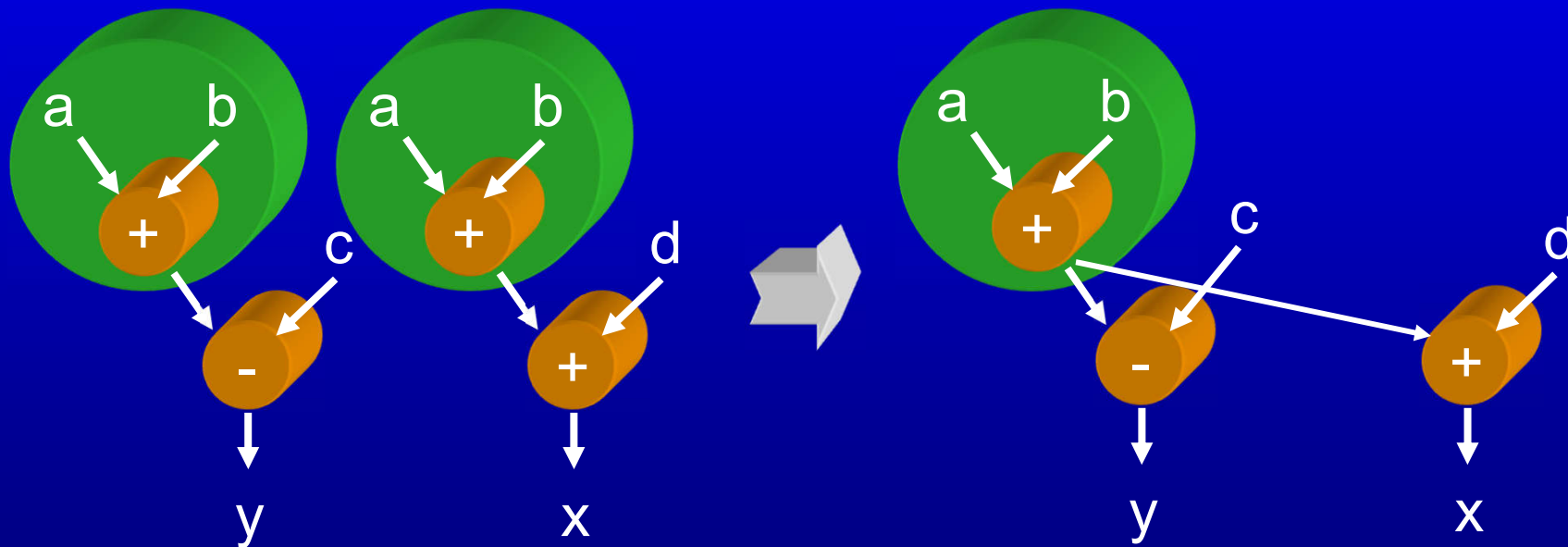
Modulo  $2^n$



Part Select

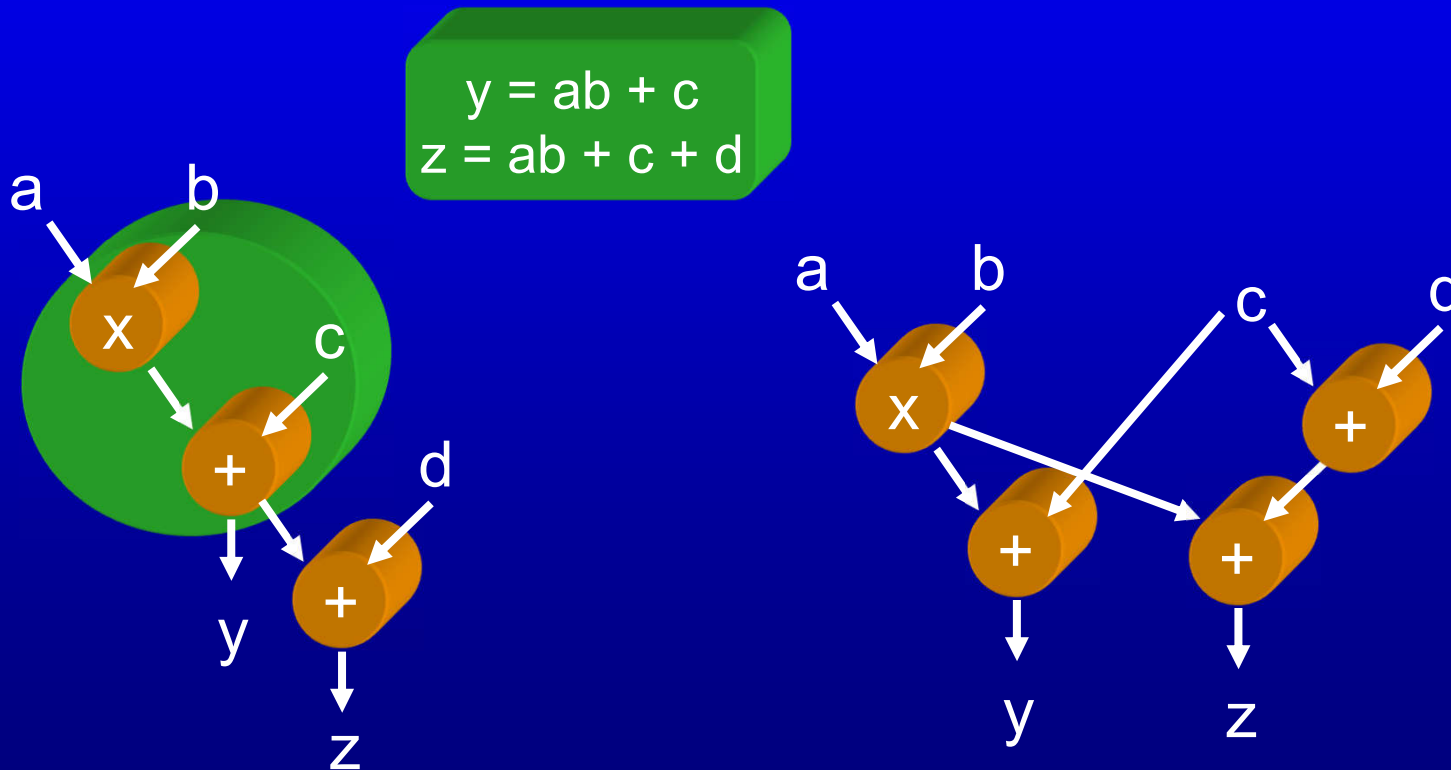
# Compiler Transformations: Common Sub-expression Elimination

- Avoid duplicating expressions



# Common Sub-expression Elimination: Area vs. Delay Trade-off

Eliminating CSE may extend critical path



# Compiler Transformations: Loop Invariant Code Motion

- Remove invariant code out of the loop

```
for (i = 0; i < 8; i++) {  
  y = A + B  
  z = y + i  
  C[i] = z  
}
```

y is invariant



```
y = A + B  
for (i = 0; i < 8; i++) {  
  z = y + i  
  C[i] = z  
}
```

(A + B) evaluated  
only once

# Compiler Transformations: Loop Unrolling

- Unrolling loop may expose parallelism

```
for (i = 0; i < 3; i++) {  
    A[i] = B[i] + x  
}
```

Loop forces  
sequential execution

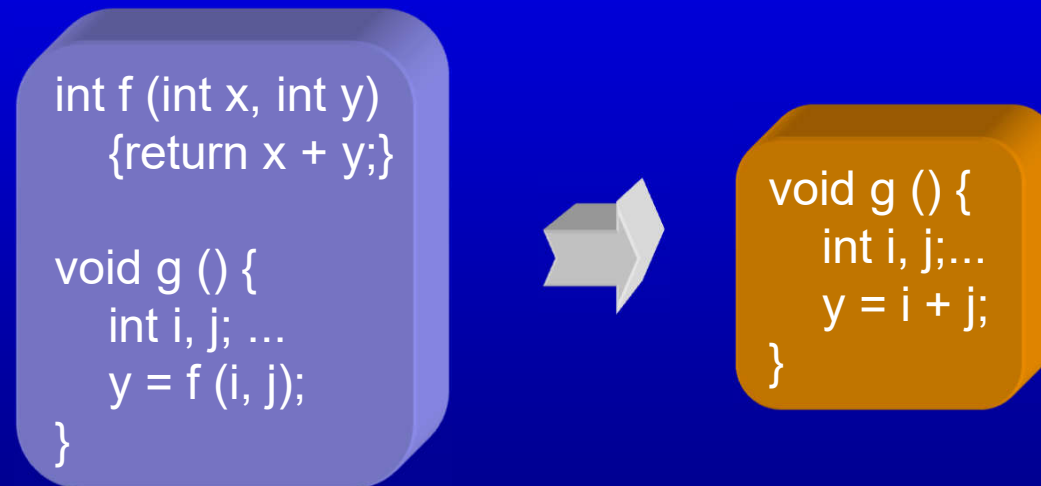


```
A[0] = B[0] + x  
A[1] = B[1] + x  
A[2] = B[2] + x
```

No dependence across  
iterations. Parallel execution  
possible

# Compiler Transformations: Function Inlining

- Inlining overcomes function call overhead





# Contents

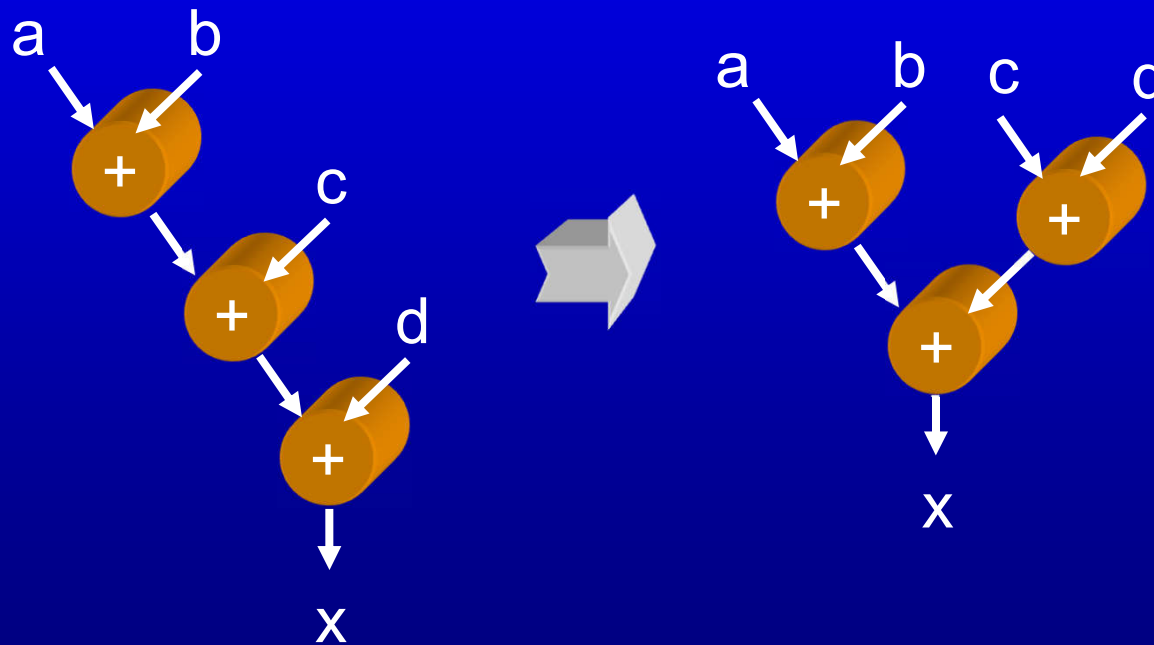
- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

# CDFG Transformations

- HW-specific Transformations
  - tree height reduction
  - control flow to data flow
  - flow graph flattening
  - boolean optimisations
  - pattern matching for complex FUs

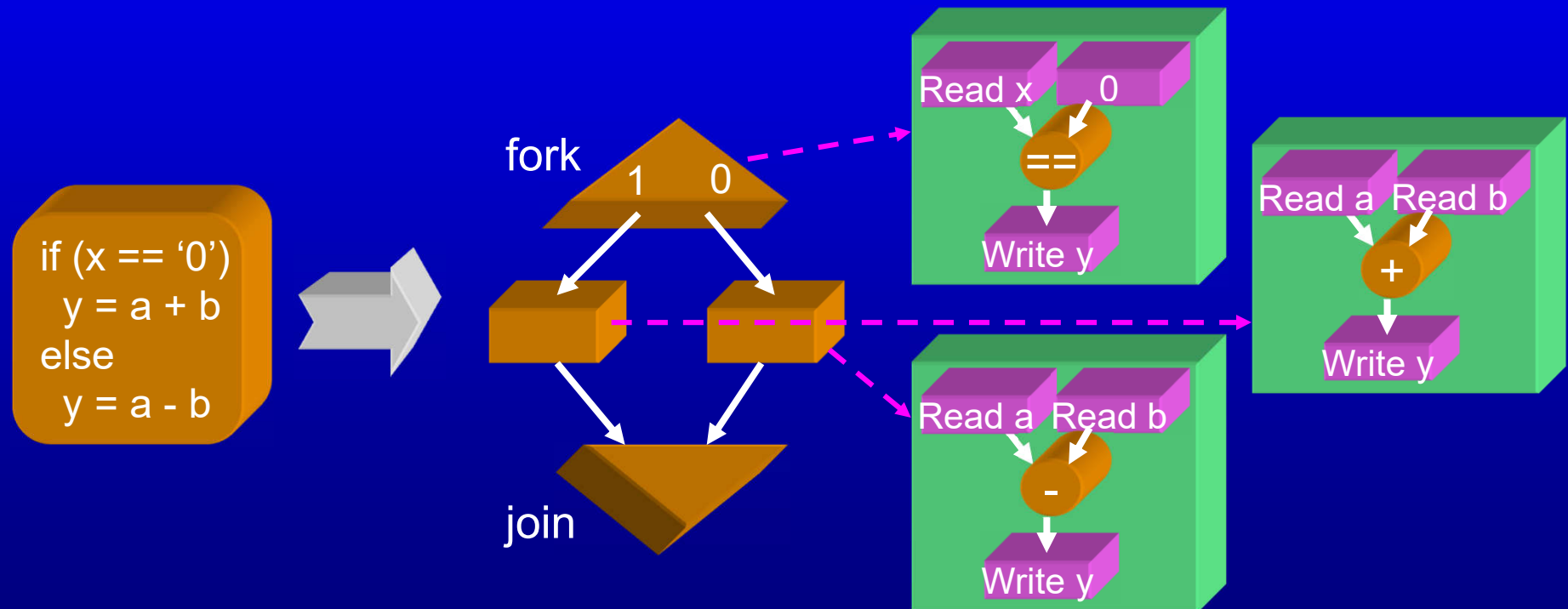
# Hardware Transformations: Tree Height Reduction

- Balancing DFG exposes parallelism



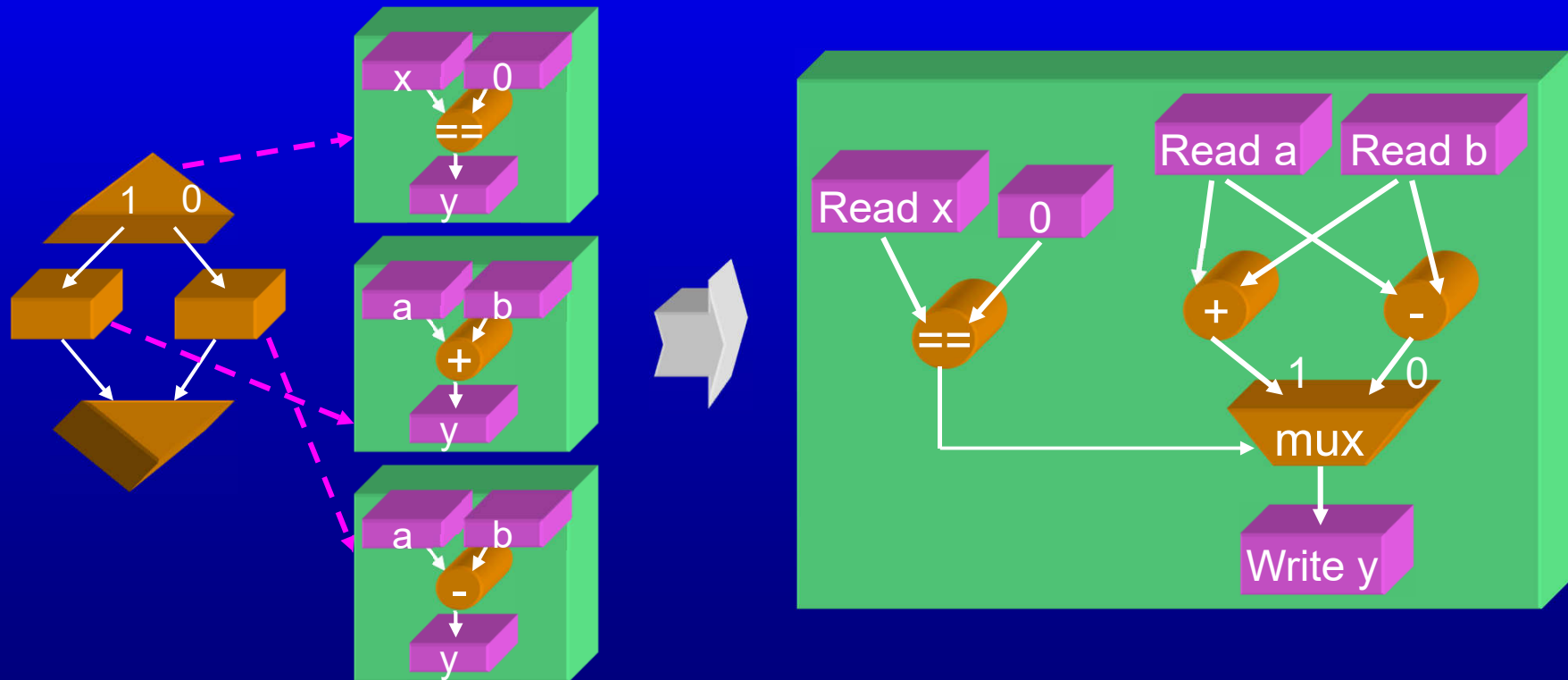
# Hardware Transformations: Control Flow to Data Flow

- Simple boolean tests don't need separate state



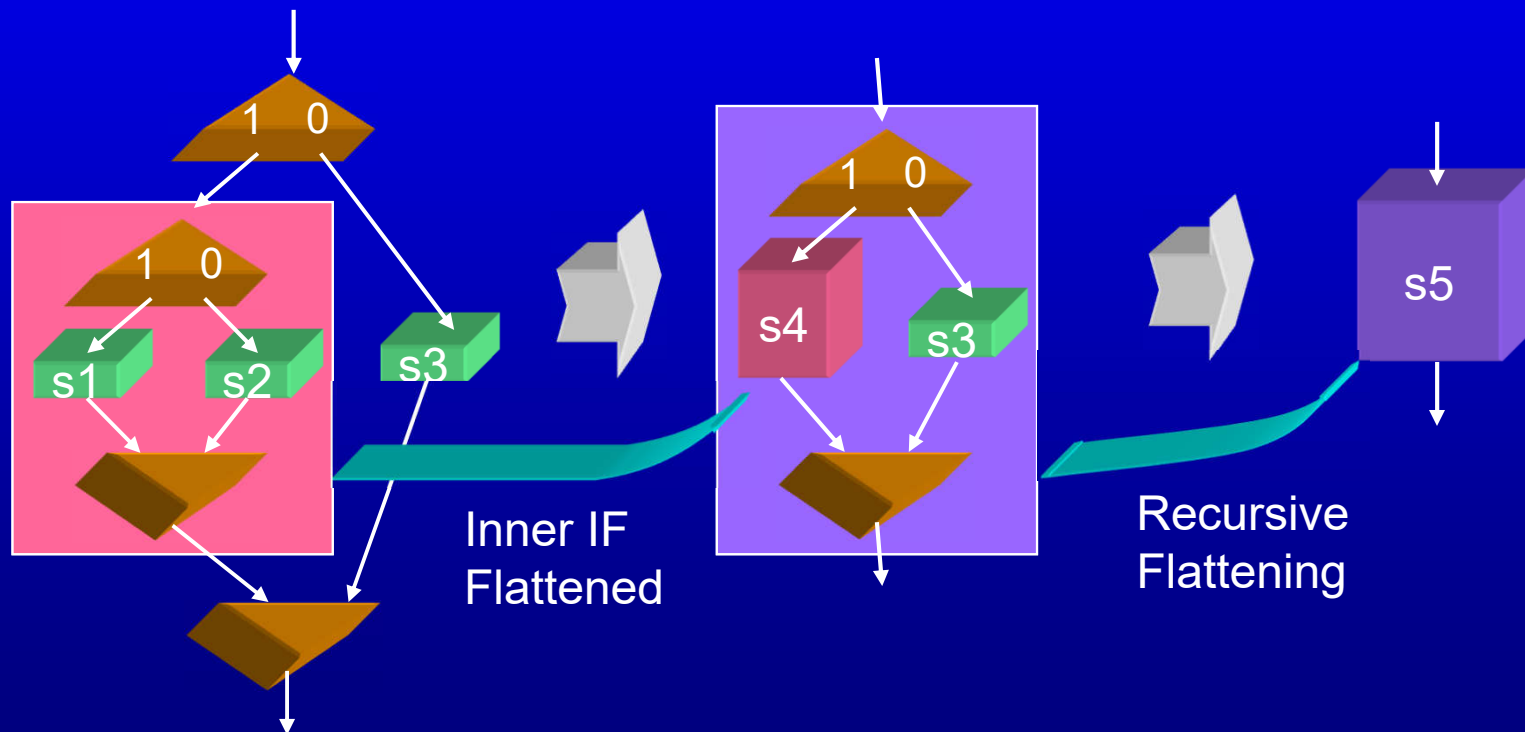
# Hardware Transformations: Control Flow to Data Flow

- Convert CFG into DFG



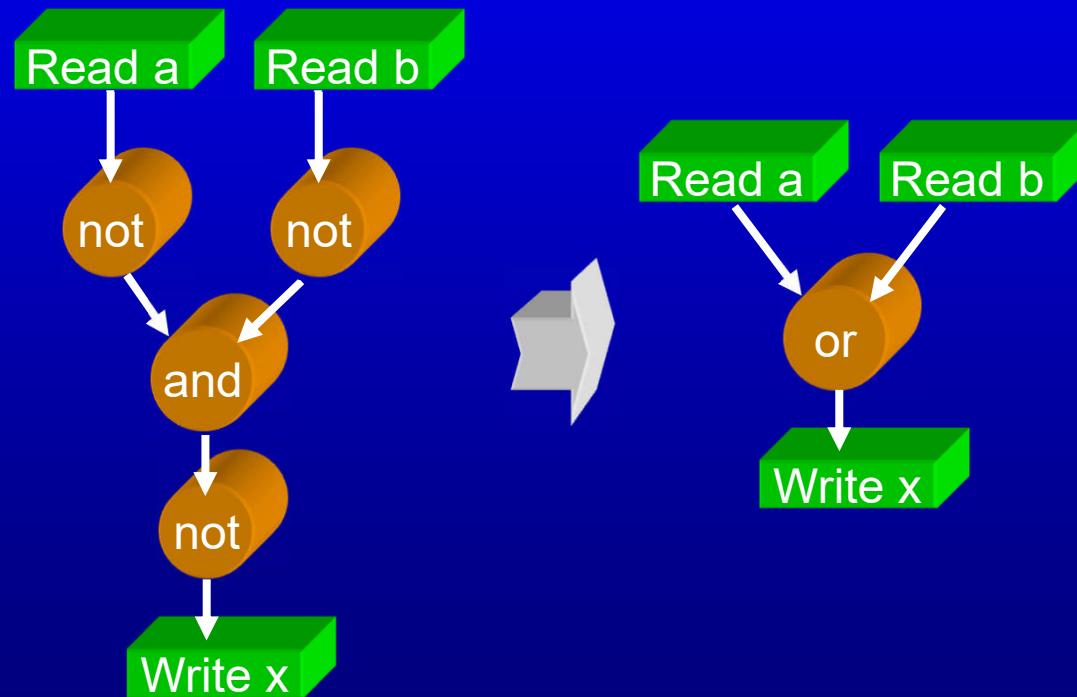
# Hardware Transformations: Flow Graph Flattening

- Repeated CFG to DFG conversion



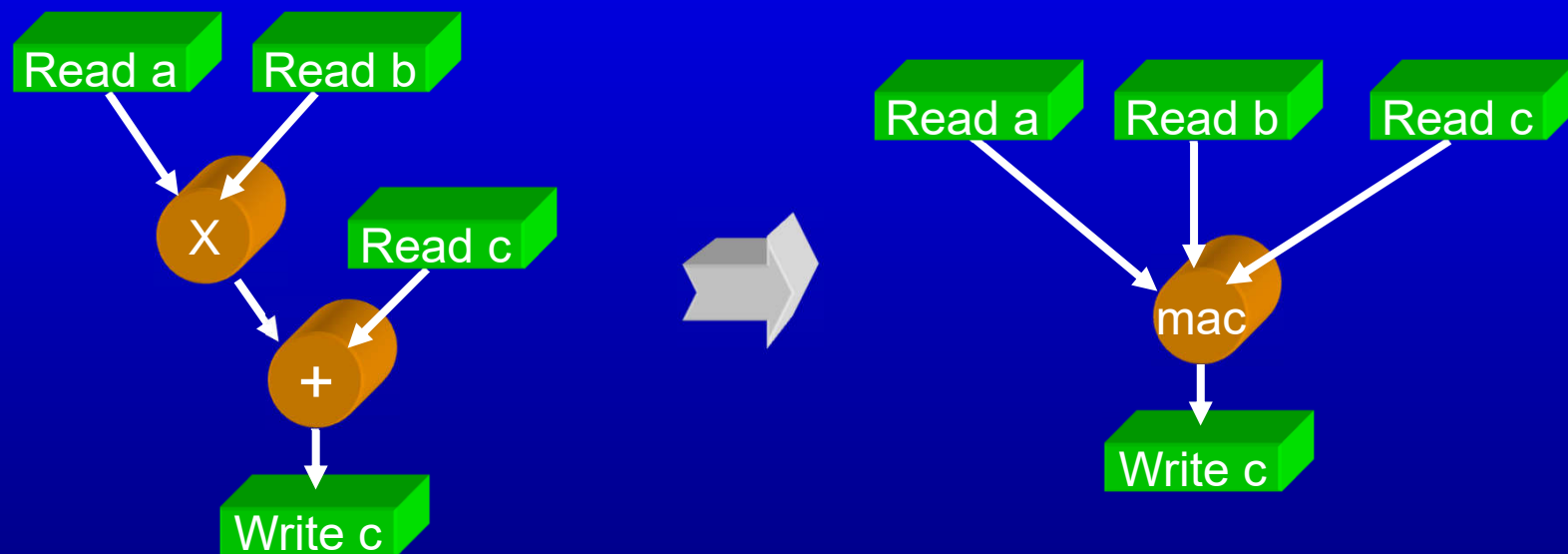
# Hardware Transformations: Boolean Optimisations

- Simple boolean transformations reduce CDFG complexity



# Hardware Transformations: Pattern Matching

- Use complex FUs in Resource Library



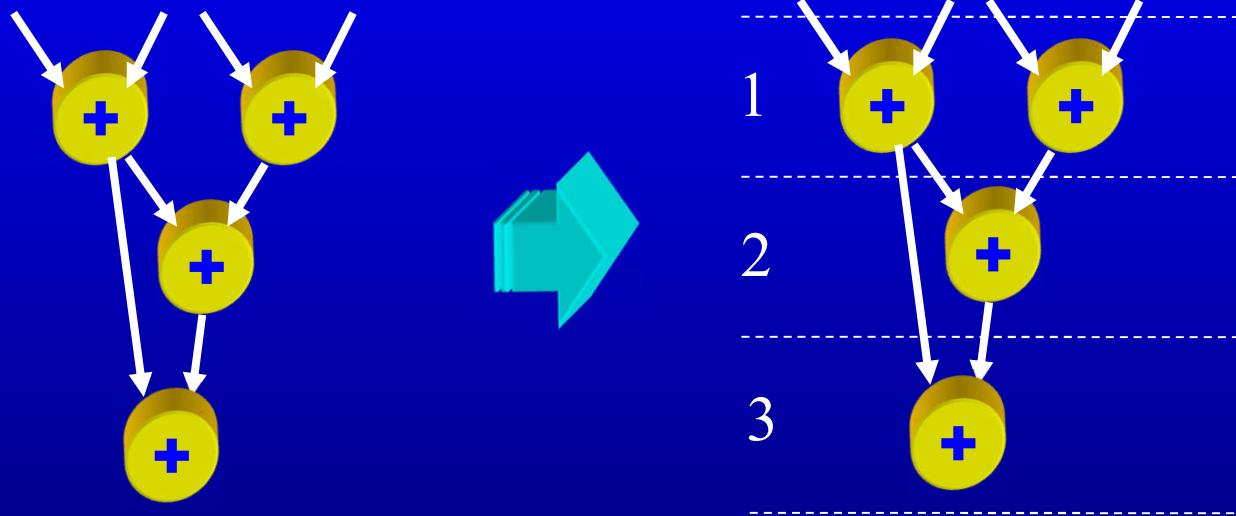


# Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

# Scheduling

Mapping operations to clock cycles



# Importance of Scheduling

- Key step in going from abstract to clocked domain
- Optimal solution difficult: NP-complete problem
  - exact polynomial-time algorithm may not exist
  - heuristics needed

# Scheduling Problem Formulation

- Resource-constrained scheduling
  - Given Resources
  - Determine minimum-delay schedule
- Time-constrained scheduling
  - Given time deadline
  - Determine minimum-resource schedule

# Basic Scheduling

- **Assumptions**
  - No conditionals – scheduling within basic block only
  - All operations take one clock cycle
  - Single function FUs
- **Assumptions relaxed later**
  - multi-cycle operations
  - chaining of operations into same cycle
  - multi-function units (e.g., ALU)
  - conditionals in CDFG

# Scheduling with Unlimited Resources

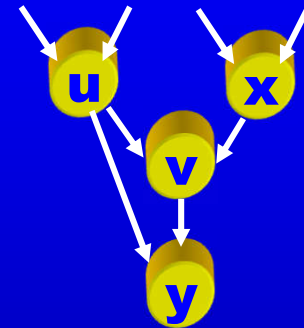
- As Soon As Possible (ASAP)
- As Late As Possible (ALAP)

Give upper and lower bounds for  $\sigma(t)$

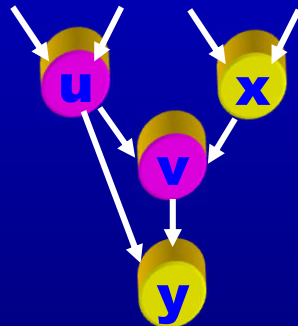
$$\sigma_{ASAP}(t) \leq \sigma(t) \leq \sigma_{ALAP}(t)$$

# Terminology

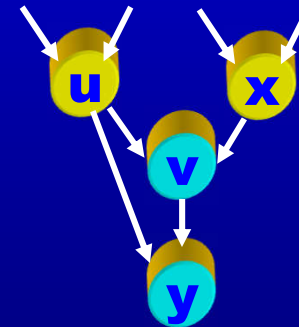
- set of nodes  $V = \{u, v, x, y\}$  represent operations
- edges  $u \rightarrow v$ , etc. represent precedence
- $u$  is an immediate predecessor of  $v$
- $v$  is an immediate successor of  $u$



$PRED(y) = \{u, v\}$   
Set of immediate  
predecessors



$SUCC(u) = \{v, y\}$   
Set of immediate  
successors



# ASAP Scheduling Algorithm

for all  $u$  with  $\text{PRED}(u) = \Phi$

$\sigma(u) = 1$

$V = V - \{u\}$

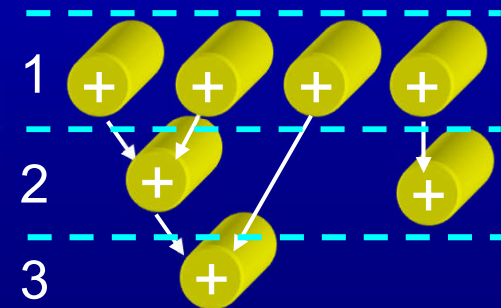
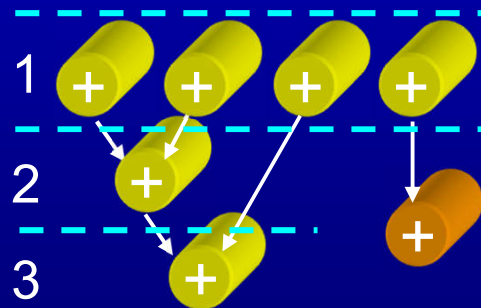
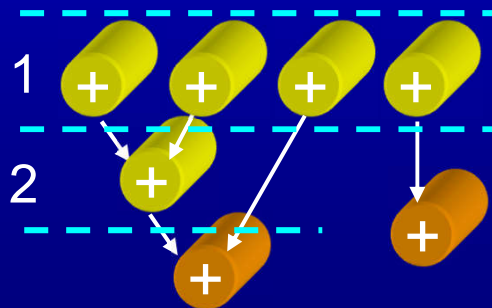
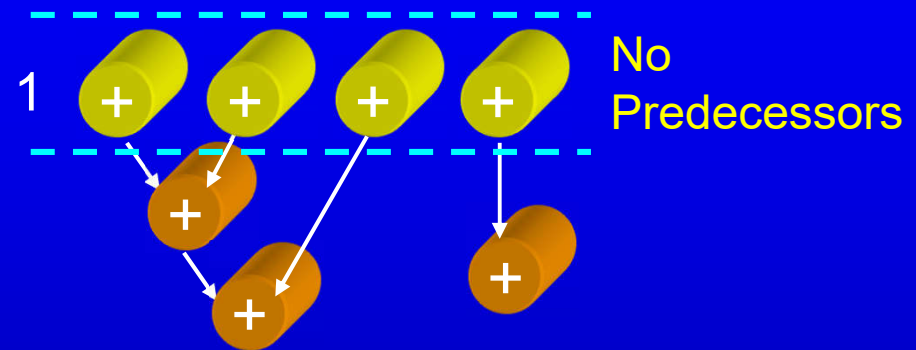
while  $V \neq \Phi$

for all  $u \in V$

if all  $\text{PRED}(u)$  already scheduled

$\sigma(u) = 1 + \text{MAX}(\sigma(w), w \in \text{PRED}(u))$

$V = V - \{u\}$





# ALAP Scheduling: Time constraint T

for all  $u$  with  $\text{SUCC}(u) = \Phi$

$\sigma(u) = T$

$V = V - \{u\}$

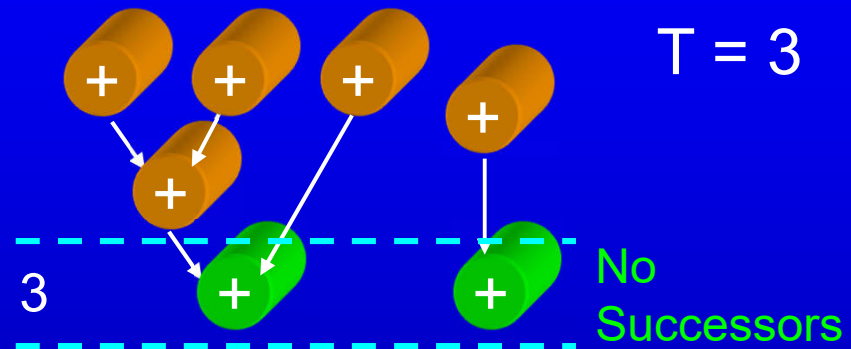
while  $V \neq \Phi$

for all  $u \in V$

if all  $\text{SUCC}(u)$  already scheduled

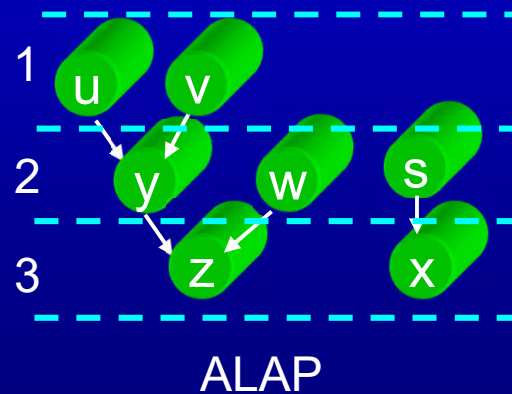
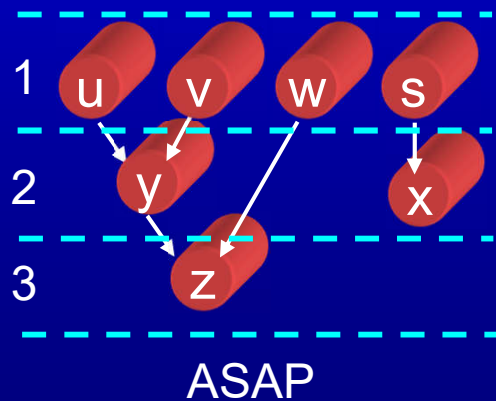
$\sigma(u) = \text{MIN}(\sigma(w), w \in \text{SUCC}(u)) - 1$

$V = V - \{u\}$



# ASAP vs. ALAP Schedule

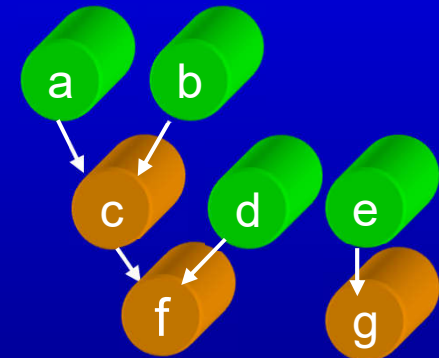
- Nodes in **critical path** have identical schedule
  - schedule is fixed for these nodes
- For other nodes, the range of possible values is:  $\sigma_{ASAP}(t) \leq \sigma(t) \leq \sigma_{ALAP}(t)$



$\sigma(u) = 1$   
 $\sigma(v) = 1$   
 $\sigma(y) = 2$   
 $\sigma(z) = 3$   
 $1 \leq \sigma(w) \leq 2$   
 $1 \leq \sigma(s) \leq 2$   
 $2 \leq \sigma(x) \leq 3$

# Resource Constrained Scheduling: List Scheduling

- Assign priorities
  - selection criteria among operations competing for resources
- At each control step (clock cycle)
  - Generate ready-list
    - set of operations whose predecessors are already scheduled
  - If size (ready-list) > #resources, schedule operations with highest priority

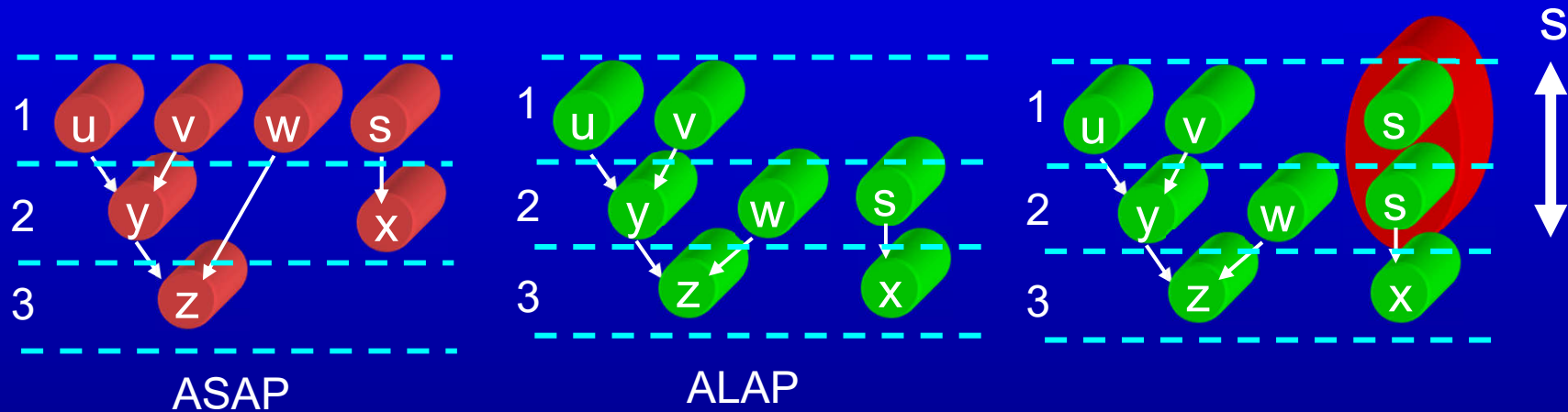


# List Scheduling

- Algorithm leaves Priority Function undefined
  - can be customised
  - can be simple or sophisticated
- Popular priority function: MOBILITY

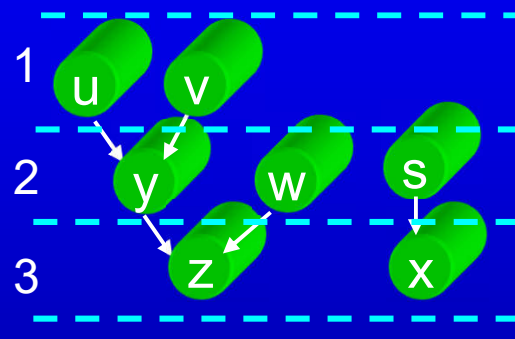
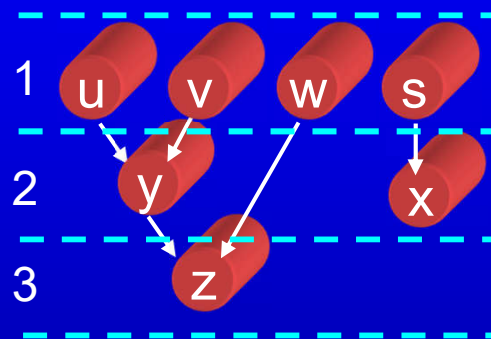
# The MOBILITY Priority Function

$$\text{Mobility } \mu(v) = \sigma_{ALAP}(v) - \sigma_{ASAP}(v)$$



$$\mu(s) = \sigma_{ALAP}(s) - \sigma_{ASAP}(s) = 2 - 1 = 1$$

# The MOBILITY Priority Function



$\sigma(u) = 1$   
 $\sigma(v) = 1$   
 $\sigma(y) = 2$   
 $\sigma(z) = 3$   
 $1 \leq \sigma(w) \leq 2$   
 $1 \leq \sigma(s) \leq 2$   
 $2 \leq \sigma(x) \leq 3$

Scheduling  
Flexibility

$\mu(u) = 0$   
 $\mu(v) = 0$   
 $\mu(y) = 0$   
 $\mu(z) = 0$   
 $\mu(w) = 1$   
 $\mu(s) = 1$   
 $\mu(x) = 1$

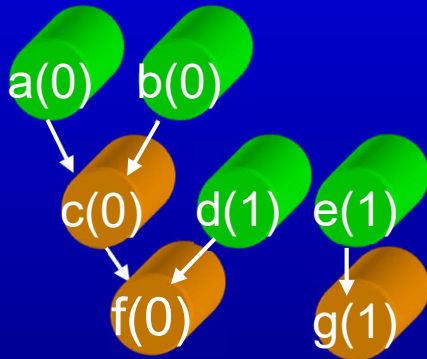
Mobility

- $\mu = 0$  implies no mobility
  - node is on critical path
- Lower  $\mu$  implies higher priority
  - delaying schedule for low- $\mu$  node increases probability of extending schedule

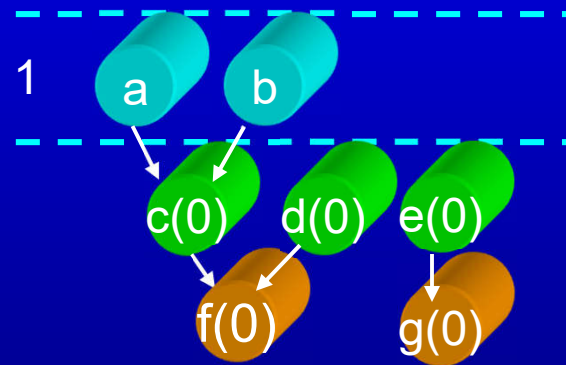
# List Scheduling using MOBILITY

$R = 2$

$V_{\text{ready}} = \{a, b, d, e\}$   
 $V_{\text{sel}} = \{a, b\}$



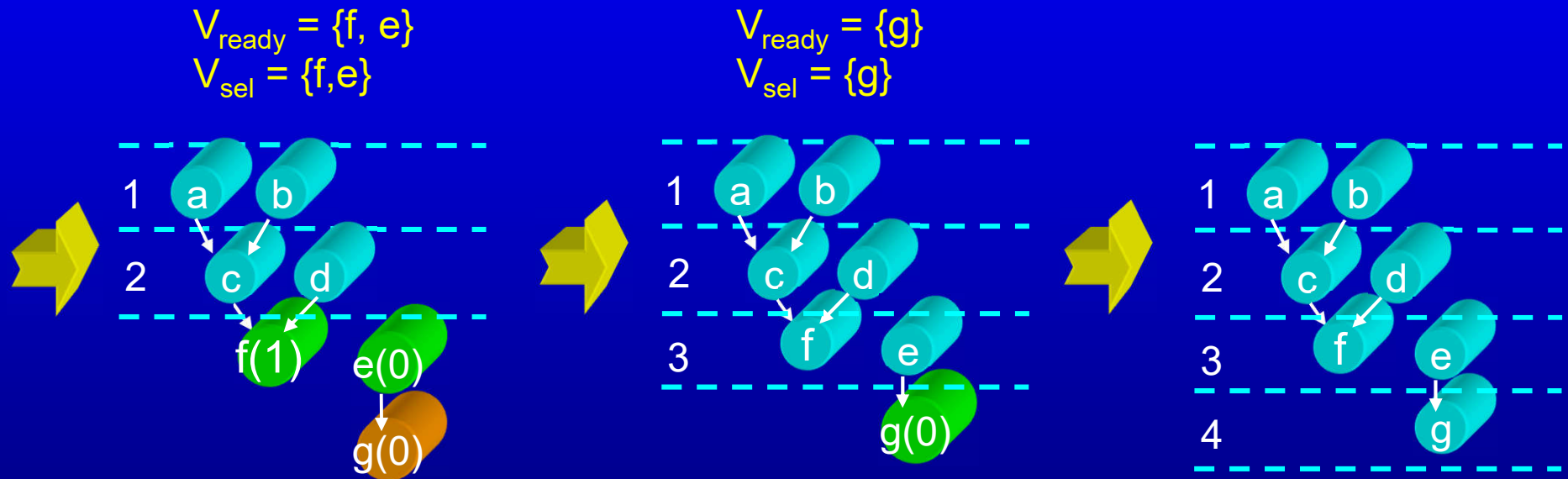
$V_{\text{ready}} = \{c, d, e\}$   
 $V_{\text{sel}} = \{c, d\}$



Mobilities Recomputed.  
Dynamic Priority Function

# List Scheduling using MOBILITY

$R = 2$





## Refined Model: Different Operation Types

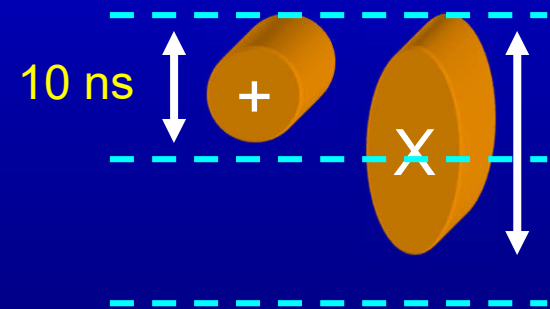
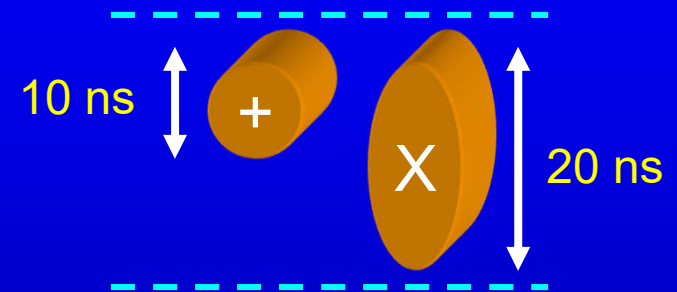
- Different operation types require different FUs
- How to handle in Scheduling algorithm?
  - maintain one ready-list per operation type

## Refined Model: Multi-function Units

- Multiple operations can be performed by same FU
  - ALU (+, -, AND, OR, etc.)
- May lead to smaller area compared to separate FUs

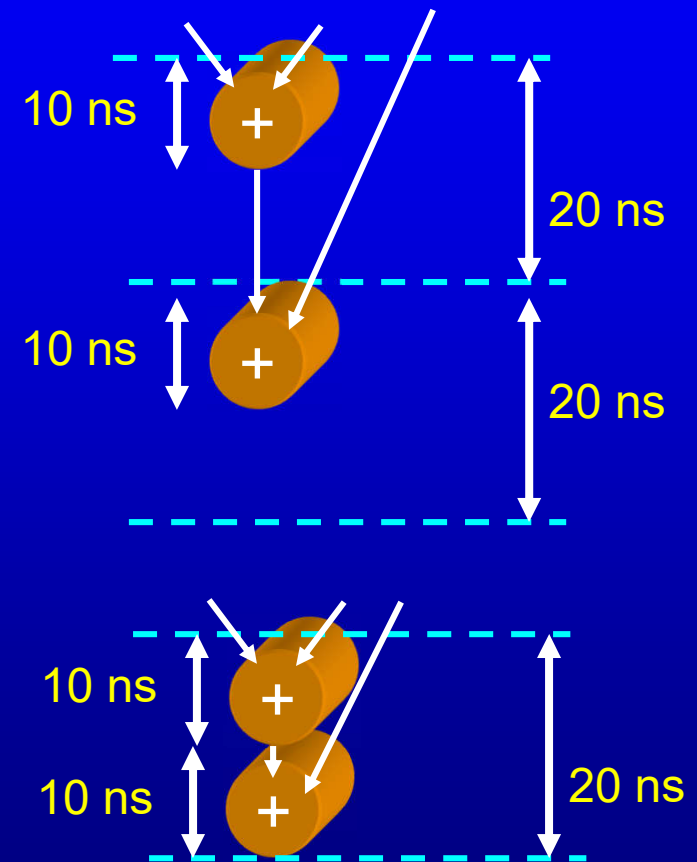
# Refined Model: Multi-cycled Operations

- FUs with different Delays
  - clock cycle  $\geq$  longest operation
  - low utilisation of faster FUs
- Multicycled Operations
  - an operation can take multiple cycles  
 $\text{cycles} = \lceil \text{delay} / \text{clock period} \rceil$
  - inputs have to be held stable
- How to extend scheduling algorithm?
  - check whether predecessor has completed



# Refined Model: Chaining

- FUs with small delays
  - clock cycle  $\gg$  operation delay
  - low utilisation of faster FUs
- Chained Operations
  - multiple FUs are required
- Extension to scheduling algorithm?
  - check whether operation can be chained with predecessor



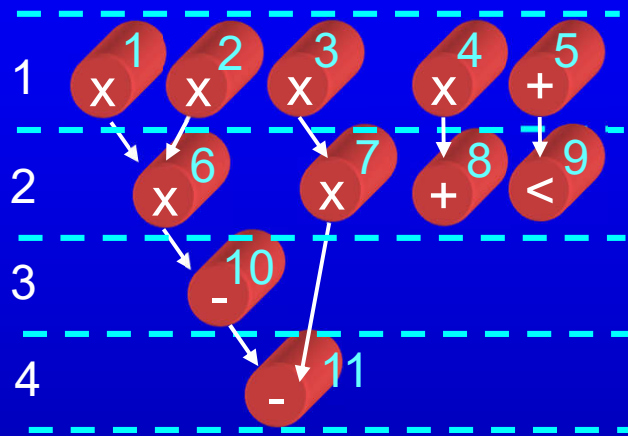
# Time Constrained Scheduling

- Latency constraint (deadline)
  - schedule a DFG in  $n$  cycles
- Time constraint between operations
  - min/max time between I/O operations
- Types
  - exact
  - min
  - max

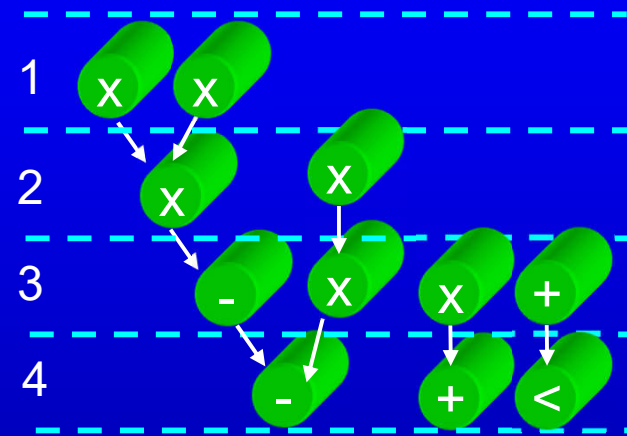
# Force Directed Scheduling (FDS)

- Classical time-constrained scheduling algorithm
- Minimises hardware subject to time constraint
- Works by uniformly distributing operations of same type across control steps
  - maximises utilisation of FU

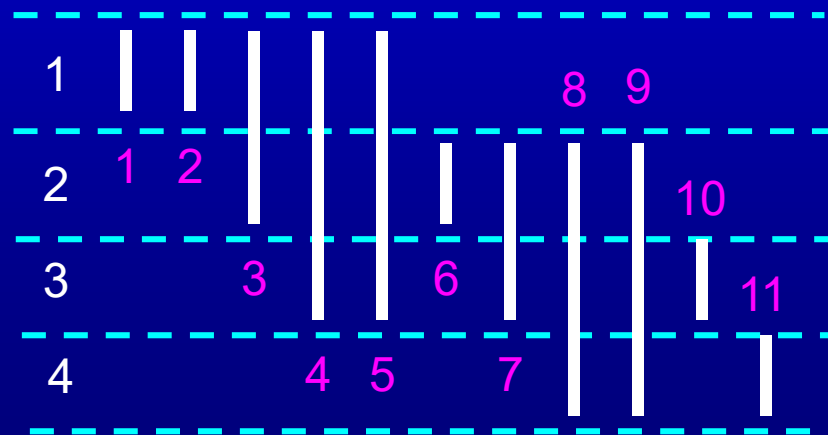
# FDS: Scheduling Ranges



ASAP

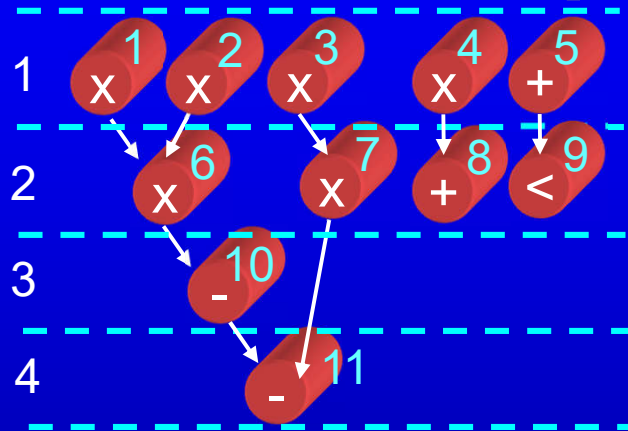


ALAP

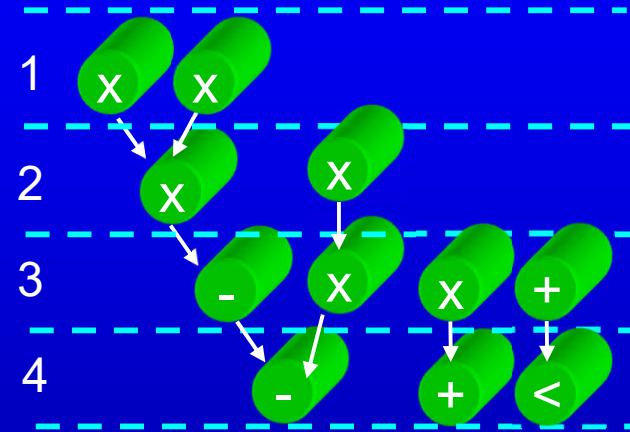


Scheduling Ranges

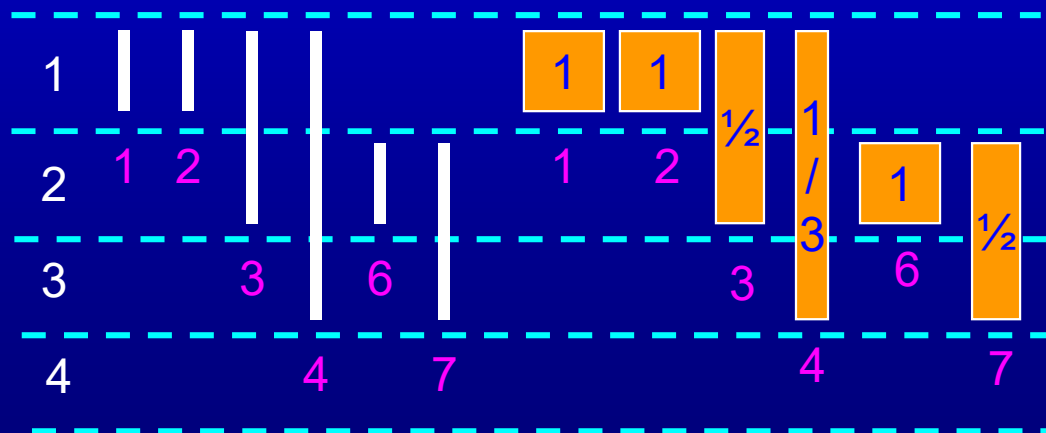
# FDS: Scheduling Probability for Multiply Operation



ASAP



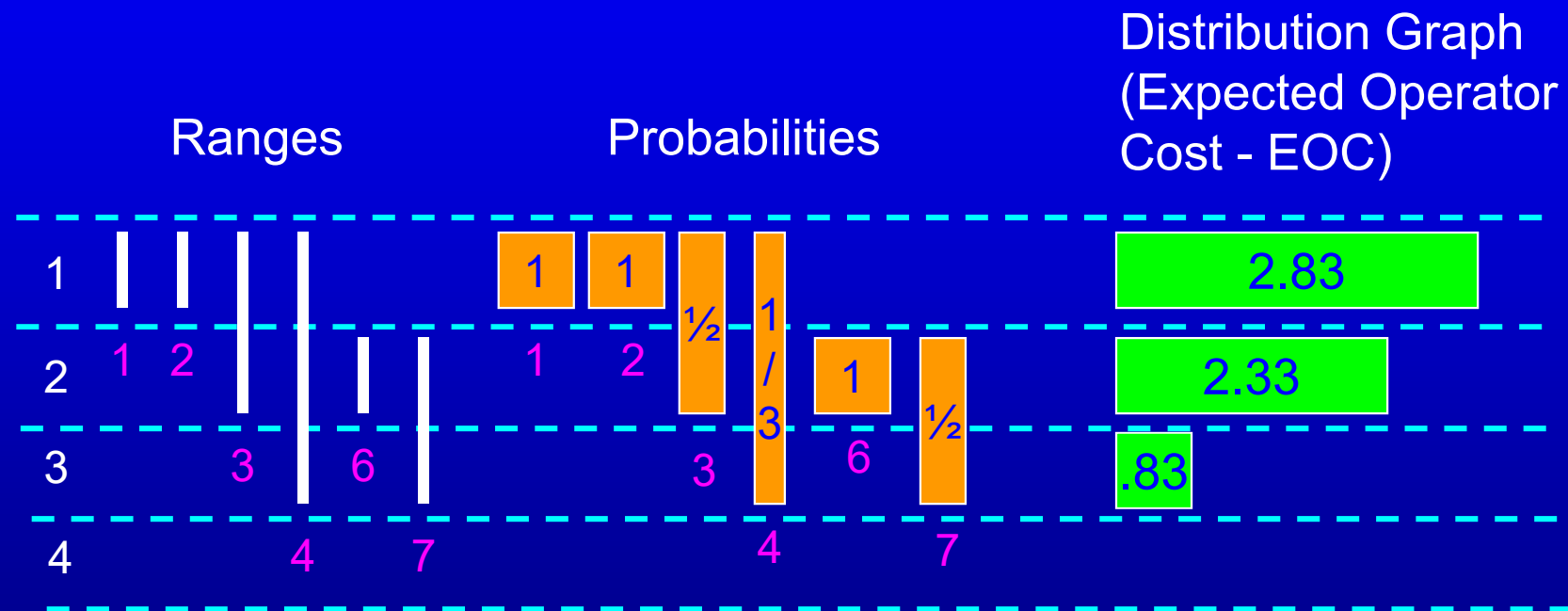
ALAP



Scheduling Probability for Multiply Operation



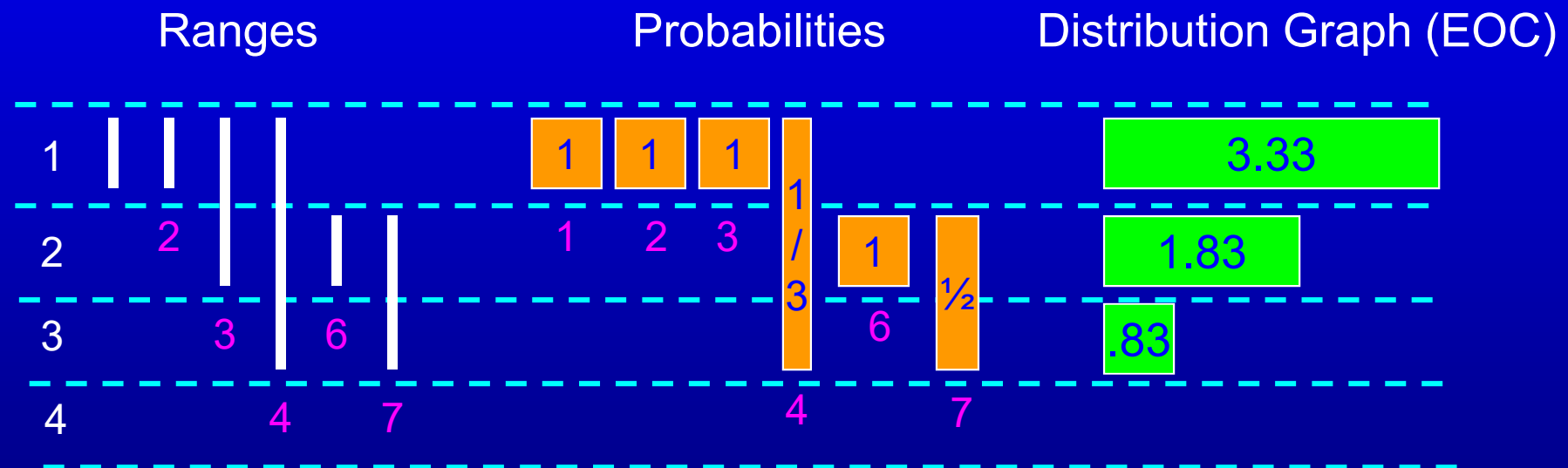
# FDS: Distribution Graph for Multiply Operation



Max. EOC gives an indication of the cost of implementing all multiplications

# FDS: Cost of Scheduling Operation into each Control Step

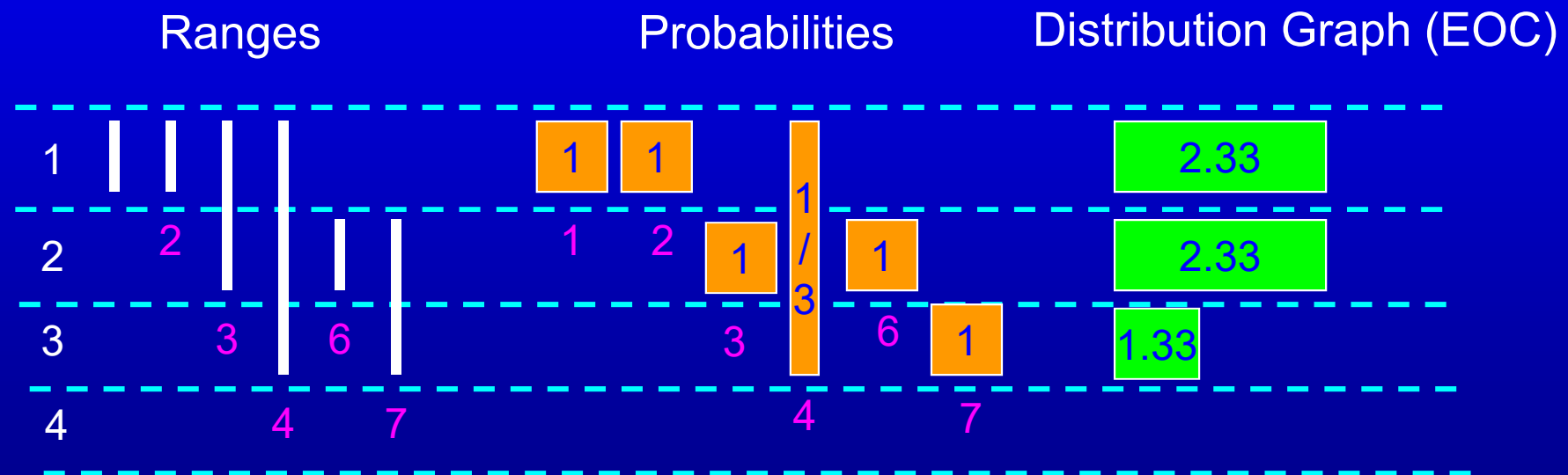
Attempt Scheduling Node 3 into Cycle 1



Note imbalanced resource utilisation

# FDS: Cost of Scheduling Operation into each Control Step

Attempt Scheduling Node 3 into Cycle 2, Node 7 to Cycle 3



Select assignment that minimises the max. EOC

# FDS: Algorithm

- At each step
  - Find assignment of operator to control step that minimises the max. EOC
  - Fix schedule of selected node (i.e., its prob. = 1)
    - constructive algorithm
    - result could be initial schedule for iterative improvement algorithm
  - Recompute probabilities and distribution graph

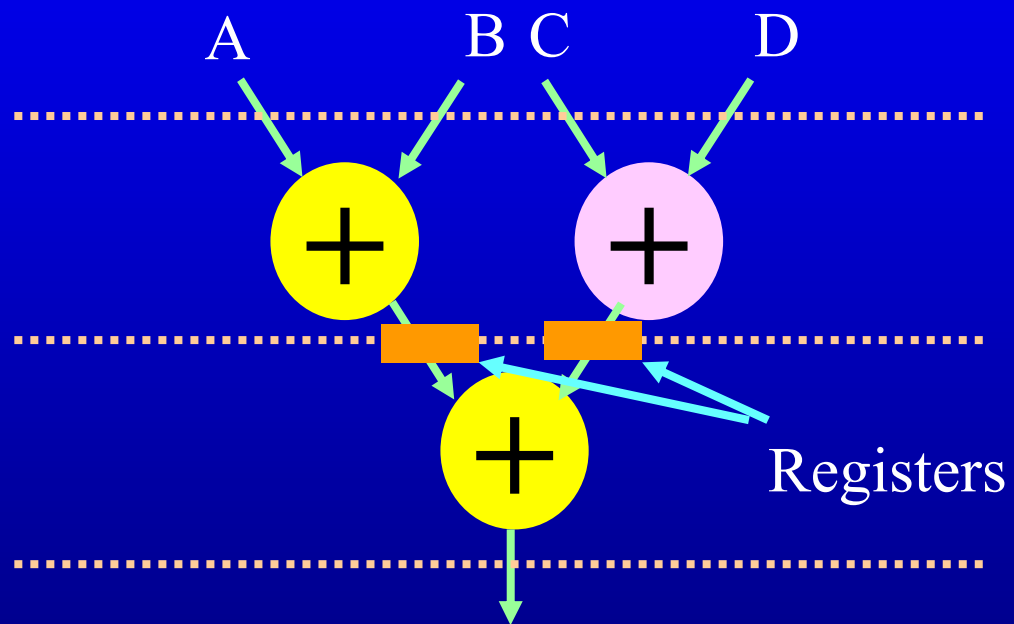
# Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Challenges

# Resource Allocation and Binding

- Binding
  - Operations to FUs
  - Variables to Registers (Register Allocation)
  - Data transfers to Buses
- Covering only Register Allocation here

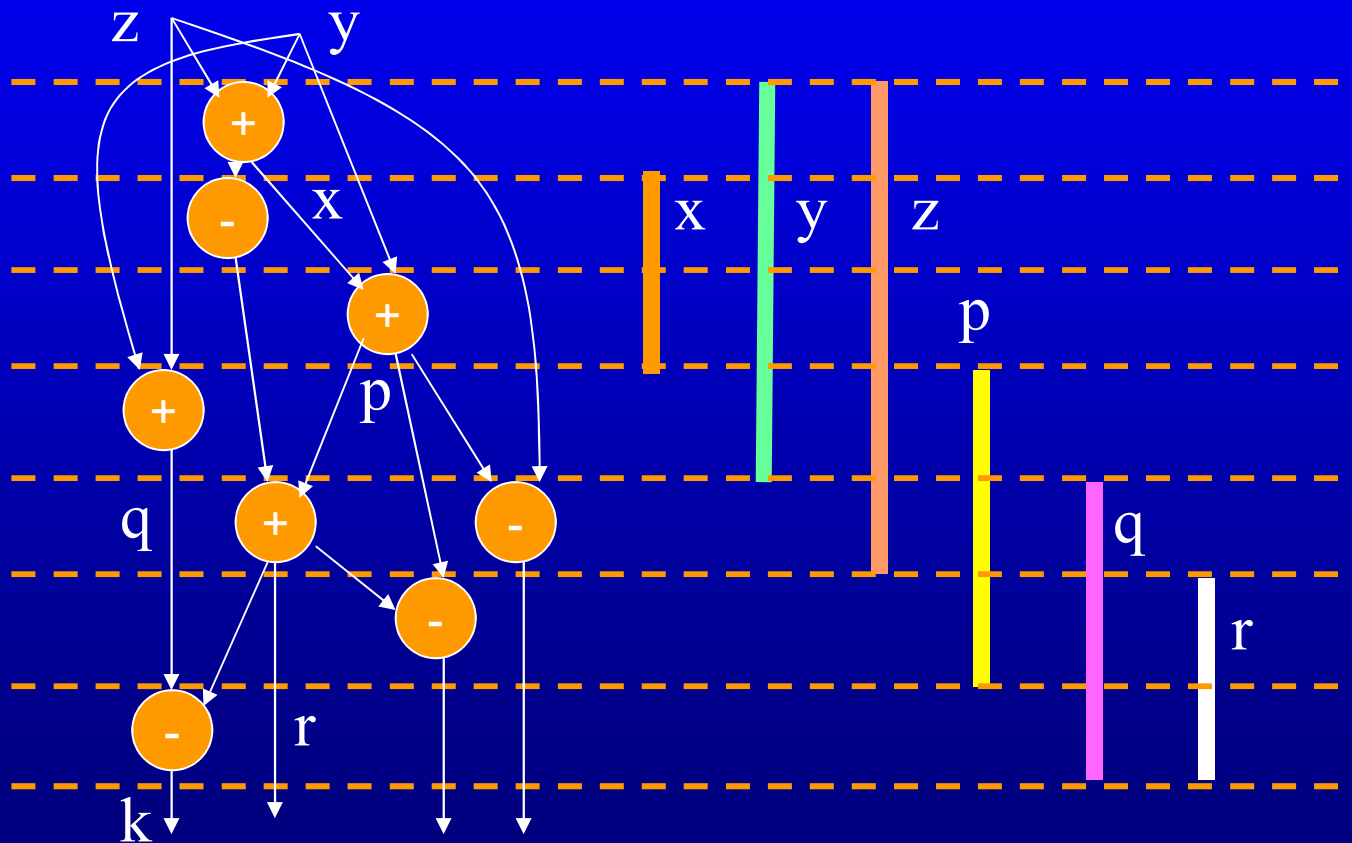
# Registers in Behavioural Synthesis



Resource Constraint -  
2 Adders

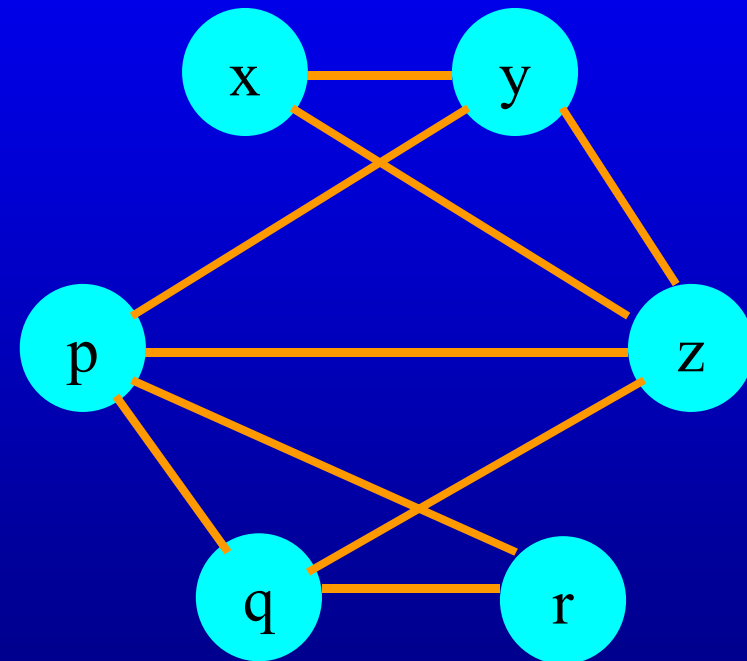
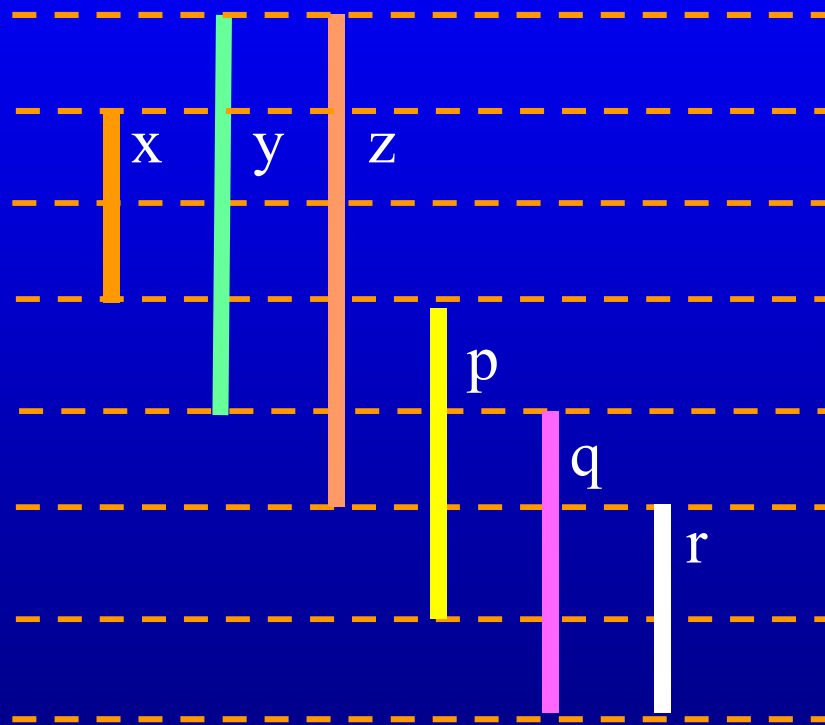
# Life-time of Variables

Life-time: definition to last use of variable





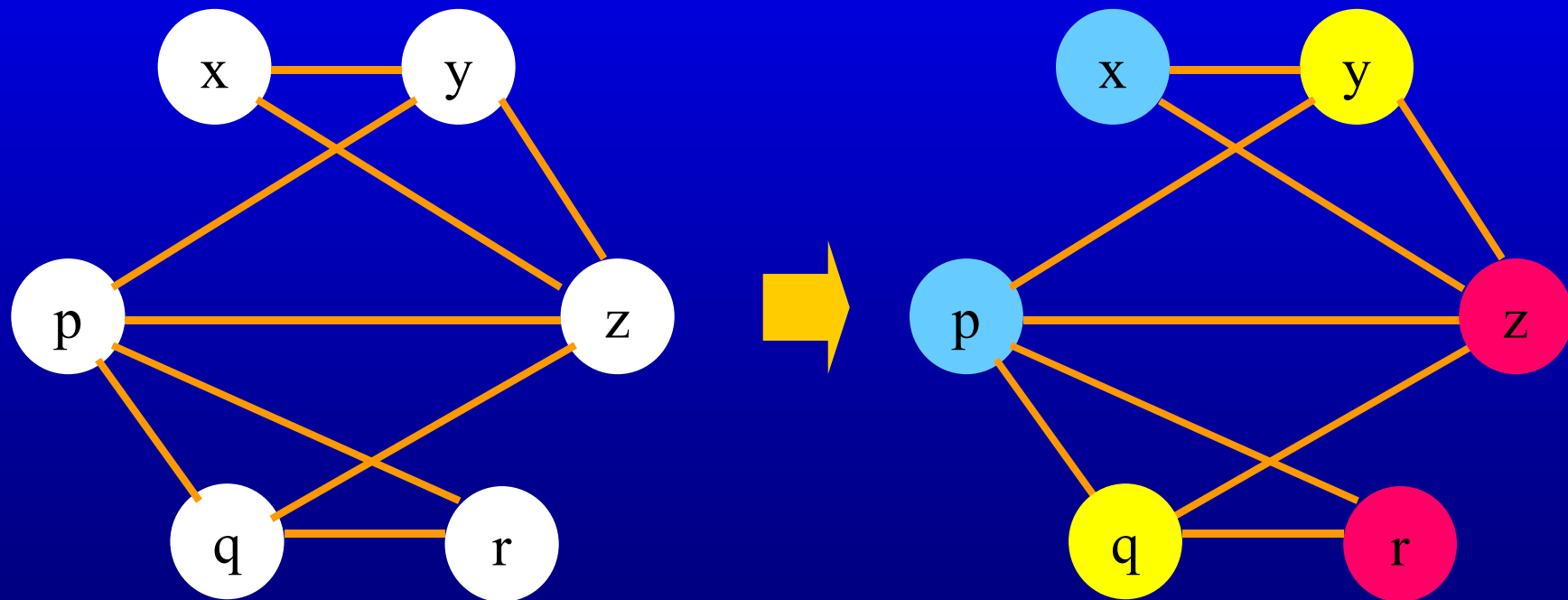
# Conflict Graph of Life-times



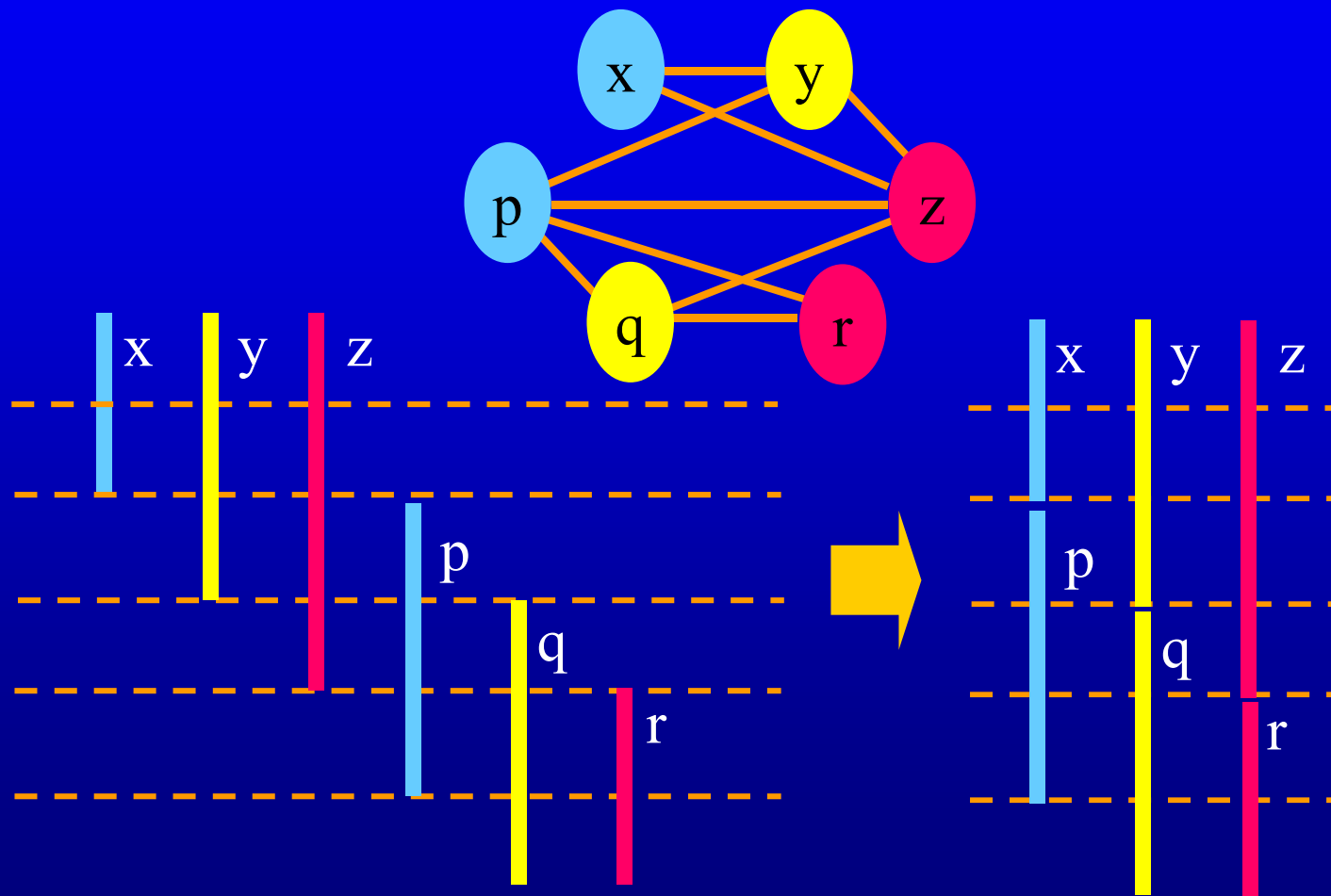
Edge = overlapping life-times

# Colouring the Conflict Graph

Minimum number of registers = Chromatic number of conflict graph



# Colouring determines Register Allocation



# Minimizing Register Count

- Graph Colouring is NP-complete
  - Heuristics (“Growing clusters”)
- Polynomial time solution exists for straight line code (no branches)
  - “Left-edge” algorithm
- Possible to incorporate other factors
  - Interconnect cost annotated as edge-weight

# Contents

- Introduction
- HLS tasks
- Design representation
- Compiler transformations
- Hardware optimisations
- Scheduling
- Register allocation
- Advanced Topics

# HLS – Advanced Topics

- Handling I/O
- Timing Challenges
  - MUX structure not known until binding
  - Wire delays unknown until layout
  - Register interconnections become complex
    - Register files
  - FSM Delays unknown
  - Bit-level timing
    - Aggressive chaining

# I/O Scheduling Modes

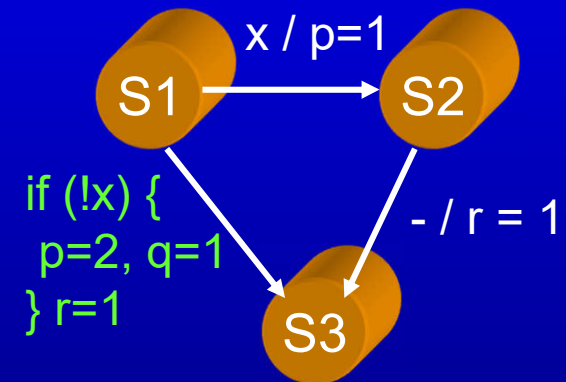
- **Exact I/O**
  - cycle-by-cycle I/O behaviour fixed
  - not allowed to add extra FSM states
- **Flexible I/O**
  - extra states can be added if necessary

# Exact I/O Scheduling

- wait statements mark cycle boundaries (cycle-fixed)
- I/O operations between two waits are constrained to be scheduled into same cycle
  - reading and writing of ports
  - mimics simulation
  - requires careful analysis!
- Non-I/O operations can be scheduled anywhere
  - arithmetic/logical operations

```
wait (); //s1
if (x) {
    p = 1;
    wait (); //s2
} else {
    p = 2;
    q = 1;
}
r = 1;
wait (); //s3
```

Behavioural Code



Inferred FSM



# Technical Difficulties: Coding-style Restrictions

- Example: if a **wait** occurs in one branch of a conditional, it should also occur in all other branches
  - is this reasonable?
  - why impose this?

```
wait ();  
if (x) {  
    p = 1;  
    wait ();  
} else {  
    p = 2;  
    q = 1;  
}  
r = 1;  
wait ();
```

Error!

```
wait ();  
if (x) {  
    p = 1;  
    wait ();  
} else {  
    p = 2;  
    q = 1;  
    wait ();  
}  
r = 1;  
wait ();
```

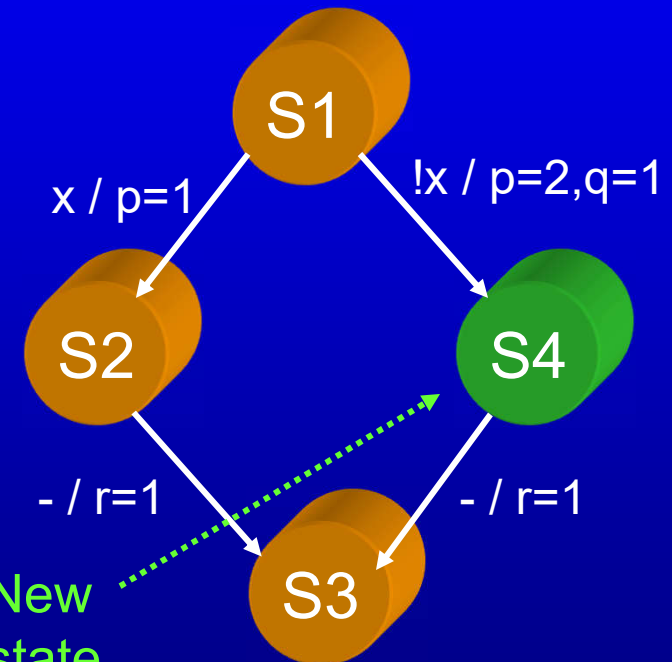
OK

# Imposing Restriction on waits Simplifies FSM Generation

- Wait imposed by coding-style restriction
- Leads to only simple actions on each transition
  - easier to generate FSM
- New state in FSM
- **Unwanted Addition!**
  - changes behaviour

```
wait (); //s1
if (x) {
    p = 1;
    wait (); //s2
} else {
    p = 2;
    q = 1;
    wait (); //s4
}
r = 1;
wait (); //s3
```

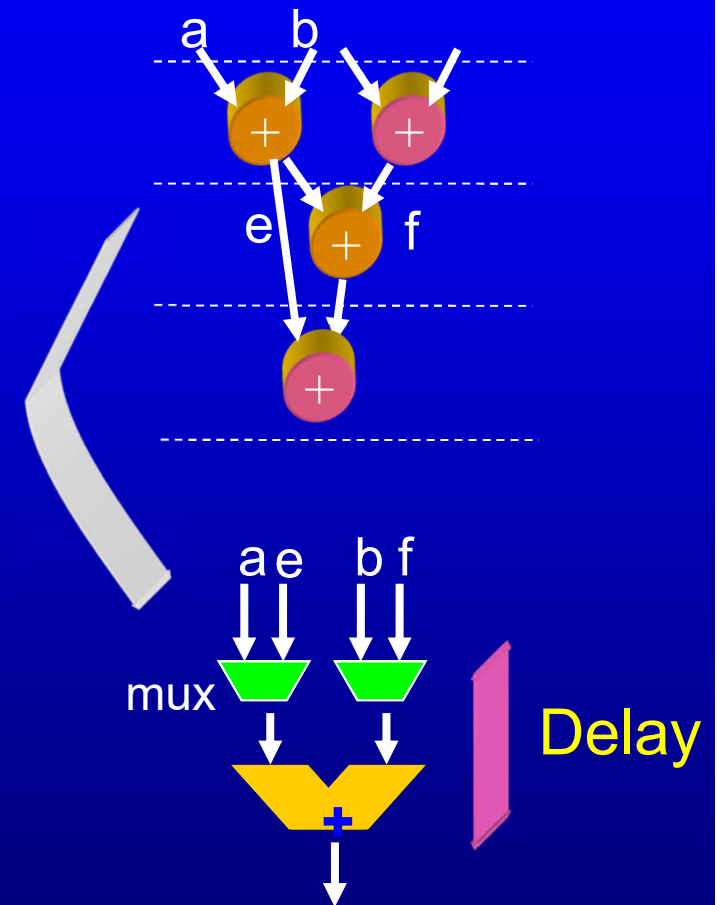
Behavioural Code



Inferred FSM

# MUX Inputs before FUs

- If binding follows scheduling...
  - an FU may perform different operations in different control steps
  - MUX implied at FU input
  - MUX delay needs to be accounted for
    - by scheduler, but it's too early!
    - scheduler cannot know MUX size

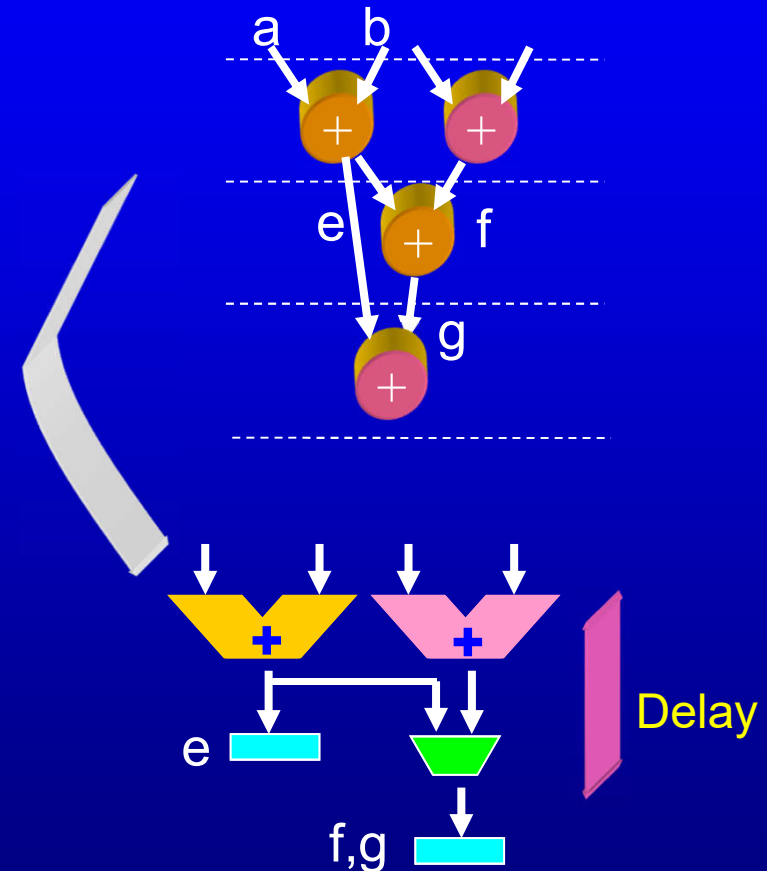


# Accounting for MUX Delays at FU Inputs

- Possible solution: integrate FU binding with Scheduling
- Iterate Scheduling and Binding
  - assume some MUX size (e.g., 4x1) during scheduling
  - binding is constrained
  - if binding cannot find a solution, increase MUX size (e.g., 8x1) and reschedule
    - results may be pessimistic
- This too isn't exact
  - What have we glossed over?

# MUX Inputs before Registers

- If register allocation follows scheduling and FU binding...
  - input to register may come from different FUs
    - due to sharing of registers
  - MUX implied at register input
  - MUX delay needs to be accounted for
    - by scheduler, but it's too early!
    - scheduler cannot know MUX size

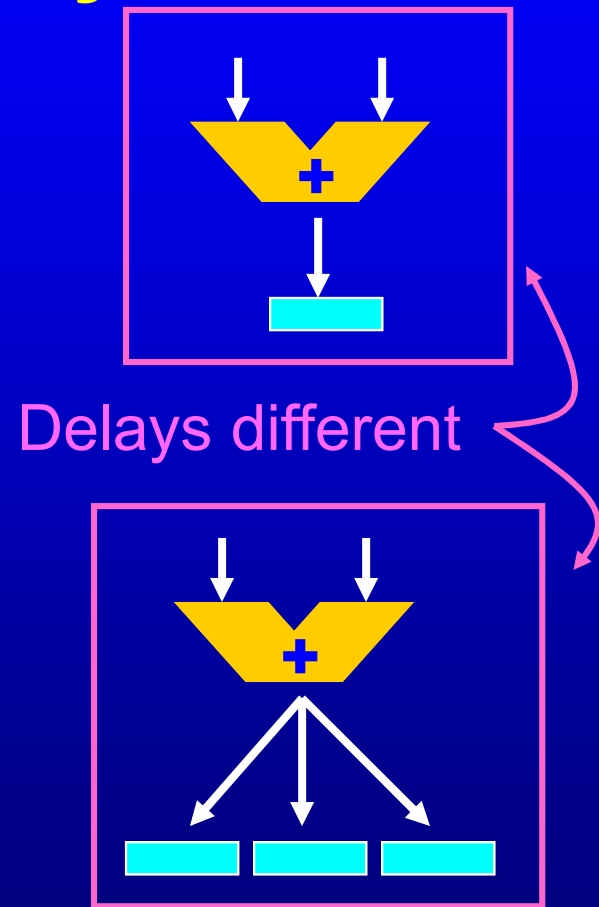


# Accounting for MUX Delays at Register Inputs

- Possible solution: integrate all steps
  - Scheduling
  - FU binding
  - Register Allocation
- Iterate Scheduling, Binding, and Register Allocation
  - assume some MUX size (e.g., 4x1) during scheduling
  - FU binding and register allocation are constrained
  - if no solution, increase MUX size (e.g., 8x1) and reschedule
    - results may be pessimistic

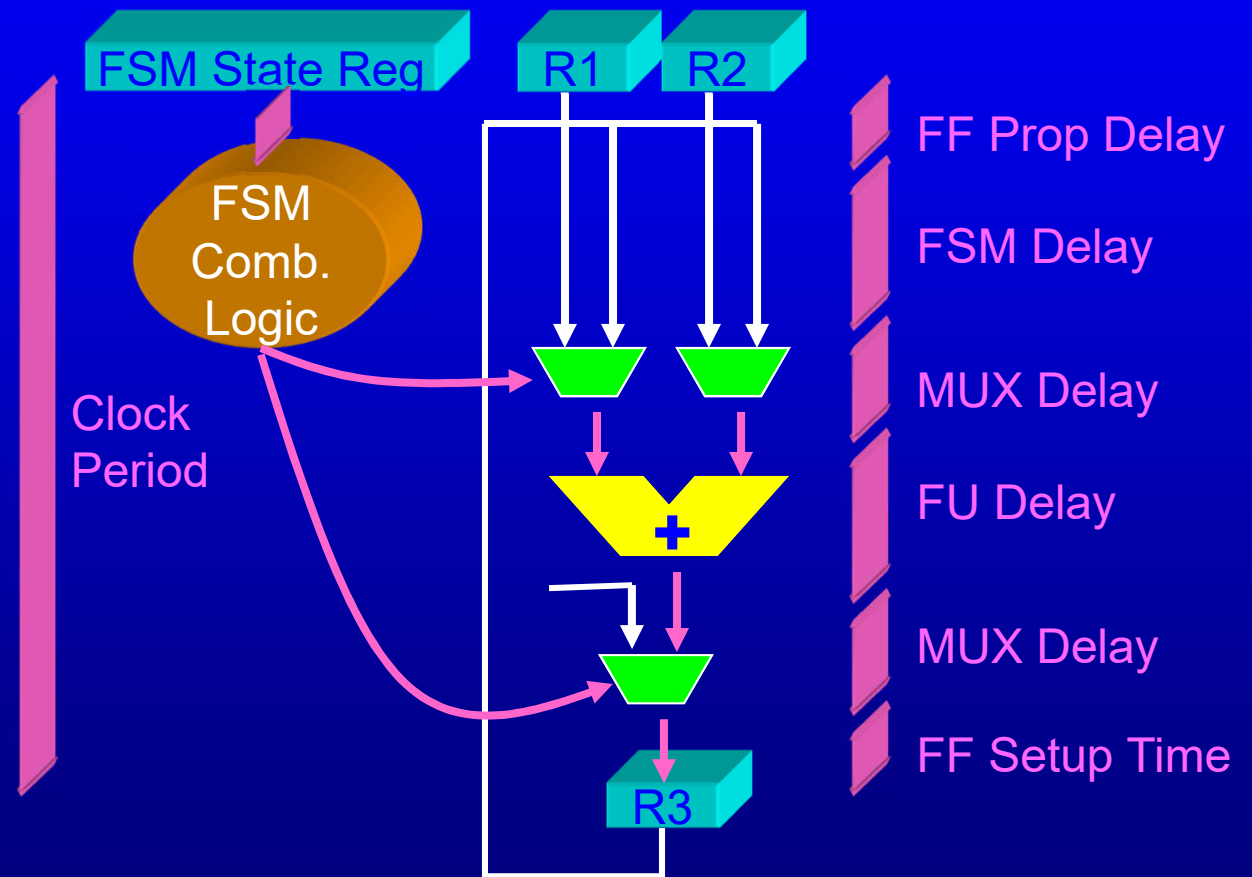
# Variable FU Delays, Unknown Wire Delays

- FU delay is not constant
  - depends on fanout/output load
- Output load not known during scheduling
  - fanout clear after FU binding and register allocation
  - wire lengths still unknown



# FSM Delays Unknown

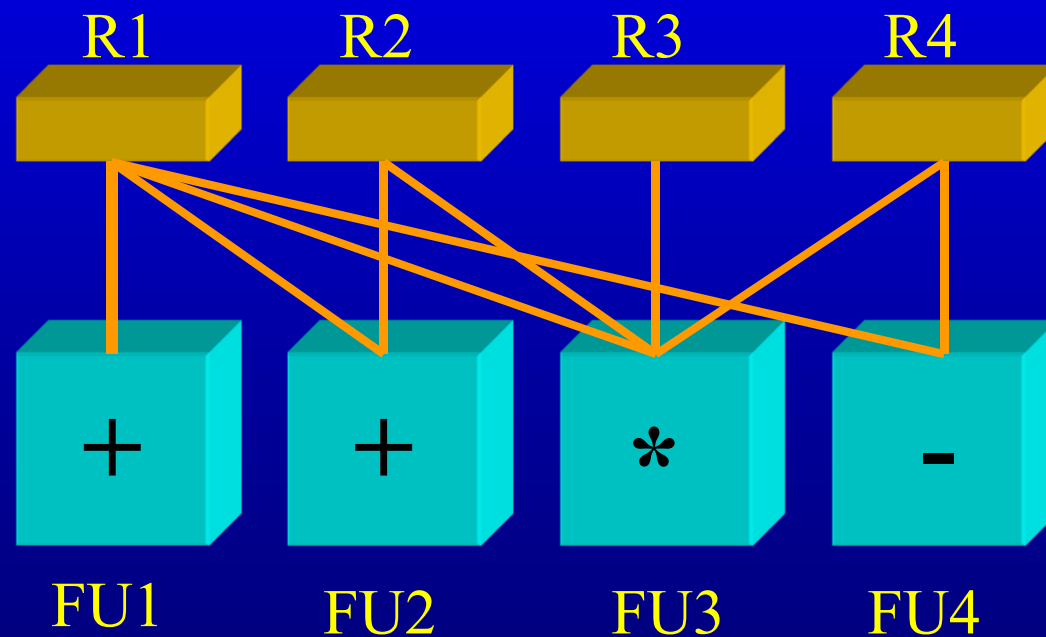
- FSM ready only after scheduling
- MUX select inputs come from FSM
- Non-zero delay through FSM



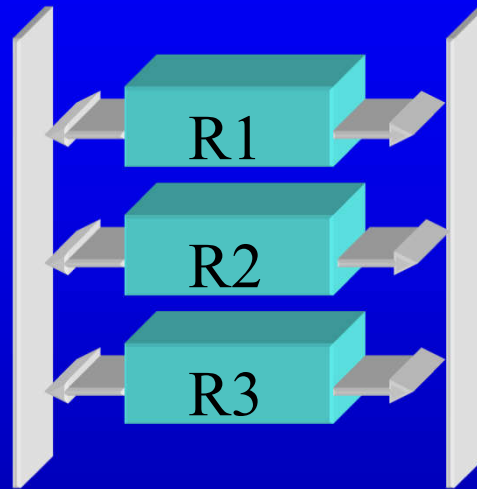


# Register-FU Interconnections

- Complex Interconnect
  - Every register connects to every FU



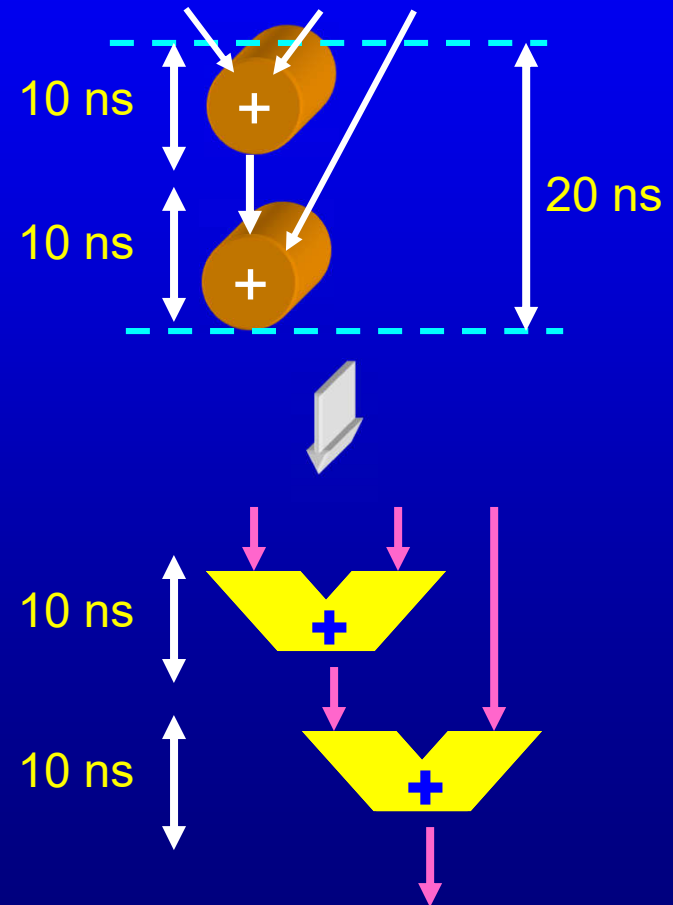
# Need for Register Files



- Modular architecture
- Limited connectivity
- New optimization opportunities

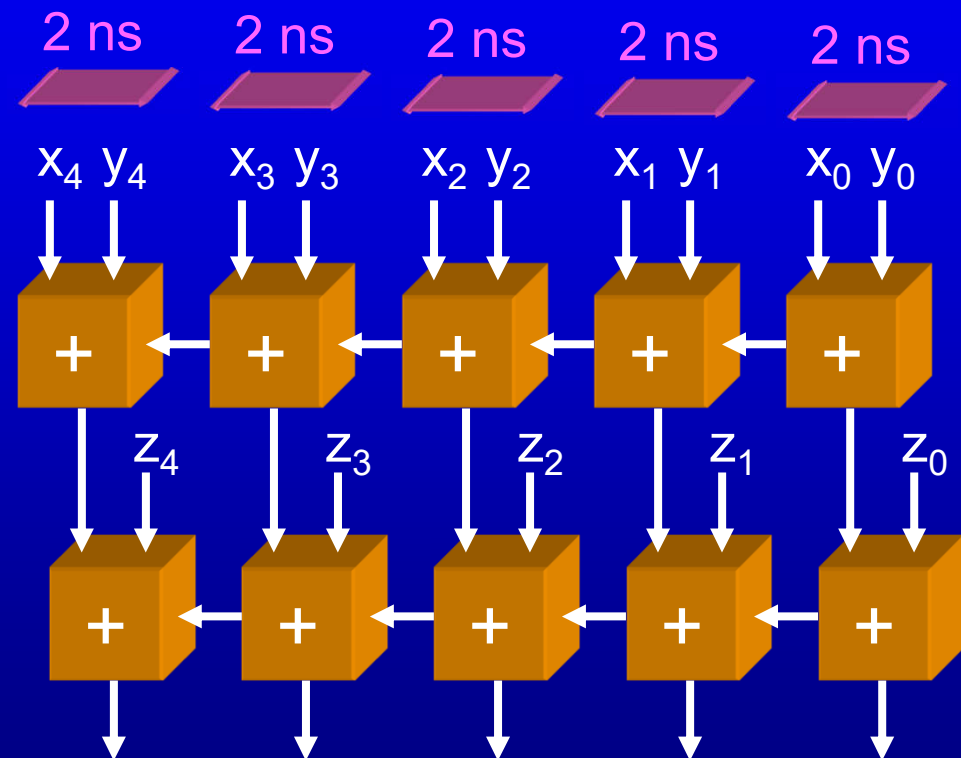
# Chained Function Units

- Chaining of operations implies chained FUs
  - o/p of one FU feeds another directly
- Max. delay through chained structure?
  - Not sum of delays!



# Chained Ripple Carry Adders

- Delay of each full adder = 2 ns
- Total delay of chain = critical path through circuit
- $x$ ,  $y$ ,  $z$  are ready at time zero
- Key observation: second adder begins operating as soon as  $(x_0 + y_0)$  is ready!



# Consequences of Chaining

- Much tighter schedules
- A lot more work for scheduler
  - max delay may not be simple equation
    - e.g., carry look-ahead adders, multipliers
  - sub-circuit may need to be synthesised and timing analysis invoked