High-Level Synthesis (HLS)

-From Algorithmic level to RTL

Song Chen

Dept. of Electronic Science & Technology, USTC

October 8, 2015

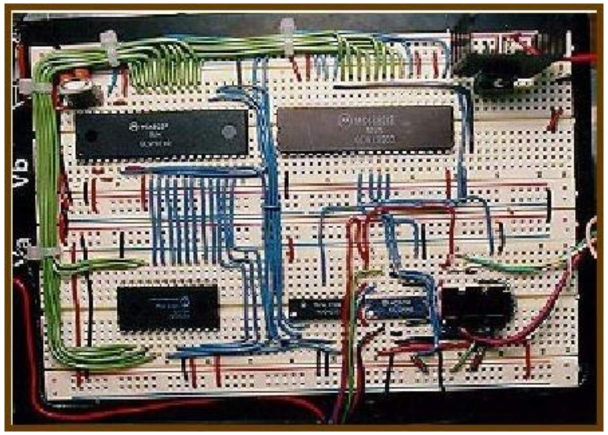
http:// staff.ustc.edu.cn/~songch/vlsi-cad.htm

Outline

- Electronic System Level Design
  - From Algorithm to RTL : Manual -> Automatic
  - A Design Example
    - Great Common Divisor (*Modified from Arvid@MIT*)
- High-level Synthesis
  - DFG/CDFG
  - Scheduling
  - Binding
  - Allocation

Originally designers used breadboards for prototyping

Solderless Breadboard



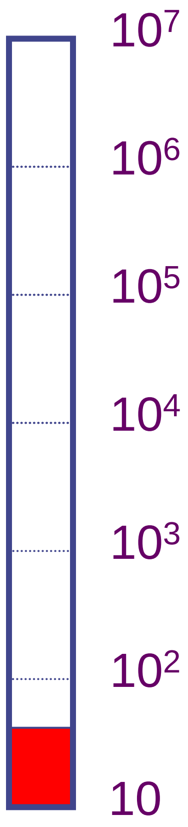
tangentsoft.net/elec/breadboard.html

No symbolic execution or testing

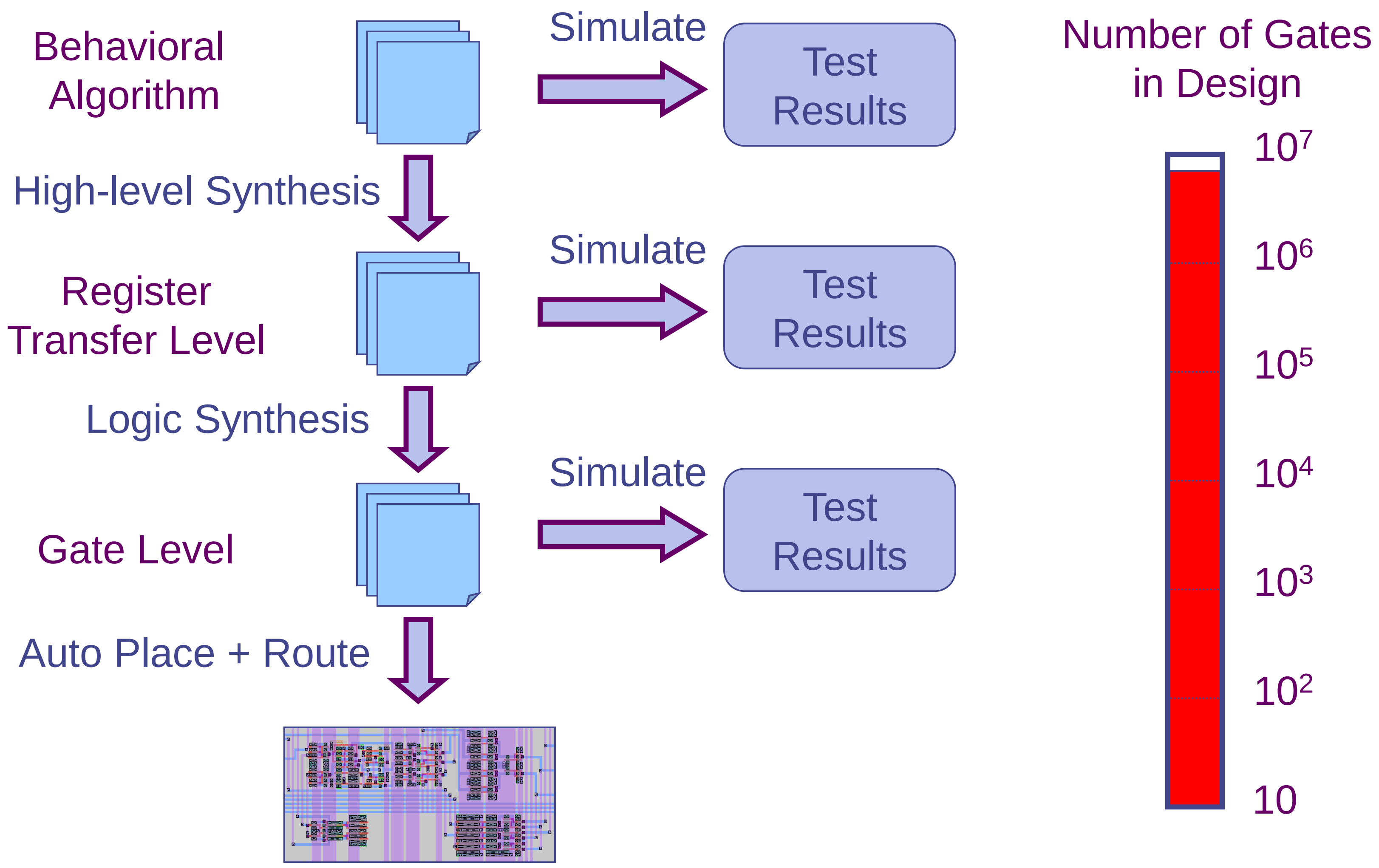
home.cogeco.ca

Printed circuit board

Number of Gates in Design



HDL + Tools for Automatic Translation



Electronic System Level Design

- System Function: Algorithmic or transfer functions
  - To cope with the challenges of designing emerging billion transistor system-on-chips
    - An abstraction level above RTL
- High-level synthesis (HLS)
  - Enabling technology for ESL design, also known as *behavioral synthesis*
  - Bridges between an algorithmic description of a design and its structural implementation at the RTL
  - Next natural step in design automation, succeeding logic synthesis

A Design Example

- Greatest Common Divisor
  - 27, 15
    - 27%15 = 12
    - 15%12 = 3
    - 12%3   = 0
    - GCD(27, 15) = 3

```
int GCD( int inA, int inB)
{    int done = 0;
     int A = inA;
     int B = inB;
     while ( !done )
     { if ( A < B )
       { swap = A;
         A = B;
         B = swap;
        }
        else if ( B != 0 )
          A = A - B;
        else
         done = 1;
       }
     return A;
}
```

Behavioral GCD in Verilog

```
module gcdGCDUnit_behav#( parameter W = 16 )
( input  [W-1:0] inA, inB,
  output [W-1:0] out );
  reg [W-1:0] A, B, out, swap;
  integer done;
  always @(*)
  begin
    done = 0; A = inA; B = inB;
    while ( !done )
    begin
      if ( A < B )
        swap = A;
        A = B; B = swap;
      else if ( B != 0 )
        A = A - B;
      else
        done = 1;
    end
    out = A;
end endmodule
```

User sets the input operands and checks the output; the answer will appear immediately, like a combinational circuit

Note data dependent loop, "`done`"

Deriving an RTL model for GCD

```
module gcdGCDUnit_behav#( parameter W = 16 )
( input  [W-1:0] inA, inB,
  output [W-1:0] out );
  reg [W-1:0] A, B, out, swap;
  integer done;
  always @(*)
  begin
    done = 0; A = inA; B = inB;
    while ( !done )
    begin
      if ( A < B )
        swap = A;
        A = B; B = swap;
      else if ( B != 0 )
        A = A - B;
      else
        done = 1;
    end
    out = A;
end endmodule
```

What does the RTL implementation need?

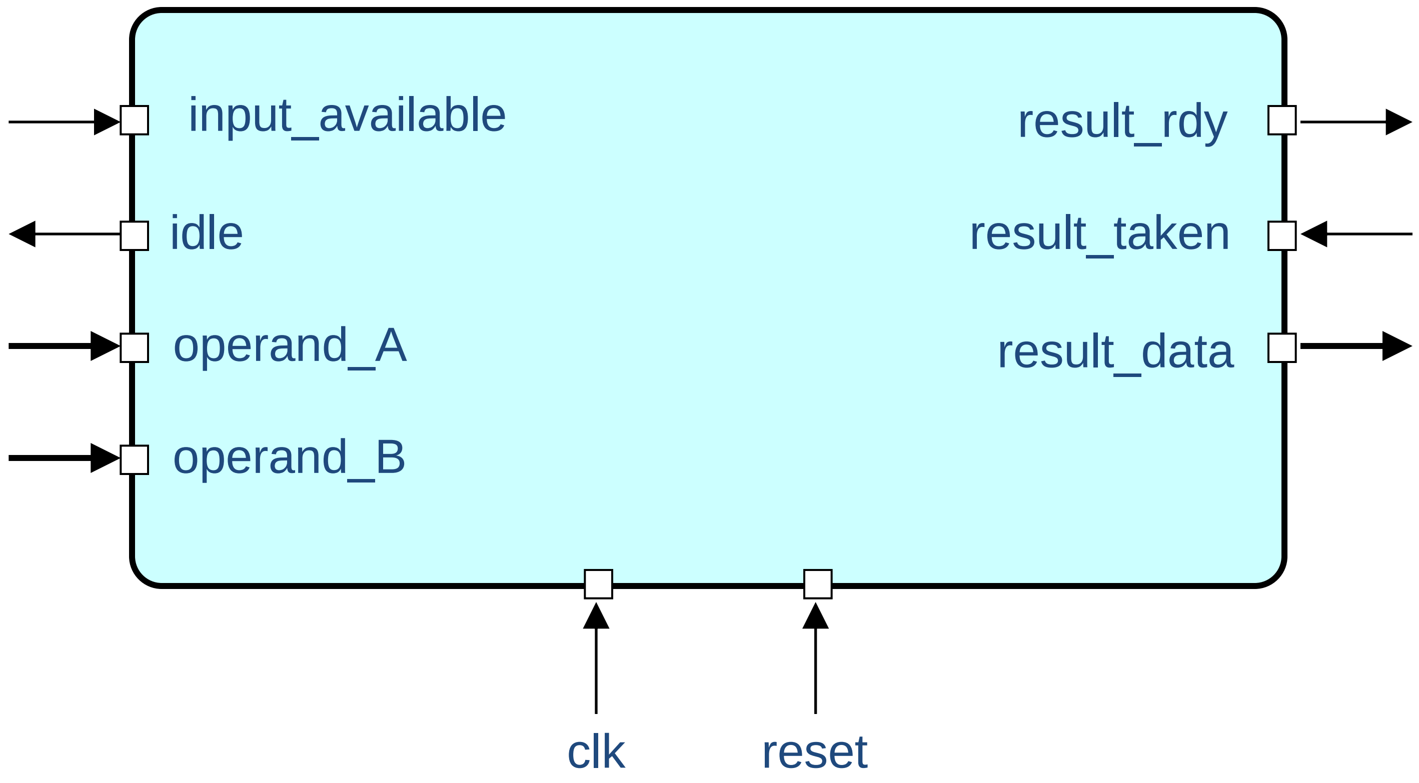
State (Register)

Less-Than Comparator

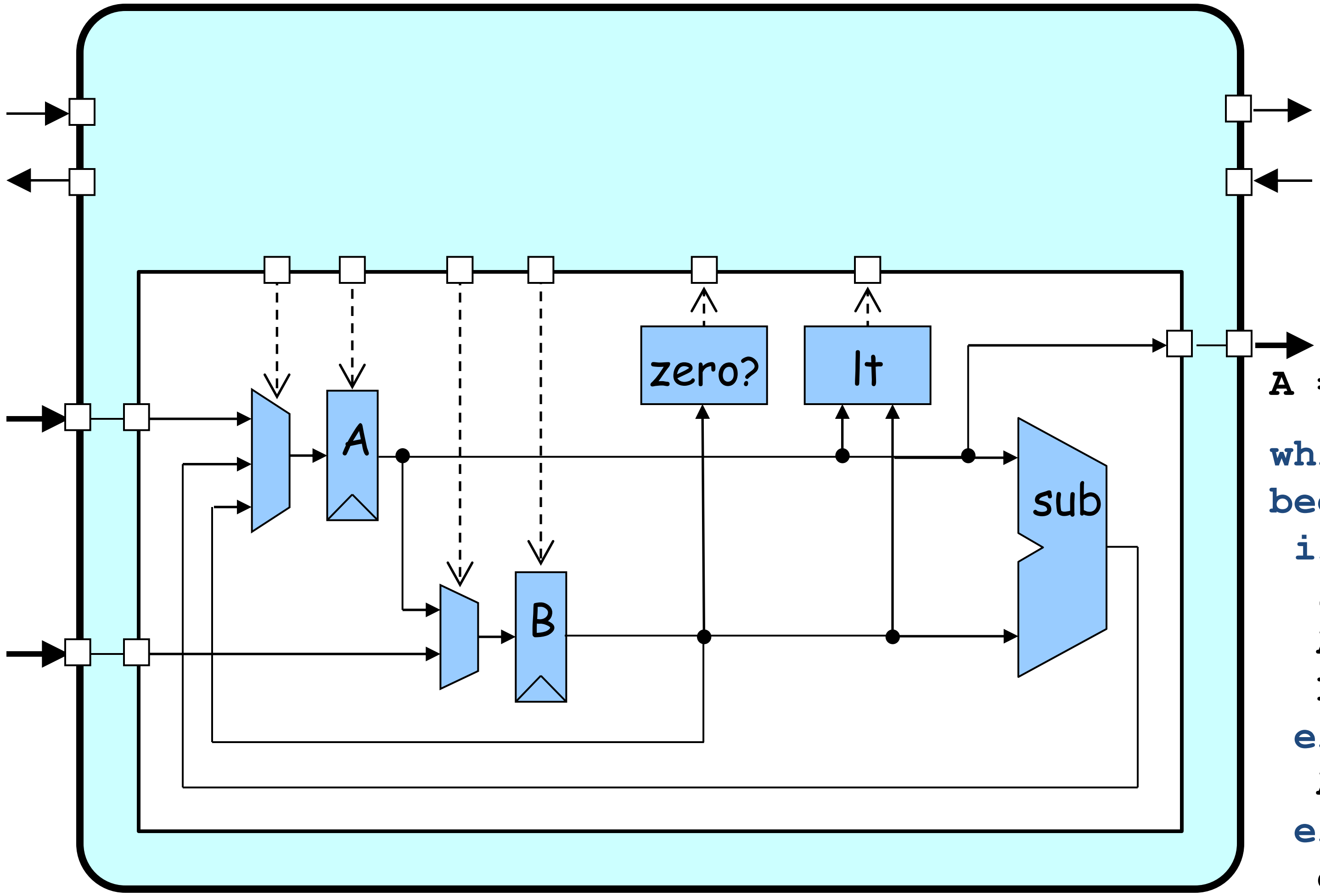
Equal Comparator

Subtractor

Step 1: Design an appropriate port interface

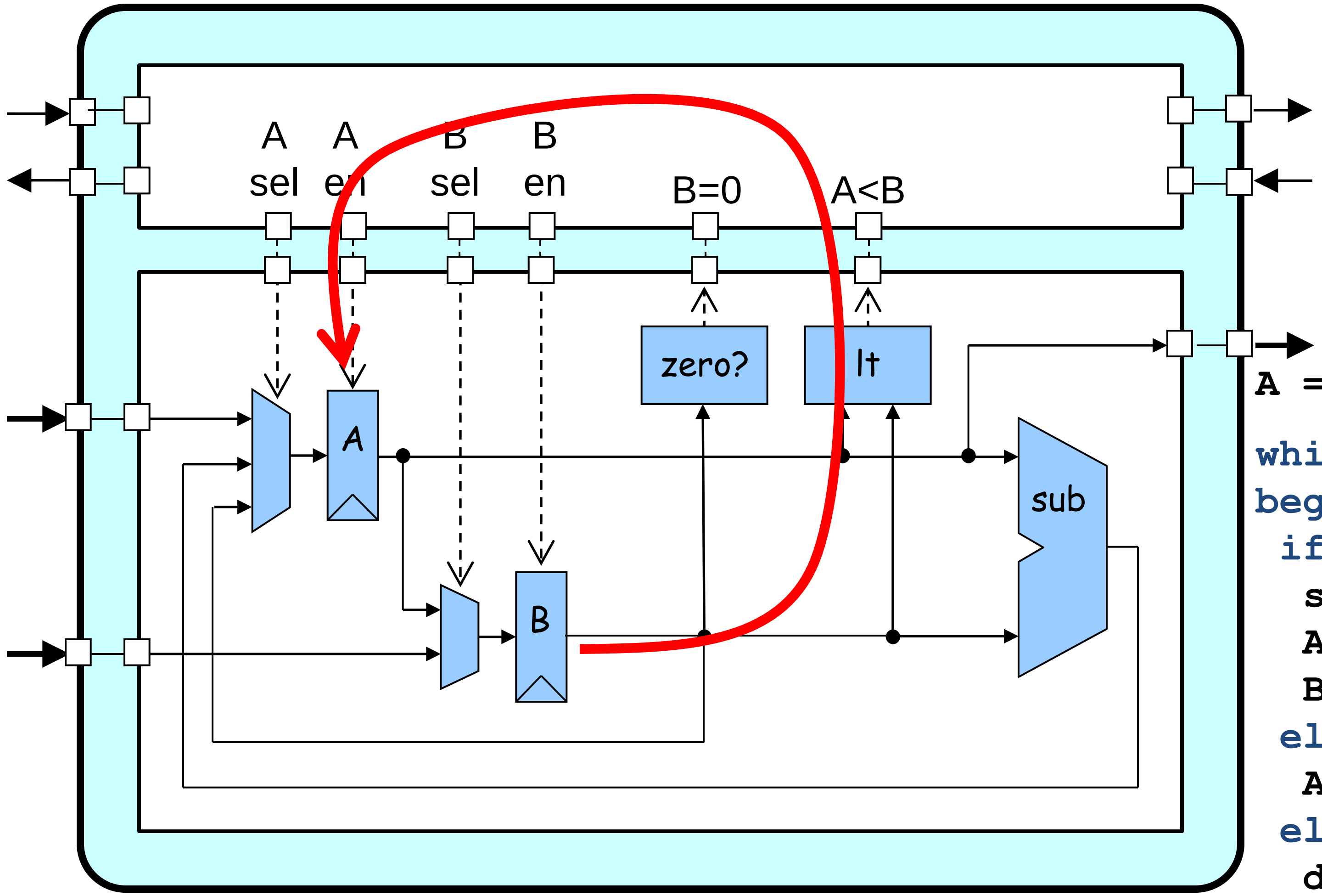


Step 2: Design a Datapath which has the functional units



```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if (B != 0)
  A = A - B;
 else
  done = 1;
End
Y = A;
```

Step 3: Add the control unit to sequence the datapath

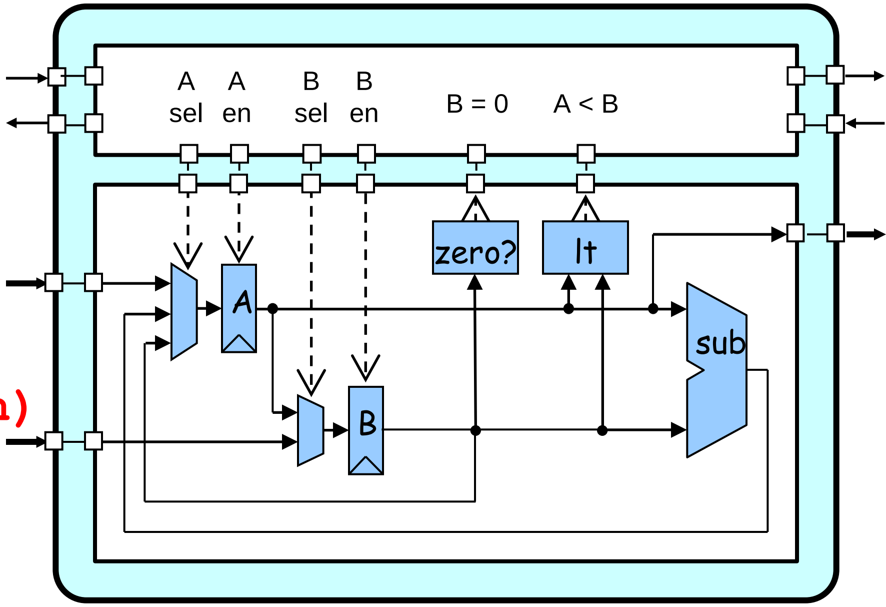


Control unit is designed to be either busy or waiting for input or waiting for output to be picked up

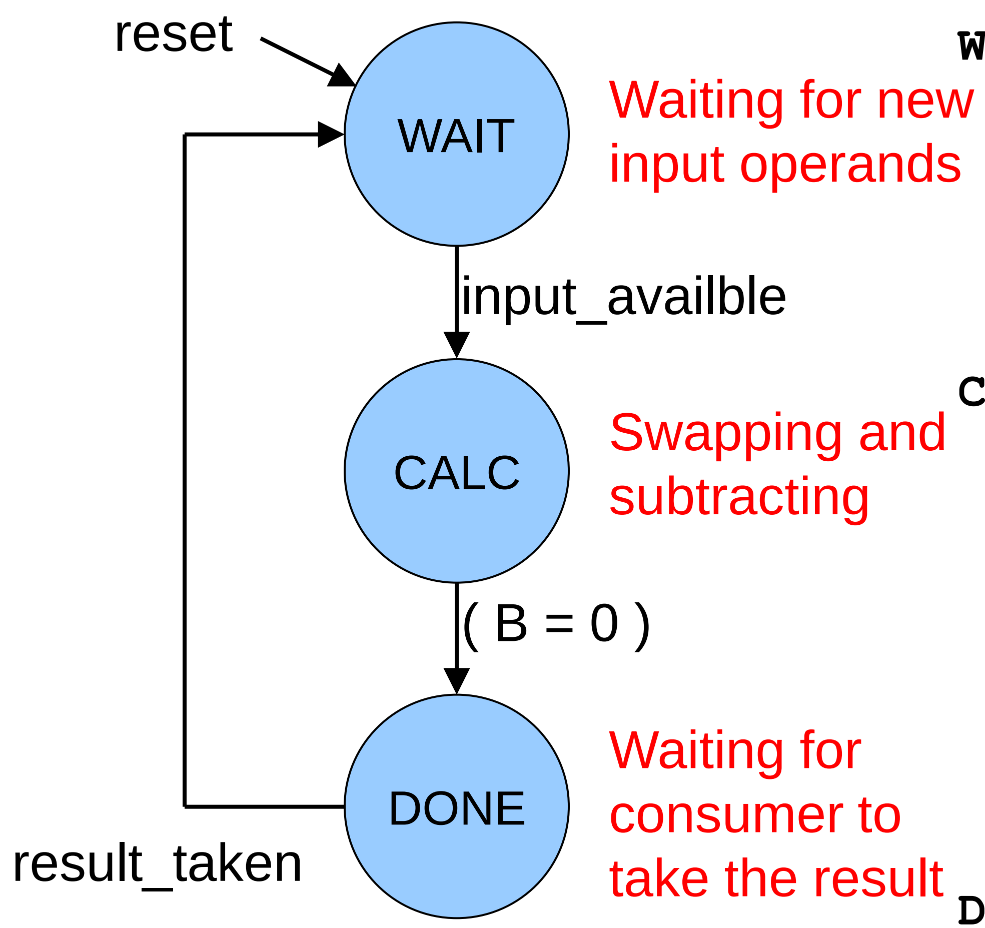
```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if (B != 0)
  A = A - B;
 else
  done = 1;
End
Y = A;
```

Datapath module interface

```
module gcdGCDUnitDpath_sstr#( parameter W = 16 )
( input        clk,

  // Data signals
  input  [W-1:0] operand_A,
  input  [W-1:0] operand_B,
  output [W-1:0] result_data,

  // Control signals (ctrl->dpath)
  input          A_en,
  input          B_en,
  input    [1:0] A_sel,
  input          B_sel,

  // Control signals (dpath->ctrl)
  output         B_zero,
  output         A_lt_B
);
```



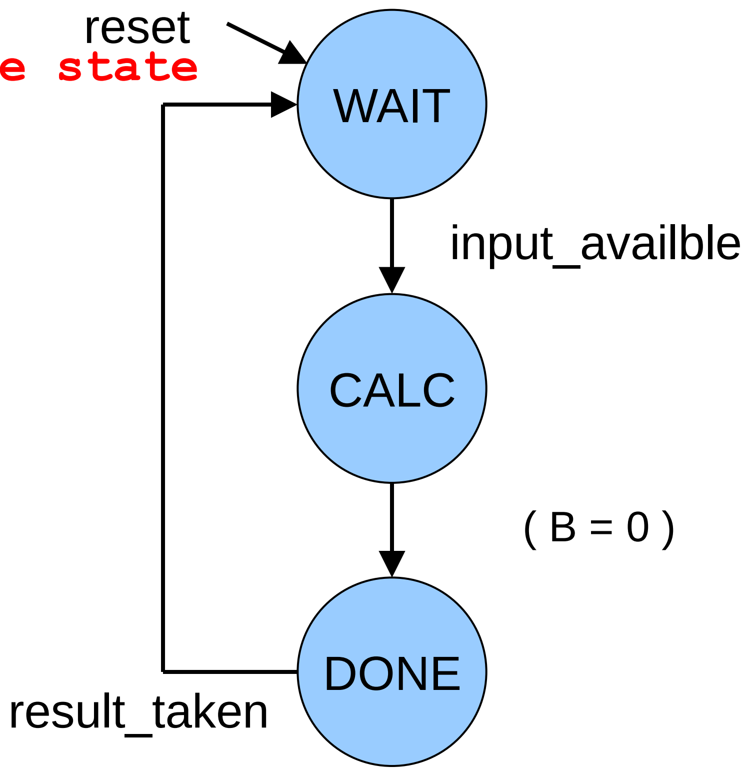
Control unit requires a state machine for valid/ready signals



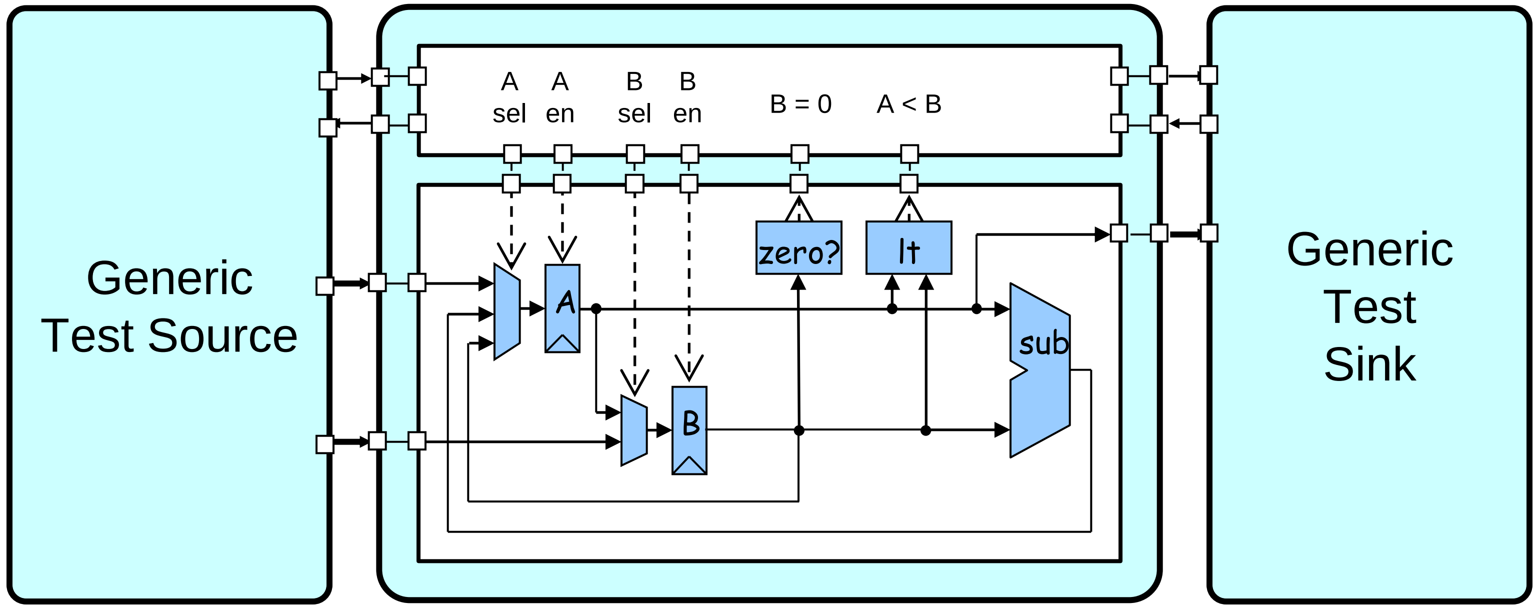
```
WAIT: begin
        A_sel     = A_SEL_IN;
        A_en      = 1'b1;
        B_sel     = B_SEL_IN;
        B_en      = 1'b1;
      end
CALC: if ( A_lt_B )
        A_sel = A_SEL_B;
        A_en       = 1'b1;
        B_sel = B_SEL_A;
        B_en       = 1'b1;
      else if ( !B_zero )
        A_sel = A_SEL_SUB;
        A_en       = 1'b1;
      end
DONE: result_rdy = 1'b1;
```

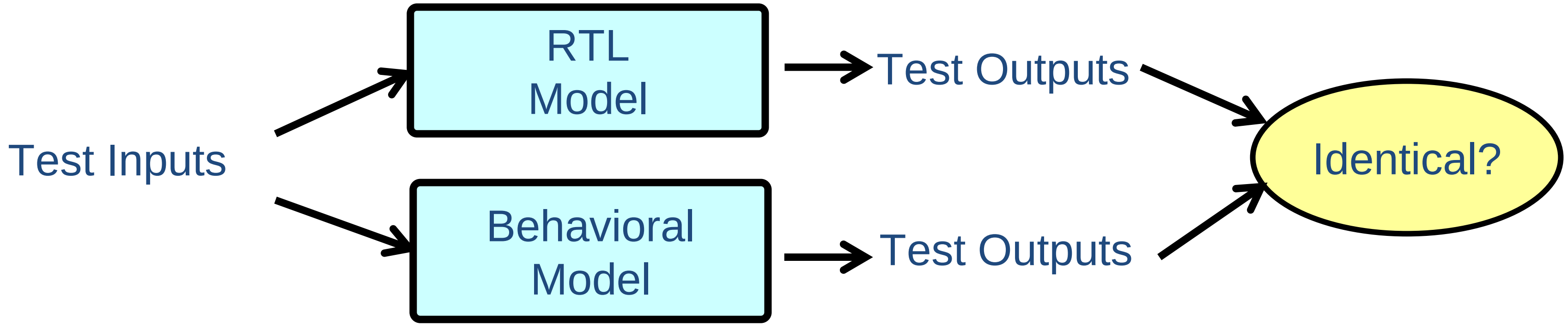
FSM state transitions

```
always @(*)
begin
  // Default is to stay in the same state
  state_next = state;

  case ( state )
    WAIT :
      if ( input_available )
        state_next = CALC;
    CALC :
      if ( B_zero )
        state_next = DONE;
    DONE :
      if ( result_taken )
        state_next = WAIT;
  endcase
end
```



RTL test harness requires proper handling of the ready/valid signals

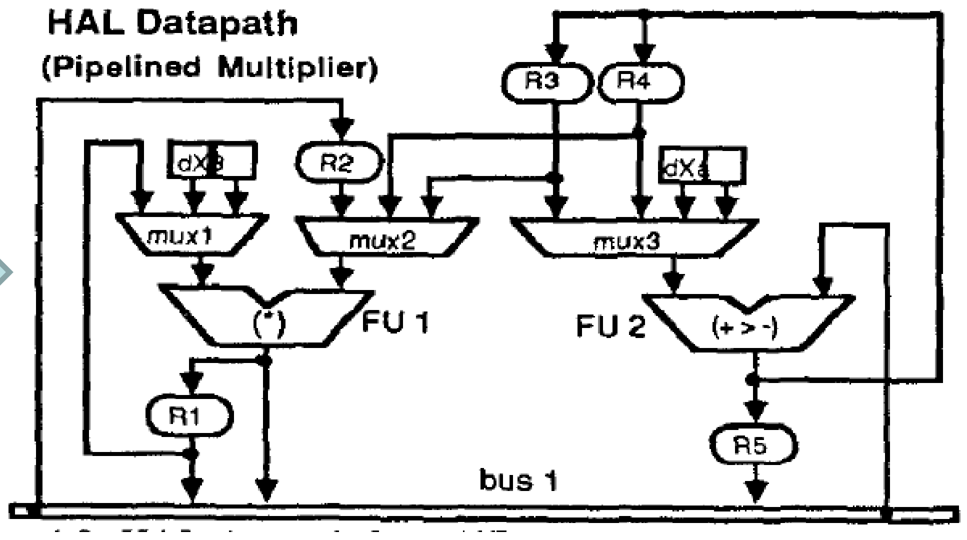




From Behavioral Description to RTL

- Datapath
  - Resources (arithmetic or logic),
  - Steering logic circuits (multiplexers, buses, etc.)
  - Registers or Memories
- Control Unit: Finite State Machine

```
while (x < a){
    x1 = x + dx;
    u1 = u - ( 3 * x * u
        * dx) - (3 * y * dx);
    y1 = y + ( u * dx);
    c = x < a;
    x = x1;
    u = u1;
 end
 y = y1;
```



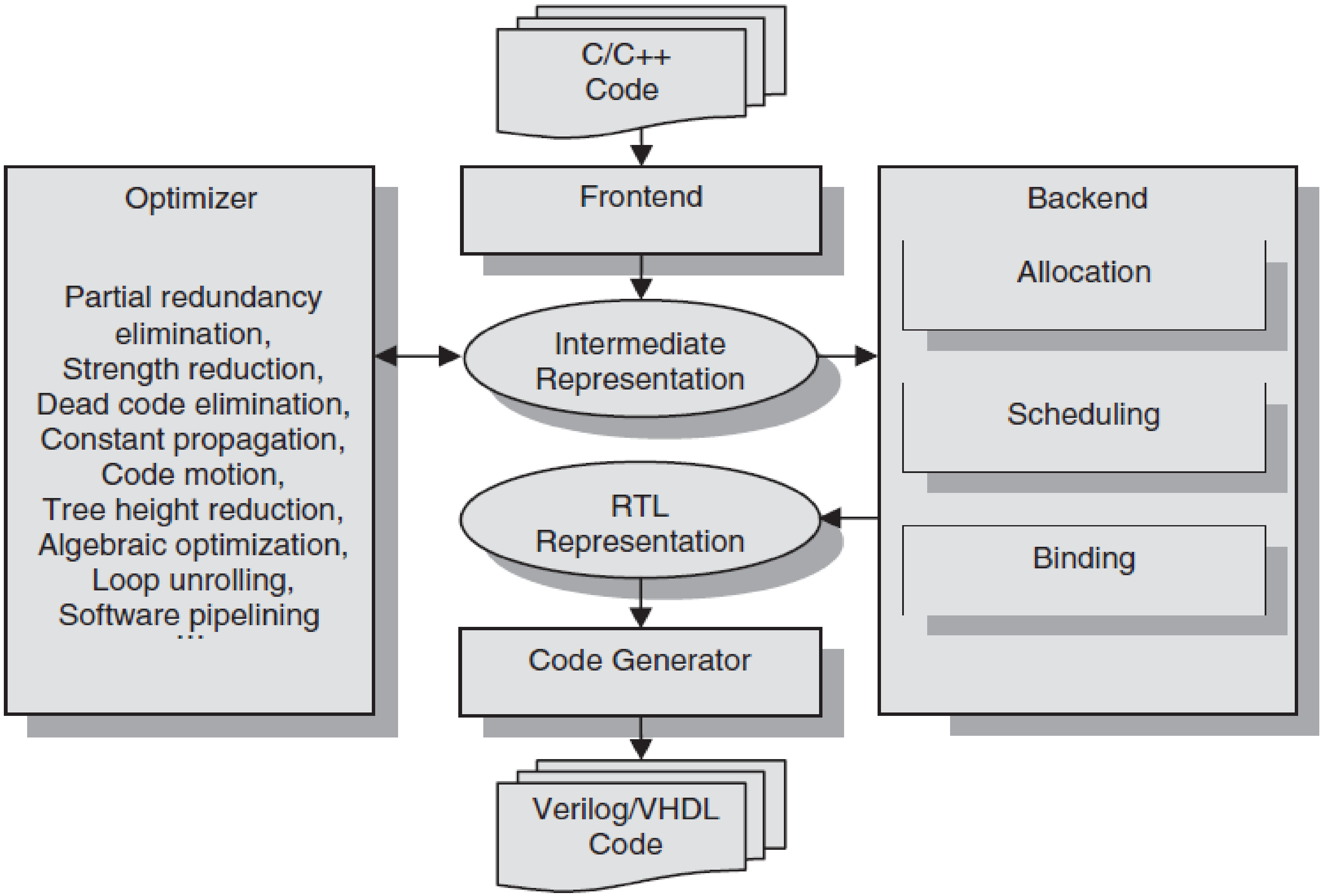
Architecture Synthesis

- Deal with “computational” behavioral descriptions
  - Behavior as a sequencing graph: Data flow graph
  - Hardware resources as library elements
    - Pipelined or non-pipelined
    - Resource performance in terms of execution delay
  - Constraints on operation timing
  - Constraints on hardware resource availability
  - Storage as registers, memories,
  - data transfer using wires

Architecture Synthesis – Cont'

- Objective
  - Generate a synchronous, single-phase clock circuit
  - Might have multiple feasible solutions (explore tradeoff)
  - Satisfy constraints, minimize objective:
    - Maximize performance subject to area constraint
    - Minimize area subject to performance constraints
    - Minimize Power subject to performance constraints

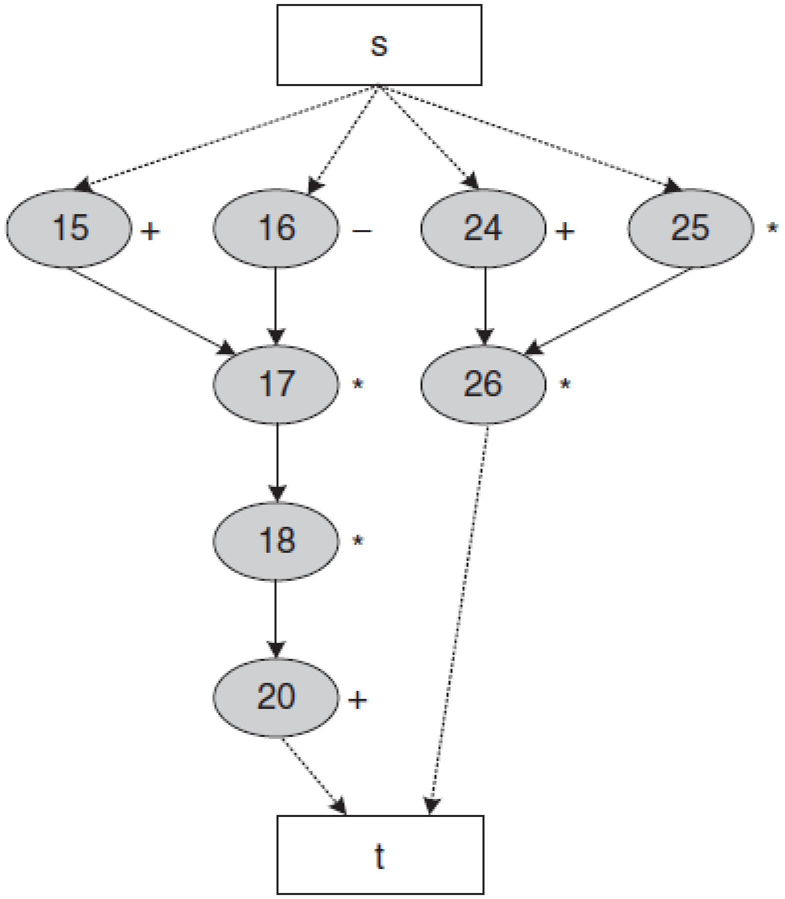
Typical HLS Flow



Data Flow Graph

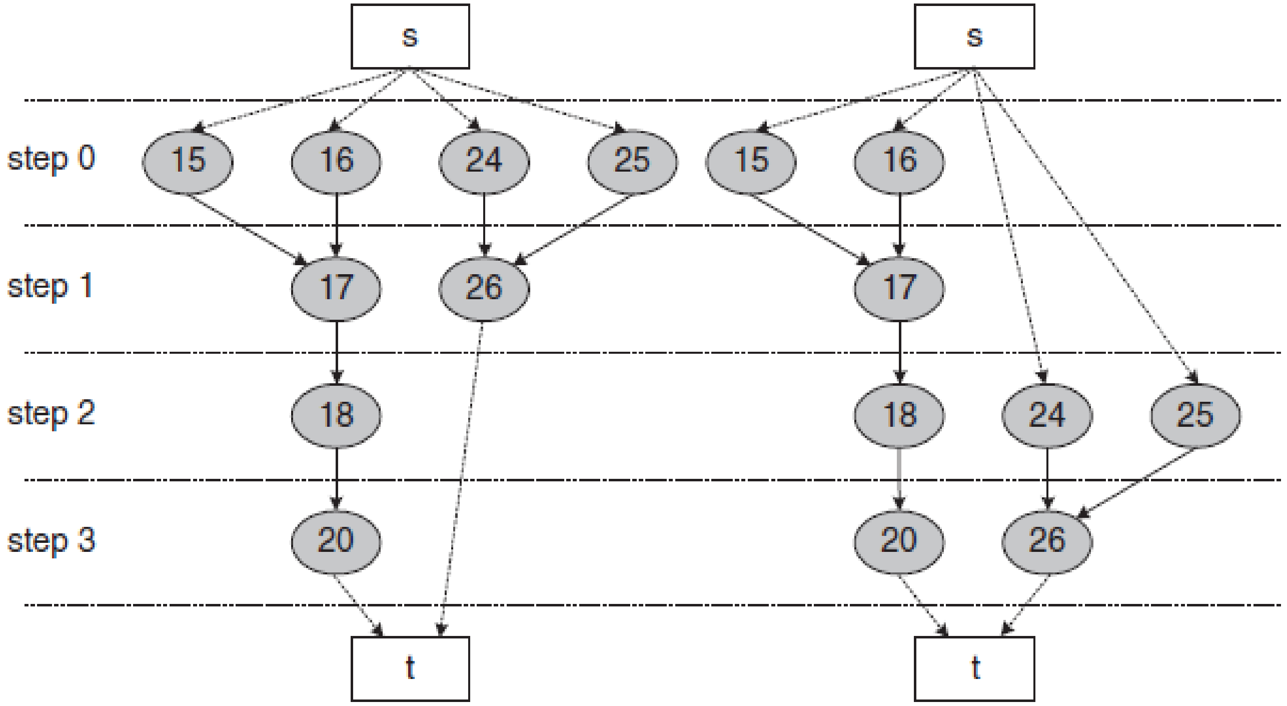
```
int a, b, c, d;
                  :
c = ...;
d = ...;

if( ... ) {
   c = (((a + b) * (a - b)) * 13) + 16;
   d = (a + 12) * (a * 12);
   }

... = c + d;
                  :
```

```
(14) cnst 13
(15) + (4) (6)
(16) - (4) (6)
(17) * (15) (16)
(18) * (14) (17)
(19) cnst 16
(20) + (18) (19)
(23) cnst 12
(24) + (4) (23)
(25) * (23) (4)
(26) * (24) (25)
(27) sts (20), c
(28) sts (26), d
```



Scheduling

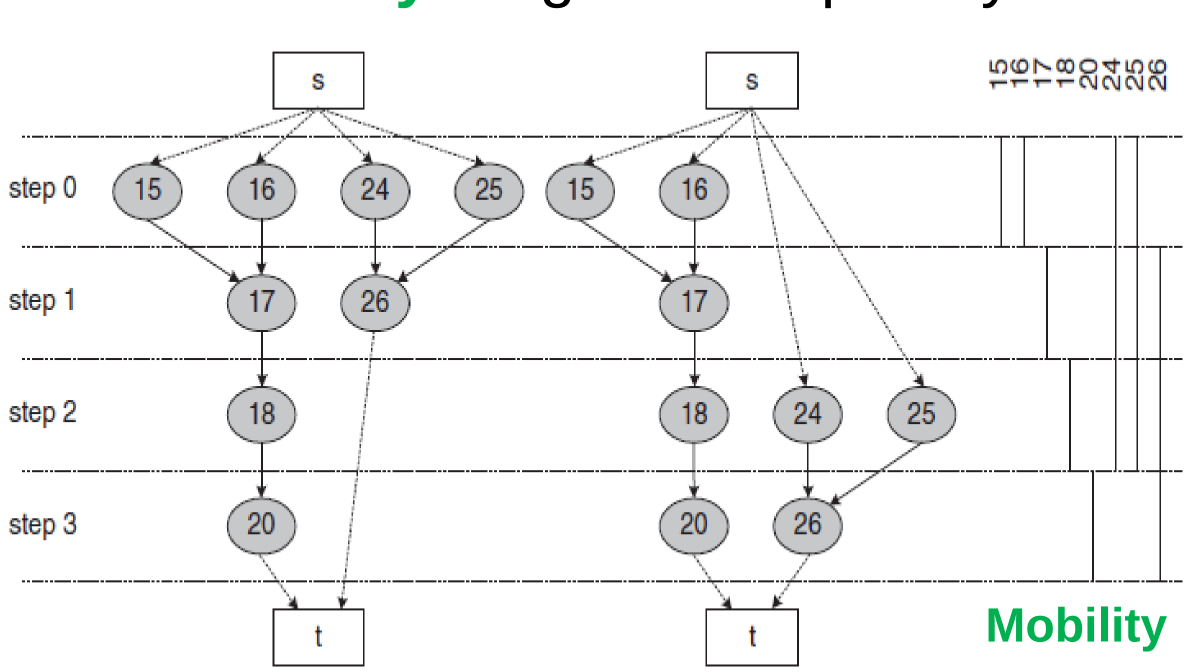
- Add/Sub: 15, 16, 20, 24;   Mult: 17, 18, 25, 26
  - ASAP Scheduling & ALAP Scheduling
    - Topological Order

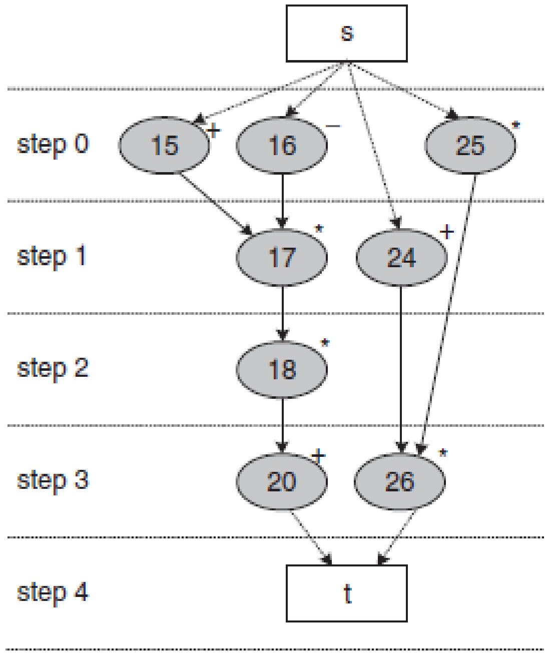


How many adders and multipliers?

List Scheduling – Resource Constrained

- Add/Sub: 15, 16, 20, 24;   Mult: 17, 18, 25, 26
    - **Mobility** range as the priority function



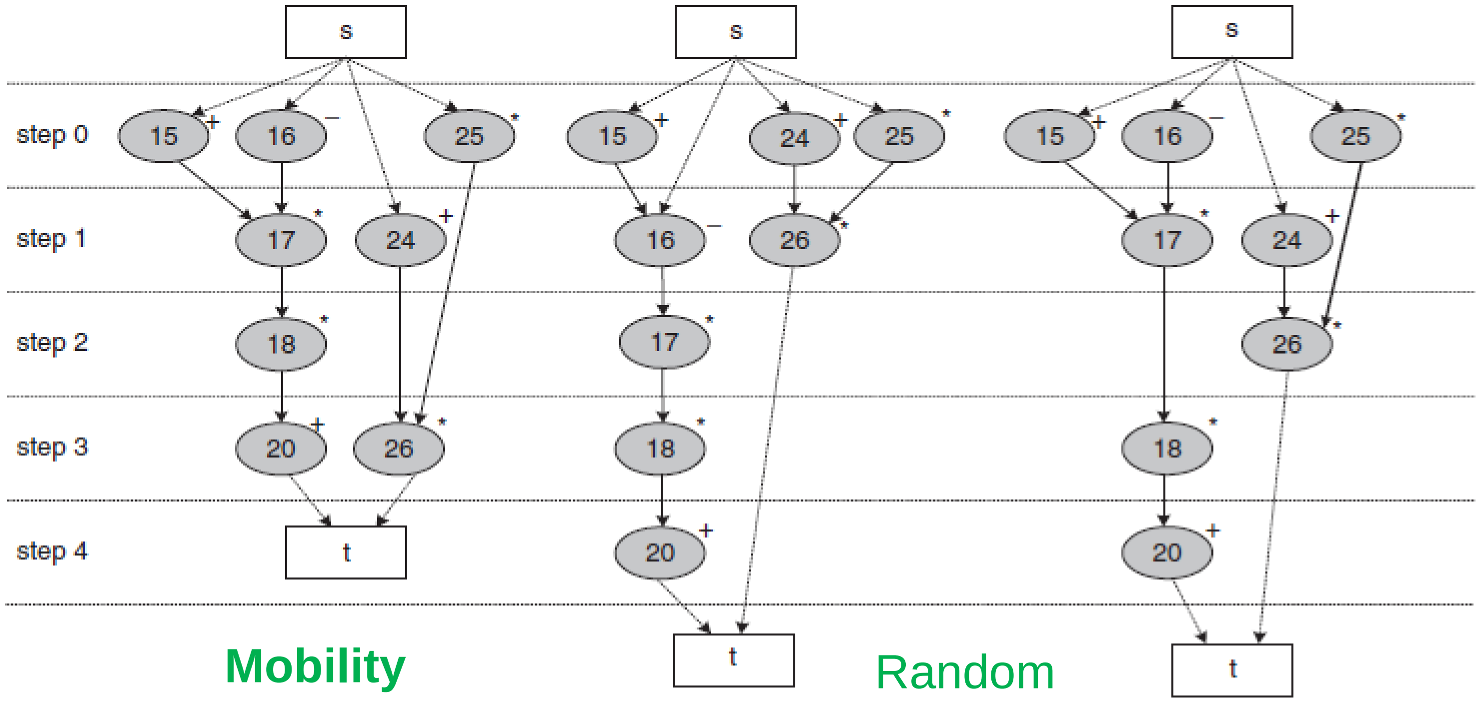


Ready List
{15, 16, 24, 25}
{24, 17}
{18, 26}
{26, 20}

STEP	ADD/SUB 1	ADD/SUB 2	MULT
0	15	16	25
1	24	-	17
2	-	-	18
3	20	-	26

List Scheduling - Example

- Add/Sub: 15, 16, 20, 24;   Mult: 17, 18, 25, 26
  - Random priority function



Ready List
{15, 16, 24, 25}
{16, 26}
{17}
{18}
{20}

STEP	ADD/SUB 1	ADD/SUB 2	MULT
0	15	24	25
1	16	-	26
2	-	-	17
3	-	-	18
4	20	-	-

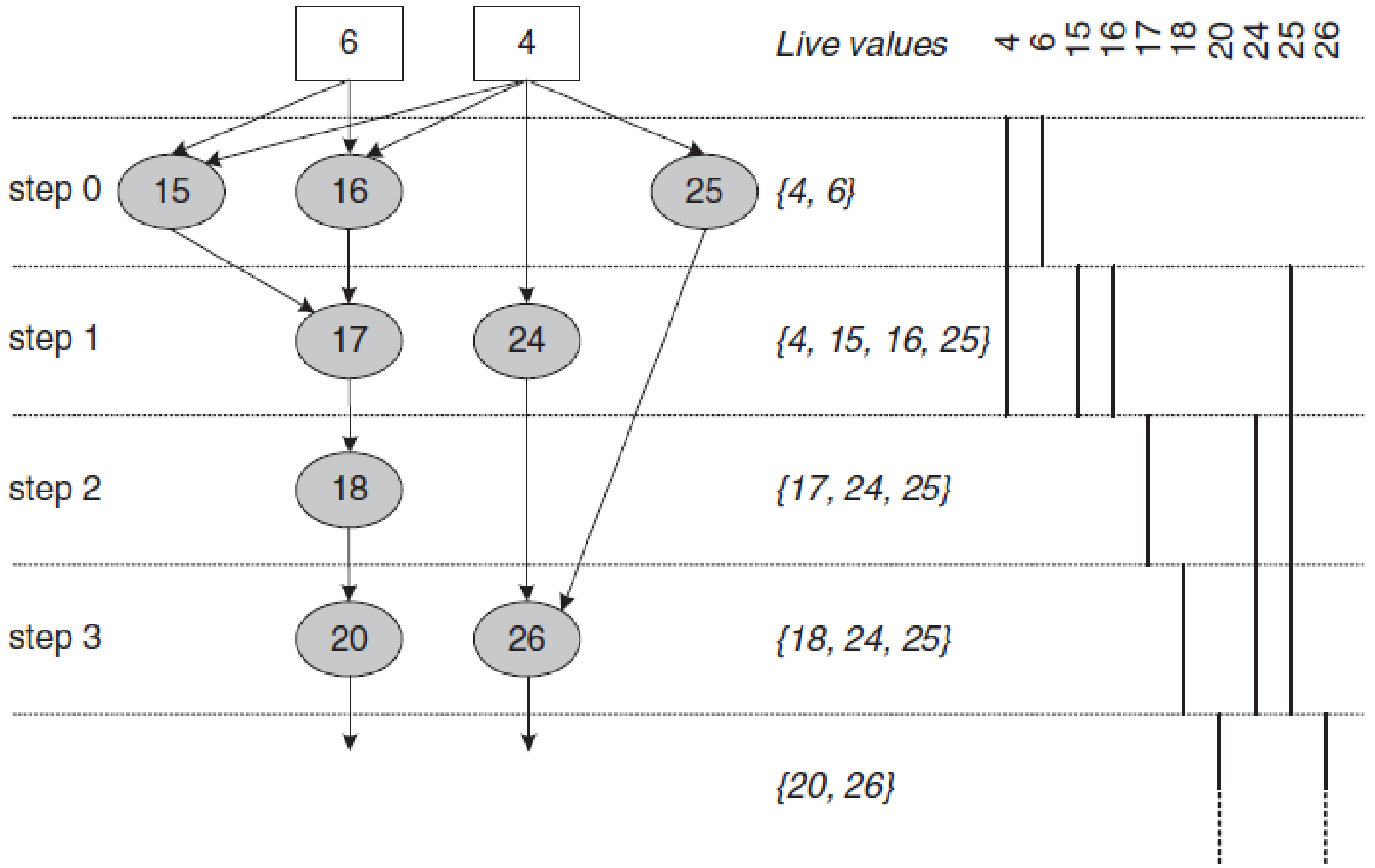
Ready List
{15, 16, 24, 25}
{24, 17}
{18, 26}
{18}
{20}

STEP	ADD/SUB 1	ADD/SUB 2	MULT
0	15	24	25
1	24	-	17
2	-	-	26
3	-	-	18
4	20	-	-

Register Binding

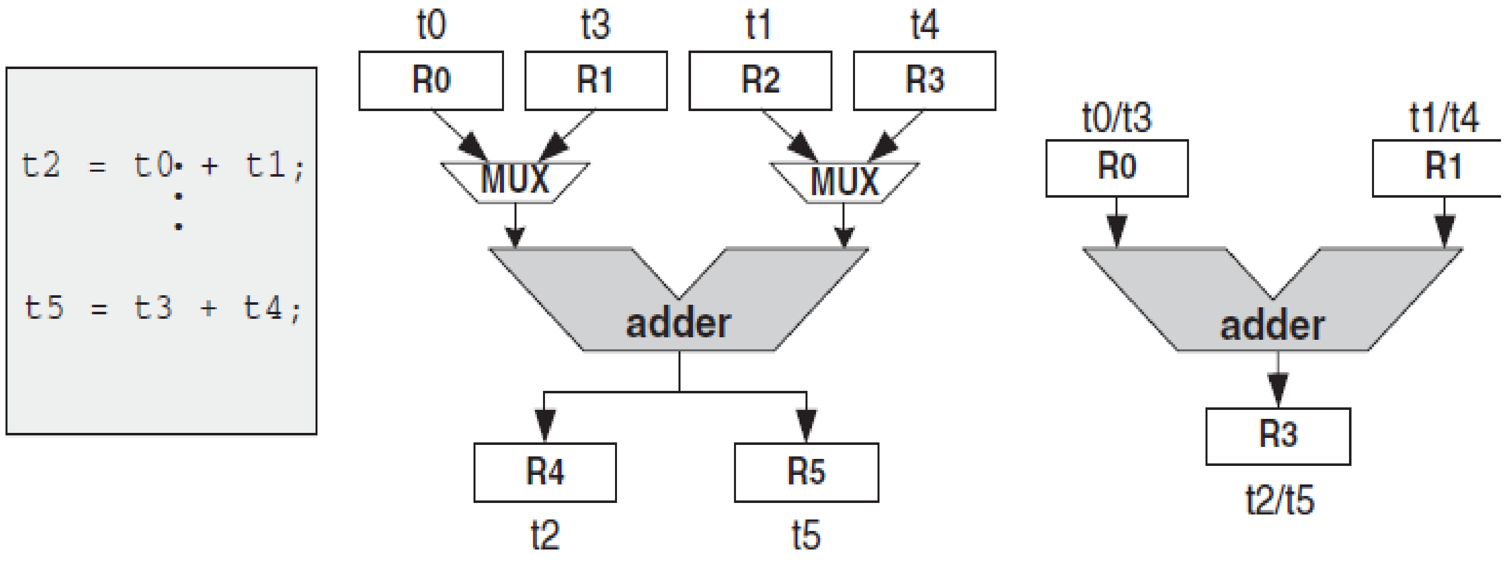
- Register sharing
  - Liveness analysis & Left edge algorithm

R0: {6, 25, 26}
R1: {16, 20, 24}
R2: {15, 17, 18}
R3: {4}



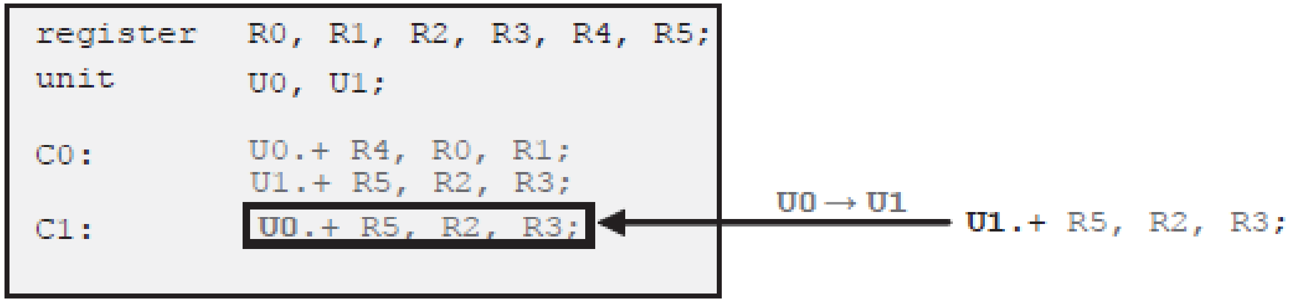
Functional Unit Binding

- Impact of register sharing on **multiplexers**
  - In an ASIC, six-input multiplexer -> an adder
  - In FPGA, typically implemented as lookup tables, often larger than adders.


```
t2 = t0 + t1;
      :
t5 = t3 + t4;
```

Functional Unit Binding

- Impaction of functional unit binding on multiplexers

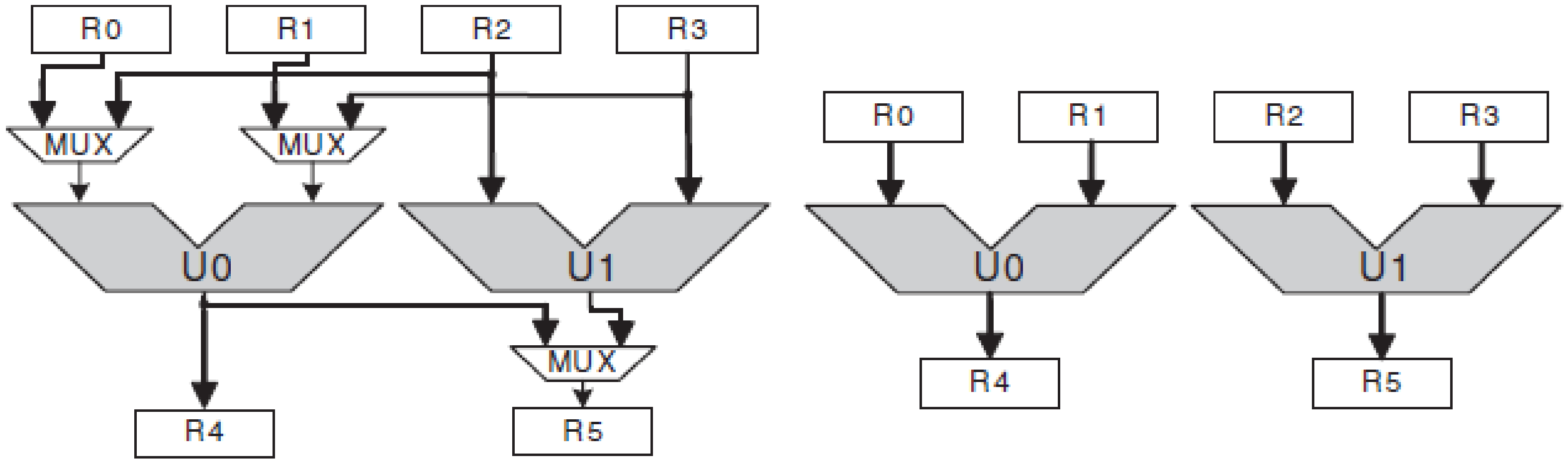


```
register    R0, R1, R2, R3, R4, R5;
unit        U0, U1;

C0:         U0.+ R4, R0, R1;
            U1.+ R5, R2, R3;
C1:         U0.+ R5, R2, R3;
```

U0 → U1

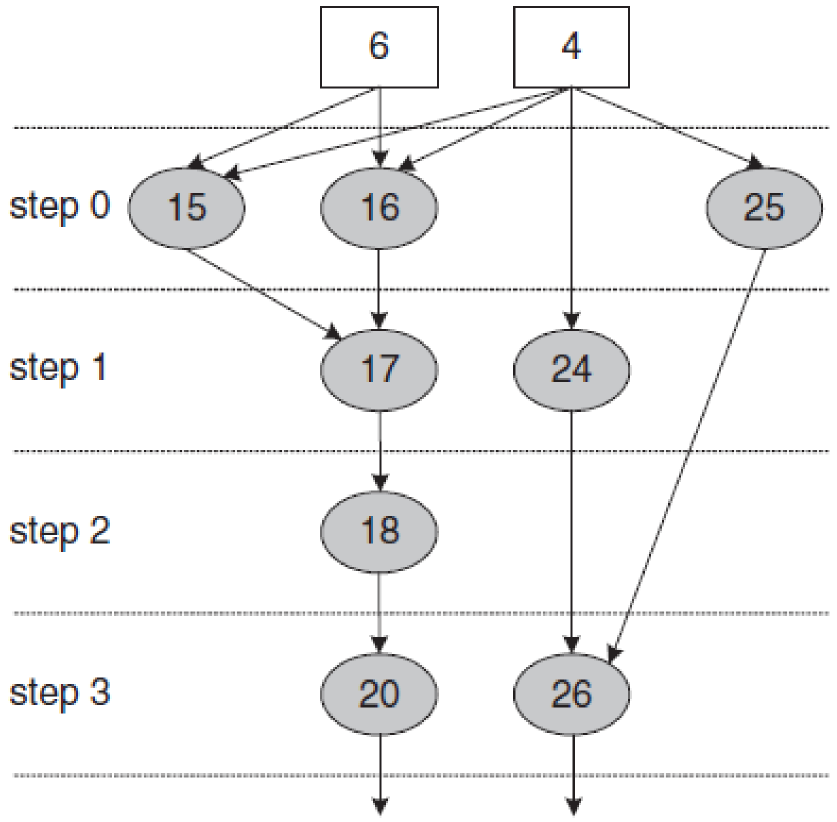
```
U1.+ R5, R2, R3;
```

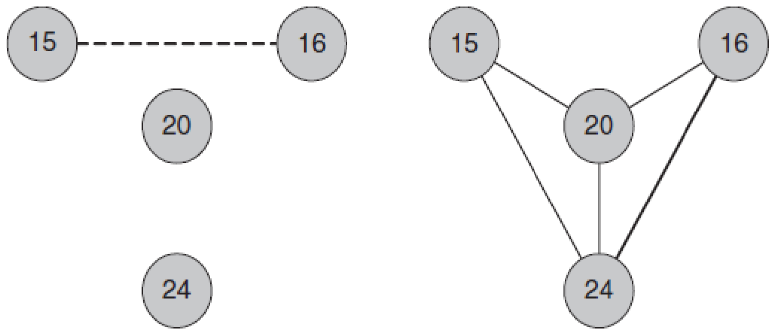
(a)



Functional Unit Binding

- Add/Sub: 15, 16, 20, 24; Mult: 17, 18, 25, 26





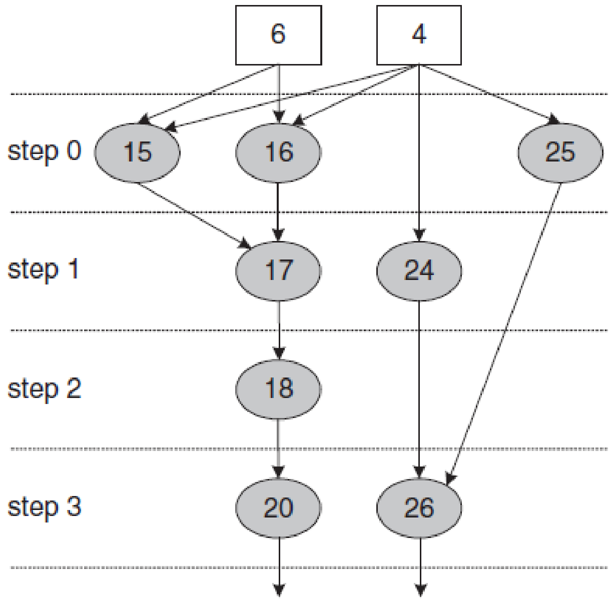
Interference graph & Compatibility graph

R0: {6, 25, 26}
R1: {16, 20, 24}
R2: {15, 17, 18}
R3: {4}

INSTRUCTION	SRC 1	SRC 2
15	4	6
16	4	6
20	18	‹16›
24	4	‹12›

Functional Unit Binding

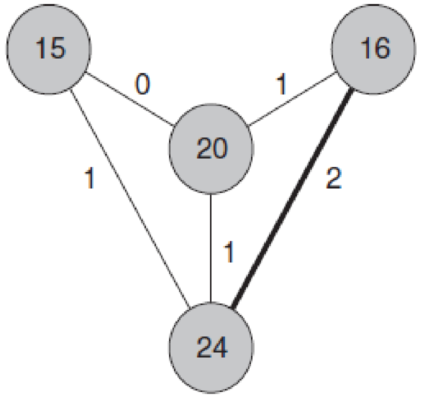
- Iterative Clique Partitioning



Iteration 1

INSTRUCTION	SRC 1	SRC 2	DEST
15	R3	R0	R2
16	R3	R0	R1
20	R2		R1
24	R3		R1

(a)

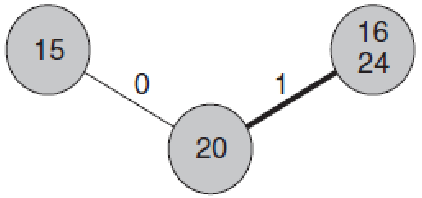


(b)

Iteration 2

INSTRUCTION	SRC 1	SRC 2	DEST
15	R3	R0	R2
16, 24	R3	R0	R1
20	R2		R1

(c)

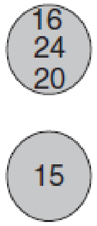


(d)

After iteration 2

INSTRUCTION	SRC 1	SRC 2	DEST
15	R3	R0	R2
16, 24, 20	R3, R2	R0	R1

(e)



(f)

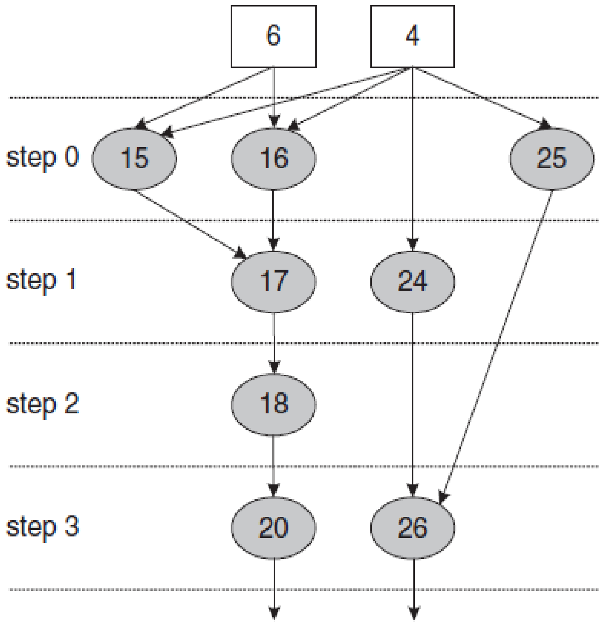
Combine Scheduling, Register/Functional Unit Binding

REGISTER	VALUES
R0	6, 25, 26
R1	16, 20, 24
R2	15, 17, 18
R3	4

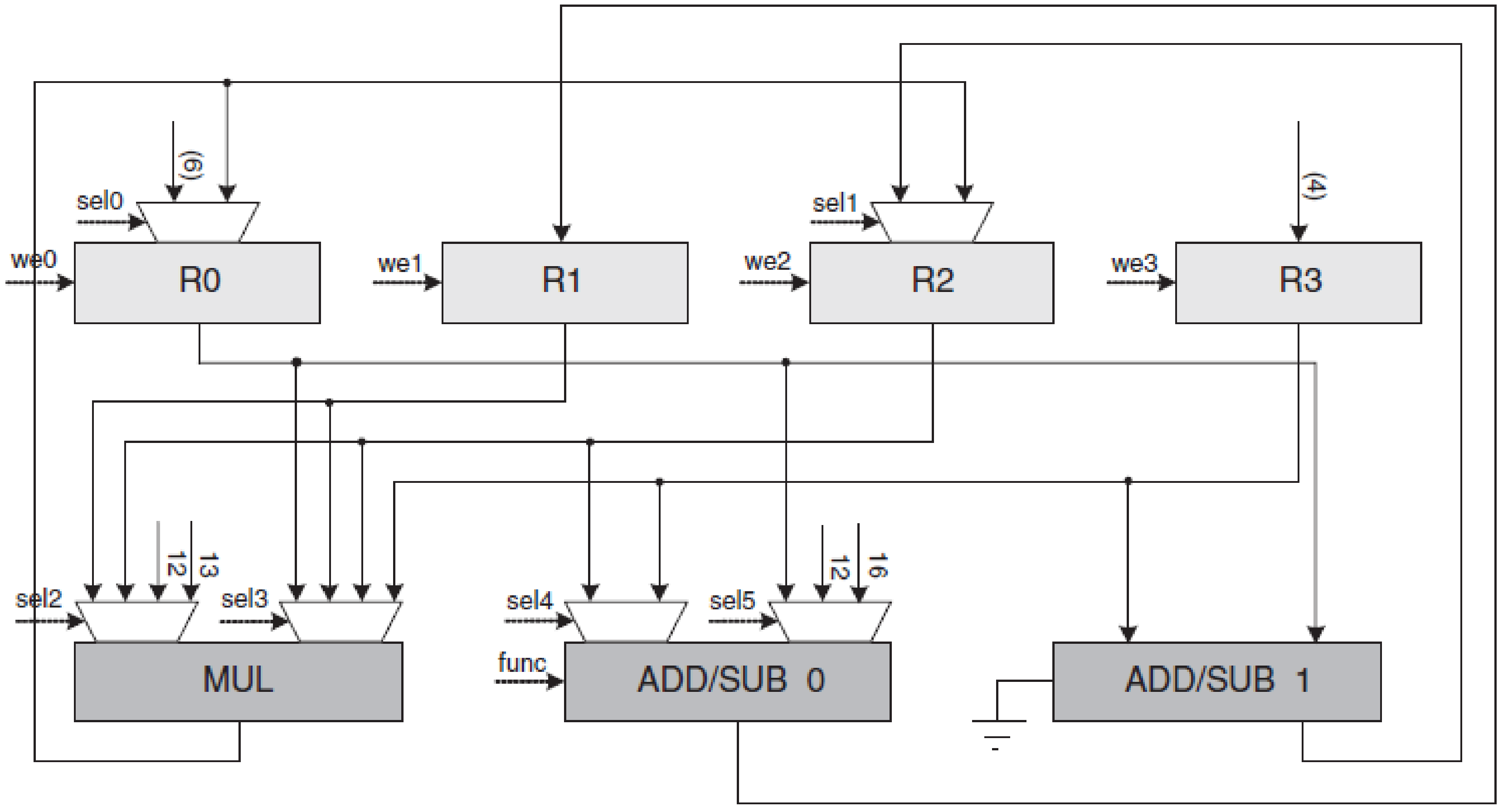
UNIT	INSTRUCTIONS
ADD/SUB 0	16, 20, 24
ADD/SUB 1	15
MULT	17, 18, 25, 26

REGISTER	INPUTS	VALUES
R0	external input	6
	MULT	25, 26
R1	ADD/SUB 0	16, 20, 24
R2	ADD/SUB 1	15
	MULT	17, 18
R3	external input	4

UNIT	INPUT PORT 1	INPUT PORT 2
ADD/SUB 0	R2, R3	R0, ‹12›, ‹16›
ADD/SUB 1	R3	R0
MULT	R1, R2, ‹12›, ‹13›	R0, R1, R2, R3

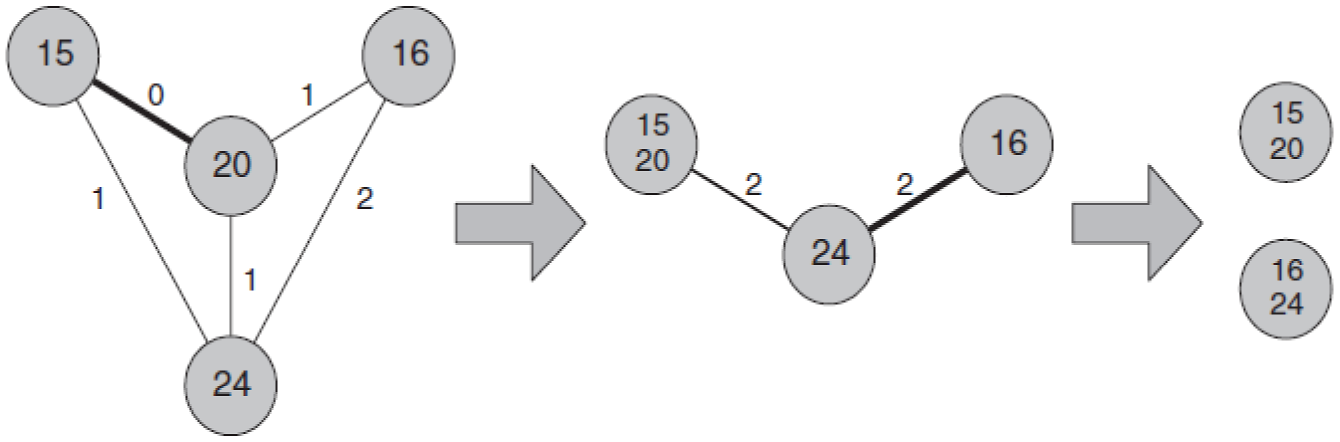


Synthesized datapath

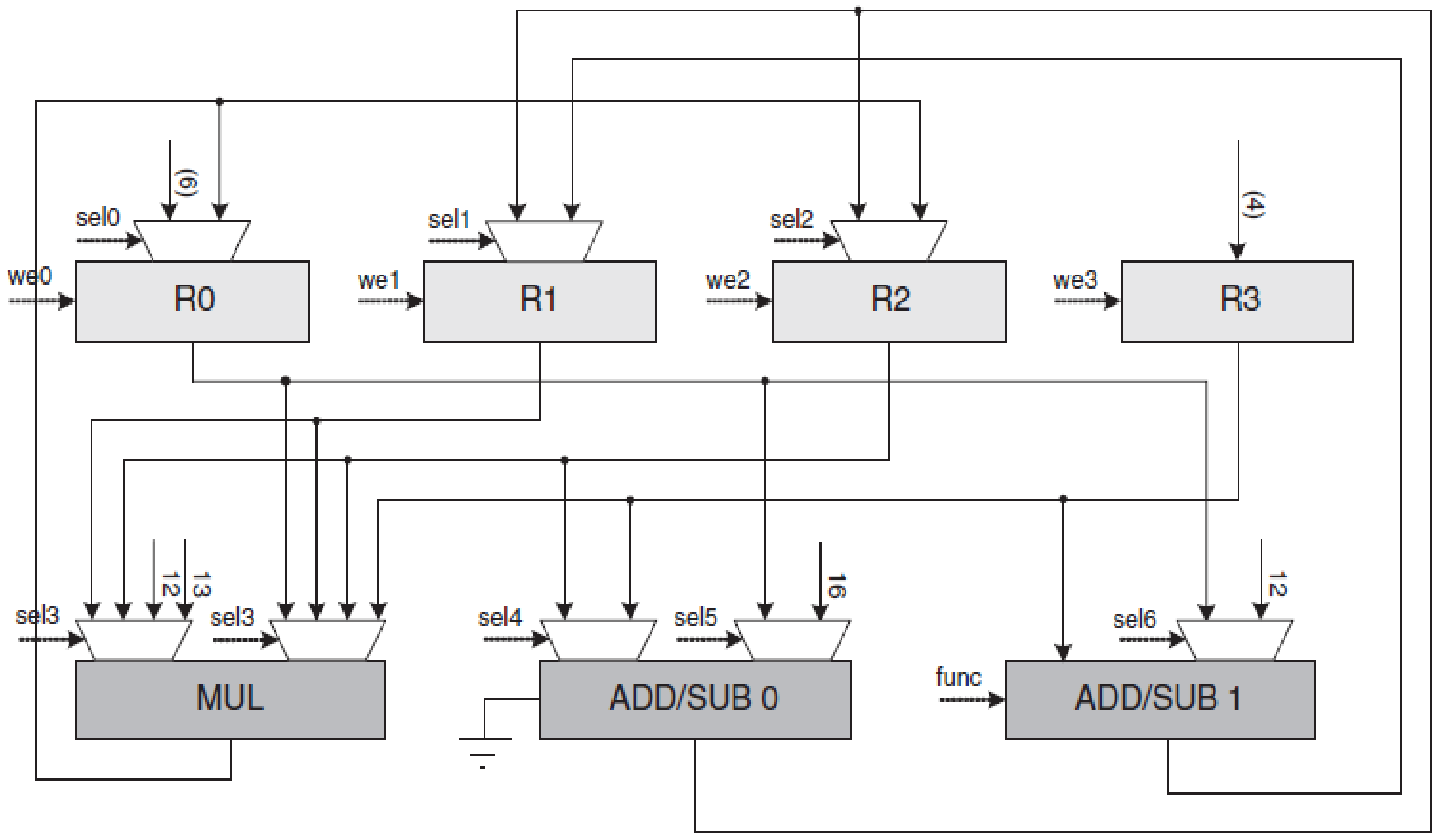


Clique partitioning binding

- Randomly select the contracted edge



Synthesized datapath



Summary

- A Design Example
  - Great Common Divisor
- High-level Synthesis
  - DFG/CDFG
  - Scheduling
    - ASAP, ALAP, and List Scheduling
  - Binding
    - Register binding, Functional unit binding (sharing)
  - Allocation