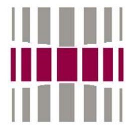


CREDES Summer School

Dependable Systems Design

June 2-3, 2011. Tallinn University of Technology, Estonia

Bottlenecks in Hardware Design and Design Automation

[Hardware Synthesis: No Pain, No Gain]

Peeter Ellervee

Department of Computer Engineering, Tallinn University of Technology, Estonia

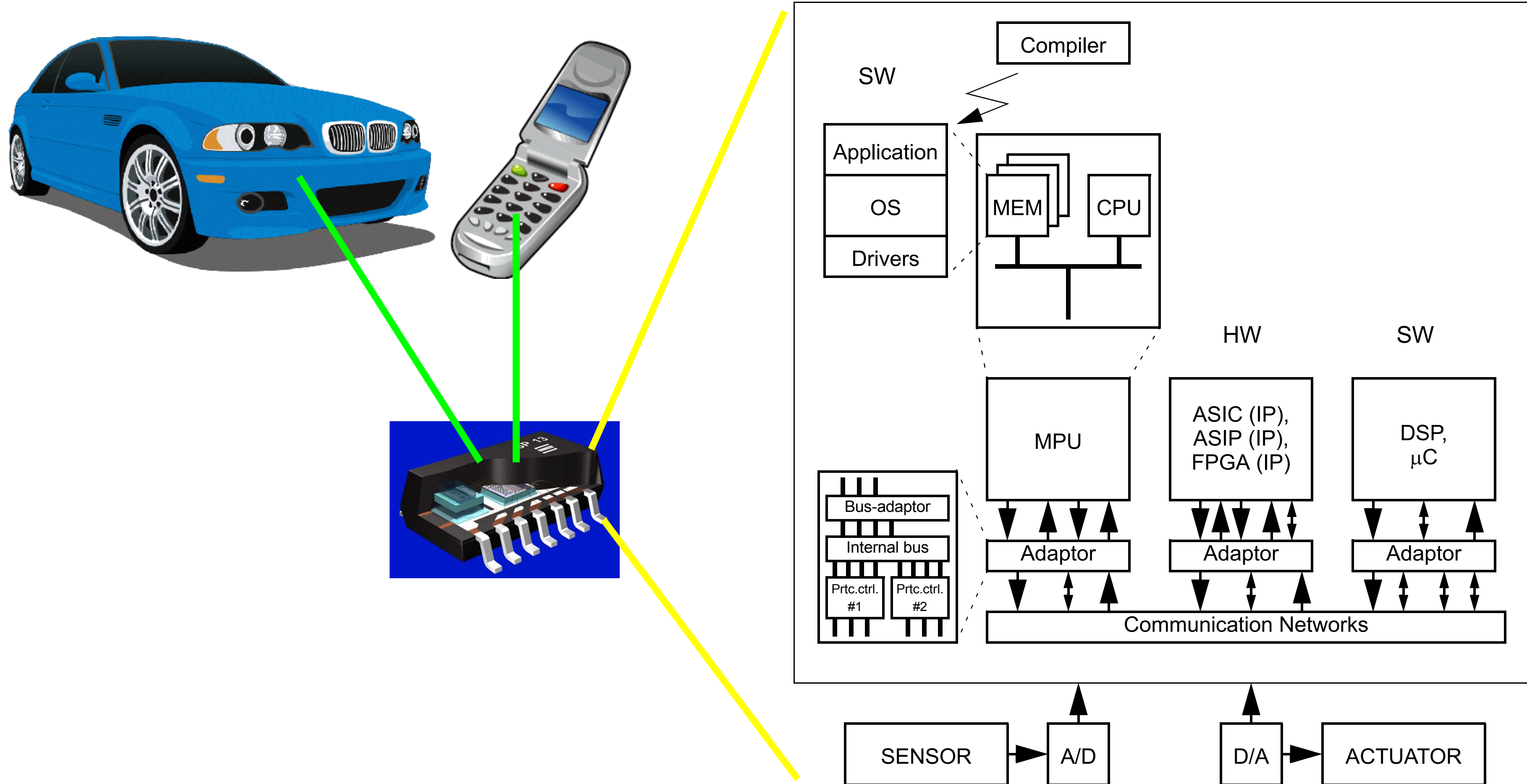
LRV@cc.ttu.ee http://www.ttu.ee/users/lrv/ http://mini.li.ttu.ee/~lrv/

© Peeter Ellervee

Motivation

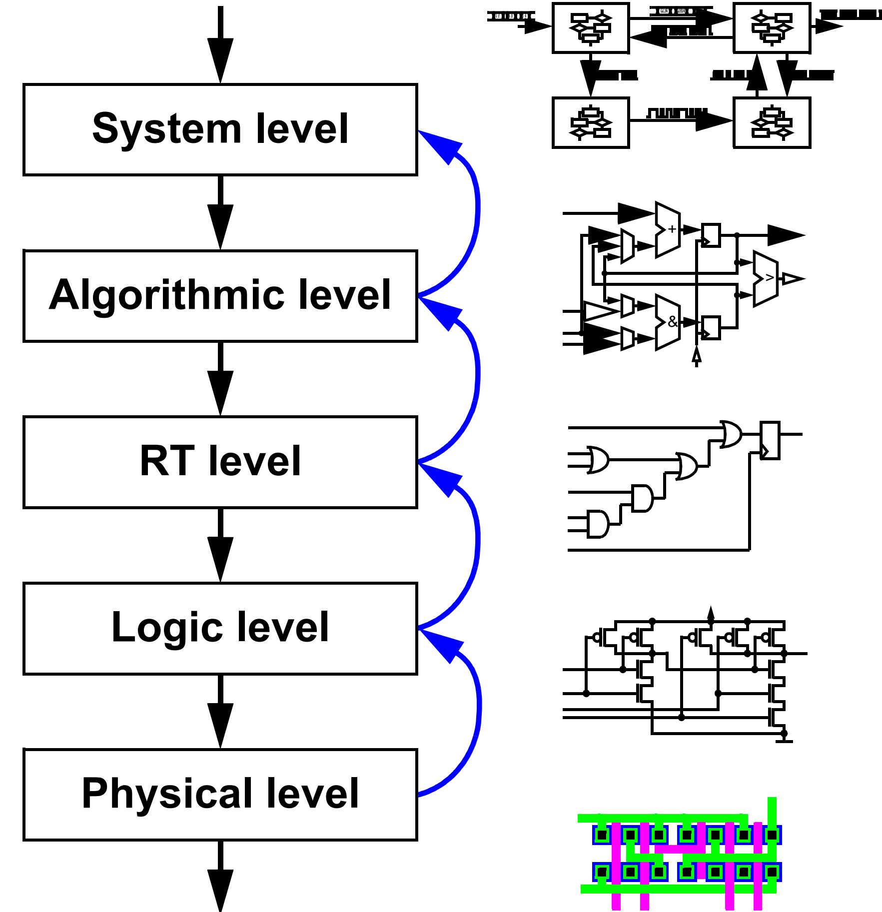
- Integrated circuits scale by Moore's Law
  - ... and this may continue for a while...
- Systems on a chip (SoC, NoC, etc.) require new design methodologies
  - to increase designers productivity
  - to get products faster into market
- There exists a demand for efficient design methodologies at higher abstraction levels
- A different thinking needed from the designers
- At higher abstraction levels
  - a designer has much wider selection of possible decisions
  - each of these decisions has also a stronger impact onto the quality of the final design

Embedded systems and chips are everywhere



Design flow

- Specification refinement
  - from higher to lower abstraction levels
  - refinement = transformations
- Algorithm selection
  - universal vs. specific
  - performance vs. memory consumption
- Partitioning
  - introducing structure
- Technology mapping
  - replacing Boolean equations with gates

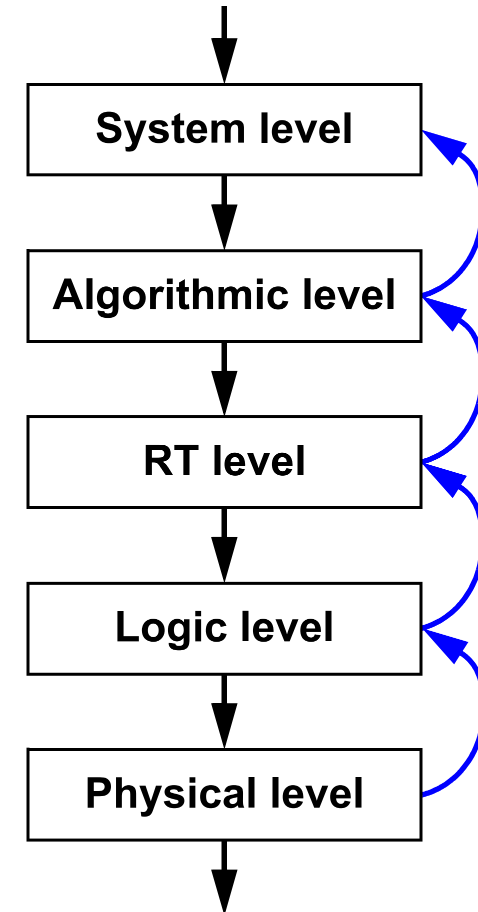


Y-chart and transformations

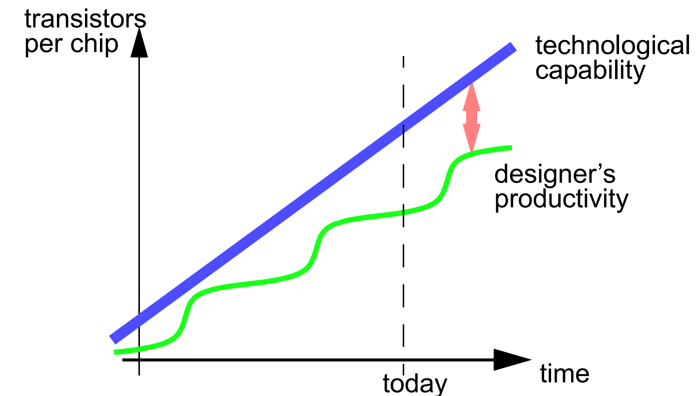
TTÜ1918

Synthesis – levels and tasks

- **System Level Synthesis**
  - **Clustering. Communication synthesis.**
- **High-Level Synthesis**
  - **Resource or time constrained scheduling.**
  - **Resource allocation. Binding.**
- **Register-Transfer Level Synthesis**
  - **Data-path synthesis. Controller synthesis.**
- **Logic Level Synthesis**
  - **Logic minimization. Optimization, overhead removal.**
- **Physical Level Synthesis**
  - **Library mapping. Placement. Routing.**



Synthesis – design automation



- **1990 – 4 Kgates / year / designer**
- **1993 – inhouse place and route – 5.6K**
- **1995 – engineer (RTL-->GDSII) – 9.1K**
- **1997 – small blocks reuse (2.5K-75K) – 40K**
- **1999 – large blocks reuse (75K-1M) – 56K**
- **2001 – synthesis (RTL-->GDSII) – 91K**
- **2003 – intelligent test-bench – 125K**
- **2005 – behavioral and architectural levels, HW/SW (co)design – 200K**
- **2007 – very large blocks reuse (>1M, IP cores) – 600K**
- **2009 – homogeneous parallel processing (multi-core) – 1200K**
- ***Future – hw/sw co-verification, executable specification, etc.***

© Peeter Ellervee

Optimizations

- **Optimizations at logic level**
  - **thousands of nodes (gates) can exist**
  - **only few possible ways exist how to map an abstract gate onto physical gate from target library**
  - **optimization algorithms can take into account only few of the neighbors**
- **Optimizations at register transfer level (RTL)**
  - **hundreds of nodes can exist (adders, registers, etc.)**
  - **there are tens of possibilities how to implement a single module**
- **At higher levels, e.g. at system level**
  - **there are only tens of nodes to handle (to optimize)**
  - **there may exist <u>hundreds</u> of ways how to implement a single node**
  - **every possible decision affects much stronger the constraints put onto neighboring nodes thus significantly affecting the quality of the whole design**

Decisions at higher abstraction levels

- Two major groups of decisions
  - selection of the right algorithm to solve a subtask
    - making transformations inside the algorithm, e.g. parallel versus sequential execution
    - affect primarily the final architecture of the chip
  - decisions about the data representation
    - e.g. floating point versus fixed point arithmetic, bit-width, precision.
- Selection of a certain algorithm puts additional constraints also onto the data representation
- Selecting a data representation narrows also the number of algorithms available

Logic synthesis

- Transforming logic functions (Boolean functions) into a set of logic gates
  - transformations at logic level from behavioral to structural domain
- Optimizations / Transformations
  - area
  - delay
  - power consumption
- Implementation of Finite State Machines (FSM)
  - state encoding
  - generating next state and output functions
  - optimization of next state and output functions

Logic synthesis – main tasks

- Optimization of (abstract) representation of logic functions
  - minimization of two-level representation
  - optimization of binary decision diagrams (BDD)
- Synthesis of multi-level combinational nets (circuits)
  - optimizations for area, delay, power consumption, and/or testability
- Optimization of state machines (FSMs)
  - state minimization, encoding
- Synthesis of multi-level sequential nets (circuits)
  - optimizations for area, delay, power consumption, and/or testability
- Library mapping
  - optimal gate selection

TTÜ1918

Register-transfer level synthesis

- Transformation from RT-level structural description (in terms of registers, multiplexers and operations) to Logic level description (in terms of combinational logic blocks and storage elements)
- Data path synthesis
  - maximizing the clock frequency
  - retiming
  - operator selection
- Controller synthesis
  - architecture selection
  - FSM optimizations for area and performance
  - state assignment / coding
  - decomposition

© Peeter Ellervee

Register-transfer level synthesis

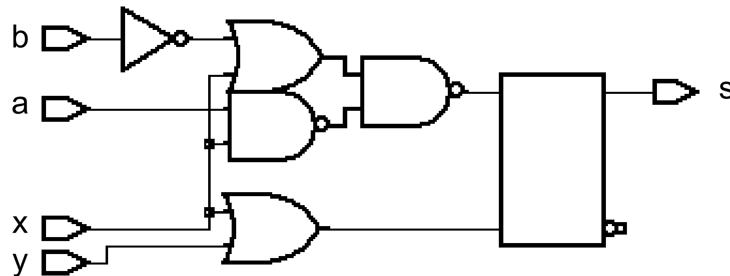
- **The most common design entry**
- **Hardware description languages (HDL) instead of schematic capture**
- **Automation well established, many good tools exist**

- **Problems**
  - **The design must be described as a <u>schematic</u> to get a good implementation**
    - **in terms of registers, functional units and multiplexers**
  - **Only sub-sets of HDL-s are synthesizable**
    - **HDL-s were created for simulation and not for synthesis**
    - **different tools support different sub-sets (extending, e.g., IEEE Std. 1076.6)**
  - **Certain rules must be followed to get desired results**
    - ***"what-you-write-is-what-you-get"***

TTÜ1918

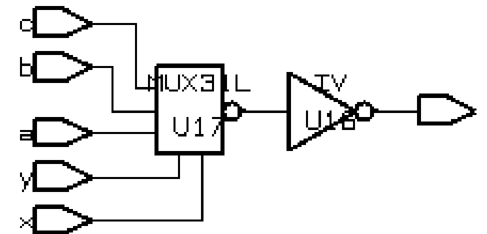
Examples

```
process (A, B, X, Y) begin
  if    X='1' then  S <= A;
  elsif Y='1' then  S <= B;
  end if;
end process;
```

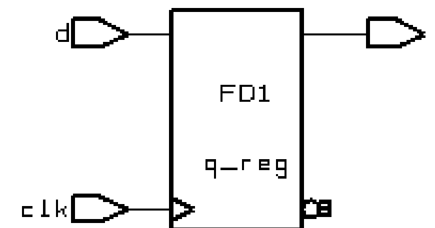


Memory element generated!

```
process (A, B, C, X, Y) begin
  if    X='1' then  S <= A;
  elsif Y='1' then  S <= B;
  else              S <= C;
  end if;
end process;
```



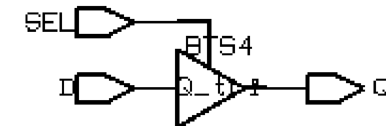

```
process (CLK) begin
  if CLK='1' and CLK'event then Q<=D;
  end if;
end process P1_FF;
```



© Peeter Ellervee

Synthesis rules

- **Guidelines in priority order:**
  - **the target signal(s) will be synthesized as flip-flops when there is a signal edge expression, e.g. CLK’event and CLK=’1’, in the process**
  - **usually, only one edge expression is allowed per process**
    - **different processes can have different clocks (tool depending)**
  - **the target signal will infer three-state buffer(s) when it can be assigned a value ’Z’**
    - **example: Q <= D when SEL=’1’ else ’Z’;**



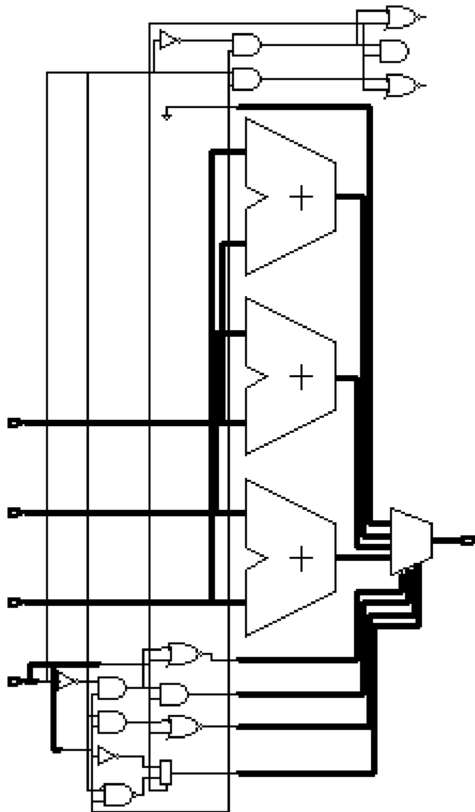
  - **the target signal will infer a latch (latches) when the target signal is not assigned with a value in every conditional branch, and the edge expression is missing**
  - **a combinational circuit will be synthesized otherwise**

- **It is a good practice to isolate flip-flops, latches and three-state buffers inferences to ensure design correctness**

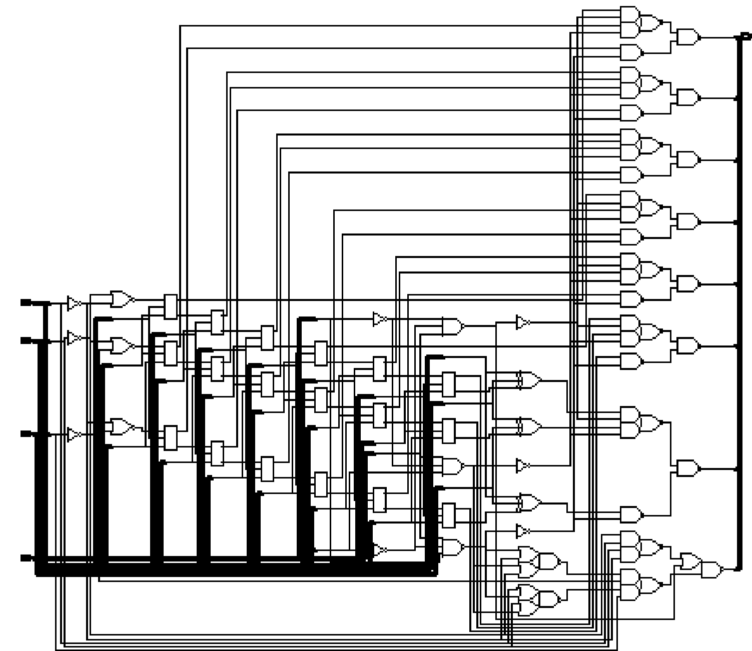
TTÜ1918

Example #2 – where to have the multiplexers?

```
process (a, b, c, x) begin
  case x is
  when ″010″ =>    o <= a+b;
  when ″011″ =>    o <= a+c;
  when ″110″ =>    o <= b+c;
  when others =>   o <= (others=>'0');
  end case;
end process;
```



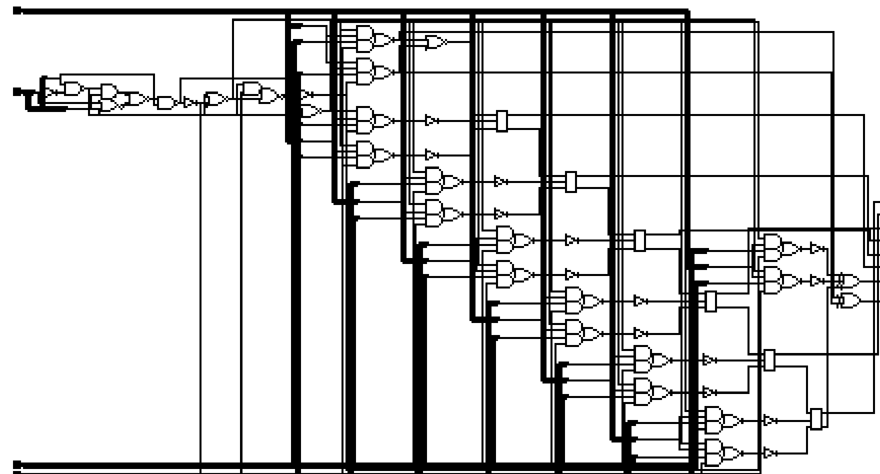
220 gates / 11.57 ns



Example #2 – explicit structure

```
architecture rtl of test is
  signal a1, a2: unsigned(7 downto 0);
  signal dc: unsigned(1 downto 0);
begin
dec: process (x) begin
  case x is
  when ″010″ =>   dc <= ″01″;
  when ″011″ =>   dc <= ″10″;
  when ″110″ =>   dc <= ″11″;
  when others =>  dc <= ″00″;
  end case;
end process dec;
m1: process (a, b, dc) begin
  case dc is
  when ″01″ =>    a1 <= a;
  when ″10″ =>    a1 <= a;
  when ″11″ =>    a1 <= b;
  when others =>  a1 <= (others=>'0');
  end case;
end process m1;
m2: process (b, c, dc) begin
  -- ...
end process m2;
o <= a1 + a2;
end architecture rtl;
```

117 gates / 19.2 ns



Example #3 – adder-subtracter

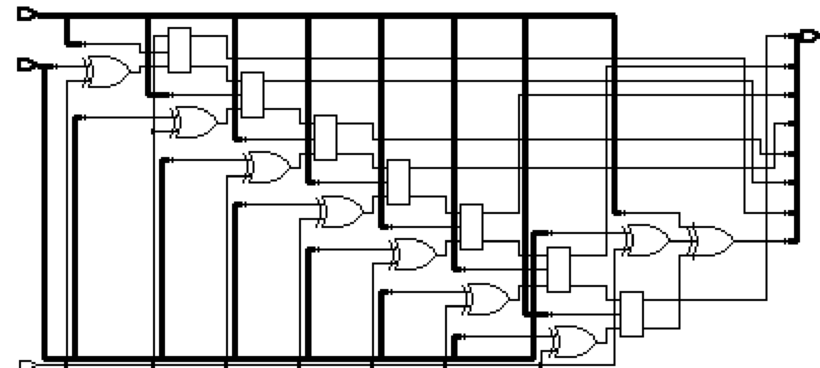
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity  add_sub  is
  port ( a, b: in unsigned(7 downto 0);
         x: in std_logic;
         o: out unsigned(7 downto 0) );
end add_sub;
architecture bhv of add_sub is begin
process (a, b, x) begin
  if x='0' then  o <= a+b;
  else           o <= a-b;  end if;
end process;
end architecture bhv;
```

```
module add_sub (a, b, x, o);
  input [7:0] a, b;
  input x;
  output [7:0] o;
  assign o = x==0 ? a+b : a-b;
endmodule // add_sub
```

145 gates / 11.64 ns

```
architecture dfl of test5 is
  signal a1, b1, o1: unsigned(8 downto 0);
begin
  a1 <= a & '1';
  b1 <= b & '0' when x='0' else
    unsigned(not std_logic_vector(b)) &
    '1';
  o1 <= a1+b1;
  o <= o1(8 downto 1);
end architecture dfl;

  /* ... */
  assign {o,t} = {a,1'b1} +
    ( x==0 ? {b,1'b0} : {~b,1'b1} );
```

87 gates / 12.45 ns



© Peeter Ellervee

CREDES

Example #4 – GCD (Greatest Common Divisor)

- Specification ~~ behavioral description
  - input/output timing fixed – handshaking signals & clock

```
process
  variable x, y: unsigned(15 downto 0);
begin
  -- Wait for the new input data
  wait on clk until clk='1' and rst='0';
  x := xi;    y := yi;    rdy <= '0';
  wait on clk until clk='1';
  -- Calculate
  while  x /= y  loop
    if  x < y  then y := y - x;
    else            x := x - y;    end if;
  end loop;
  -- Ready
  xo <= x;   rdy <= '1';
  wait on clk until clk='1';
end process;
```

Problems
- inner loop not clocked
- complex wait statement
(- multiple wait statements)

What to look for?
- different synthesis tools
- minimizing resources
- maximizing performance

Target technologies – ASIC, FPGA

Example #4 – synthesizable code?

- **Clocked behavioral style**

```
process
  variable x, y: unsigned(15 downto 0);
begin
  -- Wait for the new input data
  while  rst = '1'  loop
    wait on clk until clk='1';
  end loop;
  x := xi;     y := yi;     rdy <= '0';
  wait on clk until clk='1';
  -- Calculate
  while  x /= y  loop
    if  x < y  then y := y - x;
    else            x := x - y;    end if;
    wait on clk until clk='1';
  end loop;
  -- Ready
  xo <= x;   rdy <= '1';
  wait on clk until clk='1';
end process;
```

ASIC: synthesizable
961 e.g. / 20.0 ns
2 sub-s, 2 comp-s

FPGA: non-synthesizable
wait statements in loops :(
explicit FSM needed :(:(

Possible trade-offs
- functional unit sharing
- universal functional units
- out-of-order execution

VHDL code & testbenches
`http://mini.li.ttu.ee/~lrv/gcd/`

© Peeter Ellervee

CREDES

Example #4 – behavioral FSM

```
process begin
  wait on clk until clk='1';
  case  state  is
  -- Wait for the new input data
  when S_wait =>
   if  rst='0'  then
     x<=xi; y<=yi; rdy<='0'; state<=S_start;
   end if;
  -- Calculate
  when S_start =>
    if  x /= y  then
      if  x < y  then  y <= y - x;
      else               x <= x - y;    end if;
      state<=S_start;
    else
      xo<=x; rdy<='1'; state<=S_ready;
    end if;
  -- Ready
  when S_ready =>    state<=S_wait;
  end case;
end process;
```

ASIC: synthesizable
911 e.g. / 19.4 ns
2 sub-s, 2 comp-s

FPGA: synthesizable
108 SLC / 9.9 ns
2 sub-s, 2 comp-s

Can it be made better?

Again the possible trade-offs

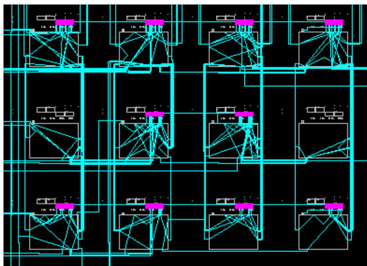
- **functional unit sharing**
  one operation per clock step
- **universal functional units**
  A<B == A-B<0 / A/=B == A-B/=0
- **out-of-order execution**
  subtracting first then deciding

Example #4 – universal functional units?

- A<B == A-B<0 / A/=B == A-B/=0

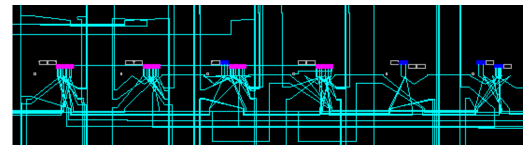
```
-- Three operations:
--   subtraction, and
--   comparisons not-equal &
--   less-than
xo <= xi - yi;

ne <= '1' when xi /= yi else '0';

lt <= '1' when xi < yi else '0';
```

ASIC: 209 e.g. / 19.9 ns
FPGA: 20 SLC / 12.1 ns
***three* adder chains!**



```
-- ALU - subtracting and then comparing
x_out <= xi - yi;    xo <= x_out;
process (x_out)
  variable or_tmp: unsigned(15 downto 0);
begin
  or_tmp(15) := x_out(15);
  for  i in 14 downto 0  loop
    or_tmp(i) := or_tmp(i+1) or x_out(i);
  end loop;
  ne <= to_bit(or_tmp(0));
end process;
lt <= to_bit(x_out(15));
```

ASIC: 148 e.g. / 21.8 ns / FPGA: 12 SLC / 14.6 ns



Example #4 – design space exploration

- Different solutions – `http://mini.li.ttu.ee/~lrv/gcd/`
  - gcd-bhv.vhdl – pure behavioral description, non-synthesizable
  - gcd-bhvc.vhdl – fully clocked behavioral style, some synthesis tools can handle
  - gcd-bfsm.vhdl – so called behavioral FSM (explicit FSM & behavioral data-path), synthesizable (but how efficient it is?)
  - gcd-rtl1.vhdl – single ALU, 3 clock cycles per iteration – 1) “not equal?”, 2) “less than?”, 3) subtract
  - gcd-rtl2.vhdl – single ALU, 2 clock cycles per iteration – 1) “not equal?” and “less than?”, 2) subtract
  - gcd-rtl3.vhdl – comparator (less than) controls subtraction, 1 clock cycle per iteration – small but slow (sequential) data-path
  - gcd-rtl4.vhdl – out-of-order execution – both subtractions are calculated first then the decision is made (one subtracter compares for “less than”, another for “not equal”)
  - gcd-rtl5.vhdl – out-of-order execution – both subtractions are calculated first then the decision is made (one subtracter compares for “less than” but separate “not equal”)

Example #4 – synthesis results

Technology	FPGA				ASIC			
Constraint[1)]	50 MHz		100 MHz		50 MHz		25 MHz	
	[SLC]	[ns]	[SLC]	[ns]	[e.g.]	[ns]	[e.g.]	[ns]
gcd-bhv[2)]	*93*	*17.3*	-	-	*1141*	*20.0*	-	-
gcd-bhvc	-	-	-	-	961	20.0	977	31.1
gcd-bfsm	108	9.9	108	9.4	911	19.4	984	30.8
gcd-rtl1	50	10.8	50	9.7	986	19.8	883	32.4
gcd-rtl2	48	10.8	48	10.0	931	19.9	882	32.3
gcd-rtl3	58	17.0	58	14.6	1134	20.0	928	40.0
gcd-rtl4	78	12.6	78	9.0	976	19.9	928	29.0
gcd-rtl5	58	8.0	58	7.6	915	20.0	932	26.9

1) Clock period was the only constraint

2) gcd-bhv was synthesized using the help prototype HLS tool xTractor

High-Level Synthesis

a.k.a. Behavioral Synthesis

a.k.a. Algorithm Level Synthesis

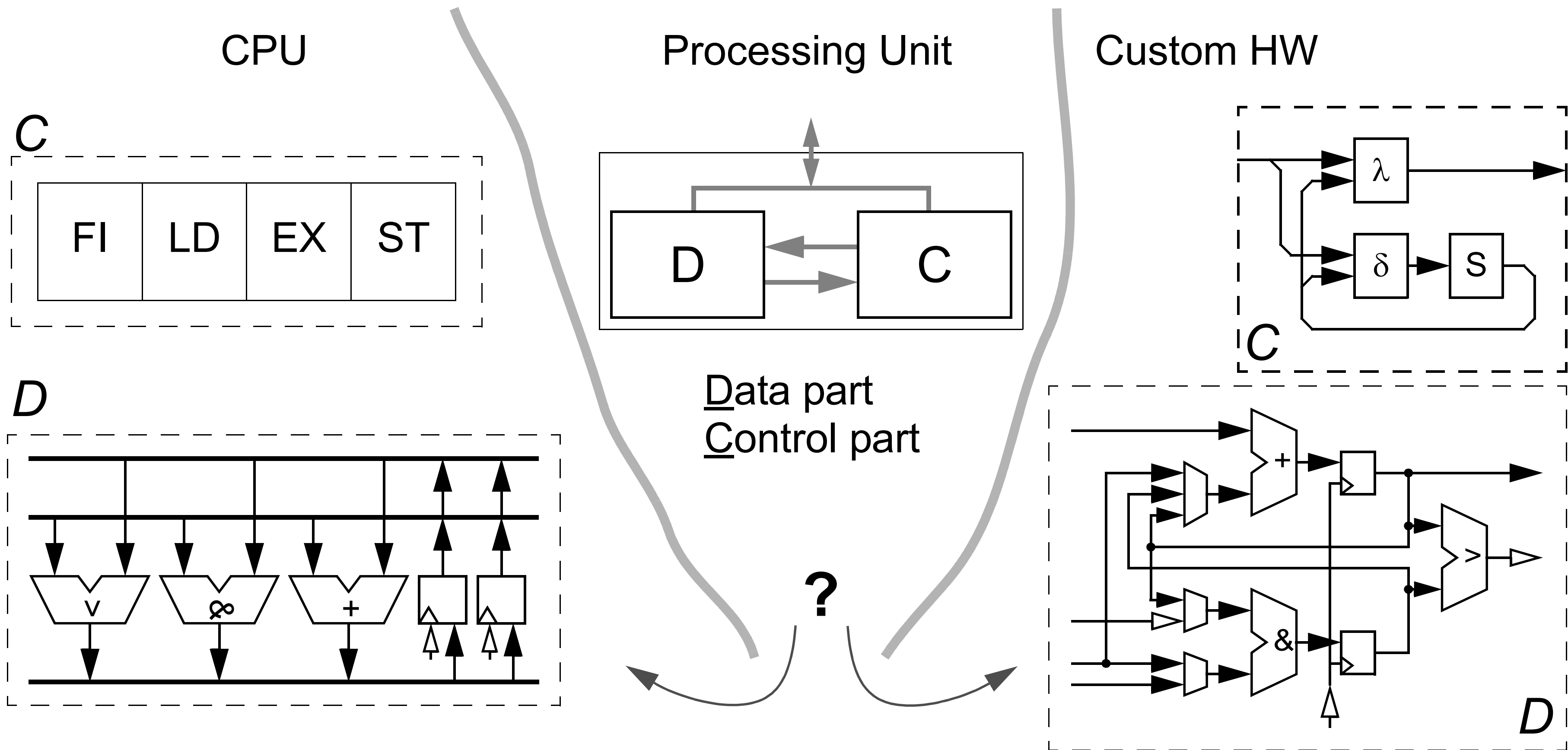
a.k.a. Silicon Compilation

- Takes a specification of the functionality of a digital system and a set of constraints, finds a structure that implements the intended behavior, and satisfies constraints
- Benefits
  - Automation simplifies handling of larger designs and speeds up exploration of different architectural solutions
  - The use of synthesis techniques promises correctness-by-construction
  - The use of higher abstraction level, i.e. the algorithmic level, helps the designer to cope with the complexity
  - An algorithm does not specify the structure to implement it, thus allowing the HLS tools to explore the design space
  - The lack of low level implementation details allows easier re-targeting and reuse of existing specifications
  - Specifications at higher level of abstraction are easier to understand thus simplifying maintenance

TTÜ1918

Software vs. hardware

Target architecture



© Peeter Ellervee

CREDES

TTÜ1918

SW compilation

- **Example – differential equation** $$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

```
{
  sc_fixed<6,10> a,dx,y,x,u,x1,x2,y1;
  while ( true ) {
    wait(); a=inport.read();
    wait(); dx=inport.read();
    wait(); y=inport.read();
    wait(); x=inport.read();
    wait(); u=inport.read();
    while ( true ) {
      for (int i=0;i<7;i++) wait();
      x1 = x + dx;  y1 = y + (u*dx);
      u = u - 5*x*(u*dx) - 3*y*dx;
      x = x1; y = y1;
      if (!(x1<a)) break;
    }
    outport.write(y);
  };
}
```

```
# R1:a, R2:dx, R3:y, R4:x, R5:u,
# R6:x1, R7:x2, R8:y1, R9:tmp
    ...
_loop_$32:
    ADD.fx    R6, R4, R2      # x1=x+dx
    MUL.fx    R9, R5, R2      # tmp=u*dx
    ADD.fx    R8, R3, R9      # y1=y+tmp
    MUL.fx    R9, R4, R9      # tmp=x*tmp
    MUL.fx    R9, R9, $5      # tmp=5*tmp
    SUB.fx    R5, R5, R9      # u=u-tmp
    MUL.fx    R9, R3, R2      # tmp=y*dx
    MUL.fx    R9, R9, $3      # tmp=3*tmp
    SUB.fx    R5, R5, R9      # u=u-tmp
    ADD.fx    R4, R6, $0      # x=x1
    ADD.fx    R3, R8, $0      # y=y1
    SUB.fx    R9, R6, R1      # tmp=x1-a
    JMP.neg   _loop_$32       # ...break
    ...
```

Target

- **SW synthesis (compilation)**
  - **input – high-level programming language**
  - **output – sequence of operations (assembler code)**
- **HW synthesis (HLS)**
  - **input – hardware description language**
  - **output – sequence of operations (microprogram)**
  - ***output – RTL description of a digital synchronous system (i.e., processor)***
    - **data part & control part**
    - **communication via flags and control signals**
    - **discrete time steps (for non-pipelined designs *time step* = *control step*)**
- **Creating the RTL structure means mapping the data and control flow in two dimensions – time and area**

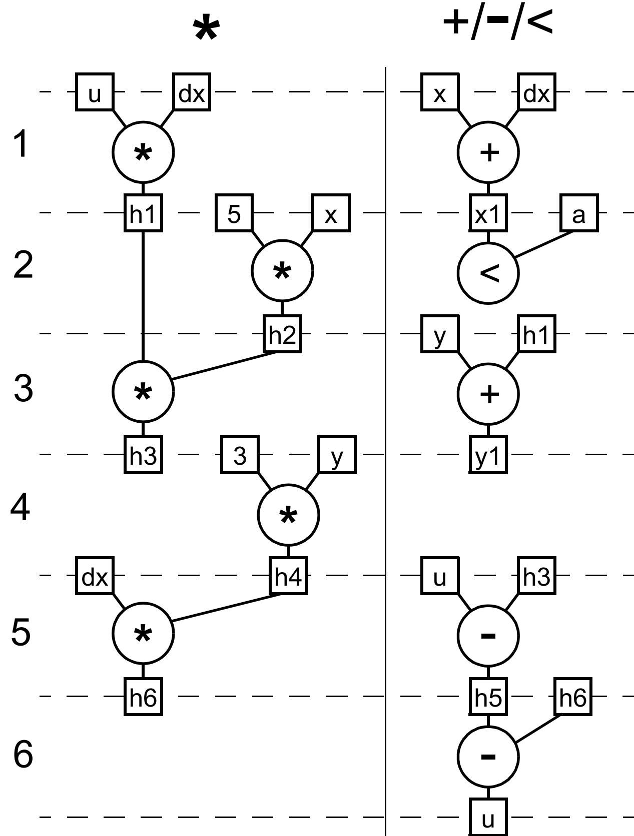
Classical high-level synthesis tasks

- **Front-end:**
  - **Deriving an internal graph-based representation equivalent to the algorithmic description of both the data flow and the control flow**
  - **Compiler optimizations**
- **Back-end:**
  - **Behavioral transformations (control and/or data graph transformations e.g. associativity, unrolling)**
  - **Transforming data and control flow into register-transfer level structure (so called *essential subtasks*)**
  - **Netlist extraction, state machine table generation**
- **Essential subtasks**
  - **Resource allocation – number and types of functional units, storage elements, and busses**
  - **Scheduling – assignment of operations to time (control steps), possibly within given constraints and minimizing a cost function**
  - **Resource binding (assignment) – operations to functional unit instances; values to be stored to instances of storage elements; and data transfers to bus instances**

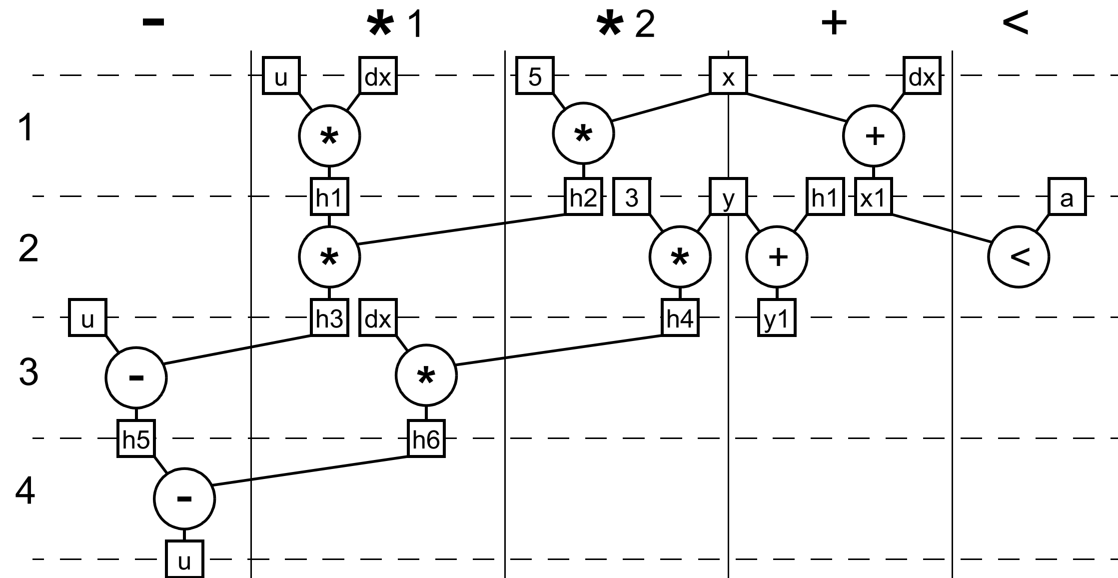
Minimal hardware vs. minimal time

Resource constrained scheduling

$$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

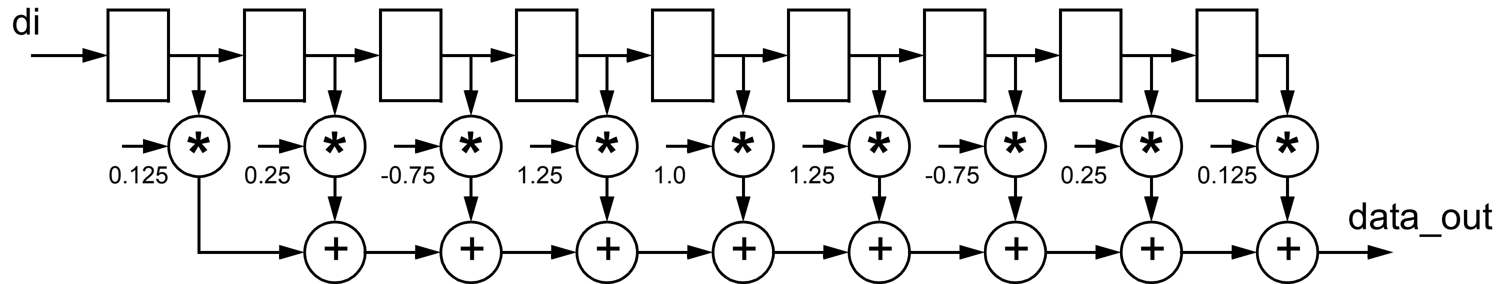


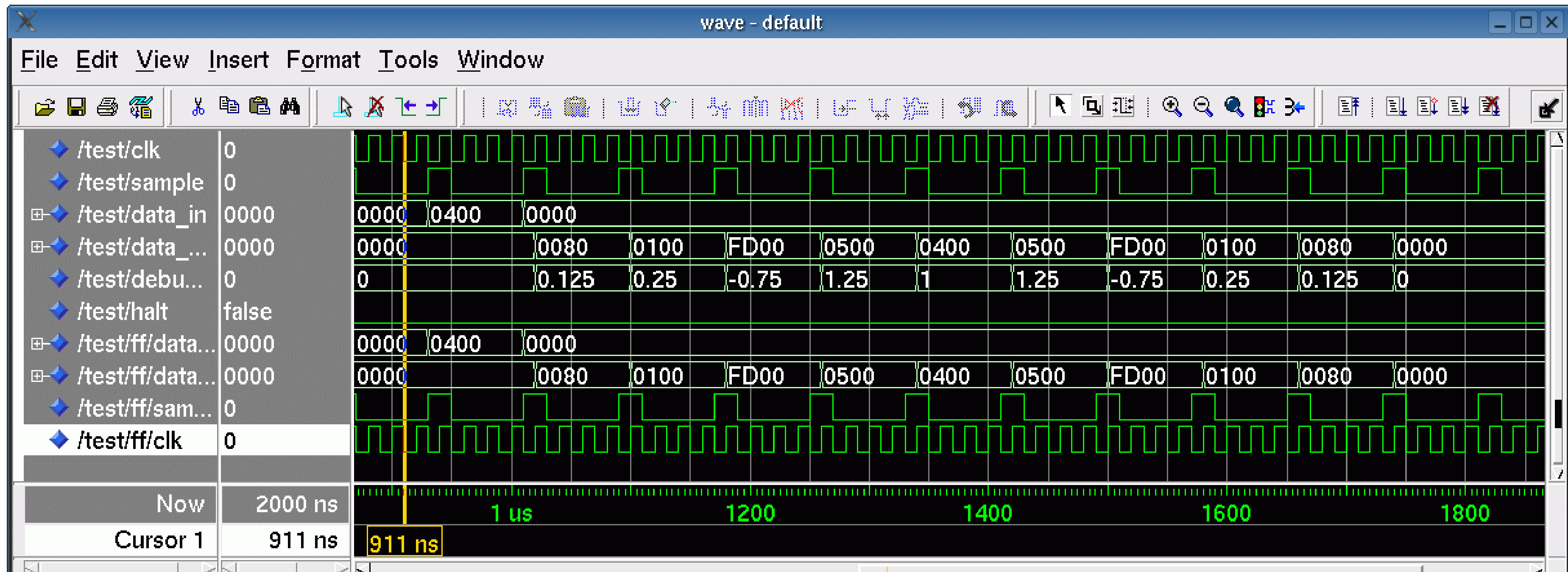
Time constrained scheduling



FIR filter implementation example

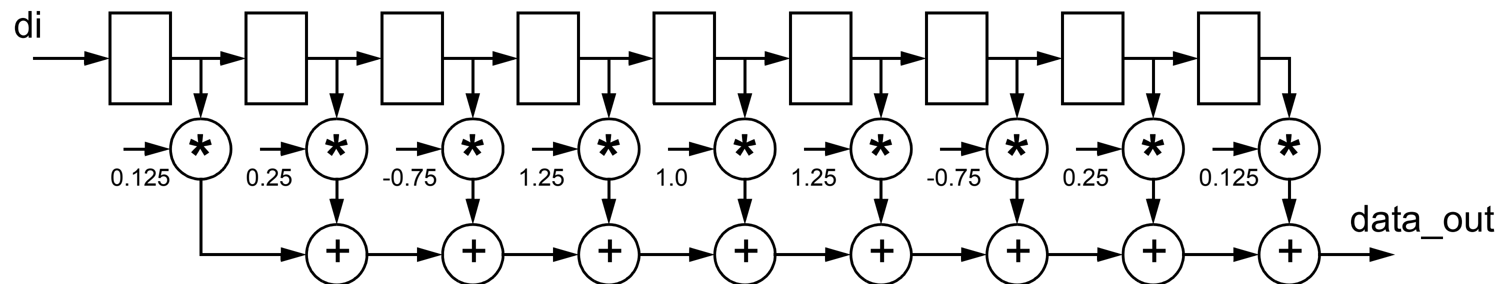
- **9-tap Finite Impulse Response (FIR) filter**





CREDES

FIR implementation example (cont.)



- **Algorithm**
  - **data_out = $0.125*di^{-0} + 0.25*di^{-1} - 0.75*di^{-2} + 1.25*di^{-3} + 1.0*di^{-4} + 1.25*di^{-5} - 0.75*di^{-6} + 0.25*di^{-7} + 0.125*di^{-8}$**
    - **9 multiplications, 8 additions/subtractions**
  - **data_out = $0.125*(di^{-0}+di^{-8}) + 0.25*(di^{-1}+di^{-7}) - 0.75*(di^{-2}+di^{-6}) + 1.25*(di^{-3}+di^{-5}) + 1.0*di^{-4}$**
    - **4 multiplications, 8 additions/subtractions**

TTÜ 1918

FIR implementation example (cont.)

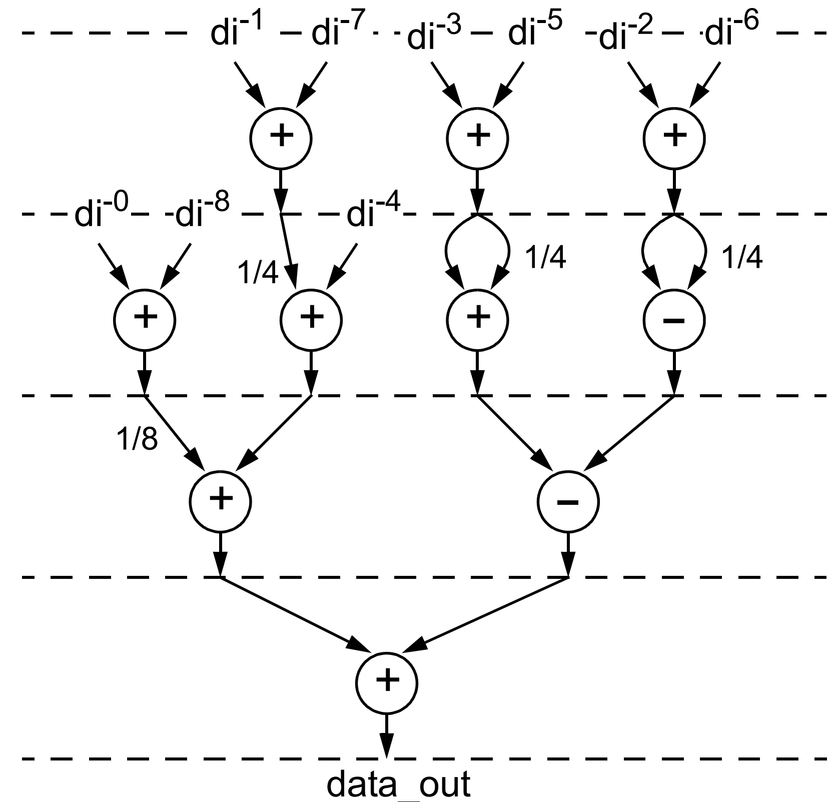
- **Multiplication is too expensive!**
  - data_out = 0.125*(di^{-0}+di^{-8}) + 0.25*(di^{-1}+di^{-7}) - 0.75*(di^{-2}+di^{-6}) + 1.25*(di^{-3}+di^{-5}) + 1.0*di^{-4}
    - 4 multiplications, 8 additions/subtractions
- **Use shift-add trees**
  - 0.125 == 1>>3 0.25 == 1>>2 0.75 == 1 - 1>>2 1.25 == 1 + 1>>2
  - data_out = ((di^{-0}+di^{-8}) >>3) + ((di^{-1}+di^{-7}) >>2) - ((di^{-2}+di^{-6}) - ((di^{-2}+di^{-6}) >> 2)) + + ((di^{-3}+di^{-5}) + ((di^{-3}+di^{-5}) >> 2)) + di^{-4}
    - 12 additions/subtractions; 10 after common sub-expression elimination
- **Time constrained scheduling: 10 operations in 4 clock steps**
- **At least three functional units needed**
  - $\lceil 10 / 4 \rceil = 3$

FIR scheduling example #1

- **Algebraic transformations**
  - **addition is commutative**
    - **a+b == b+a**
  - **double “inversion”**
    - **(a+b)-(c+d) == (a-d)-(c-b) == (a-c)-(d-b)**
    - **10 operations & 9 variables**

	additions	subtractions
1	$v1=di^{-1}+di^{-7}$; $v2=di^{-2}+di^{-6}$; $v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}$; $v5=di^{-4}+v1/4$; $v6=v3+v3/4$	v7=v2-v2/4
3	v8=v4/8+v5	v9=v6-v7
4	data_out=v8+v9	

- **(4 functional units & 4 registers)**

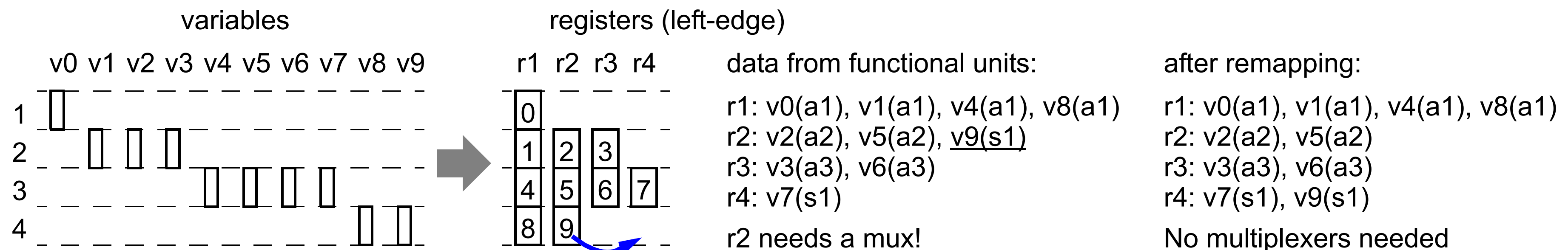


FIR binding example #1

- **4 steps, 10 operations, 9 variables**
- **Assumptions – sample in (di^{-0}) & result out (v0) at step 1; di^{-n} are shifted at step 4**

	additions	**subtraction**
1	$v1=di^{-1}+di^{-7}$; $v2=di^{-2}+di^{-6}$; $v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}$; $v5=di^{-4}+v1/4$; $v6=v3+v3/4$	v7=v2-v2/4
3	v8=v4/8+v5	v9=v6-v7
4	data_out=v8+v9	

	add #1	**add #2**	**add #3**	**sub #1**
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	$v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}$	$v5=di^{-4}+v1/4$	v6=v3+v3/4	v7=v2-v2/4
3	v8=v4/8+v5			v9=v6-v7
4	v0=v8+v9			



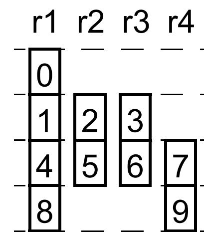
© Peeter Ellervee

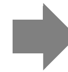
CREDES

FIR binding example #1 (cont.)

- **Storing data from functional units into registers**

data from functional units:
r1: v0(a1), v1(a1), v4(a1), v8(a1)
r2: v2(a2), v5(a2)
r3: v3(a3), v6(a3)
r4: v7(s1), v9(s1)





	r1	r2	r3	r4
1	a1	a2	a3	--
2	a1	a2	a3	s1
3	a1	--	--	s1
4	a1	--	--	--

a1 - writes new value
[ ] - keeps previous value
-- - don't care

- **Multiplexers at functional units' inputs (plus shifters '>')**

	add #1	add #2	add #3	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	$v3=di^{-3}+di^{-5}$	
2	$v4=di^{-0}+di^{-8}$	$v5=di^{-4}+v1/4$	v6=v3+v3/4	v7=v2-v2/4
3	v8=v4/8+v5			v9=v6-v7
4	v0=v8+v9			

	a1 L	a1 R	a2 L	a2 R	a3 L	a3 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-2}	di^{-6}	di^{-3}	di^{-5}	--	--
2	di^{-0}	di^{-8}	di^{-4}	r1>	r3	r3>	r2	r2>
3	r1>	r2	--	--	--	--	r3	r4
4	r1	r4	--	--	--	--	--	--

- **Components: 3 add, 1 sub, 4 reg, 2 4-mux, 6 2-mux**
  - **3*125+139+4*112+(2*3+6)*48 = 1538 (+controller)**

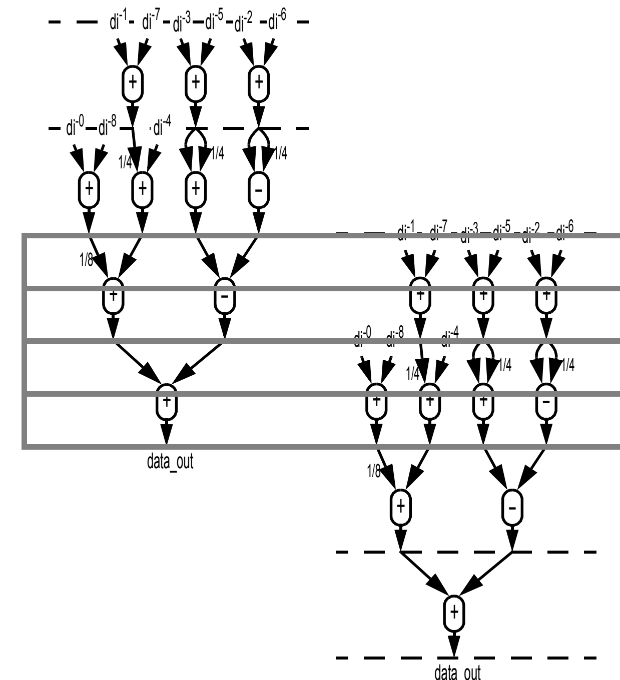
© Peeter Ellervee

CREDES

TTÜ1918

FIR scheduling example #2

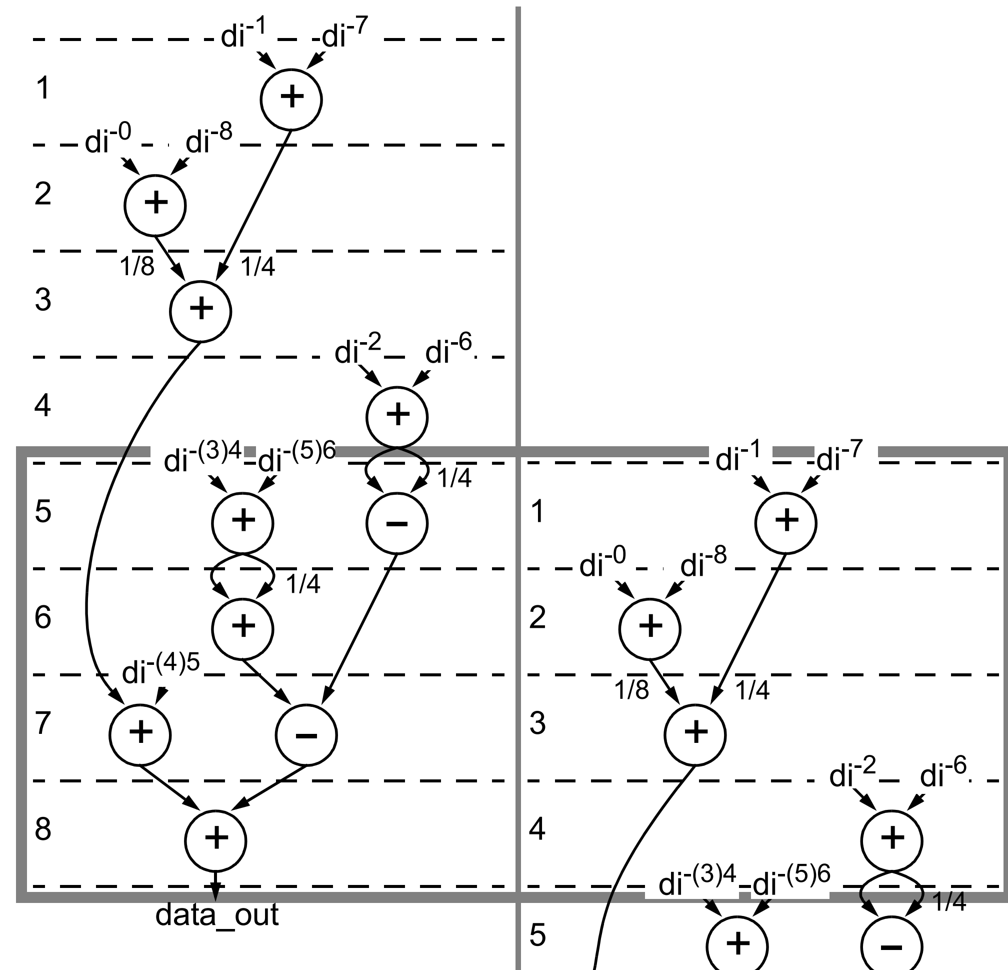
- 10 operations & 4 functional units
  - at least <u>three</u> functional units – ⌈ 10 / 4 ⌉ = 3
- 10 operations & <u>3</u> functional units?
  - <u>7</u> operations should be executed during the first two clock steps
- Solution – pipeline
  - output data can be delayed
  - two samples processed simultaneously
  - 8 clock steps per sample
  - 10+10 operations over 4+4 clock steps



FIR scheduling example #2 (cont.)

- **Introducing <u>pipelining</u> – additional delay at the output**
- **Distribution of operations must be analyzed at both stages**
- **10 operations & 9 variables**

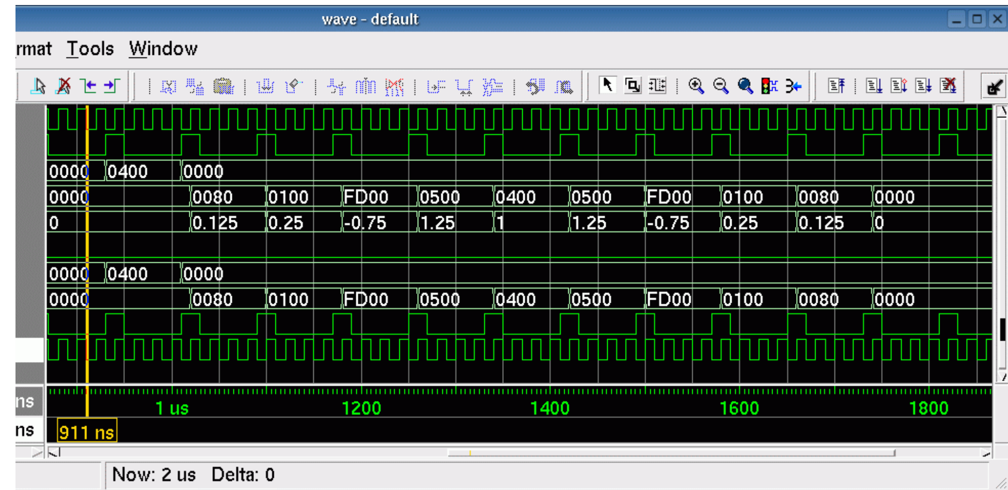
	additions	**subtractions**
1	$v1=di^{-1}+di^{-7}$	
(5)	$v5=di^{-4}+di^{-6}$	$v6=v4-(v4/4)$
2	$v2=di^{-0}+di^{-8}$	
(6)	$v7=v5+(v5/4)$	
3	$v3=(v2/8)+(v1/4)$	
(7)	$v8=v3+di^{-5}$	$v9=v7-v6$
4	$v4=di^{-2}+di^{-6}$	
(8)	data_out=v8+v9	

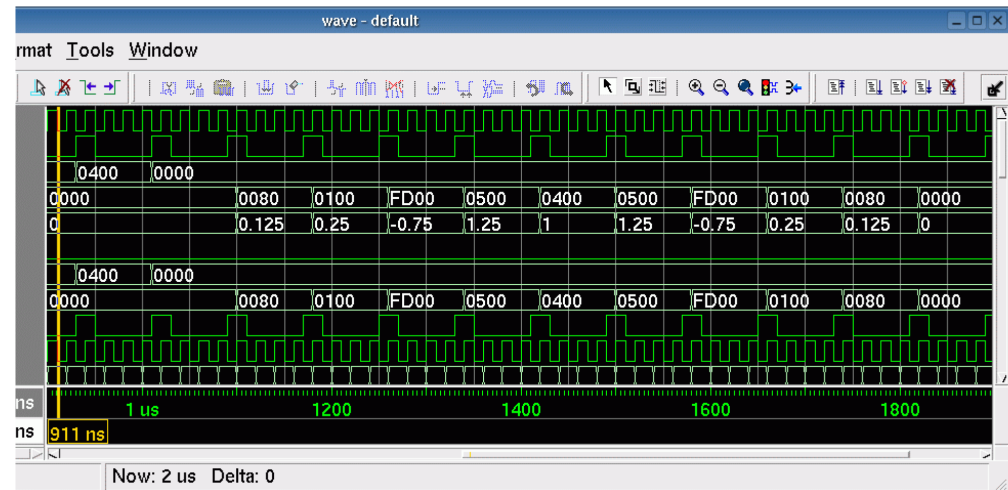


FIR scheduling example #2 (cont.)

Result delayed for one sample cycle

	sample #1	sample #2
1	$v1=di^{-1}+di^{-7}$	
2	$v2=di^{-0}+di^{-8}$	
3	v3=v2/8+v1/4	
4	$v4=di^{-2}+di^{-6}$	
5	$v5=di^{-3}+di^{-5}$ v6=v4-v4/4	$v1=di^{-1}+di^{-7}$
6	v7=v5+v5/4	$v2=di^{-0}+di^{-8}$
7	$v8=v3+di^{-4}$ v9=v7-v6	v3=v2/8+v1/4
8	data_out=v8+v9	$v4=di^{-2}+di^{-6}$
9		$v5=di^{-3}+di^{-5}$ v6=v4-v4/4
10		v7=v5+v5/4
11		$v8=v3+di^{-4}$ v9=v7-v6
12		data_out=v8+v9



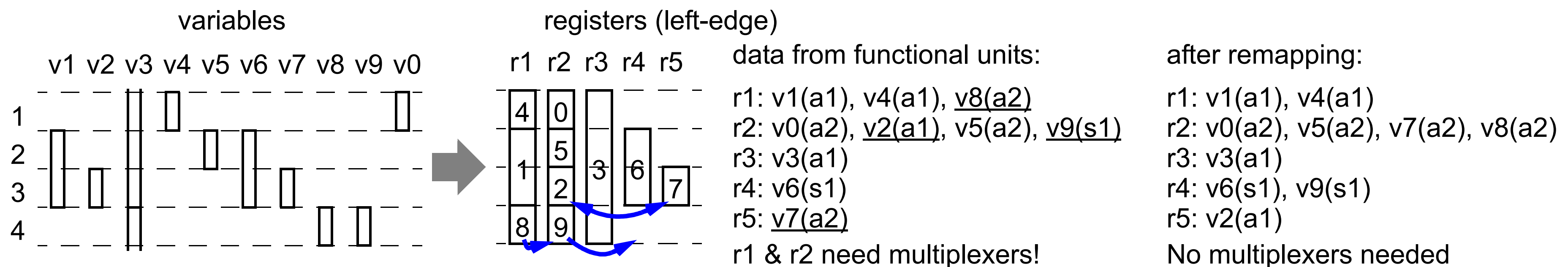


FIR binding example #2

- **4+4 steps, 10 operations, 9 variables**
- **Assumptions – sample in (di^{-0}) & result out (v0) at step 1; di^{-n} are shifted at step 4**

	additions	**subtraction**
1 (5)	$v1=di^{-1}+di^{-7}$; $v5=di^{-4}+di^{-6}$	v6=v4-(v4/4)
2 (6)	$v2=di^{-0}+di^{-8}$; v7=v5+(v5/4)	
3 (7)	v3=(v2/8)+(v1/4); $v8=v3+di^{-5}$	v9=v7-v6
4 (8)	$v4=di^{-2}+di^{-6}$; data_out=v8+v9	

	add #1	**add #2**	**sub #1**
1	$v1=di^{-1}+di^{-7}$	$v5=di^{-4}+di^{-6}$	v6=v4-(v4/4)
2	$v2=di^{-0}+di^{-8}$	v7=v5+(v5/4)	
3	v3=(v2/8)+(v1/4)	$v8=v3+di^{-5}$	v9=v7-v6
4	$v4=di^{-2}+di^{-6}$	v0=v8+v9	



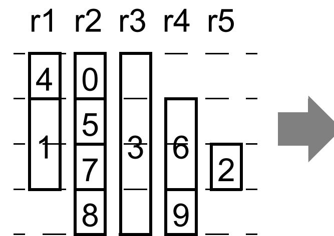
© Peeter Ellervee

FIR binding example #2 (cont.)

- **Storing data from functional units into registers**

data from functional units:

r1: v1(a1), v4(a1)
r2: v0(a2), v5(a2), v7(a2), v8(a2)
r3: v3(a1)
r4: v6(s1), v9(s1)
r5: v2(a1)



	r1	r2	r3	r4	r5
1	a1	a2	[ ]	s1	--
2	[ ]	a2	[ ]	[ ]	a1
3	--	a2	a1	s1	--
4	a1	a2	[ ]	--	--

a1 - writes new value
[ ] - keeps previous value
-- - don't care

- **Multiplexers at functional units' inputs (plus shifters '>')**

	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v5=di^{-4}+di^{-6}$	v6=v4-(v4/4)
2	$v2=di^{-0}+di^{-8}$	v7=v5+(v5/4)	
3	v3=(v2/8)+(v1/4)	$v8=v3+di^{-5}$	v9=v7-v6
4	$v4=di^{-2}+di^{-6}$	v0=v8+v9	

	a1 L	a1 R	a2 L	a2 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-4}	di^{-6}	r1	r1>
2	di^{-0}	di^{-8}	r2>	r2	--	--
3	r5>	r1>	r3	di^{-5}	r2	r4
4	di^{-2}	di^{-6}	r5	r4	--	--

- **Components: 2 add, 1 sub, 5 reg, 4 4-mux, 2 2-mux**
  - **2*125+139+5*112+(4*3+2)*48 = 1621 (+controller) – less FU-s (-1) but more reg-s (+1) & mux-s (+2)**

© Peeter Ellervee

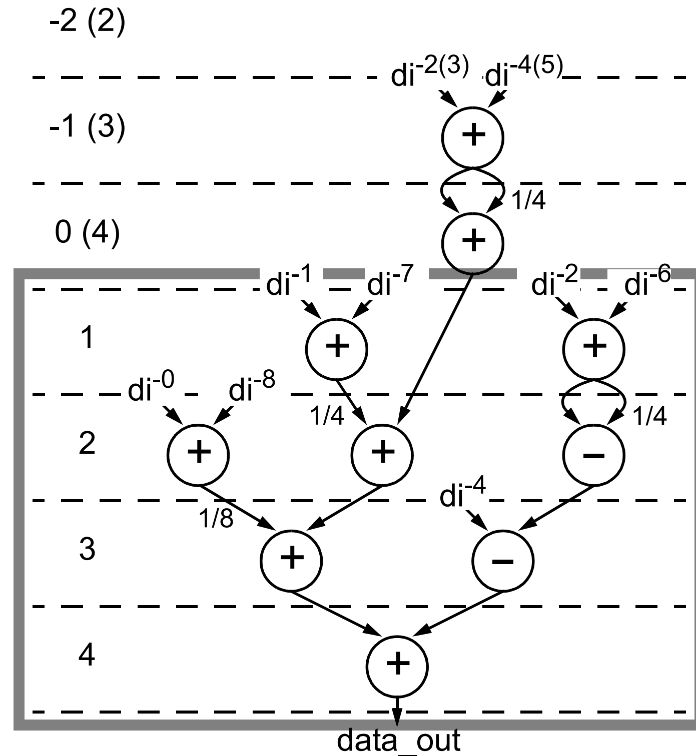
TTÜ1918

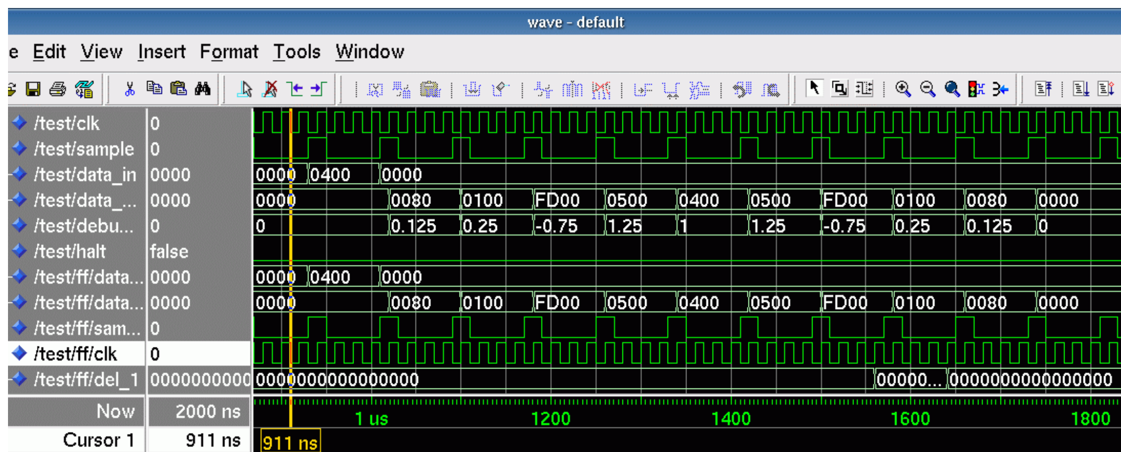
FIR scheduling example #3

- **Out-of-order execution (functional pipelining)**
- **Earlier samples are available!**

	additions	**subtractions**
1	$v1=di^{-1}+di^{-7}$; $v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$; $v4=(v1/4)+v9$	$v5=v2-(v2/4)$
3	$v6=(v3/8)+v4$; [$v8=di^{-2}+di^{-4}$]	$v7=di^{-4}-v5$
4	data_out=v6+v7; [$v9=v8+(v8/4)$]	

- **(3 functional units & 3 registers!)**





TTÜ1918

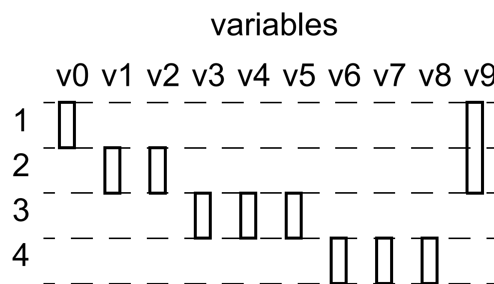
FIR binding example #3

- **4 steps, 10 operations, 9 variables, out-of-order execution**
- **Assumptions – sample in (di^{-0}) & result out (v0) at step 1; di^{-n} are shifted at step 4**

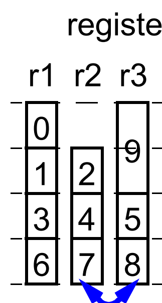
	additions	**subtraction**
1	$v1=di^{-1}+di^{-7}$; $v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$; v4=(v1/4)+v9	v5=v2-(v2/4)
3 (-1)	v6=(v3/8)+v4; [$v8=di^{-2}+di^{-4}$]	$v7=di^{-4}$-v5
4 (0)	data_out=v6+v7; [v9=v8+(v8/4)]	

	add #1	**add #2**	**sub #1**
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$	v4=(v1/4)+v9	v5=v2-(v2/4)
3	v6=(v3/8)+v4	$v8=di^{-2}+di^{-4}$	$v7=di^{-4}$-v5
4	v0=v6+v7	v9=v8+(v8/4)	

variables



registers (left-edge)



data from functional units:

r1: v0(a1), v1(a1), v3(a1), v6(a1)
r2: v2(a2), v4(a2), v7(s1)
r3: v5(s1), v8(a2), v9(a2)

r2 & r3 need multiplexers!

after remapping:

r1: v0(a1), v1(a1), v3(a1), v6(a1)
r2: v2(a2), v4(a2), v8(a2)
r3: v5(s1), v7(s1), v9(a2)

Only r3 needs a multiplexer

© Peeter Ellervee

CREDES

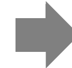
FIR binding example #3 (cont.)

- **Storing data from functional units into registers**

data from functional units:

r1: v0(a1), v1(a1), v3(a1), v6(a1)
r2: v2(a2), v4(a2), v8(a2)
r3: v5(s1), v7(s1), v9(a2)

r1	r2	r3
0		9
1	2	9
3	4	5
6	8	7



	r1	r2	r3
1	a1	a2	[ ]
2	a1	a2	s1
3	a1	a2	s1
4	a1	--	a2

a1 - writes new value
[ ] - keeps previous value
-- - don't care

- **Multiplexers at functional units' inputs (plus shifters '>')**

	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$	v4=(v1/4)+v9	v5=v2-(v2/4)
3	v6=(v3/8)+v4	$v8=di^{-2}+di^{-4}$	$v7=di^{-4}-v5$
4	v0=v6+v7	v9=v8+(v8/4)	

	a1 L	a1 R	a2 L	a2 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-2}	di^{-6}	--	--
2	di^{-0}	di^{-8}	r1>	r3	r2	r2>
3	r1>	r2	di^{-2}	di^{-4}	di^{-4}	r3
4	r1	r3	r2	r2>	--	--

- **Components: 2 add, 1 sub, 3 reg, 3 4-mux, 1 3-mux, 3 2-mux**
  - **2*125+139+3*112+(3*3+5)*48 = 1397 (+controller) – less FU-s (-1) & reg-s (-1) / more mux-s (+2)**

© Peeter Ellervee

FIR binding example #3.1

- **Multiplexers at functional units' inputs (plus shifters '>', ver. #3)**

	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$	v4=(v1/4)+v9	v5=v2-(v2/4)
3	v6=(v3/8)+v4	$v8=di^{-2}+di^{-4}$	$v7=di^{-4}-v5$
4	v0=v6+v7	v9=v8+(v8/4)	

	a1 L	a1 R	a2 L	a2 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-2}	di^{-6}	--	--
2	di^{-0}	di^{-8}	r1>	***r3***	r2	r2>
3	r1>	***r2***	di^{-2}	di^{-4}	di^{-4}	r3
4	r1	***r3***	***r2***	r2>	--	--

- **Swapping add operations on the last clock step**

	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$	v4=(v1/4)+v9	v5=v2-(v2/4)
3	v6=(v3/8)+v4	$v8=di^{-2}+di^{-4}$	$v7=di^{-4}-v5$
4	v9=v8+(v8/4)	v0=v6+v7	

	a1 L	a1 R	a2 L	a2 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-2}	di^{-6}	--	--
2	di^{-0}	di^{-8}	r1>	r3	r2	r2>
3	r1>	r2	di^{-2}	di^{-4}	di^{-4}	r3
4	r2>	r2	r1	r3	--	--

- **Components: 2 add, 1 sub, 3 reg, 1 4-mux, 3 3-mux, 3 2-mux**
  - **2*125+139+3*112+(3+6+3)*48 = 1301 (+controller) – less FU-s (-1) & reg-s (-1)**

Problems with classical HLS

- Huge number of possible data part architectures makes it very hard to automate HLS
- Dividing HLS into sub-tasks reduces optimality of the result
- Different application types require different approaches
  - data dominated vs. control dominated
- Parallelism is not fully explored
  - out of basic block analysis
- Not taken into account – power/energy consumption, testability, reliability, layout, etc.

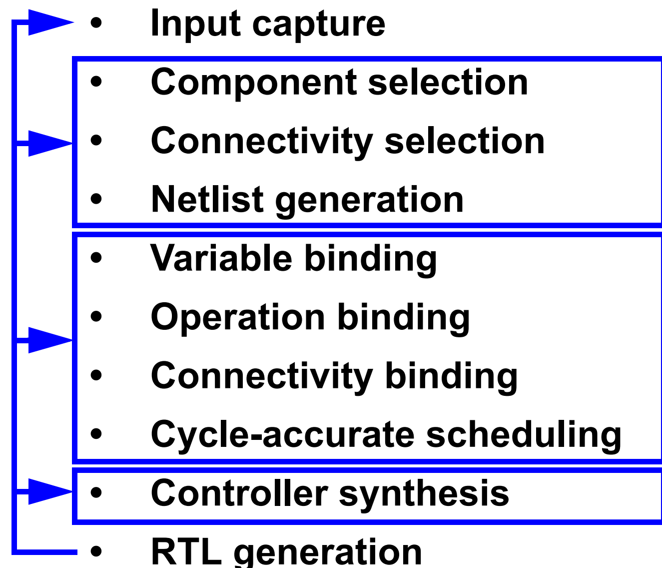
Creating synthesizable code

- Behavioral level code is not synthesizable (so far)
- Register-transfer level code is synthesizable
- What about “Behavioral RTL” or the other intermediate levels?
- Step-by-step code refinement
  - from idea to model – validating model’s behavior by simulation
  - from model to structure – transforming behavioral level code into (pure) RTL code
  - from structure to schematics (==synthesis)

Improving HLS

- Example – NISC technology
  - No Instruction Set Computer
  - Standard processor – custom SW on fixed HW
  - High-level synthesis – fixed FSM on fixed datapath
  - Custom IP (NISC) – custom SW on custom HW

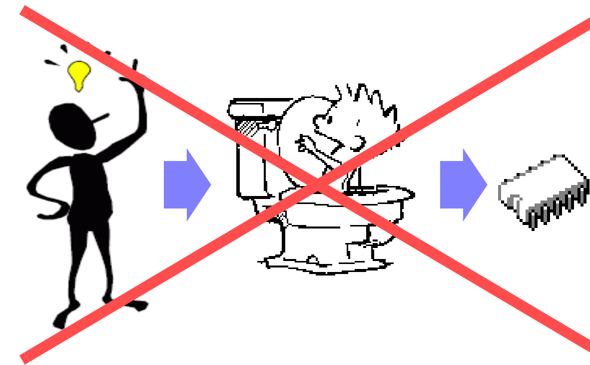
- Iterative synthesis



- Processor synthesis
  - Input capture
  - DP architecture selection
  - Cycle-accurate scheduling
  - Variable binding
  - Operation binding
  - Bus binding
  - Controller synthesis
  - RTL generation

But there are more...

- Efficient implementation is not only the hardware optimization



- Efficient implementation is
  - selecting the right algorithm
  - selecting the right data types
  - selecting the right architecture
  - making the right modifications
  - and optimizing...

  - right == the most suitable

- Think first, implement later!