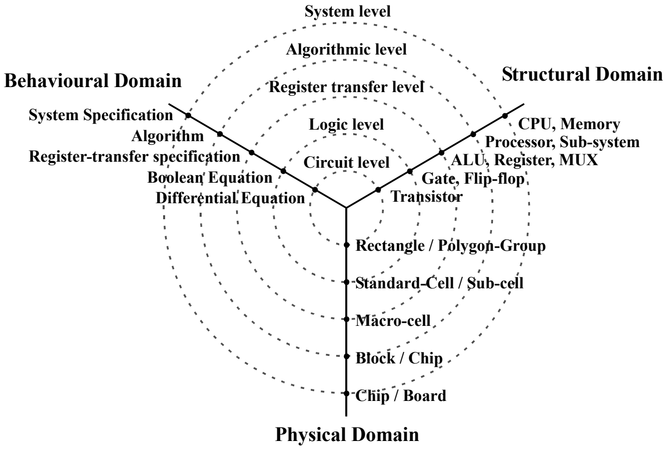
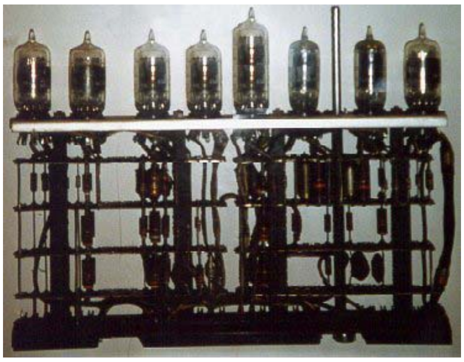
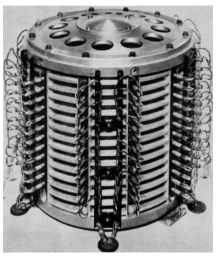
Synthesis at different abstraction levels

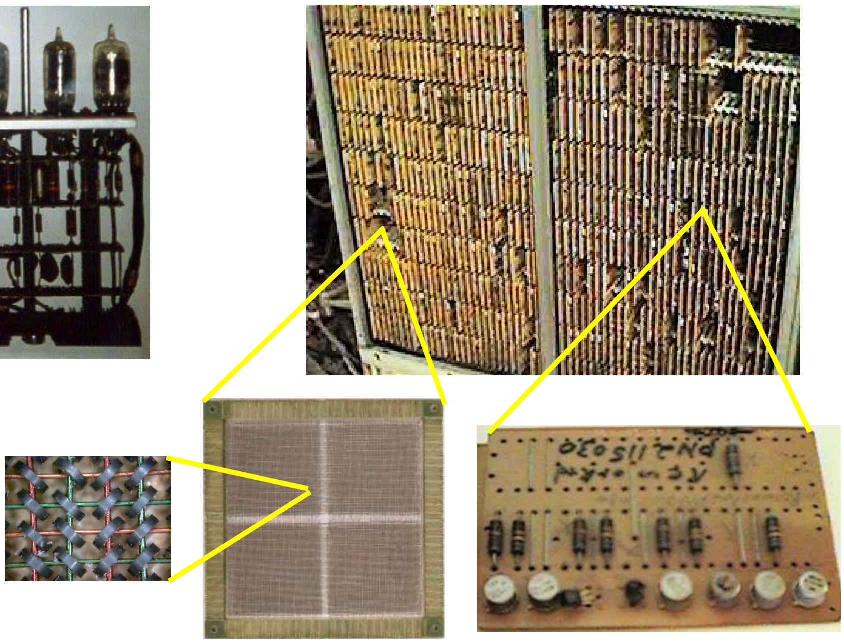
- System Level Synthesis
  - Clustering.
    Communication synthesis.
- High-Level Synthesis
  - Resource or
    time constrained scheduling
  - Resource allocation. Binding
- Register-Transfer Level Synthesis
  - Data-path synthesis.
    Controller synthesis
- Logic Level Synthesis
  - Logic minimization.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



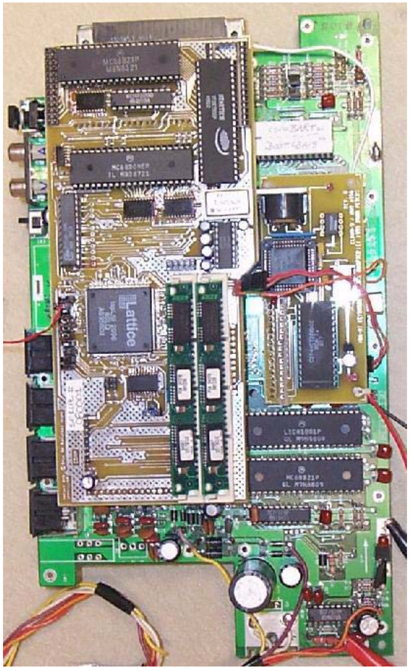
Physical implementation – history

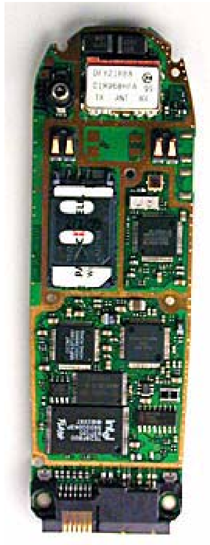






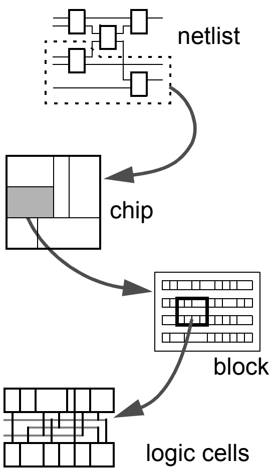
Physical implementation – history

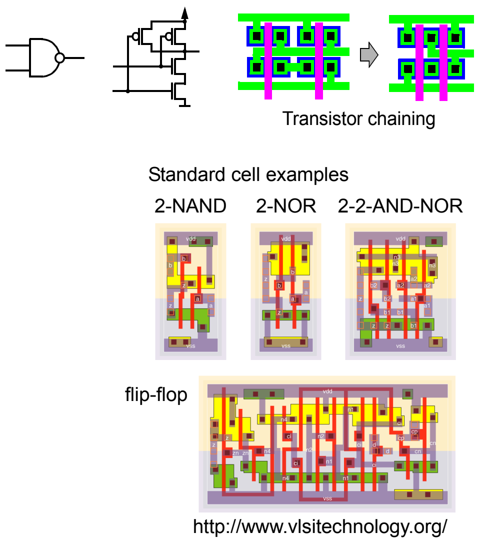




Chip design and fabrication

- **Partitioning**
- **Floorplanning**
  - **initial placement**
- **Placement**
  - **fixed modules**
- **Global routing**
- **Detailed routing**
- **Layout optimization**
- **Layout verification**

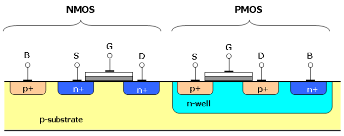




- **Fabrication – http://jas.eng.buffalo.edu/ [e.g., 7.2]**

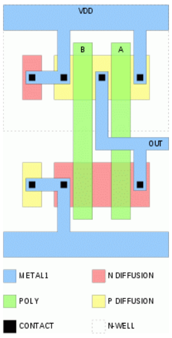
CMOS chip fabrication

CMOS transistors

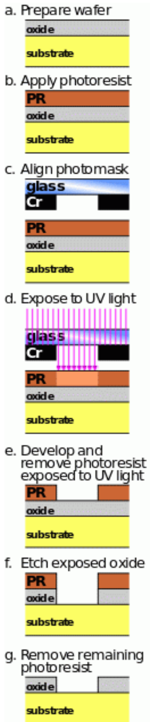


n - electrons [P, As, Sb]
p - holes [B, Al]

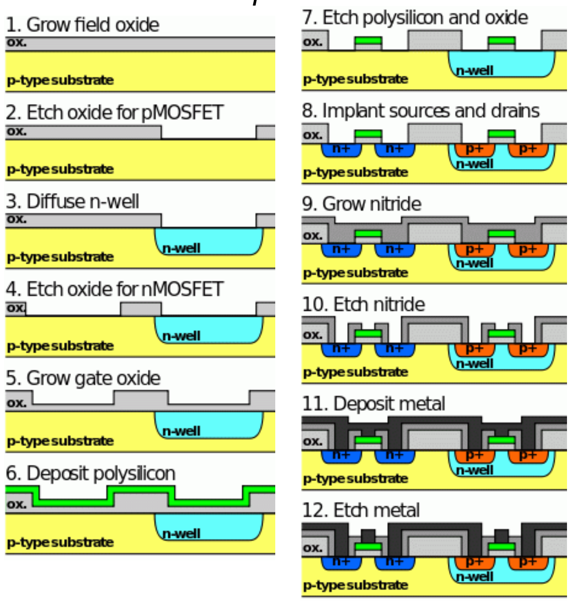
2-NAND layout



etching steps



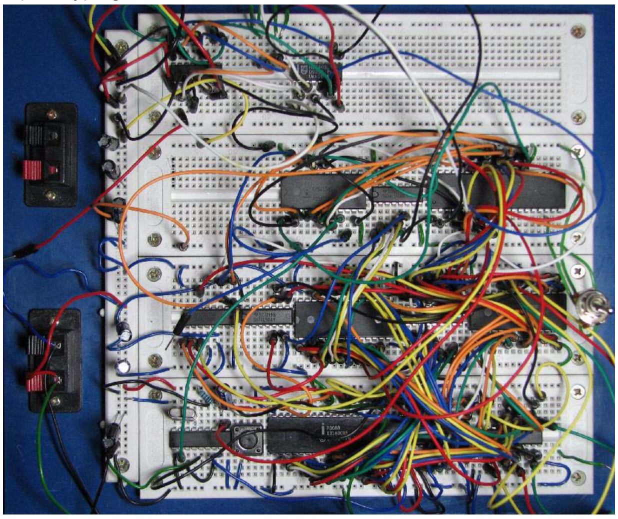
fabrication steps



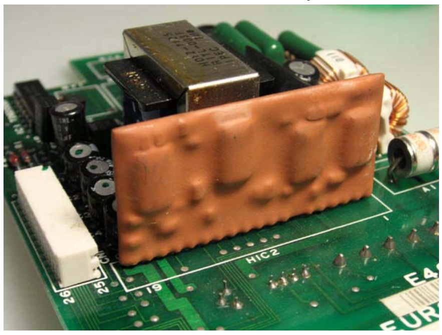
www.wikipedia.org

Packaging examples

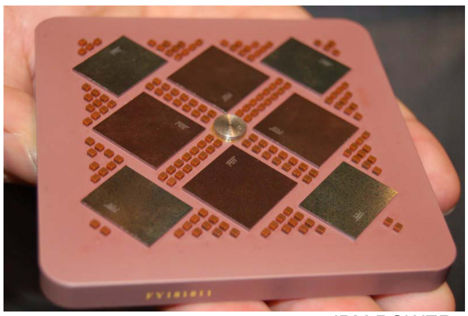
prototyping



hybrid circuit



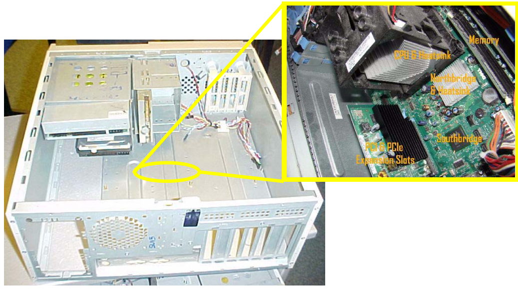
3D assembly

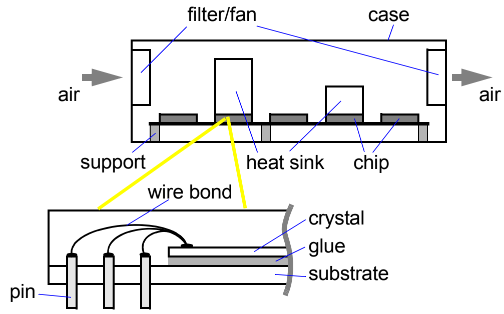


IBM POWER5

Packaging issues

- **Mechanical requirements and constraints**
  - **size, interfaces**
  - **durability – dust, vibration**
- **Thermal requirements and constraints**
  - **work temperature range**
  - **cooling / heating**
- **Electrical requirements and constraints**
  - **power supply**
  - **protection – voltage, electromagnetic fields**
- **Ergonomic requirements and constraints**
  - **appearance, user interface, noise**





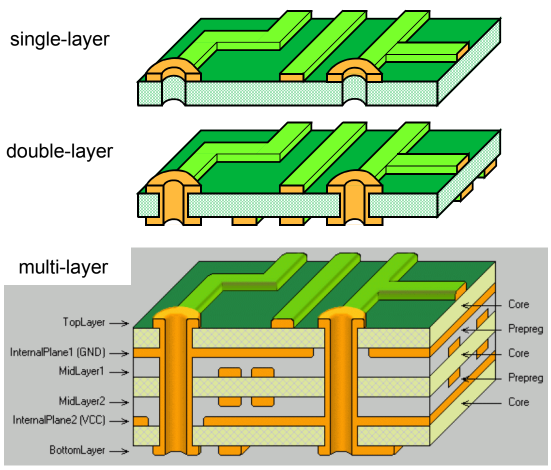
Department of Computer Engineering

Printed Circuit Board (PCB)

- **Components**
  - **chips, transistors,
resistors, capacitors, etc.**
- **Connections / interfaces / mounting**

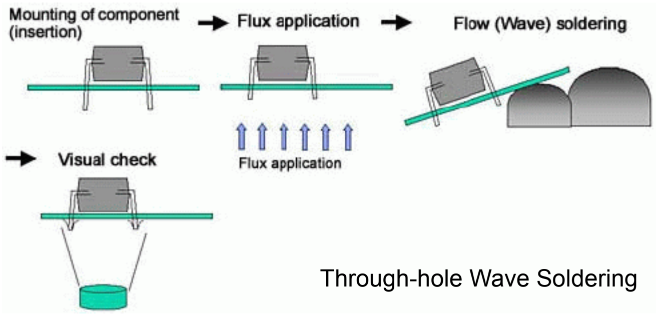
- **PCB manufacturing**
- **Component placement (and fixing)**
- **Electrical connections (e.g. soldering)**

- **Single-layer PCB**
  - **wires (bottom side)**
- **Double-layer PCB**
  - **wires + metallized vias**
- **Multi-layer PCB**
  - **multiple double-layer PCB-s**
  - **location of vias!**

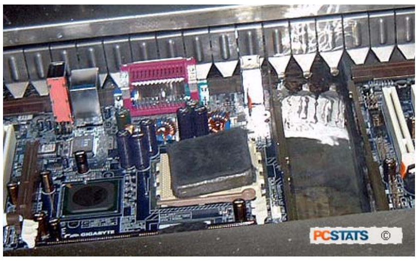


PCB manufacturing

- **Manufacturing**
  - **component mounting**
  - **soldering**
    - **solder paste / tin**
  - **thermal problems**
    - **large copper surfaces**
    - **component over-heating**
  - **quality check**
    - **visual inspection**
  - **final finish**
    - **cleaning**
    - **protective lacquering**
  - **final test**
    - **functional test**



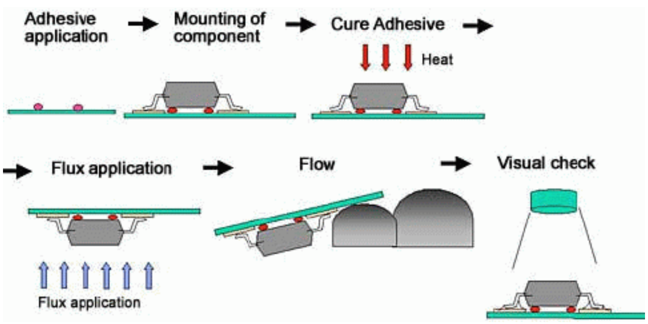
Through-hole Wave Soldering



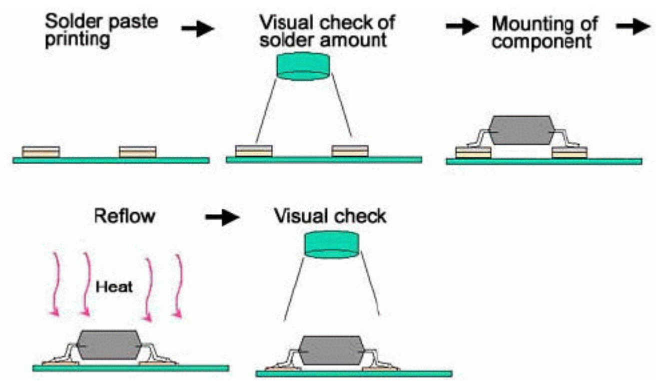
Gigabyte Factory Tour
PCSTATS.com

PCB manufacturing

SMD Wave Soldering



SMD Reflow Soldering



Logic synthesis

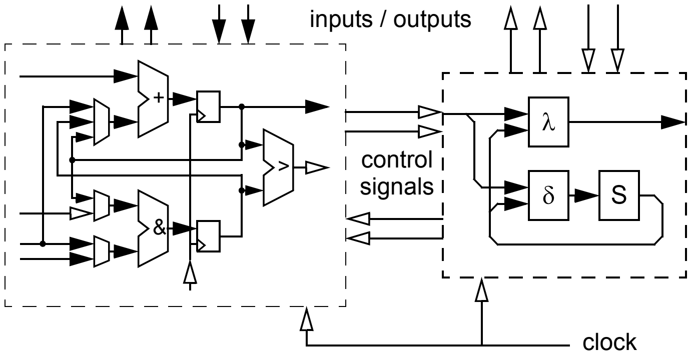
- Transforming logic functions (Boolean functions) into a set of logic gates
  - transformations at logic level from behavioral to structural domain
- Optimizations / Transformations
  - area
  - delay
  - power consumption
- Implementation of Finite State Machines (FSM)
  - state encoding
  - generating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

Logic synthesis – main tasks

- Optimization of 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
  - state minimization, encoding
- Synthesis of multi-level sequential nets (circuits)
  - optimizations for area, delay, power consumption, and/or testability
- Library mapping
  - optimal gate selection

Register-transfer level synthesis

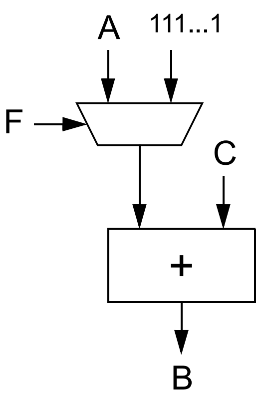
Digital system @RTL = data-path + controller



- *T*ransformation from RT-level structural description to logic level description
- Data-path – storage units (registers, register files, memories) and combinational units (ALU-s, multipliers, shifters, comparators etc.), connected by buses
  - Data path synthesis – maximizing the clock frequency, retiming, operator selection
- Controller – Finite State Machine (FSM) – state register and two combinational units (next state and output functions)
  - Controller synthesis – architecture selection, FSM optimizations, state encoding, decomposition

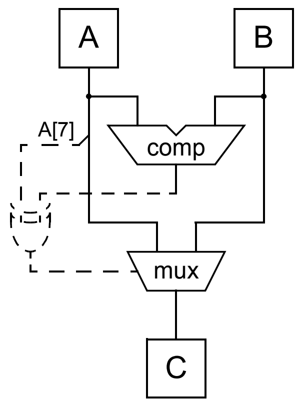
Data-path optimization

```
if F=0 then B := A + C
        else B := C - 1;
```



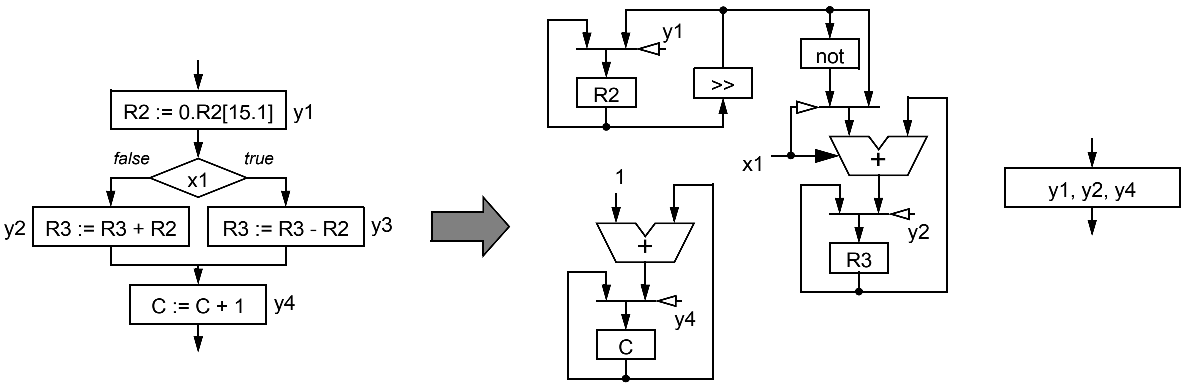

```
read_port(A);
read_port(B);
if A(7)='0' then
  if A>B then
    C := A;
  else
    C := B;
  end if;
else
  if A>B then
    C := B;
  else
    C := A;
  end if;
end if;
```

data flow only
(w/o controller)



Data-path optimization

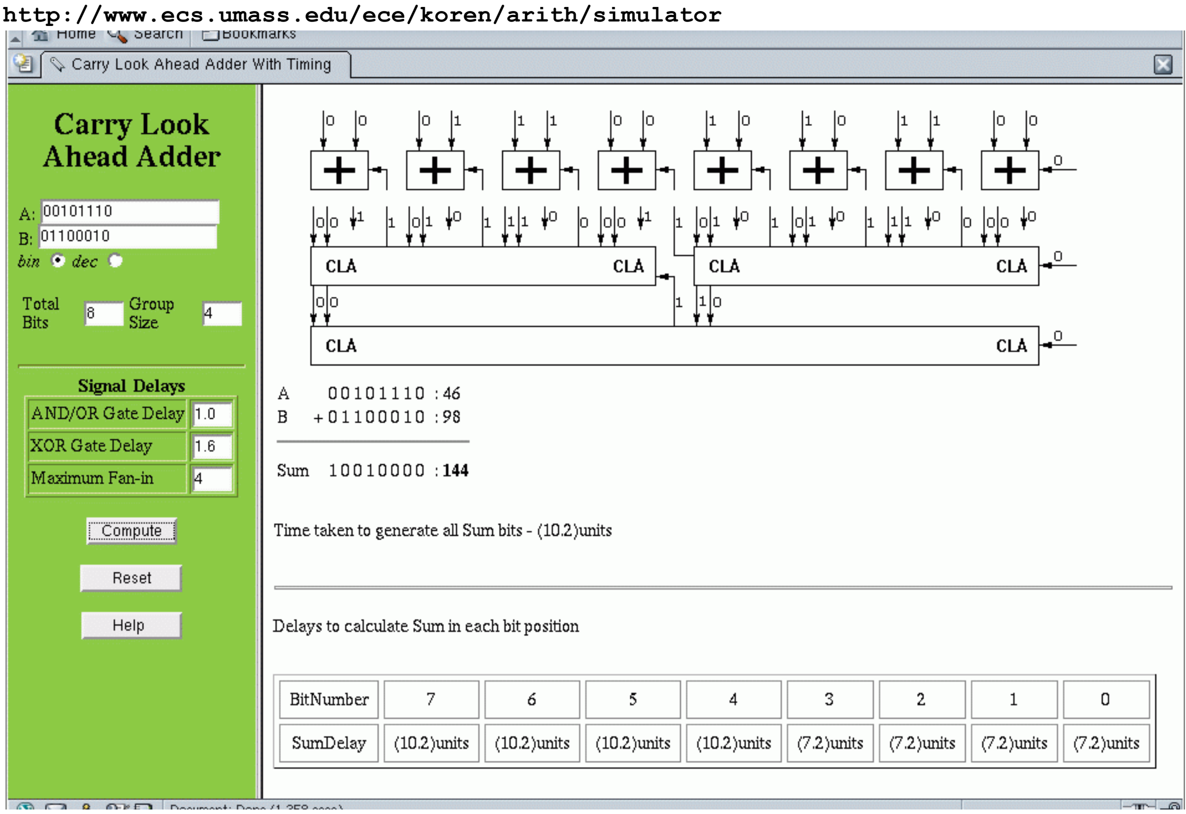
- **Hardware cost of an operation**
  - **shifting == rerouting wires**
  - **the same functional unit for addition and subtraction – adder (+carry)**
- **Independent operations can be executed simultaneously**
  - **analysis of real data dependencies, e.g., result of shifting R2**



Arithmetic unit architecture selection

- **Parallel versus sequential execution**
- **Adder/subtracter architectures**
  - **ripple-carry – sequence of full-adders, small but slow**
  - **carry-look-ahead – separate calculation of carry generation and/or propagation**
    - **generation: $g_i=a_i \cdot b_i$; propagation: $p_i=a_i+b_i$; carry: $c_i=g_i+p_i \cdot c_{i-1}$**
  - **carry-select adders – duplicated hardware plus selectors**
    - **speculative calculation one case with carry and another without, the answer will be selected when the actual carry has arrived**
- **Multiplier architectures**
  - **sequential algorithms – register + adder, 1/2/... bit(s) at a time**
  - **“parallel” algorithms – array multipliers – AND gates + full-adders**
- **Multiplication/division with constant**
  - **shift+add – 5*n = 4*n + n = (n<<2) + n**

http://www.ecs.umass.edu/ece/koren/arith/simulator



Controller synthesis

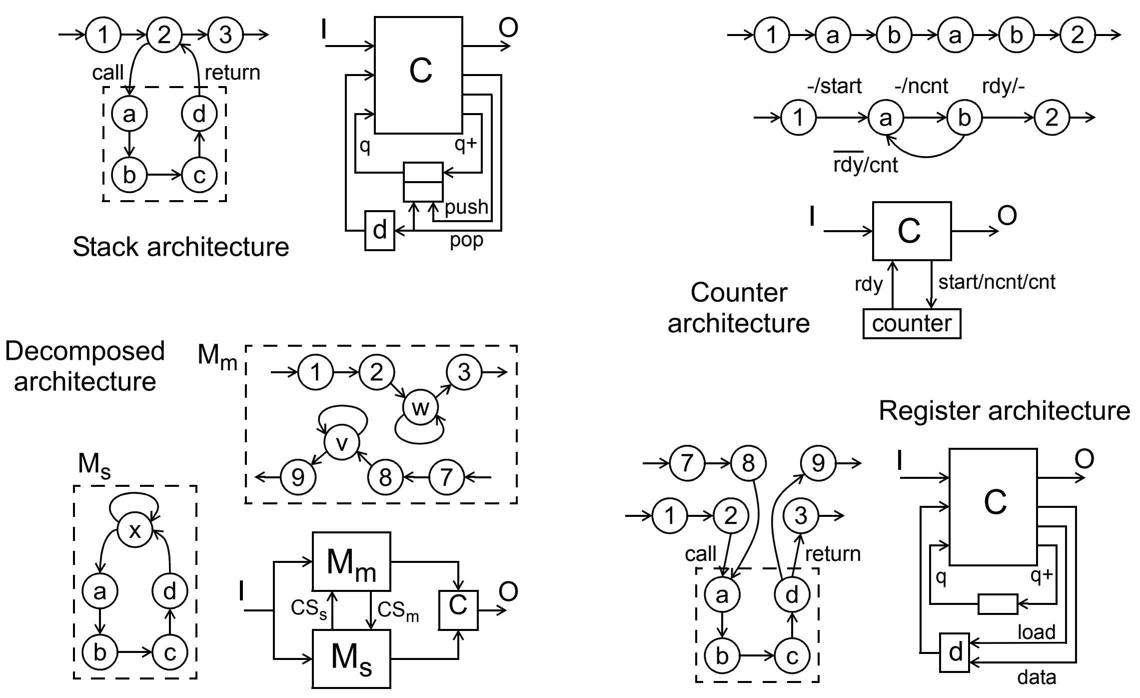
- **Controller == Finite State Machine (FSM)**
- **Controller synthesis is also a task at the algorithmic level. Controller is the implementation of the operation scheduling task in hardware.**
- ***The canonical implementation* of a sequential system is based directly on its state description. It consists of *state register*, and a *combinational network* to implement the transition and output functions.**
- **Sub-tasks**
  - **generation of the state graph**
  - **selecting the proper controller architecture**
  - **finite state machine optimization for area, performance, testability, etc.**

FSM encoding

- **Input/output encoding – symbols –> binary code**
- **State encoding – states –> binary code**
- ***z* – number of symbols/states –> minimum code length is $t=ceil(log_2 z)$**
- **Two border cases – minimal code length encoding & one-hot-encoding**
- **General encoding strategy**
  - **Identification of sets of states (adjacent groups) in the state table such that, if encoded with the minimal Hamming distance, lead to a simplification of the corresponding next-state and output equations after logic minimization.**
  - **The groups and their intersections are analyzed with the respect of the degree of potential minimizations during the subsequent logic minimization. Results are reflecting the potential gains in the cost of the final logic.**
  - **Coding constraints and calculated gains control the encoding heuristics which try to satisfy as much constraints as possible.**

TTÜ1918 ▪ Department of Computer Engineering ▪

FSM architectures



High-Level Synthesis

a.k.a. Behavioral Synthesis a.k.a. Algorithm Level Synthesis

a.k.a. Silicon Compilation

- *High-Level Synthesis* (HLS) 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
  - Automatization simplifies handling of larger designs and speeds up exploration of different architectural solutions.
  - The use of synthesis techniques promises correctness-by-construction. This both eliminates human errors and shortens the design time.
  - 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.

Example – differential equation

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

```
  variable a,dx,x,u,y,x1,y1: integer;
begin
  cycles(sysclock,1); a:=inport;
  cycles(sysclock,1); dx:=inport;
  cycles(sysclock,1); y:=inport;
  cycles(sysclock,1); x:=inport;
  cycles(sysclock,1); u:=inport;
  loop
    cycles(sysclock,7);
    x1 := x + dx; y1 := y + (u * dx);
    u := u-5 * x * (u * dx) - 3 * y * dx;
    x := x1; y := y1;
    exit when not (x1 < a);
  end loop;
```

SW compilation

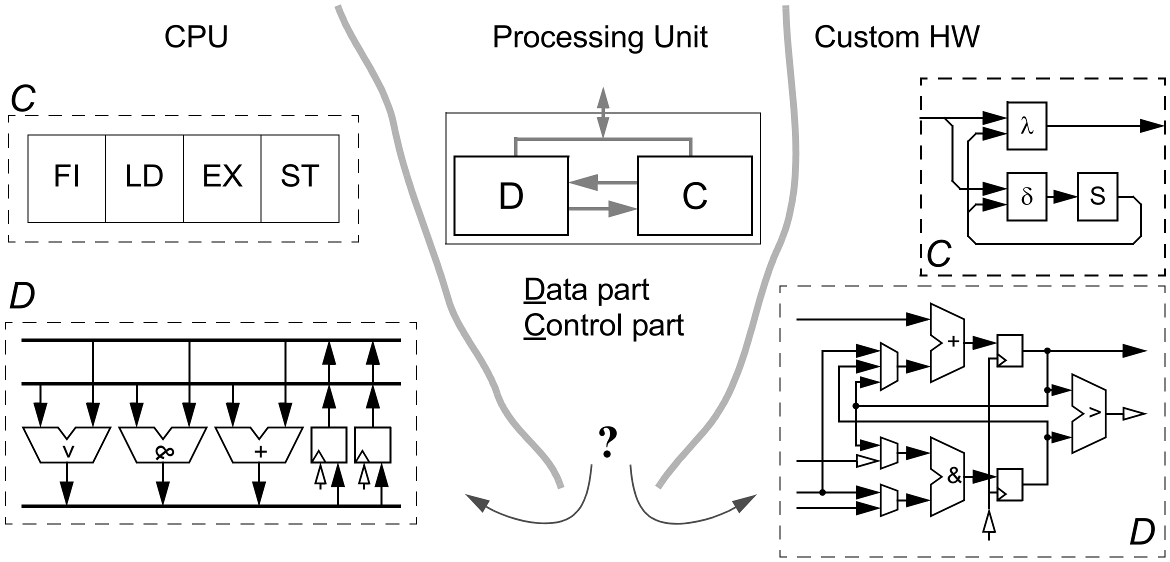
- **Example – differential equation** $$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

```
{
  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
   ...
```


Software vs. hardware

The target architecture



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**

Department of Computer Engineering

The classical High-Level Synthesis tasks

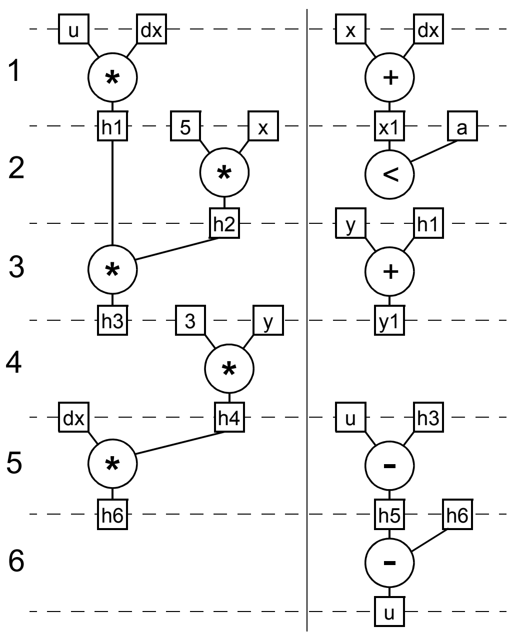
- **Front-end:**
  - **Deriving an internal graph-based representation equivalent to the algorithmic description of both the data flow and the control flow**
  - **Compiler optimizations**
- **Back-end:**
  - **Behavioral transformations (control and/or data graph transformations e.g. associativity, unrolling)**
  - **Transforming data and control flow into register-transfer level structure (so called *essential subtasks*)**
  - **Netlist extraction, state machine table generation**

Essential subtasks

- **Scheduling**
  - **Assignment of operations to time steps subject to certain constraints and minimizing some objective function**
    - **Time is abstracted to the number of needed time steps.**
    - **Depending on whether the time constraint or the area constraint is more difficult to meet, *resource constrained scheduling* or *time constrained scheduling* has to be chosen.**
- **Resource allocation**
  - **Number and types of functional units**
  - **Number and type of storage elements**
  - **Number and type of busses**
- **Resource assignment**
  - **Operations to functional unit instances**
  - **Values to be stored to instances of storage elements**
  - **Data transfers to bus instances**

Department of Computer Engineering

Minimal hardware

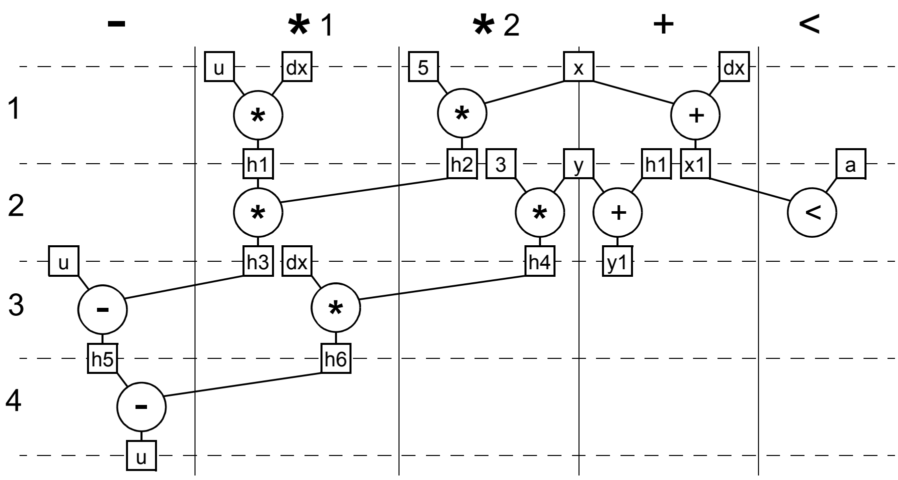


$$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

*	FU1
+/−/<	FU2

Minimal time

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



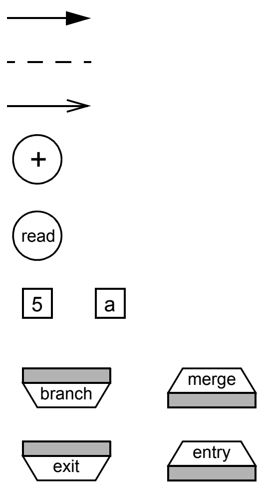
Internal representation of the algorithmic description

- **Control flow model – CFG(V,E)**
  - **nodes – basic blocks**
  - **edges – flow of control**
- **Data flow model – DFG(V,E)**
  - **nodes – actors, representing operations**
  - **edges – links, representing data conveying paths**
  - **DFG specifies a partial order of operations**
- **Control flow and data flow can be combined in a single graph – CDFG (Control and Data Flow Graph)**

Eindhoven Silicon Compiler

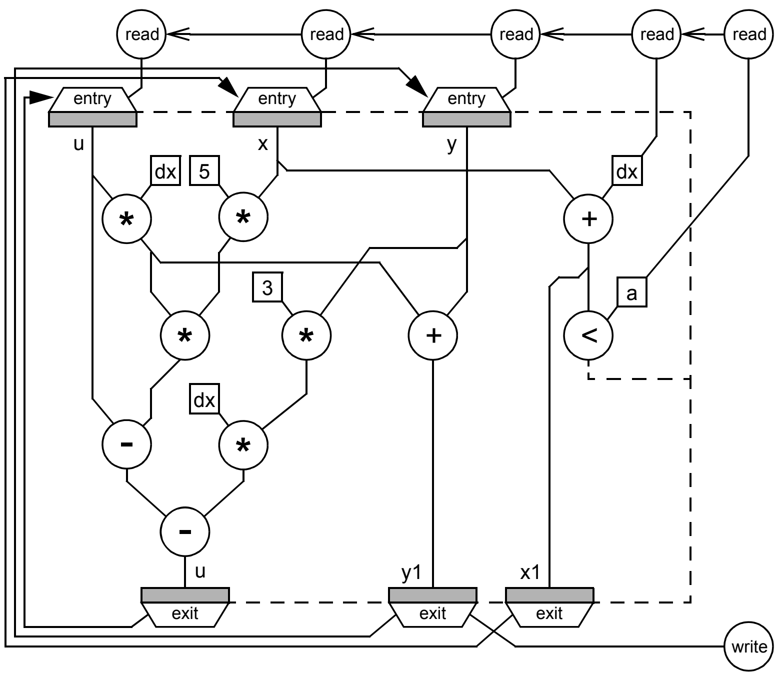
DFG(V_o,V_c,E,E_c,E_s)

- **Data flow edges**
- **Control edges**
- **Sequence edges**
- **Operation vertices**
- **Read and write vertices**
- **Constant and variable vertices**
- **Branch/merge vertices**
- **Loop entry/exit vertices**



Department of Computer Engineering

Differential equation example



Synthesis

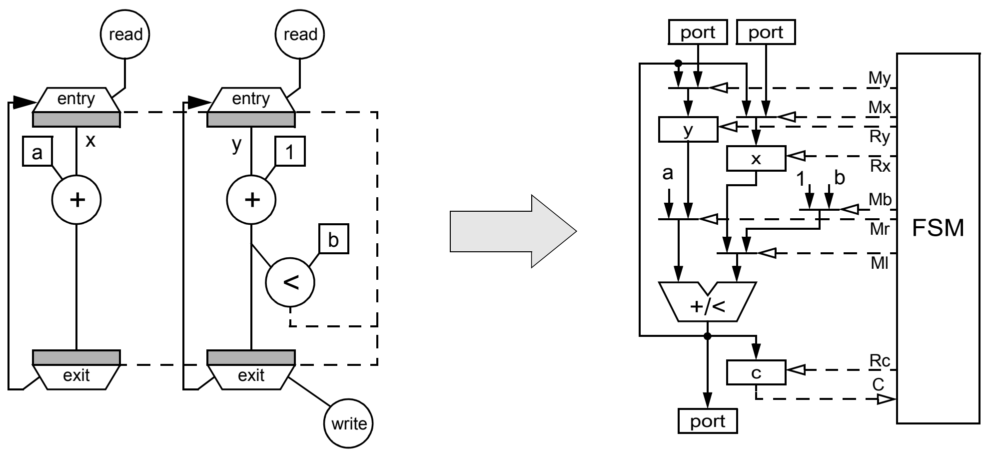
- **Synthesizing an appropriate RT level structure implies meeting hardware constraints such as area, clocking frequency, delay, power consumption, etc.**
  Physical parameters, however, can be estimated from the physical parameters of the hardware components in the library.

Component	Delay	Area
ALU(+,-,<)	24 ns	208
Adder	18 ns	125
Subtracter	19 ns	139
Comparator (<)	16 ns	72
Parallel Multiplier	49 ns	2284
2:1 Multiplexer || Tristate driver	2 ns	48
Register	1 ns	112
2-stage Pipelined Multiplier	28 ns	2624

example parameters - LSI-10K, 16-bit units

Synthesized structure

- **Internal representation: netlist G(C,N,E) and FSM(S,X,Y,f,g)**



Scheduling

- Scheduling – assignment of operations to time (control steps), possibly within given constraints and minimizing a cost function
  - transformational and constructive algorithms
  - use potential parallelism, alternation and loops
  - many good algorithms exist, well understood

- Definition
  - Given a set T of tasks of equal length 1, a partial order $\prec$ on T, a number of $m \in Z^+$ processors, and an overall deadline $D \in Z^+$.
  - *Precedence constrained scheduling* is defined as the following problem: Is there a schedule $\sigma : T \rightarrow \{0,1,...,D\}$ such that $|\{ t \in T : \sigma(t)=s \}| \leq m \ \forall s \in \{0,1,...,D\}$ and $t_i \prec t_j \Rightarrow \sigma(t_i) < \sigma(t_j)$?
  - Precedence constrained scheduling is NP-complete task.

Hierarchy of FU and operation types

- Relation $o_t \in r_k$ – functional unit (FU) r_k is capable of executing operation o_t
- R is the set of FUs
- R_k is the set of FUs of type k
- |K| is the number of FU types
- |T| is the number of operation types
- Uniform FU type $o_t \in r_1, \ \forall \ t \in T$
- Disjoint operation type sets
  $\{ t \in T : o_t \in r_{k_1} \} \cap \{ t \in T : o_t \in r_{k_2} \} = \emptyset \quad \forall \ k_1 \neq k_2 \in K$
- Overlapping functionality
  $\{ t \in T : o_t \in r_{k_1} \} \cap \{ t \in T : o_t \in r_{k_2} \} \neq \emptyset$ for some $k_1 \neq k_2 \in K$

Operation timing

- Single-cycle $\delta(o_t) \leq t_{cycle}$
- Multi-cycle $\delta(o_t) > t_{cycle}$
- Chaining (multiple operation with one clock)

- The *simple scheduling problem* is defined as the following problem:
  - Is there a schedule $\delta : V \rightarrow \{1,...,S\}$ such that
    $| \{ o_t \in V \wedge o_t \in r_k : \sigma(o_t)=s \} | \leq | R_k | \; \forall \; s \in \{1,...,S\}, k \in \{1,...,K\}$ and $(o_i \prec o_j) \Rightarrow \sigma(o_i) < \sigma(o_j)$?
  - Here, $|R_k|$ is the number of functional units of type k
  - With unlimited FUs available, the minimum schedule length S corresponds to the *critical path*

Scheduling problem

- Resource constrained scheduling (RCS)
  - A typical heuristic for RCS is *list scheduling*

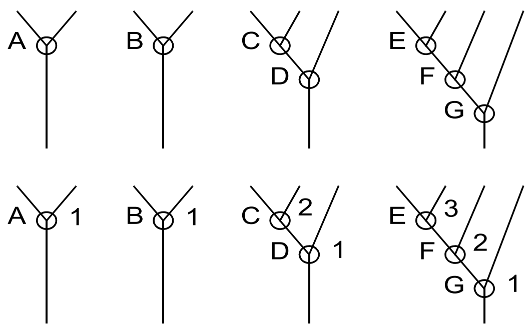
- Time constrained scheduling (TCS)

- Can be stated in terms of Integer Linear Programming (ILP)
  Given a cost function $c^T \bullet x$ and a constraints set of integer equations $A \bullet x = b$; A_{ij}, b_i - integer, find a parameter configuration x meeting the constraints such that the cost function is minimized and entries x_i are positive and integer.
  - Cost function – (1) schedule length or (2) resource cost
  - In general NP-complete problem

List scheduling

- **The basic idea is to sort the operations in a priority list in order to provide a selection criterion if operations compete for resources**
- **List scheduling is constructive method proceeding form control step to control step**
  - **for every step there are candidate “*ready*” operations**
  - **if the number of ready operations exceeds the number of FUs available, the operations with the highest priority are selected for being scheduled**
- **HU’s algorithm (RCS)**
  - **Polynomial time algorithm**
  - **Restrictions – DFG(V,E) is a forest (set of trees); single-cycle operations; uniform FUs**
  - **Consists of two steps – (1) labeling (bottom-up) & (2) scheduling according to resources available (top-down)**
- **Extensions**
  - **disjoint operation type sets**
  - **multi cycling and chaining (separated)**
- **Priority function (mobility) is used to sort “ready” operations**

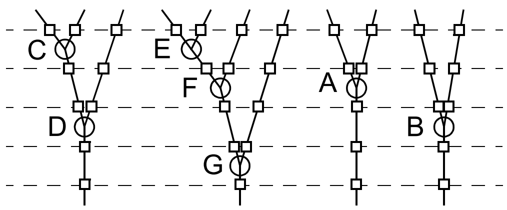
HU’s algorithm example



1. Labeling

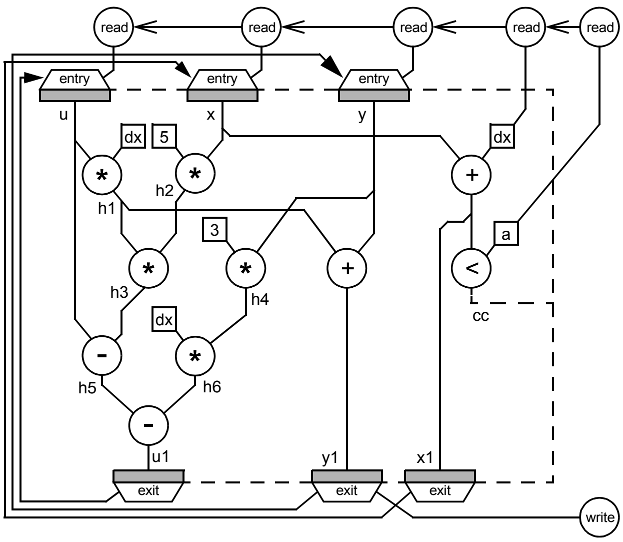
2. Scheduling (R=2)

```
ready := A B C E   ->  E C
         1 1 2 3
ready := A B D F   ->  F A
         1 1 1 2
ready := B D G     ->  B D
         1 1 1
ready := G         ->  G
         1
```



List scheduling and “diffeq” example

- Operations
  - h1 = u * dx
  - h2 = 5 * x
  - h3 = h1 * h2
  - h4 = 3 * y
  - h5 = u - h3
  - h6 = dx * h4
  - u1 = h5 + h6
  - x1 = x + dx
  - cc = x1 < a
  - y1 = h1 + y
- Data ready: a, dx, u, x, y, 3, 5
- 1 MUL (priorities)
  - h1(4), h2(4), h3(3), h4(3), h6(2)
- 1 ALU (priorities)
  - h5(2), x1(2), cc(1), u1(1), y1(1)

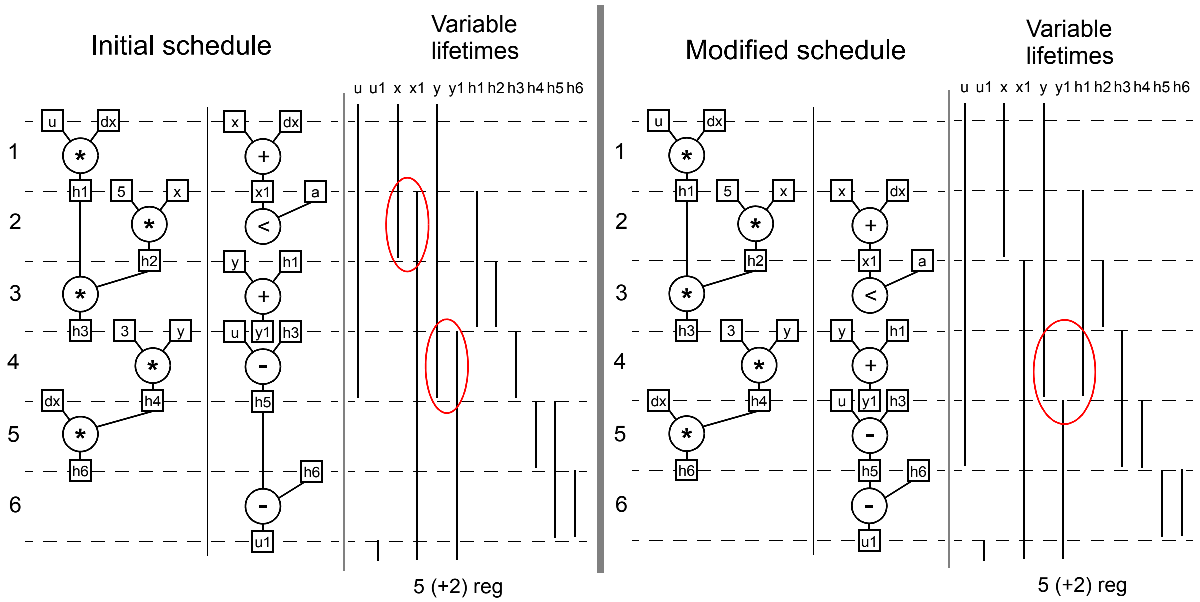


“diffeq” example – 1 MUL & 1 ALU

- All operations
  - MUL: h1:4[u,dx], h2:4[5,x], h3:3[h1,h2], h4:3[3,y], h6:2[dx,h4]
  - ALU: h5:2[u,h3], x1:2[x,dx], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]

Step	Data ready (variables)	MUL ready list	ALU ready list
1	a, dx, u, x, y (5)	**h1:4[u,dx]**, h2:4[5,x], h4:3[3,y] ~~h3:3[h1,h2], h6:2[dx,h4]~~	**x1:2[x,dx]** ~~h5:2[u,h3], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]~~
2	a, dx, u, x, y, h1, x1 (7)	**h2:4[5,x]**, h4:3[3,y] ~~h3:3[h1,h2], h6:2[dx,h4]~~	**cc:1[x1,a]**, y1:1[h1,y] ~~h5:2[u,h3], u1:1[h5,h6]~~
3	a, dx, u, y, h1, h2, x1 (7)	**h3:3[h1,h2]**, h4:3[3,y] ~~h6:2[dx,h4]~~	**y1:1[h1,y]** ~~h5:2[u,h3], u1:1[h5,h6]~~
4	a, dx, u, y, h3, x1, y1 (7)	**h4:3[3,y]** ~~h6:2[dx,h4]~~	**h5:2[u,h3]** ~~u1:1[h5,h6]~~
5	a, dx, h4, h5, x1, y1 (6)	**h6:2[dx,h4]**	~~u1:1[h5,h6]~~
6	a, dx, h5, h6, x1, y1 (6)		**u1:1[h5,h6]**

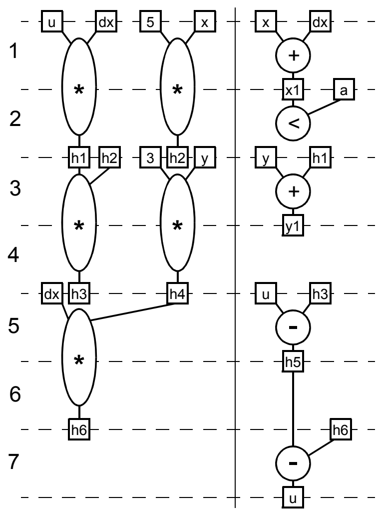
"diffeq" example – variables' lifetimes



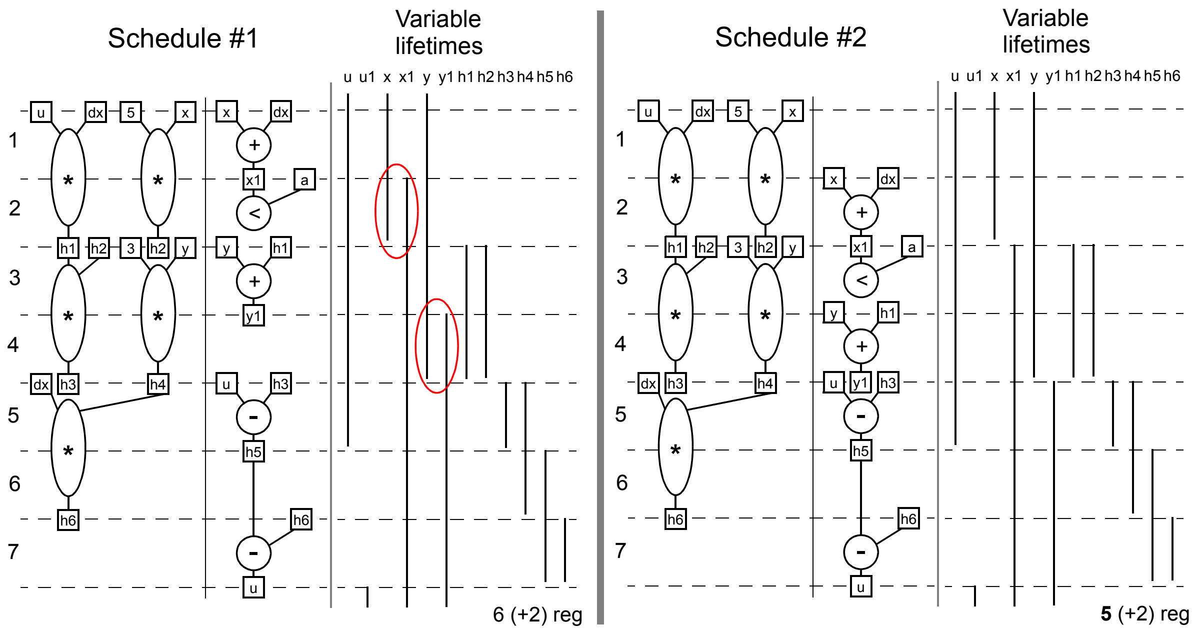
"diffeq" example #2 – 2 MUL (2-cycle) & 1 ALU

- **All operations**
  - **MUL: h1:4[u,dx], h2:4[5,x], h3:3[h1,h2], h4:3[3,y], h6:2[dx,h4]**
  - **ALU: h5:2[u,h3], x1:2[x,dx], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]**

Step	Data ready (variables)	MUL ready list	ALU ready list
1	a, dx, u, x, y (5)	**h1:4[u,dx]**, **h2:4[5,x]**, h4:3[3,y] ~~h3:3[h1,h2], h6:2[dx,h4]~~	**x1:2[x,dx]** ~~h5:2[u,h3], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]~~
2	a, dx, u, x, y, x1 (6)	**h1:4[u,dx]**, **h2:4[5,x]**, h4:3[3,y] ~~h3:3[h1,h2], h6:2[dx,h4]~~	**cc:1[x1,a]** ~~h5:2[u,h3], u1:1[h5,h6], y1:1[h1,y]~~
3	a, dx, u, y, h1, h2, x1 (7)	**h3:3[h1,h2]**, **h4:3[3,y]** ~~h6:2[dx,h4]~~	**y1:1[h1,y]** ~~h5:2[u,h3], u1:1[h5,h6]~~
4	a, dx, u, y, h1, h2, x1, y1 (8)	**h3:3[h1,h2]**, **h4:3[3,y]** ~~h6:2[dx,h4]~~	~~h5:2[u,h3], u1:1[h5,h6]~~
5	a, dx, u, h3, h4, x1, y1 (7)	**h6:2[dx,h4]**	**h5:2[u,h3]** ~~u1:1[h5,h6]~~
6	a, dx, h4, h5, x1, y1 (6)	**h6:2[dx,h4]**	~~u1:1[h5,h6]~~
7	a, dx, h5, h6, x1, y1 (6)		**u1:1[h5,h6]**



"diffeq" example #2 – variables' lifetimes



ASAP and ALAP

- **ASAP – "as soon as possible"**
  - **assignment to the earliest control step $\sigma_{ASAP}(o)$ possible**
- **ALAP – "as late as possible"**
  - **assignment to the latest control step $\sigma_{ALAP}(o)$ possible**
- **ASAP and ALAP scheduling are used for**
  - **calculate ASAP and ALAP times**
  - **calculate critical path(s)**
  - **find a average distribution of operation types in a control step**
  - **calculate mobility of operations: $M_o = \sigma_{ALAP}(o) - \sigma_{ASAP}(o)$**

Time constrained scheduling

- **TCS is performed subject to time constraints with the objective function to minimize the hardware to be allocated**
- **Categories**
  - **constraints to throughput or sampling rate have to be met (signal processing applications)**
  - **time constraints are spread over an algorithmic description (control-dominated applications)**
- **Basic method: Force directed scheduling**

Neural net based schedulers

- **Based on self-learning / self-adjusting features of artificial neural nets**
  - **efficient solving of hard task**
  - **simple use of multidimensional cost functions**
  - **can be painful to tune a neural net for a particular task**

Iterative improvements

step \ space	FU1 (<>)	FU2 (+-)	FU3 (+-)
#1		-a = b + c	d = e - f
#2	h = a / 2		
#3	g = a * 2	k = h - c	
#4			j = a + d

- - - data dependencies
——▷ valid moves

- **Simulated annealing**

$$p(\Delta E) = e^{-\frac{\Delta E}{KT}} \qquad \Delta E \sim \Delta c$$

K - Boltzmann's constant

$$\begin{cases} \Delta c \le 0 \Rightarrow accept \\ \Delta c > 0 \Rightarrow accept\ with\ probability\ p(\Delta c) = e^{-\frac{\Delta c}{T}} \end{cases}$$

- **Kohonen's self-organizing networks**

Path-based scheduling

- **AFAP (as fast as possible) scheduling minimizes the length of each program path**
  - **a CFG is used as the underlying internal representation**
- **The basic idea is (1) to optimize the control step assignment for every path separately and then (2) to minimize the number of states needed for the complete program when the paths are combined**

Path traversing schedulers

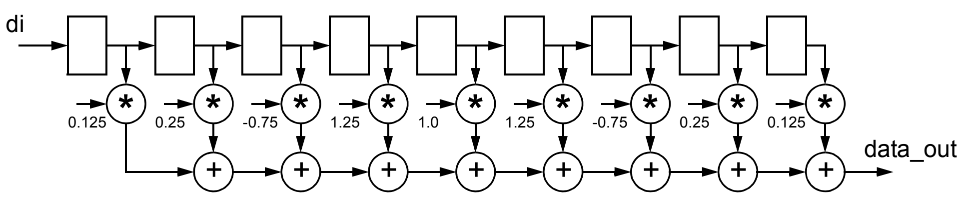
- **Control-flow oriented like path based scheduling**
  - **avoids construction of all possible path by traversing the CDF**
  - **states are assigned to satisfy rules and constraints**
  - **heuristic rules allow to prioritize constraints**
    - **I/O constraints, timing constraints resource constraints**

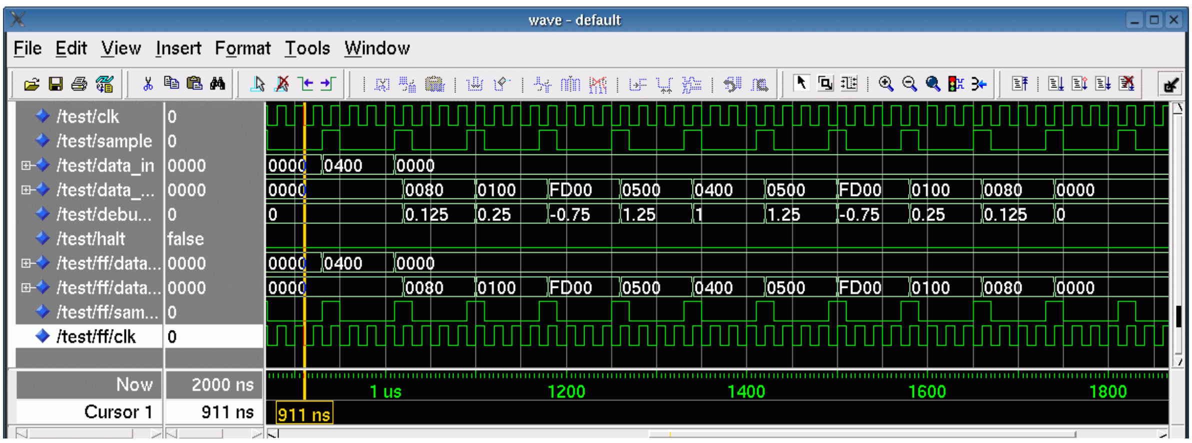
Data-flow versus control-flow

- DFG vs. CFG based
  - data-dependency vs. control-dependency dominance
- Data-flow based
  - exploits well (fine grain) parallelism
  - problems with control defined timing constraints
  - problems with operation chaining (especially when $\Delta(o) \ll 1$)
  - efficient for data dominated applications
- Control-flow based
  - exploits well operation chaining possibilities
  - may suffer from path explosion
  - efficient for control dominated applications
- Universally good scheduling algorithms?
  - data-flow based take into account control-flow
  - control-flow based take into account data-flow

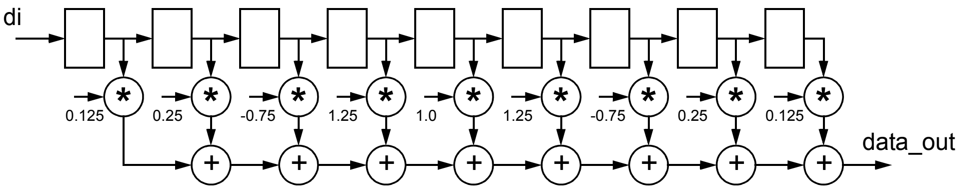
Scheduling example

- 9-tap Finite Impulse Response (FIR) filter





Scheduling 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**
- **More about FIR filters**
  http://www.falstad.com/dfilter/
  http://en.wikipedia.org/wiki/Finite_impulse_response

Scheduling example (cont.)

```
architecture  behave  of  fir_filter  is
  type array_type is array (1 to 9) of signed (15 downto 0);
  -- (0.125, 0.25, -0.75, 1.25, 1.0, 1.25, -0.75, 0.25, 0.125)
  constant coeffs: array_type := (
    "0000000010000000", "0000000100000000", "1111110100000000",
    "0000010100000000", "0000010000000000", "0000010100000000",
    "1111110100000000", "0000000100000000", "0000000010000000" );
begin
process
    variable delayed: array_type;
    variable sum: signed (15 downto 0);
    variable tmp: signed (31 downto 0);
  begin
    wait on clk until clk='1' and sample='1';     -- Waiting for a new sample
    data_out <= sum;                              -- Outputting results
    delayed (1 to 8) := delayed (2 to 9); delayed (9) := data_in; -- Shift and latch
    sum := (others=>'0');                                          -- Calculate
    for i in array_type'range loop
      tmp := coeffs(i) * delayed(i);   sum := sum + tmp(25 downto 10);
    end loop;
  end process;
end behave;
```

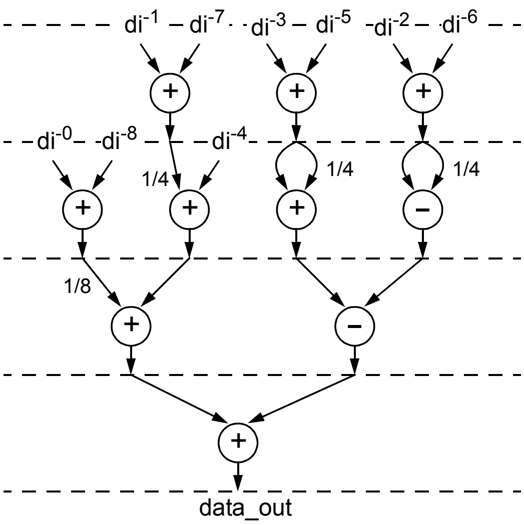
Scheduling example (cont.)

- **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 <u>three</u> functional units**
  - **$\lceil 10 / 4 \rceil = 3$**

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**

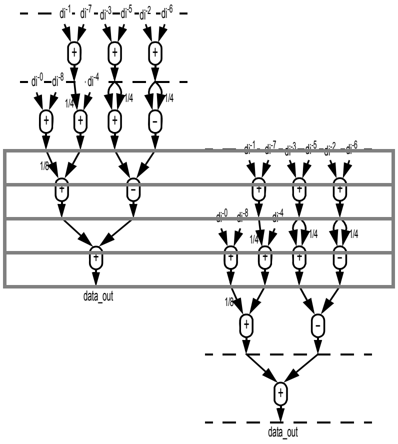
	additions	subtractions
1	v1=di^{-1}+di^{-7}; v2=di^{-2}+di^{-6}; v3=di^{-3}+di^{-5}	
2	v4=di^{-0}+di^{-8}; v5=di^{-4}+v1/4; v6=v3+v3/4	v7=v2-v2/4
3	v8=v4/8+v5	v9=v6-v7
4	data_out=v8+v9	



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

Scheduling example #2

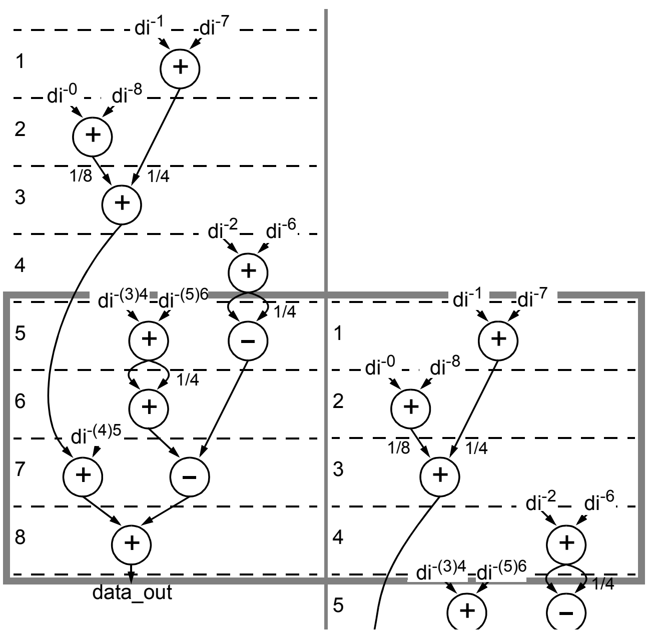
- **10 operations & 4 functional units**
  - **at least <u>three</u> functional units – $\lceil 10 / 4 \rceil = 3$**
- **10 operations & <u>3</u> functional units?**
  - **<u>7</u> operations should be executed during the first two clock steps**
- **Solution – pipeline**
  - **output data can be delayed**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



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**

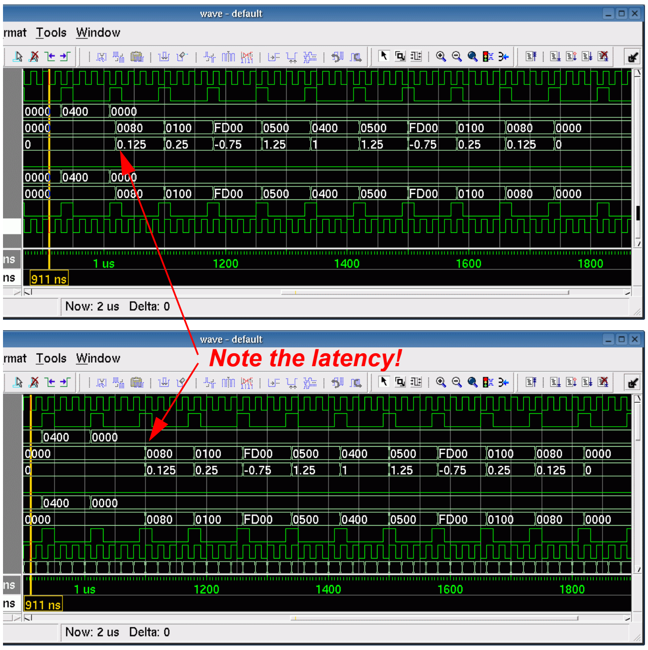
	additions	subtractions
1	v1=di^{-1}+di^{-7}	
(5)	v5=di^{-4}+di^{-6}	v6=v4-(v4/4)
2	v2=di^{-0}+di^{-8}	
(6)	v7=v5+(v5/4)	
3	v3=(v2/8)+(v1/4)	
(7)	v8=v3+di^{-5}	v9=v7-v6
4	v4=di^{-2}+di^{-6}	
(8)	data_out=v8+v9	



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

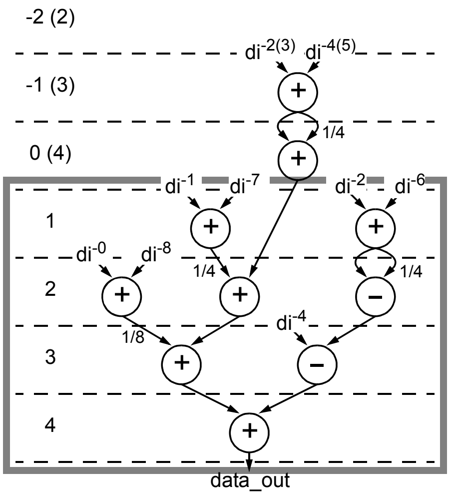


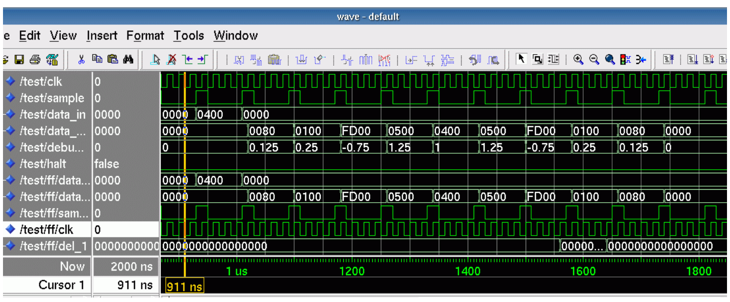
Scheduling example #3

- **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!)**





Allocation and binding

- High-level synthesis tasks, i.e., scheduling, resource allocation, and resource assignment neither need to be performed in a certain sequence nor to be considered as independent tasks
- Allocation is the assignment of operations to hardware possibly according to a given schedule, given constraints and minimizing a cost function
- Functional unit, storage and interconnection allocations
  - slightly different flavors:
    - module selection – selecting among several ones
    - binding – to particular hardware (a.k.a. assignment)
- Other HLS tasks...
    - *Memory management*: deals with the allocation of memories, with the assignment of data to memories, and with the generation of address calculation units
    - *High-level data path mapping*: partitions the data part into application specific units and defines their functionality
    - *Encoding* data types and control signals

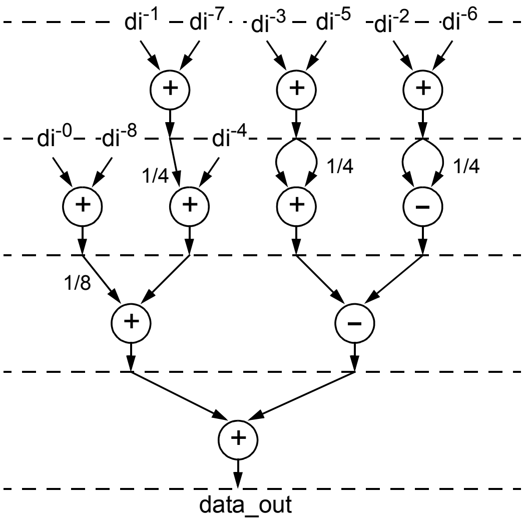
Department of Computer Engineering

Completing the Data Path

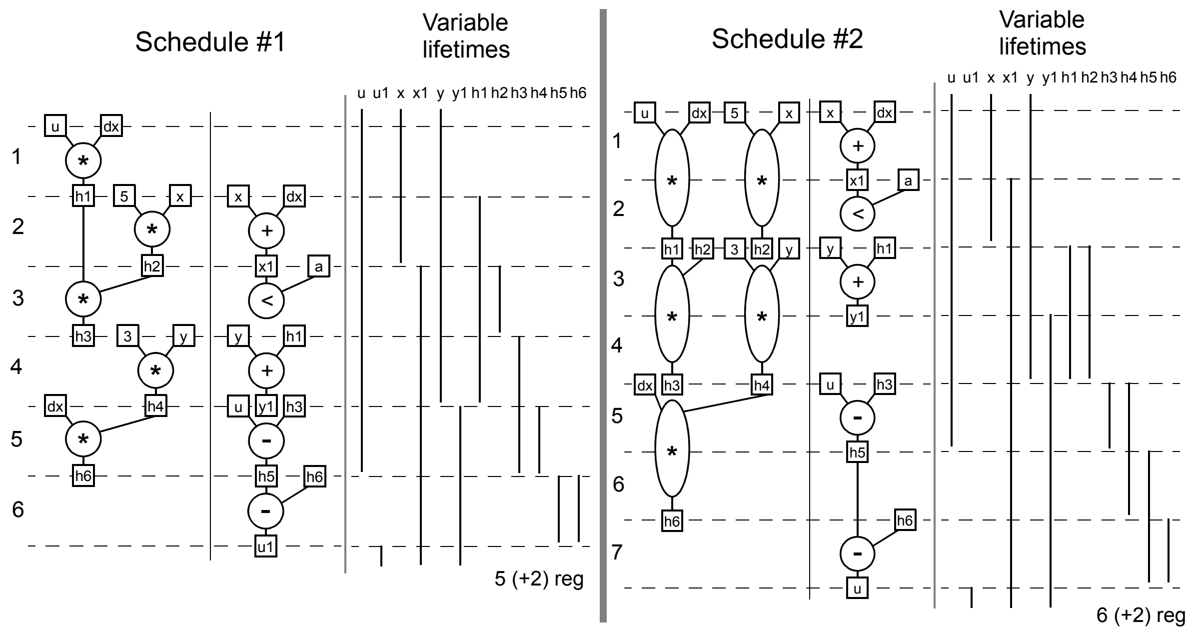
- Subtasks after scheduling
  - Allocation
    - Allocation of FUs (if not allocated before scheduling)
    - Allocation of storage (if not allocated before scheduling)
    - Allocation of busses (if buses are required and not allocated in advance)
  - Binding (assignment)
    - Assignment of operations to FU instances
      (if not assignment before scheduling as in the partitioning approach)
    - Assignment of values to storage elements
    - Assignment of data to be transferred to buses (if busses are used)
- Allocation and binding approaches
    - Rule based schemes (Cathedral II), used before scheduling
    - Greedy (e.g., Adam)
    - Iterative methods
    - Branch and bound (interconnect levels)
    - Integer linear programming (ILP)
    - Graph theoretical (clicks, node coloring)

Operation types and functional units

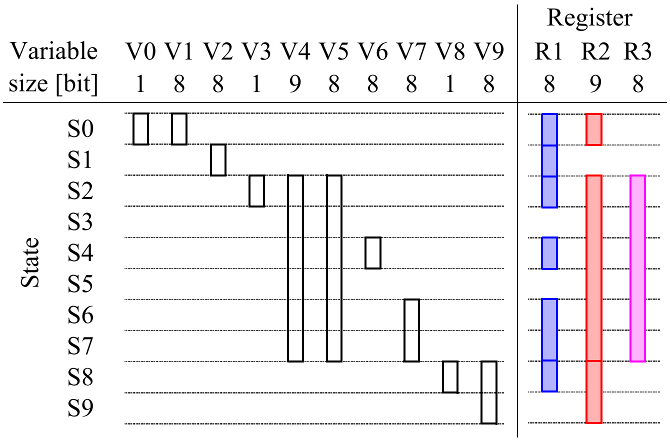
- An operation can be mapped onto different functional units
  - bit-width
    - 12-bit addition & 16-bit adder
  - supported operations
    - addition & adder/subtracter
  - cost trade-offs
    - universal modules are always more expensive
- Algebraic transformations
  - addition is commutative
    - a+b == b+a
  - double “inversion”
    - (a+b)-(c+d) == (a-d)-(c-b) == (a-c)-(d-b)

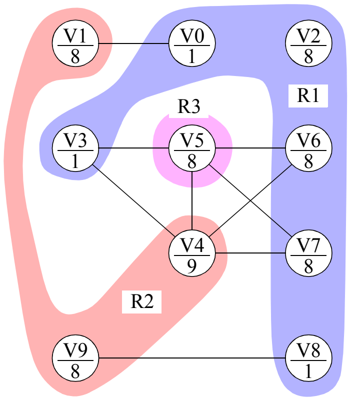


DFG and values lifetime table



Left-Edge algorithm and graph coloring



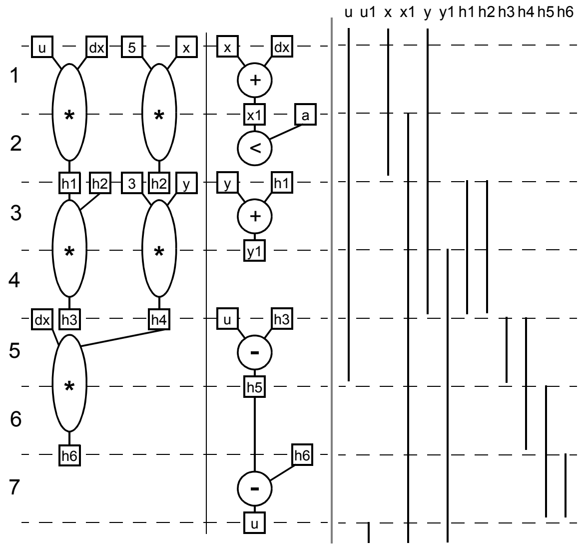


- a) lifetime moments of variables and allocated registers
- b) colored conflict graph

Example synthesis approach

- **Differential Equation example, multiplexed data part architecture**
- **Functional unit allocation**
- **Resource constrained scheduling**
- **Functional unit assignment**
- **Register allocation**
- **Register assignment**
- **Multiplexer extraction**

Schedule and lifetime table



- **Functional unit (FU) assignment**

FU	M1	M2	ALU
result	h1, h3, h6	h2, h4	x1, cc, y1, h5, u1

- **Register assignment**

Reg.	R1	R2	R3	R4	R5	R6
var.	u (u1) h5	x y1	y	x1	h1 h3 h6	h2 h4

Multiplexer optimization

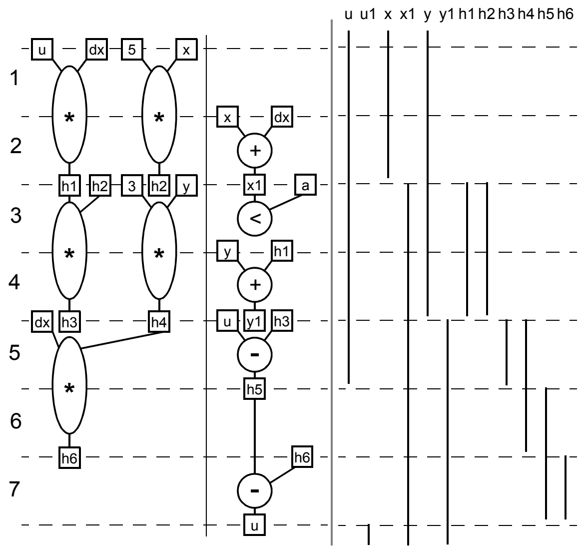
- **Functional units & registers**
  - **M1: h1:4[u,dx], h3:3[h1,h2], h6:2[dx,h4]; M2: h2:4[5,x], h4:3[3,y]**
  - **ALU: h5:2[u,h3], x1:2[x,dx], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]**
  - **Ra; Rdx; R1 (u,u1,h5); R2 (x,y1); Ry; Rx1; R5 (h1,h3,h6); R6 (h2,h4)**
- **Multiplexers**

step		1	2	3	4	5	6	7
L	M1	R1	R1	R5	R5	Rdx	Rdx	-
R		Rdx	Rdx	R6	R6	R6	R6	-
L	M2	5	5	3	3	-	-	-
R		R2	R2	Ry	Ry	-	-	-

	1	2	3	4	5	6	7
ALU	R2	Rx1	Ry	-	R1	-	R1
	Rdx	Ra	R5	-	R5	-	R5

  - **M1.L - 3, M1.R - 2, M2.L - 2, M2.R - 2, ALU.L - 4, ALU.R - 3**
  - **M1 has the same source (Rdx) on both multiplexers - swap inputs at the first step**
- **Result – 22 multiplexer inputs:**
  - **M1.L - 2 (Rdx, R5), M1.R - 2 (R1, R6), M2.L - 2 (5, 3), M2.R - 2 (R2, Ry), ALU.L - 4 (R2, Rx1, Ry, R1), ALU.R - 3 (Rdx, Ra, R5), Ra - 0 (inp), Rdx - 0 (inp), R1 - 2 (inp, ALU), R2 - 3 (inp, ALU, Rx1), Ry - 2 (inp, R2), Rx1 - 0 (ALU), R5 - 0 (M1), R6 - 0 (M2)**

Schedule and lifetime table – example #2



- **Functional unit (FU) assignment**

FU	M1	M2	ALU
result	h1, h3, h6	h2, h4	x1, cc, y1, h5, u1

- **Register assignment**

Reg.	R1	R2	R3	R4	R5
var.	u (u1) h5	x x1	y y1	h1 h3 h6	h2 h4

Multiplexer optimization – example #2

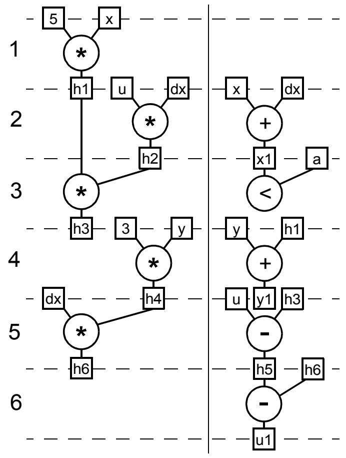
- **Functional units & registers**
  - **M1: h1:4[u,dx], h3:3[h1,h2], h6:2[dx,h4]; M2: h2:4[5,x], h4:3[3,y]**
  - **ALU: h5:2[u,h3], x1:2[x,dx], u1:1[h5,h6], cc:1[x1,a], y1:1[h1,y]**
  - **Ra; Rdx; R1 (u,u1,h5); Rx (x,x1); Ry (y,y1); R4 (h1,h3,h6); R5 (h2,h4)**
- **Multiplexers**

step		1	2	3	4	5	6	7
L	M1	R1	R1	R4	R4	Rdx	Rdx	-
R		Rdx	Rdx	R5	R5	R5	R5	-
L	M2	5	5	3	3	-	-	-
R		Rx	Rx	Ry	Ry	-	-	-

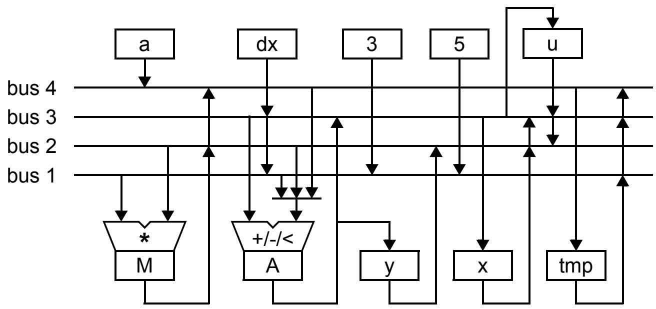
	1	2	3	4	5	6	7
ALU	-	Rx	Rx	Ry	R1	-	R1
	-	Rdx	Ra	R4	R4	-	R4

  - **M1.L - 3, M1.R - 2, M2.L - 2, M2.R - 2, ALU.L - 3, ALU.R - 3**
  - **M1 has the same source (Rdx) on both multiplexers - swap inputs at the first step**
- **Result – 20 multiplexer inputs:**
  - **M1.L - 2 (Rdx, R4), M1.R - 2 (R1, R5), M2.L - 2 (5, 3), M2.R - 2 (Rx, Ry), ALU.L - 3 (Rx, Ry, R1), ALU.R - 3 (Rdx, Ra, R4), Ra - 0 (inp), Rdx - 0 (inp), R1 - 2 (inp, ALU), Rx - 2 (inp, ALU), Ry - 2 (inp, ALU), R4 - 0 (M1), R5 - 0 (M2)**

Bidirectional bus architecture



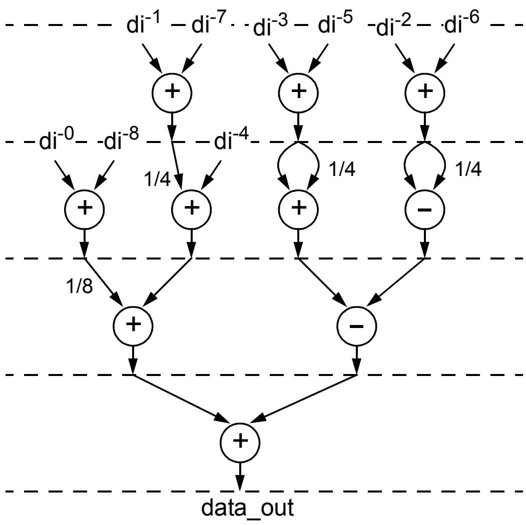
	bus 1	bus 2	bus 3	bus 4	
1	5 -> M.l	x -> M.r	A -> u	-	
2	dx -> M.l, A.r	u -> M.r	x -> A.l	M -> tmp	
3	tmp -> M.l	M -> M.r	A -> A.l, x	a -> A.r	
4	3 -> M.l	y -> M.r, A.r	tmp -> A.l	M -> tmp	
5	dx -> M.l	M -> M.r	u -> A.l	tmp -> A.r	A -> y
6	-	M -> A.r	A -> A.l	-	



Binding example #1

- **FIR filter**
  - **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}**
  - **transformations**
    - **shift-add trees & input swapping**
    - **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	

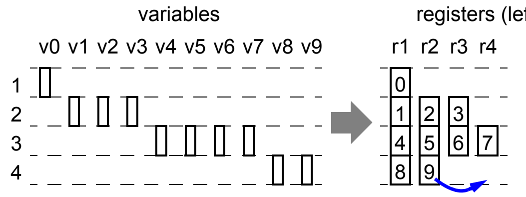


Binding example #1 (cont.)

- **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			



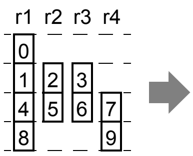
data from functional units:
r1: v0(a1), v1(a1), v4(a1), v8(a1)
r2: v2(a2), v5(a2), v9(s1)
r3: v3(a3), v6(a3)
r4: v7(s1)
r2 needs a mux!

after remapping:
r1: v0(a1), v1(a1), v4(a1), v8(a1)
r2: v2(a2), v5(a2)
r3: v3(a3), v6(a3)
r4: v7(s1), v9(s1)
No multiplexers needed

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			

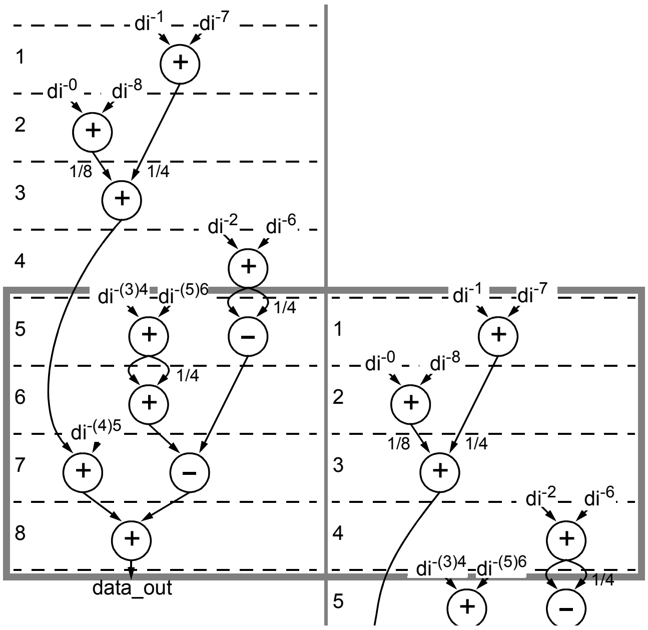
	a1 L	a1 R	a2 L	a2 R	a3 L	a3 R	s1 L	s1 R
1	di^{-1}	di^{-7}	di^{-2}	di^{-6}	di^{-3}	di^{-5}	--	--
2	di^{-0}	di^{-8}	di^{-4}	r1>	r3	r3>	r2	r2>
3	r1>	r2	--	--	--	--	r3	r4
4	r1	r4	--	--	--	--	--	--

- **Components: 3 add, 1 sub, 4 reg, 2 4-mux, 6 2-mux**
  - **3*125+139+4*112+(2*3+6)*48 = 1538 (+ controller & buffer)** ***[3332 eq.gates in total]***

Binding example #2

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

	additions	subtractions
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	

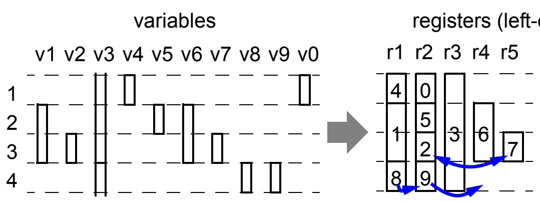


Binding example #2 (cont.)

- **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	



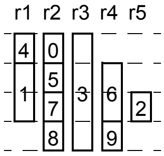
data from functional units:
r1: v1(a1), v4(a1), v8(a2)
r2: v0(a2), v2(a1), v5(a2), v9(s1)
r3: v3(a1)
r4: v6(s1)
r5: v7(a2)
r1 & r2 need multiplexers!

after remapping:
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)
No multiplexers needed

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 '>')**

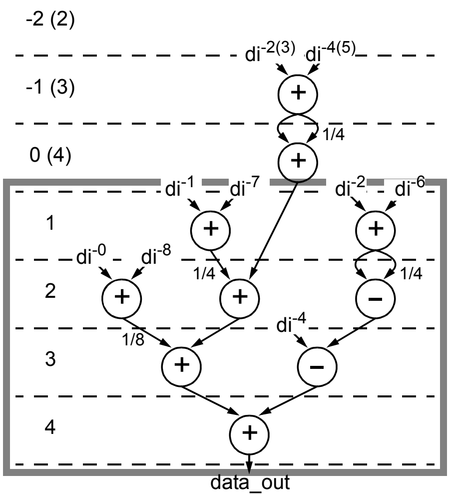
	add #1	add #2	sub #1
1	$v1=di^{-1}+di^{-7}$	$v5=di^{-4}+di^{-6}$	v6=v4-(v4/4)
2	$v2=di^{-0}+di^{-8}$	v7=v5+(v5/4)	
3	v3=(v2/8)+(v1/4)	$v8=v3+di^{-5}$	v9=v7-v6
4	$v4=di^{-2}+di^{-6}$	v0=v8+v9	

	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 – less FU-s (-1) but more reg-s (+1) & mux-s (+2)** ***[3440 e.g.]***

Binding example #3

- **Out-of-order execution (functional pipelining)**
- **Earlier samples are available!**
- **4 steps, 10 operations, 9 variables**



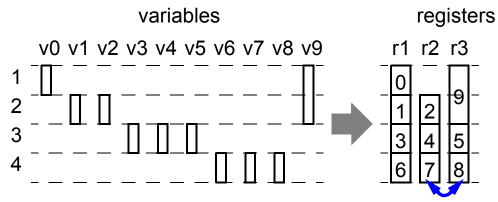
	additions	subtractions
1	$v1=di^{-1}+di^{-7}$; $v2=di^{-2}+di^{-6}$	
2	$v3=di^{-0}+di^{-8}$; v4=(v1/4)+v9	v5=v2-(v2/4)
3	v6=(v3/8)+v4; [$v8=di^{-2}+di^{-4}$]	$v7=di^{-4}-v5$
4	data_out=v6+v7; [v9=v8+(v8/4)]	

Binding example #3 (cont.)

- **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)	



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!

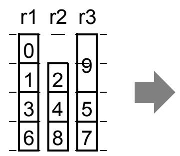
after remapping:
r1: v0(a1), v1(a1), v3(a1), v6(a1)
r2: v2(a2), v4(a2), v8(a2)
r3: v5(s1), v7(s1), v9(a2)

Only r3 needs a multiplexer

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
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 – less FU-s (-1) & reg-s (-1) but more mux-s (+2)** ***[3075 e.g.]***

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 – less FU-s (-1) & reg-s (-1)** ***[3055 e.g. = -8.3% vs. #1]***

Creating synthesizable code

- **Behavioral level code is not synthesizable**
- **Register-transfer level code is synthesizable**
- **What about "Behavioral RTL"?**
  - **... or 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 RT level code**
    - **pure RTL gives the best results (FSM & data-path == no ambiguities)**
  - **from structure to schematics (==synthesis)**

Creating synthesizable code

- Use bit-vector data types
  - corresponds to actual implementation, e.g. no overflow detection
- Simplify behavioral hierarchy
  - avoid timing control in subroutines
- Introduce structural hierarchy
  - only few processes per design unit
    - one process would be ideal
- No tricks with clock signal(s)
- Follow coding rules to avoid
  - latches in combinational processes
  - duplication of registers

- Behavioral level construct

```
wait until sign_1 = val_2 for 25 sec;
```

- Behavioral RT level (not synthesizable)
  - timer & counter introduced

```
for counter in 0 to 49 loop -- 25 sec
  exit when sign_1 = val_2;
  wait on timer until timer='1';
end loop;
```

- Behavioral RT level (synthesizable)
  - synthesizable counter

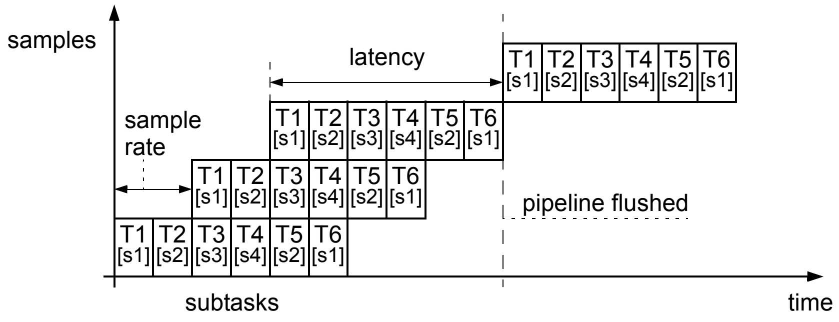
```
counter := 0; -- 25 sec
while counter < 50 and
      sign_1 /= val_2 loop
  counter := counter + 1;
  wait on timer until timer='1';
end loop;
```

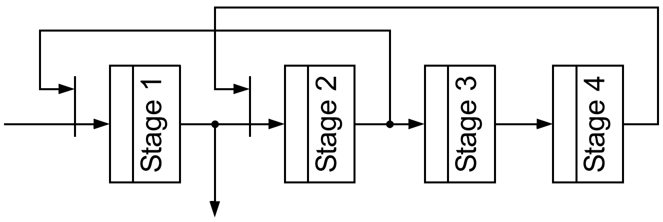
- Pure RTL == FSM + data-path
  - one 'wait' statement ~~ one state in FSM

Pipelining

- *Pipelining* - an implementation technique whereby multiple instructions are overlapped in execution
- *Latency (L)* - total number of time units needed to complete the computation on one input sample
- *Sample rate (R)* - the number of time units between two consecutive initiations, where initiation is the start of a computation on an input sample
- *A (pipe) stage* is a piece of HW that is capable of executing certain subtask of the computation
- *The reservation table* is a two-dimensional representation of the data flow during one computation. One dimension corresponds to the stages, and the other dimension corresponds to time units.
- Actions in pipeline: *flushing*, *refilling*, *stalling*.

Pipeline - example





Pipeline measurements

- **Average initiation rate (measure of pipeline performance):**

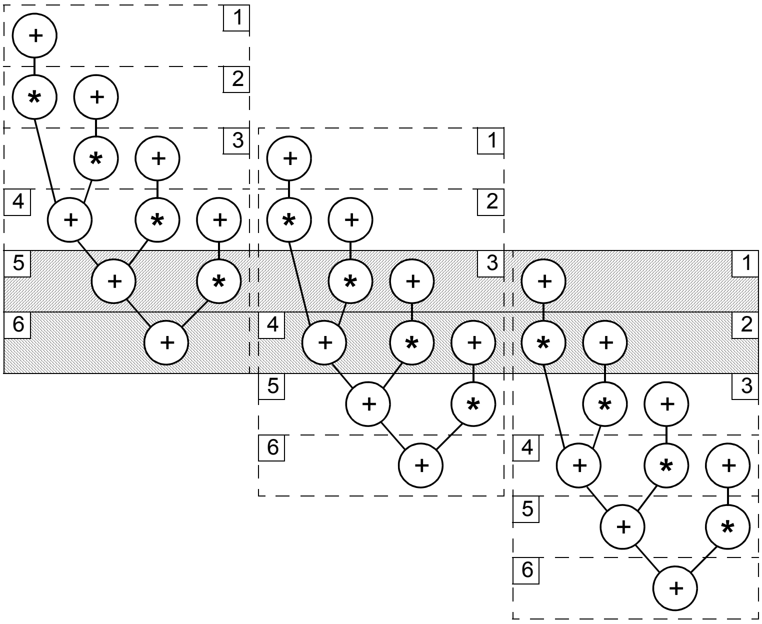
  $R_{init,N\to\infty} = 1 / (R \times t_{stage} + r_{synchro} \times (L\text{-}R) \times t_{stage})$

  - **R -sample rate, L - latency,**
  - **t_{stage} - the time one stage needs to complete its subtask,**
  - **$r_{synchro} = N_{flush} / N$ - resynchronization rate,**
  - **N_{flush} - the number of input samples that cause flushing,**
  - **N - number of input samples.**

Functional pipelining

- **In conventional pipelining, stages have physical equivalents, i.e. the stage hardware is either shared completely in different time units or not shared at all.**
- **In the case of large functional units, there is no physical stage corresponding to the logical grouping of operations in a time step.**
- **A *control step* corresponds to a group of time steps that overlap in time. Operations belonging to different control steps may share functional units without conflict.**
- **Operations, belonging to the time steps s+n×L, for n≥0, are executed simultaneously and cannot share hardware.**

Functional pipelining of 8-point FIR filter



Retiming

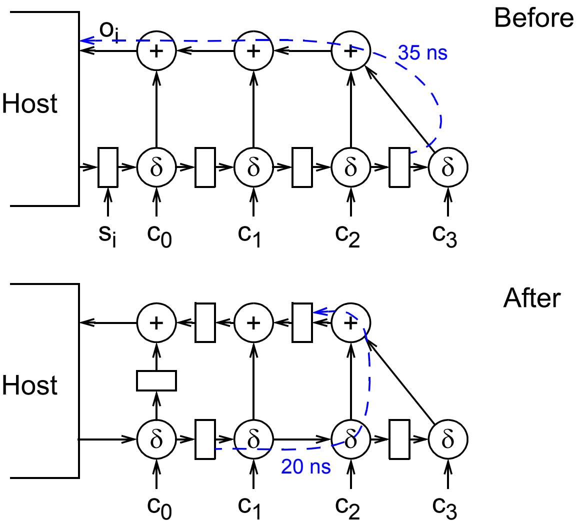
- Minimization of the cycle-time or the area of synchronous circuits by changing the position of the registers
  - cycle-time <- critical path
- The number of registers may increase or decrease
  - area minimization corresponds to minimizing the number of registers
  - combinational circuits are not affected (almost)
- Synchronous logic network
  - variables / boolean equations / synchronous delay annotation

Retiming at higher abstraction levels?

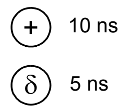
- The same, in principle, as for logic networks
  - operation nodes - functions / delay nodes - e.g. shared resources (memories)
- More possibilities to manipulate the functions - higher complexity of the optimization task
  - partitioning/merging functions
  - reorganizing shared resources

Retiming: example #2

Digital correlator

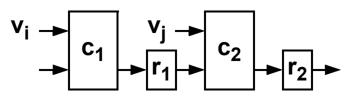


$$o_i = \sum_{j=0}^{3} \delta(s_{i-j}, c_j)$$

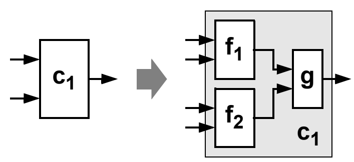


Retiming at higher abstraction levels

- **Optimization:**
  - **control-step #1:** $r_1 \leftarrow c_1(v_i,...)$,
  - **control-step #2:** $r_2 \leftarrow c_2(r_1,v_j,...)$
  - r_1,r_2 **- registers;** c_1,c_2 **- combinational blocks;** v_i,v_j **- variables**



  - f_{max} **= 1 / max(delay(**c_1**),delay(**c_2**)),**
  - **delay(**c_1**) > delay(**c_2**) : then** $c_1^{new} = g(\ f_1(v_i,...),\ f_2(v_i,...)\ )$



- **After resynthesis, delay(g)+delay(**c_2**) < delay(**c_1**) :**
  - **control-step #1:** $r_1 \leftarrow f_1(v_i,...);\quad r_x \leftarrow f_2(v_i,...)$
  - **control-step #2:** $r_2 \leftarrow c_2(g(r_1,r_x),v_j,...)$