Scheduling

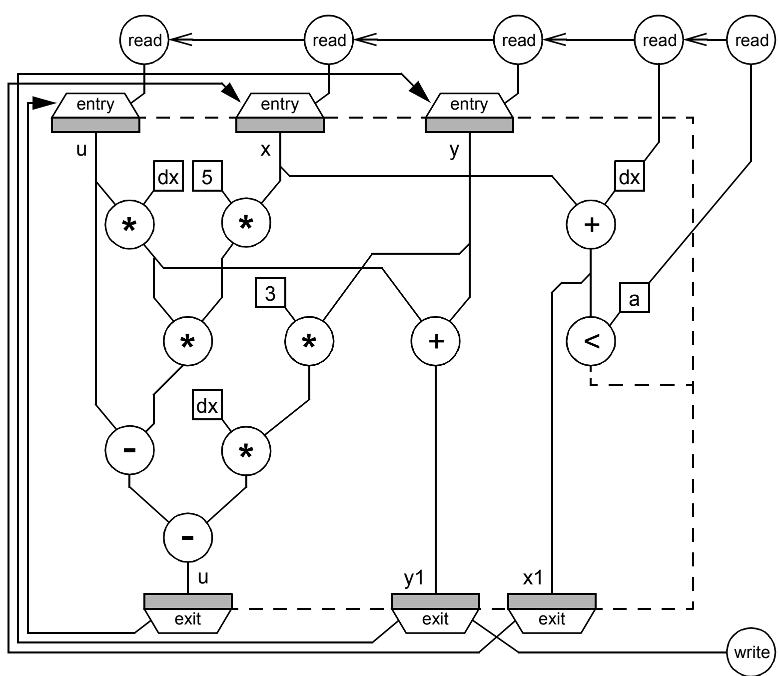
- Synthesis processes – a sequence of transformations
  - the main goal – to refine the specification in such a manner that all given initial constraints of the design are satisfied
- High-level synthesis (HLS) is usually divided into four optimization tasks
    - authors may classify differently
    - tools may execute in a different order
  - partitioning
    - dividing the behavioral description into sub-descriptions in order to reduce the size of the problem or to satisfy some external constraints
  - allocation
    - assigning operations onto available functional unit types (available in the target technology)
  - scheduling
    - assigning operations to control steps in order to minimize the amount of used hardware
    - imposes additional constraints how the operations can be allocated when performed before allocation (and binding)
  - binding
    - assigning operations to specific functional units in order to minimize the interconnection cost

Example – differential equation solver (diffeq)

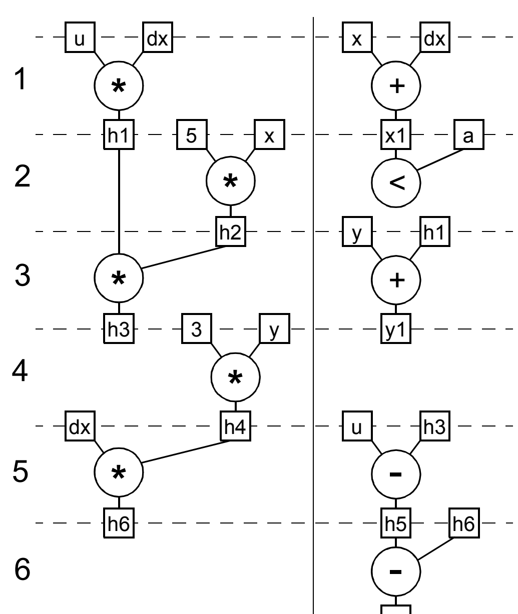
$$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

```
  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;
```

Differential equation solver example



Minimal hardware

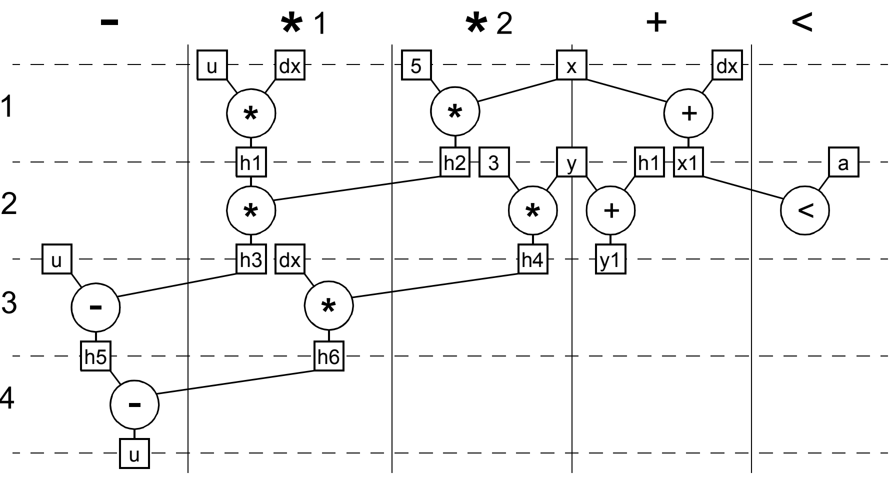


$$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

*	FU1
+/-/<	FU2

Minimal time

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



© Kalle Tammemäe / Peeter Ellervee

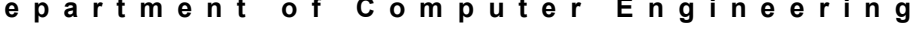
- **Scheduling – assignment of operations to time (control steps), possibly within given constraints and minimizing a cost function**
  - **transformational and constructive algorithms**
  - **use potential parallelism, alternation and loops**
  - **many good algorithms exist, well understood**

- **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.**

© Kalle Tammemäe / Peeter Ellervee

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
  $\{\ t \in T : o_t \in r_{k_1}\ \} \cap \{\ t \in T : o_t \in r_{k_2}\ \} = \varnothing \quad \forall\ k_1 \neq k_2 \in K$
- Overlapping functionality
  $\{\ t \in T : o_t \in r_{k_1}\ \} \cap \{\ t \in T : o_t \in r_{k_2}\ \} \neq \varnothing$ 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

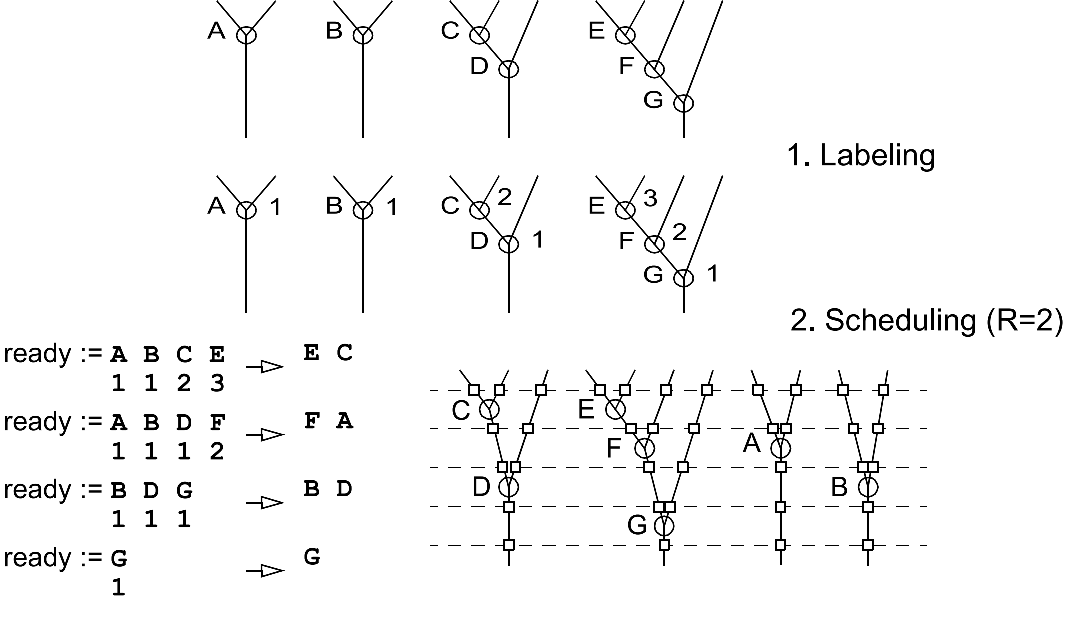
- Resource constrained scheduling (RCS)
- Time constrained scheduling (TCS)

- 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

Resource constrained scheduling

- A typical heuristic for RCS is *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 – a set of trees
    - Single-cycle operations
    - Uniform FUs
  - Consists of two steps
    - Labeling (bottom-up)
    - Scheduling according to resources available (top-down)

HU's algorithm example



```
{ Labeling routine: Assign labels P(o) to operations }
Λ ← ∅;
label ← 1;
    WHILE    V \ Λ ≠ ∅      DO BEGIN
    V_end ← { o∈V \ Λ : Succ(o,DFG)⊆Λ };
    P(o∈V_end) ← label;
    Λ ← Λ ∪ V_end;
    label ← label +1;
END
{ Schedule routine: Assign operations to control step σ(o) }
Σ ← ∅;
s ← 0;
WHILE    Σ ≠ V     DO BEGIN
    s ← s + 1;
    V_ready ← { o∈V \ Σ : Pred(o,DFG)⊆Σ };
    IF    |V_ready|>|R_1|     THEN BEGIN
        V_sel ← ∅;
        L_sorted ← Sort V_ready with decreasing P;
        WHILE    L_sorted ≠ NIL ∧ |V_sel|<|R_1|     DO BEGIN
            V_sel ← V_sel ∪ {HEAD(L_sorted)};
            L_sorted ← TAIL(L_sorted);
        END
    END ELSE    V_sel ← V_ready;
    σ(o∈V_sel) ← s;
    Σ ← Σ ∪ V_sel;
END
{ Schedule's length - s }
```

List scheduling

- **Enables**
  - **disjoint operation type sets**
  - **multi cycling and chaining (separated)**

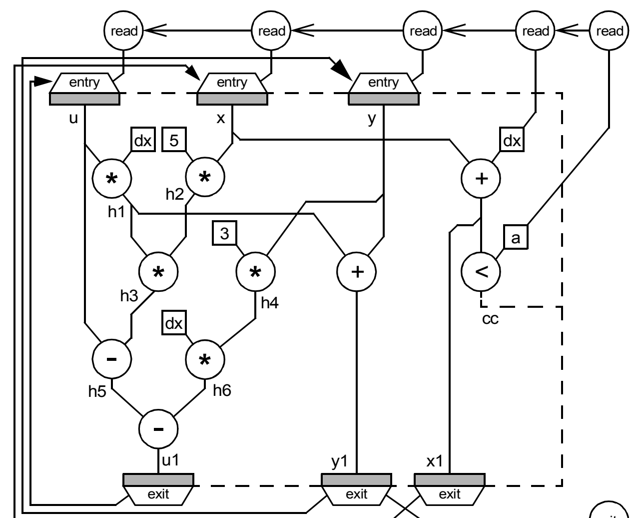
  - **$\Delta(o) < 1$ for fast operations (occupies resource for full cycle)**
  - **$\Delta(o) = 1$ for single-cycle operations**
  - **$\Delta(o) = \lceil \delta(o) / t_{cycle} \rceil$ for multi-cycle operations**

- **Priority function (mobility) is used to sort “ready” operations**

© Kalle Tammemäe / Peeter Ellervee

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)**



© Kalle Tammemäe / Peeter Ellervee

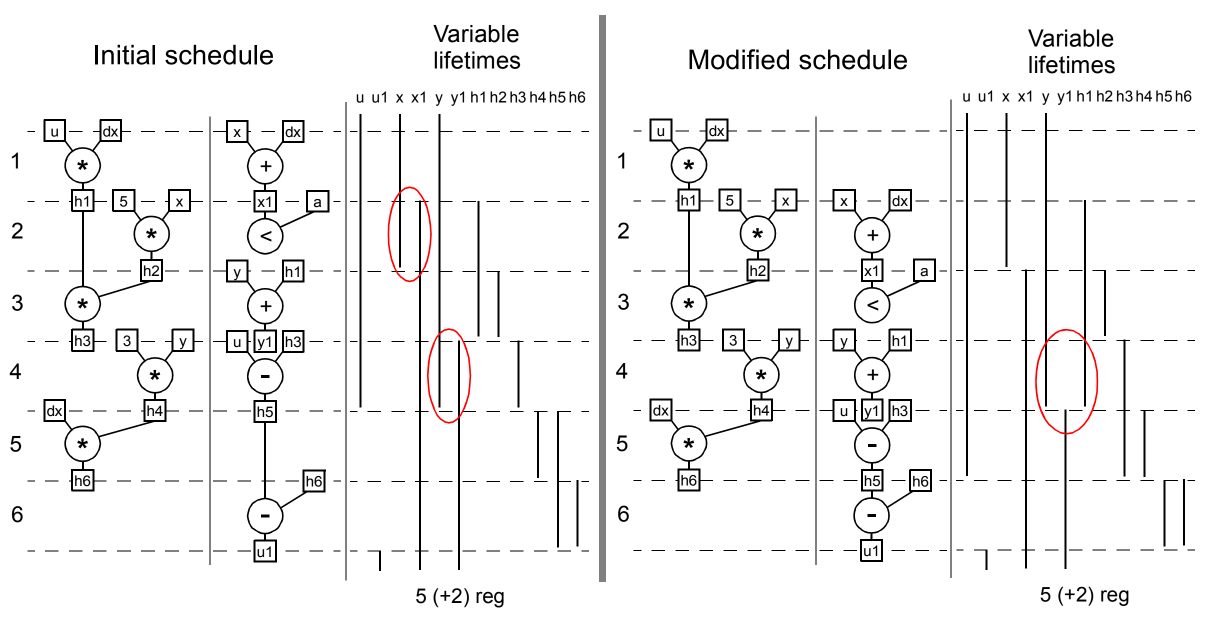
"diffeq" example – 1 MUL & 1 ALU

- **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]**

© Kalle Tammemäe / Peeter Ellervee

"diffeq" example – variables' lifetimes

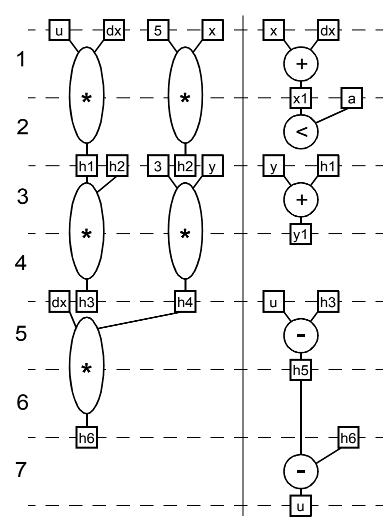


© Kalle Tammemäe / Peeter Ellervee

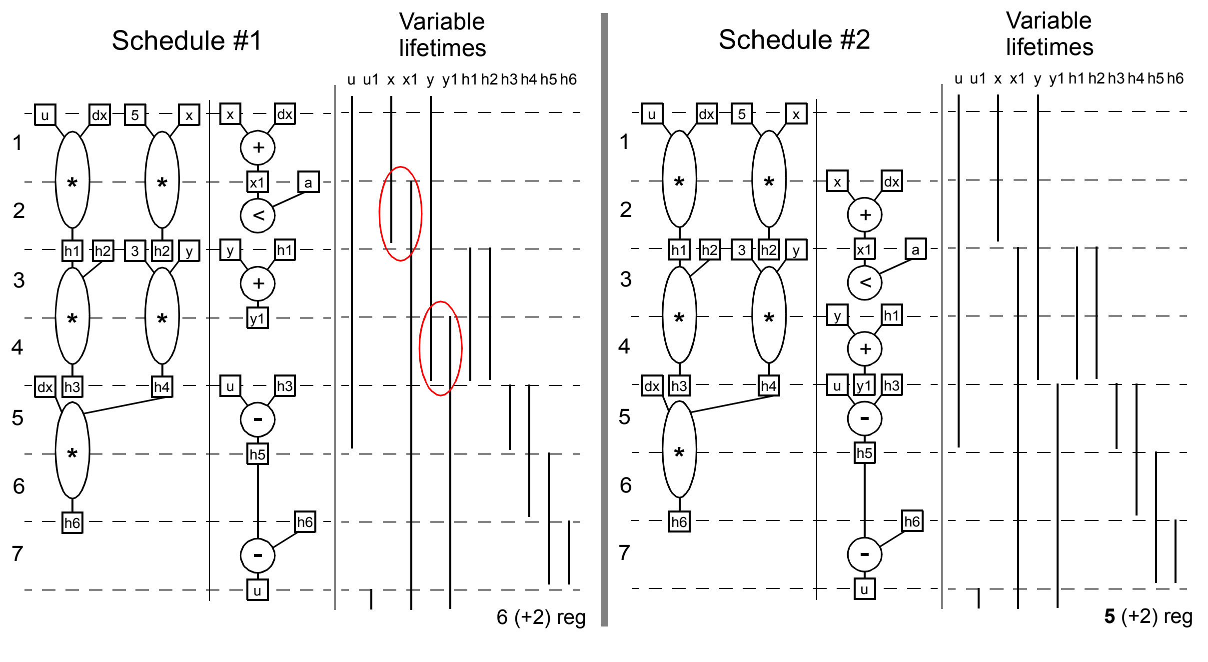
“diffeq” example #2 – 2 MUL (2-cycle) & 1 ALU

- **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]



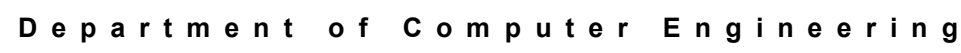
“diffeq” example #2 – variables’ lifetimes



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: $M_o = \sigma_{ALAP}(o) - \sigma_{ASAP}(o)$**

© Kalle Tammemäe / Peeter Ellervee



Department of Computer Engineering

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**

© Kalle Tammemäe / Peeter Ellervee

Force directed scheduling

- **The most popular method under a global and exact time constraint**
- **The goal – minimize HW subject to the given time constraint by balancing the concurrency of operations, value to be stored, and data transfers**
- **Force-directed scheduling successively selects, among all operations and control steps, an operation and the control step it is to be assigned to according to a measure called *force***
  - **Self force:**

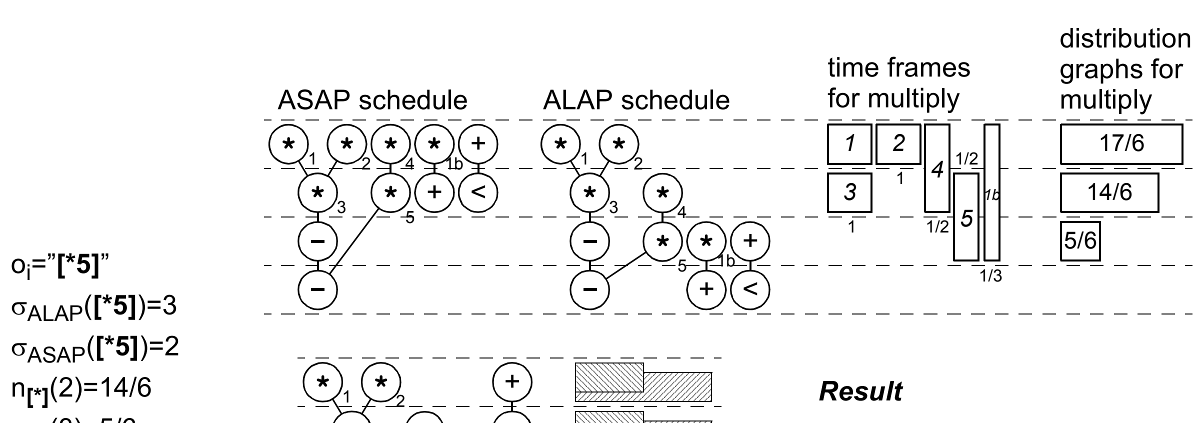
$$SF_{(\sigma(o_i)=s_j)} = n_k(s_j) - \frac{1}{\Delta T(o_i)} \sum_{s=\sigma_{ASAP}(o_i)}^{\sigma_{ALAP}(o_i)} n_k(s)$$

  - **operation o_i (of type k) at control step s_j ;**
  - **$n_k(s)$ - distribution probability; $\Delta T(o_i) = \sigma_{ALAP}(o_i) - \sigma_{ASAP}(o_i) + 1$**

Force directed scheduling - example

$$SF_{(\sigma(o_i)=s_j)} = n_k(s_j) - \frac{1}{\Delta T(o_i)} \sum_{s=\sigma_{ASAP}(o_i)}^{\sigma_{ALAP}(o_i)} n_k(s)$$

operation o_i (of type k) at control step s_j ;
$n_k(s)$ - distribution probability; $\Delta T(o_i) = \sigma_{ALAP}(o_i) - \sigma_{ASAP}(o_i) + 1$



o_i="[*5]"
σ_{ALAP}([*5])=3
σ_{ASAP}([*5])=2
$n_{[*]}(2)=14/6$
$n_{[*]}(3)=5/6$

$SF_{([*5]=2)}=0.75$
$SF_{([*5]=3)}=-0.75$

ASAP/ALAP schedules, time frames and distribution graphs

Algorithm for force directed scheduling

- **Force directed scheduling under a global time constraint τ_{exact}**

REPEAT
Evaluate and update actual ASAP and ALAP schedules and ΔTs
Create or update $n_k(s)$ (# of FUs type k in control step s)
For every unscheduled operation and for every
feasible control step calculate $SF_{(\sigma(o)=s)}$
Traverse the DFG for predecessor/successor contributions
Schedule operations with lowest overall force F
***UNTIL* all operations are scheduled**

- **predecessor/successor force ($\tilde{o}_i$ - reduced mobility of o_i):**

$$PSF_{(\sigma(o_i) = s_j)} = \frac{1}{\Delta T(\tilde{o}_i)} \sum_{s = \sigma_{ASAP}(\tilde{o}_i)}^{\sigma_{ALAP}(\tilde{o}_i)} n_k(s) - \frac{1}{\Delta T(o_i)} \sum_{s = \sigma_{ASAP}(o_i)}^{\sigma_{ALAP}(o_i)} n_k(s)$$

© Kalle Tammemäe / Peeter Ellervee

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

© Kalle Tammemäe / Peeter Ellervee

Iterative scheduling

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

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

K - Boltzmann's constant

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

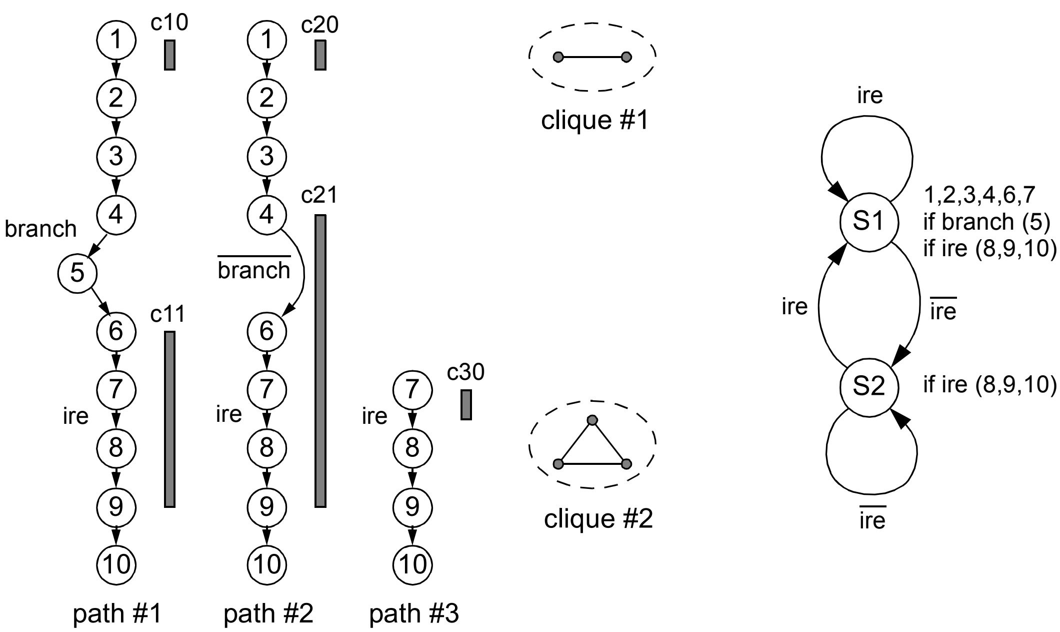
© Kalle Tammemäe / Peeter Ellervee

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**
- **The first task is equivalent with finding a minimum clique cover (NP) of an undirected graph G(V,E) where constraint violations are vertices v∈V connected with edge e∈E only when constraint violations intersect**
  - **G(V,E) is “interval graph”, there exists polynomial algorithm**
- **The second task (combining the paths) is equivalent with finding the minimum click cover, but due to graph structure there exists no polynomial-time algorithm**

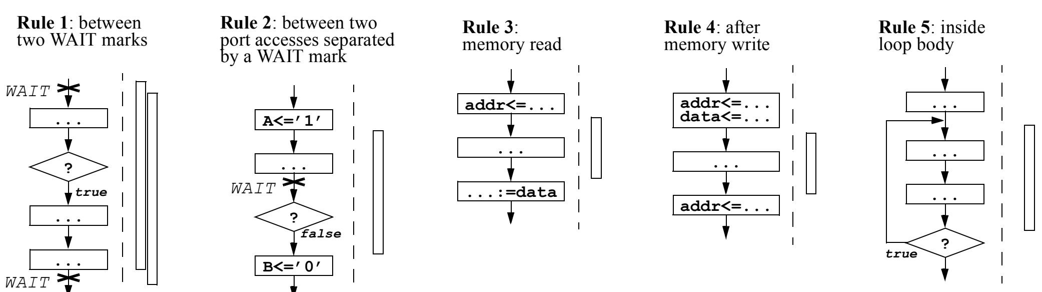
© Kalle Tammemäe / Peeter Ellervee

Path-based scheduling - example



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 $\Delta(o) << 1$)
  - efficient for data dominated applications

© Kalle Tammemäe / Peeter Ellervee

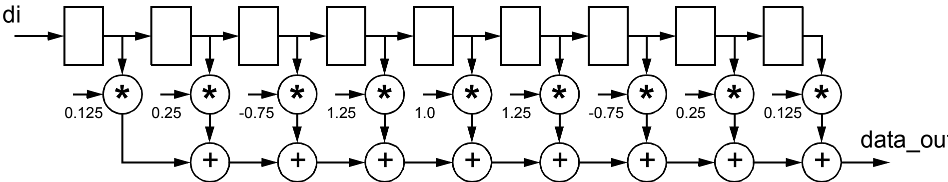
Data-flow versus control-flow (cont.)

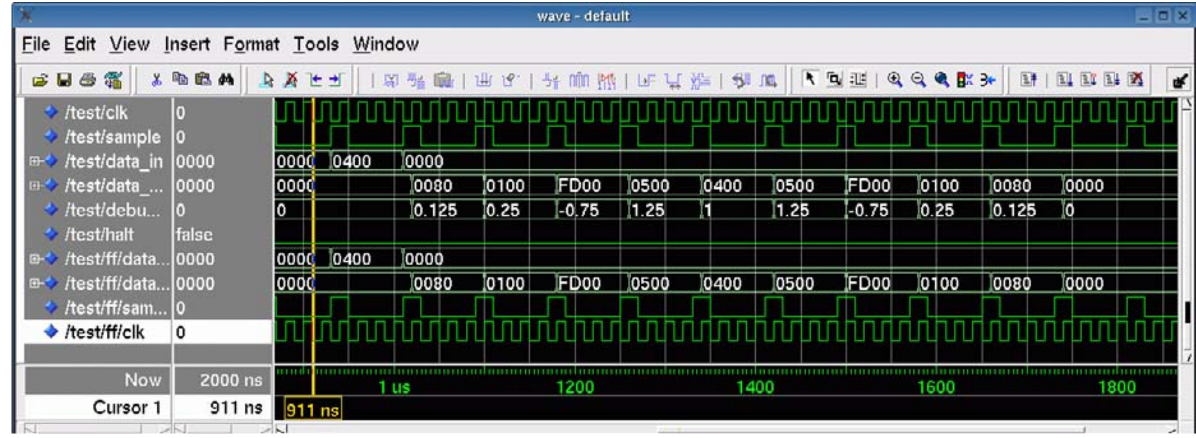
- Control-flow based
  - exploits well operation chaining possibilities
  - may suffer from path explosion
  - efficient for control dominated applications
- Universally good scheduling algorithms?
  - data-flow based take into account control-flow
  - control-flow based take into account data-flow

© Kalle Tammemäe / Peeter Ellervee

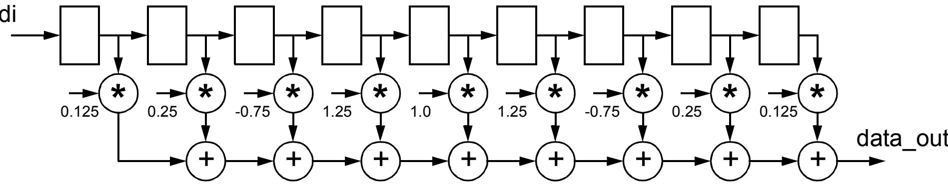
Scheduling example

- **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**

Scheduling example (cont.)

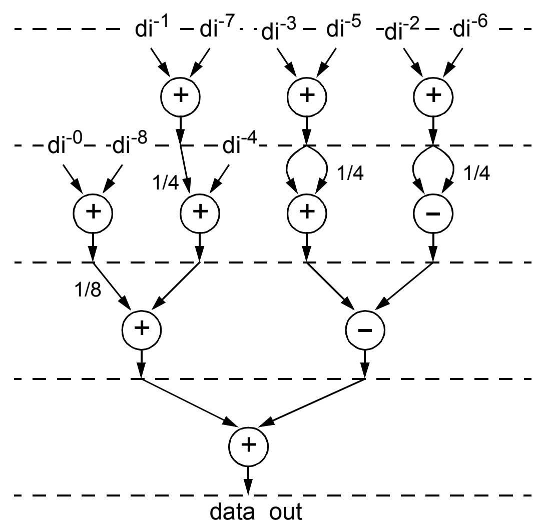
```
architecture  behave  of  fir_filter  is
  type array_type is array (1 to 9) of signed (15 downto 0);
  -- (0.125, 0.25, -0.75, 1.25, 1.0, 1.25, -0.75, 0.25, 0.125)
  constant coeffs: array_type := (
    "0000000010000000", "0000000100000000", "1111110100000000",
    "0000010100000000", "0000010000000000", "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 (cont.)

- **Algebraic transformations**
  - **addition is commutative**
    - **a+b == b+a**
  - **double "inversion"**
    - **(a+b)-(c+d) == (a-d)-(c-b) == (a-c)-(d-b)**



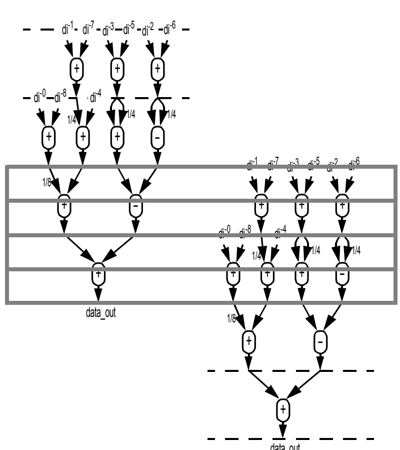
- **10 operations & 9 variables**

	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 (cont.)

- **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**

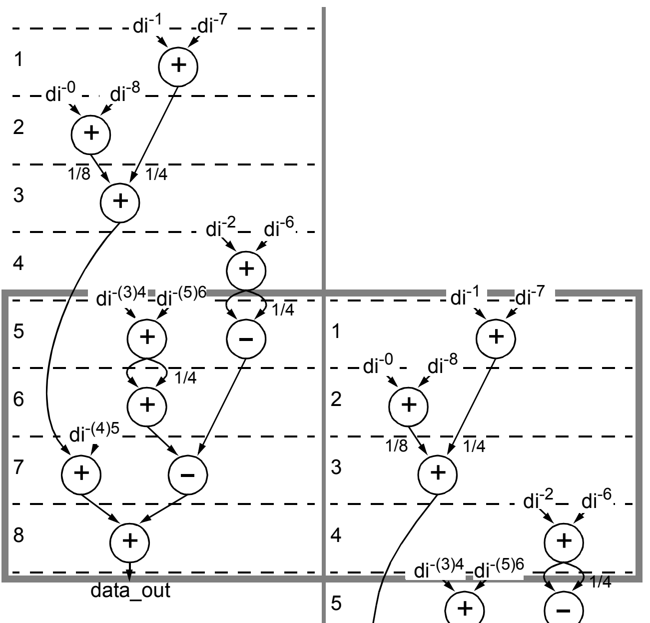


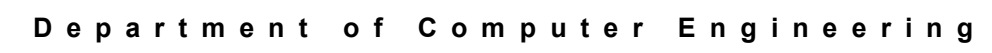
- **Solution – pipeline**
  - **output data can be delayed**
  - **two samples processed simultaneously**
  - **8 clock steps per sample**
  - **10+10 operations over 4+4 clock steps**

Scheduling example (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	

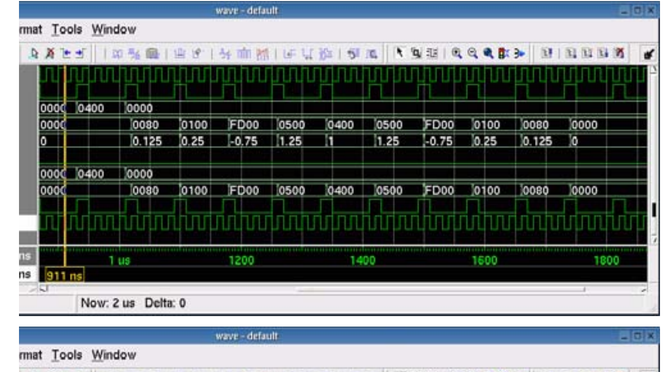


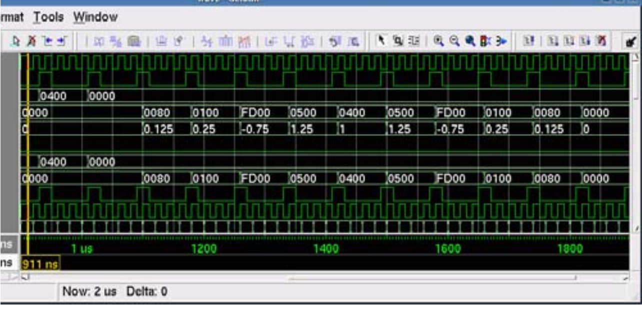


Scheduling example (cont.)

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



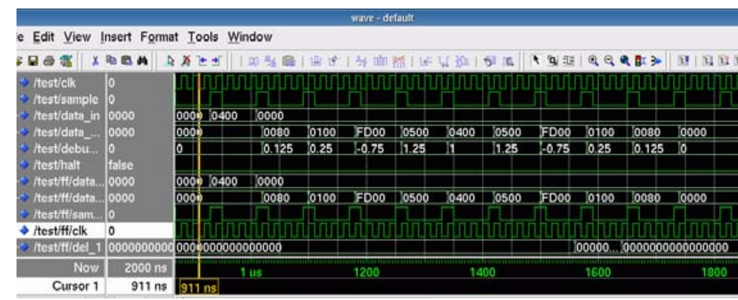


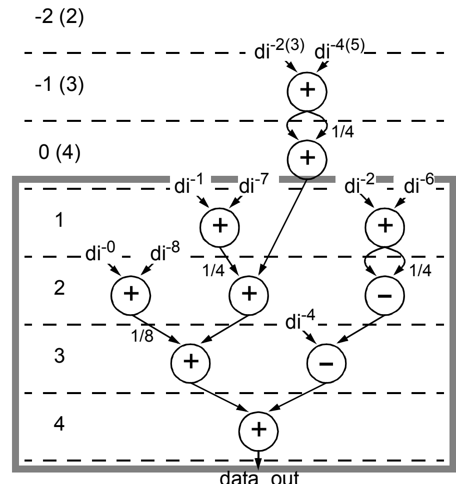
Scheduling example (cont.)

- **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!)**





Scheduling of high performance applications

Critical path

- **Sequence of operations with the longest finishing time**
  - **the longest path in the (weighted) graph**
- **Critical path in clock steps**
- **Critical path inside clock period (in time units)**
- **Topological path**
- **False path**
- **Latency ~ clock_period * critical_path_in_clock_steps**

Scheduling of high performance applications

Speculative execution

- Out of order execution
- Execution of operations in a conditional branch can be started <u>before</u> the branch itself starts
- Extra resources required
- Speeding up the algorithm
  - guaranteed speed-up
  - overall speed-up (statistical)
- Scheduling outside basic blocks
  - “flattening” hierarchy
  - increase in the optimization complexity

Scheduling of high performance applications

Loop folding and pipelining

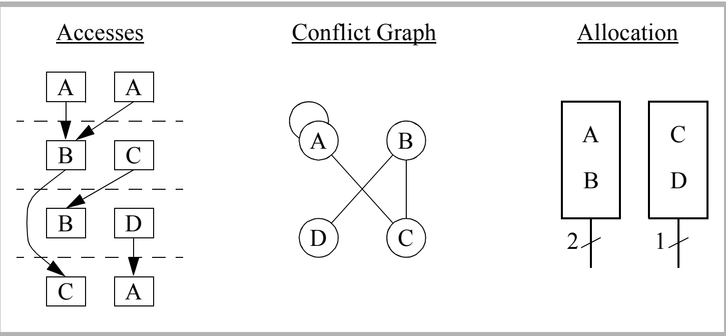
- Loop folding
  - fixed number of cycles
  - no need to calculate conditions in loop header
    - iterations of loop body can overlap
- Loop pipelining
  - executing some iterations of loop body in parallel
    - no data dependencies

Scheduling of high performance applications

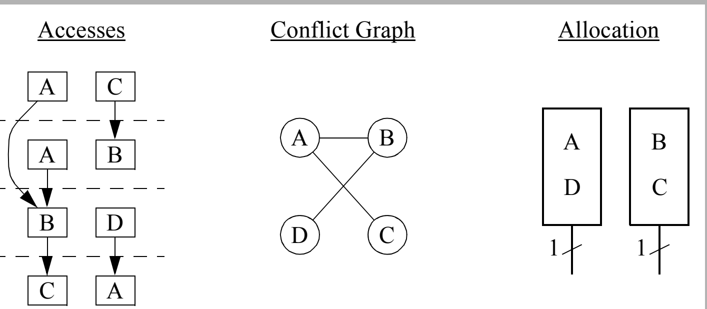
Storage operation scheduling

- Usually scheduled together with other operations
- Affects memory architecture
  - number of memories
  - single-port versus multi-port memories
- Memory operations can be scheduled before others to explore possible memory architectures
  - puts additional constraints for ordinary scheduling
  - reducing the number of possible combinations

© Kalle Tammemäe / Peeter Ellervee



Initial



After scheduling of memory accesses

© Kalle Tammemäe / Peeter Ellervee