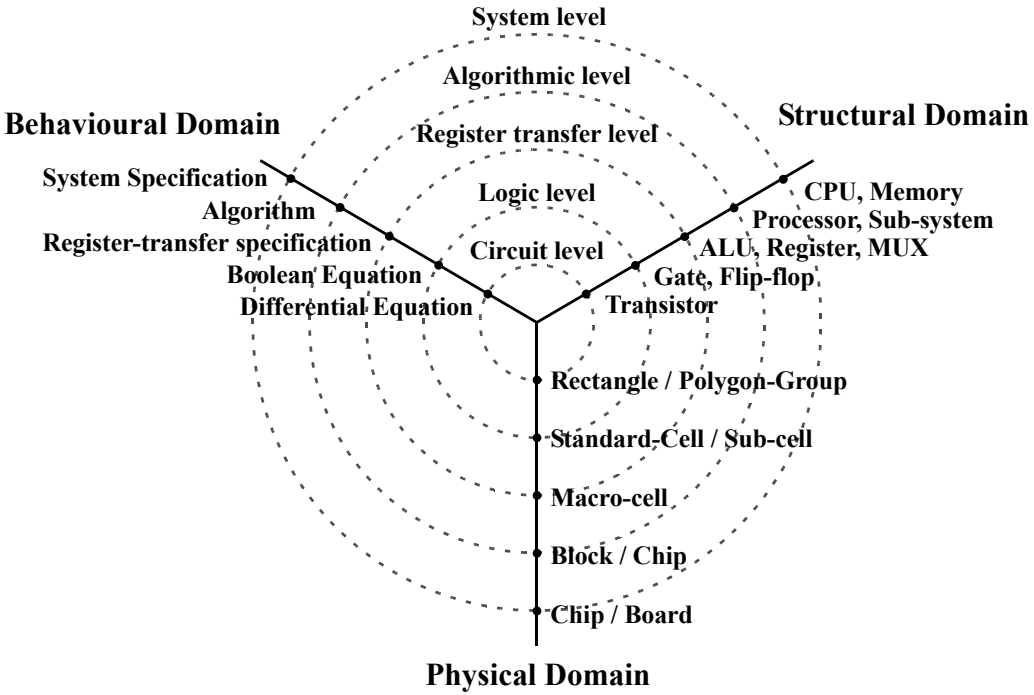
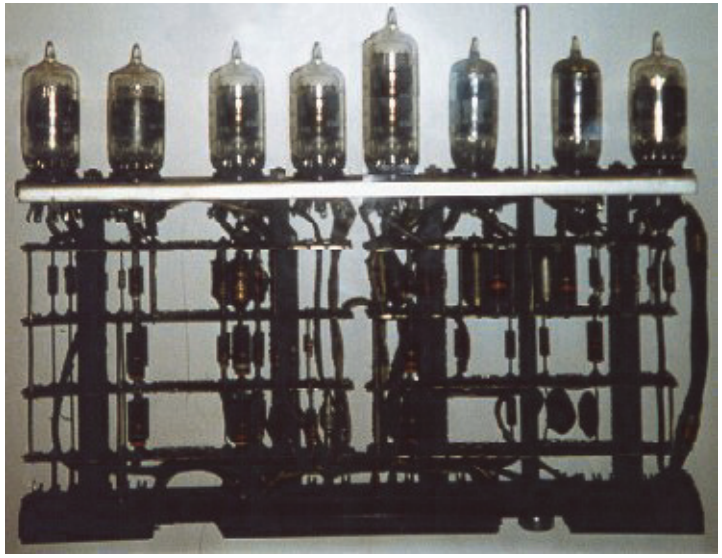
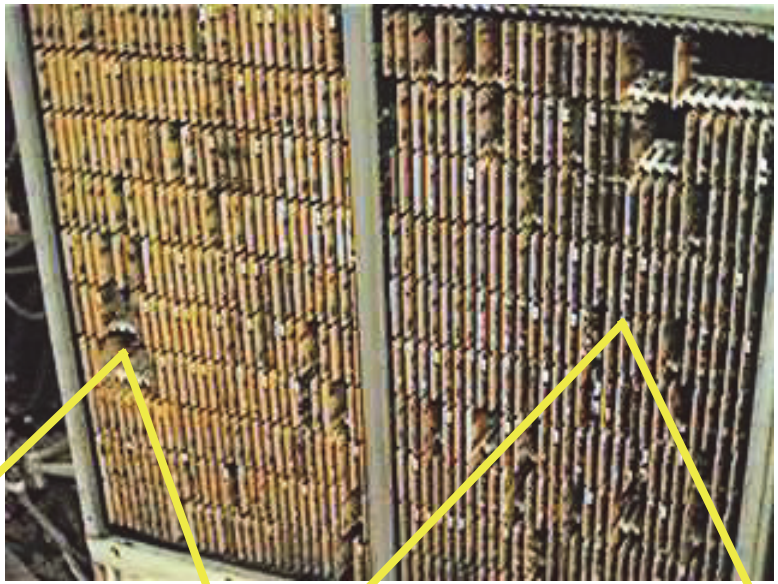
Synthesis at different abstraction levels

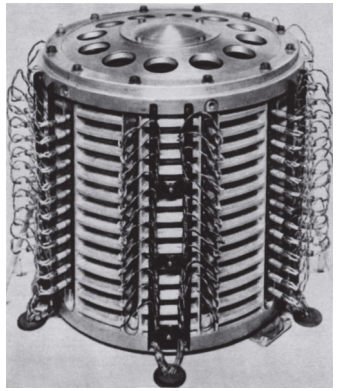
- System Level Synthesis
  - Clustering.
    Communication synthesis.
- High-Level Synthesis
  - Resource or
    time constrained scheduling
  - Resource allocation. Binding
- Register-Transfer Level Synthesis
  - Data-path synthesis.
    Controller synthesis
- Logic Level Synthesis
  - Logic minimization.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

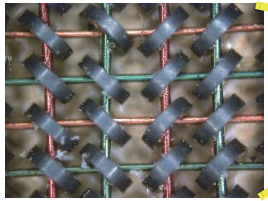


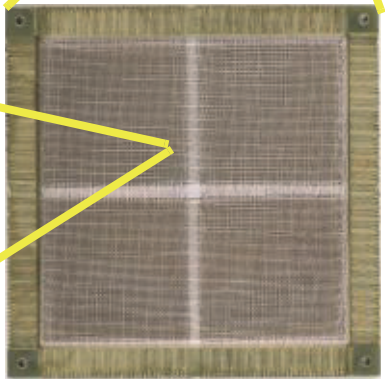
Physical implementation – history

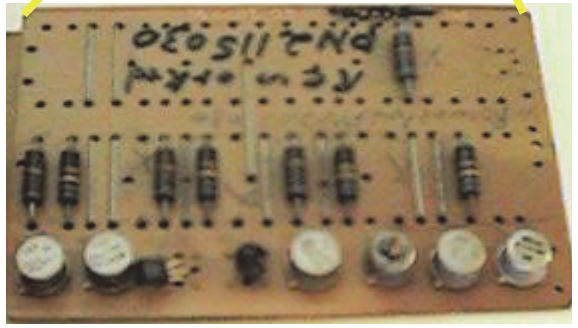




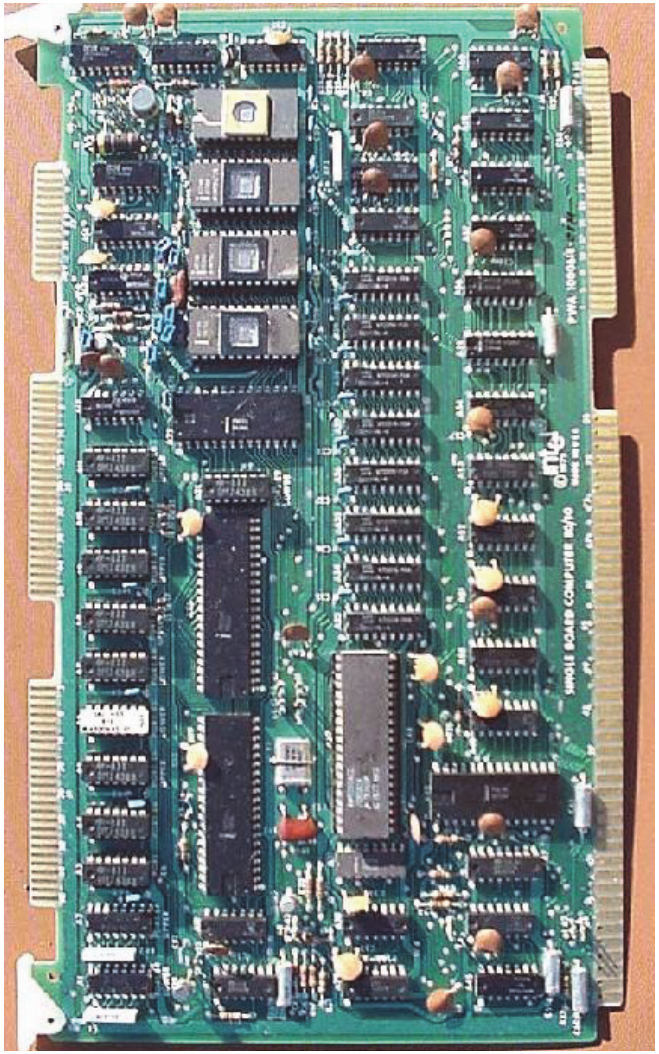


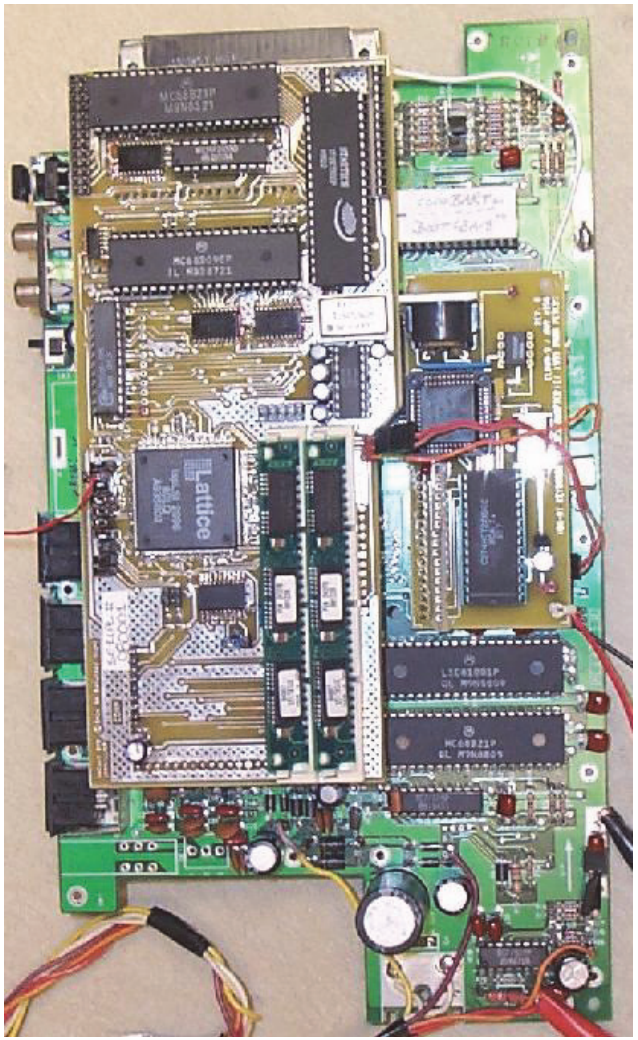


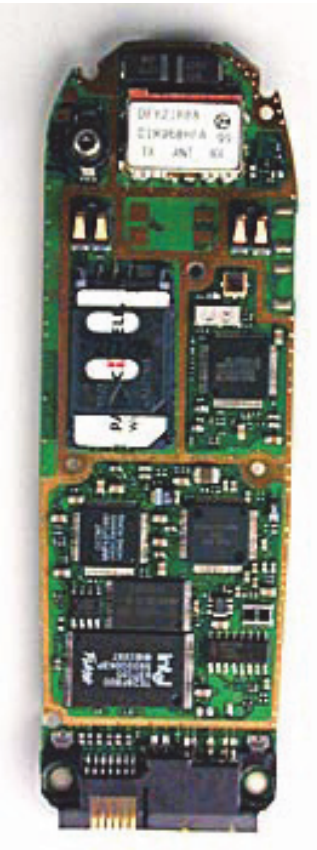




Physical implementation – history

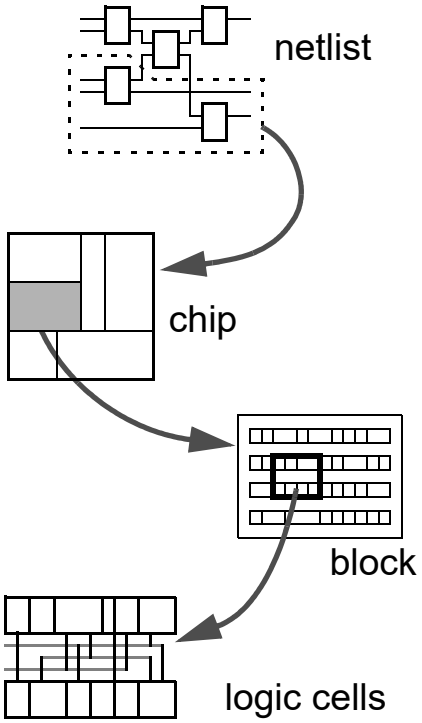


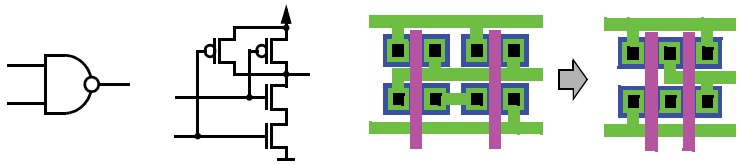




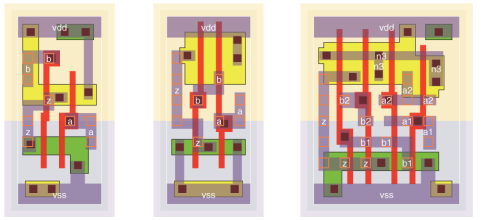
Chip design and fabrication

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

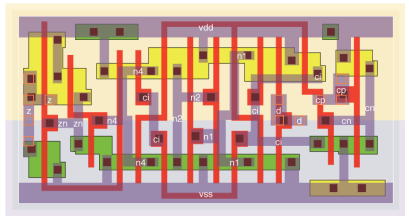




Standard cell examples



flip-flop

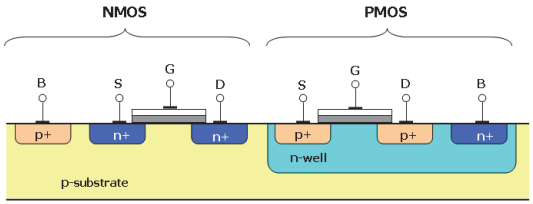


http://www.vlsitechnology.org/

https://www.tel.com/museum/exhibition/process/process1.html
https://www.apogeeweb.net/electron/semiconductor-manufacturing-steps-with-flow-charts.html

CMOS chip fabrication

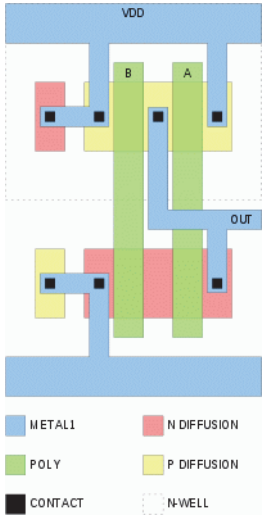
CMOS transistors



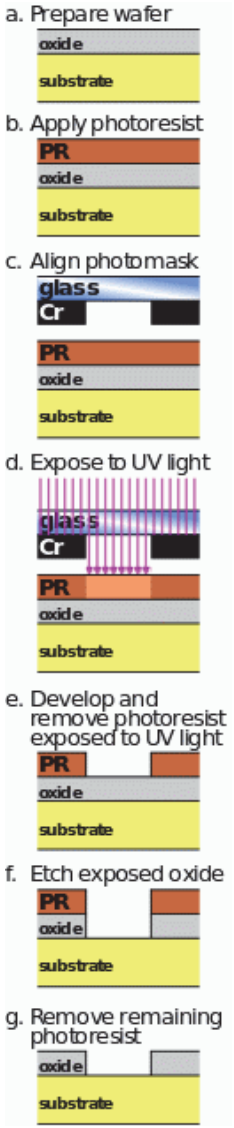
n - electrons [P, As, Sb]

p - holes [B, Al]

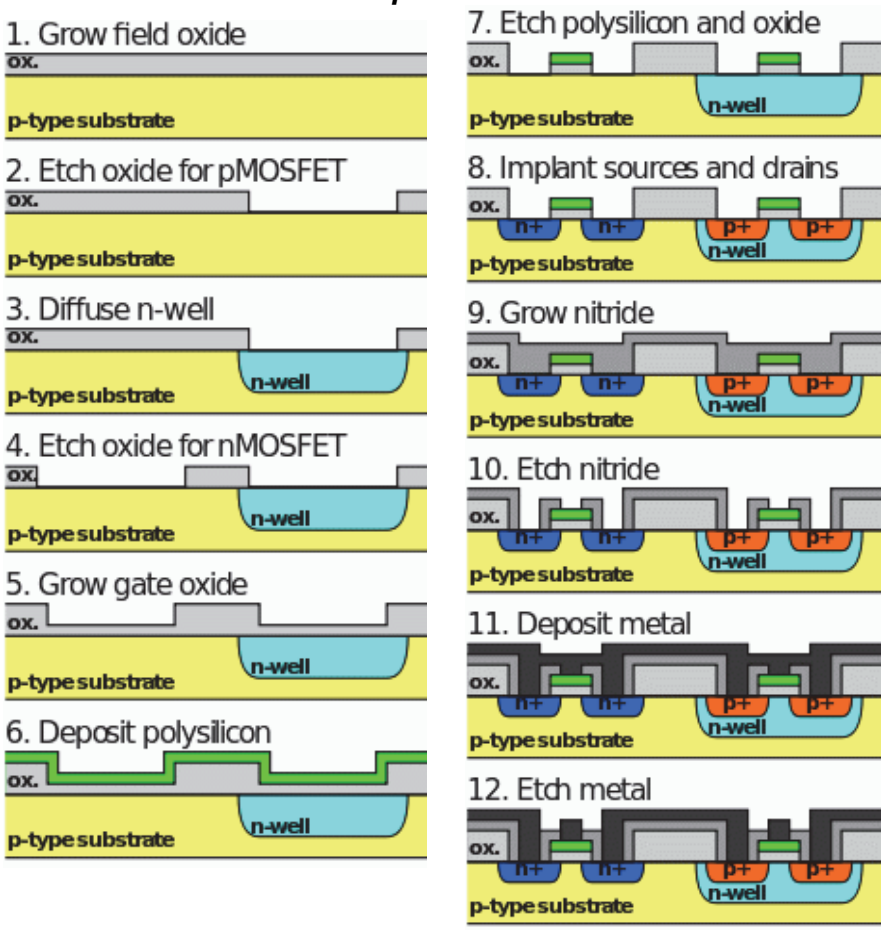
2-NAND layout



etching steps



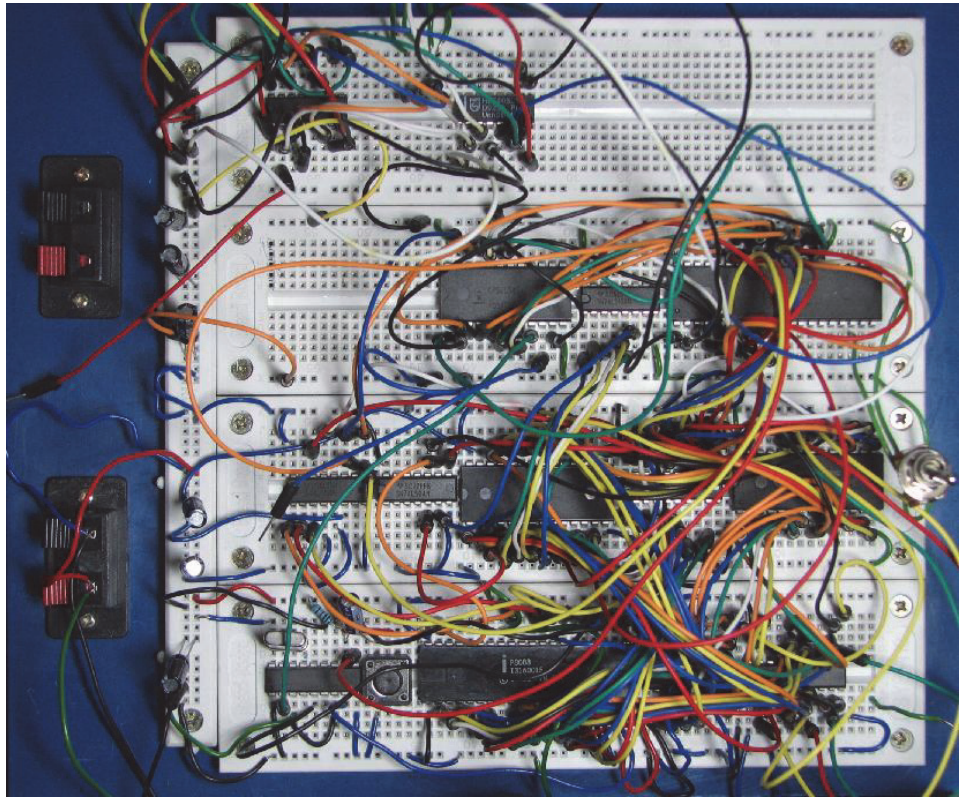
fabrication steps



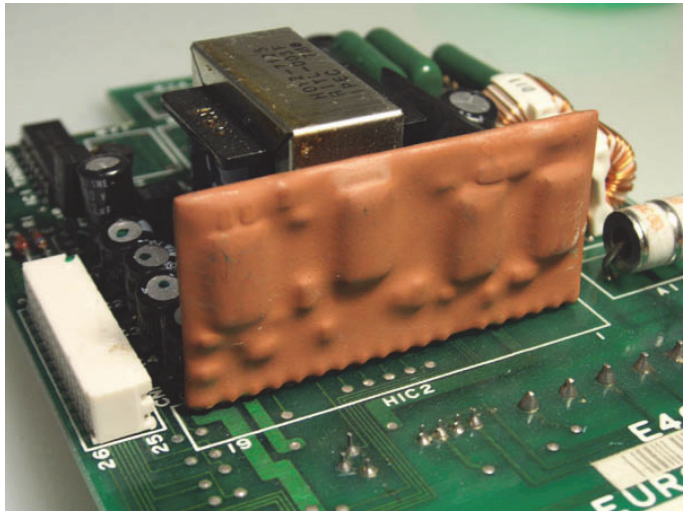
www.wikipedia.org

Packaging examples

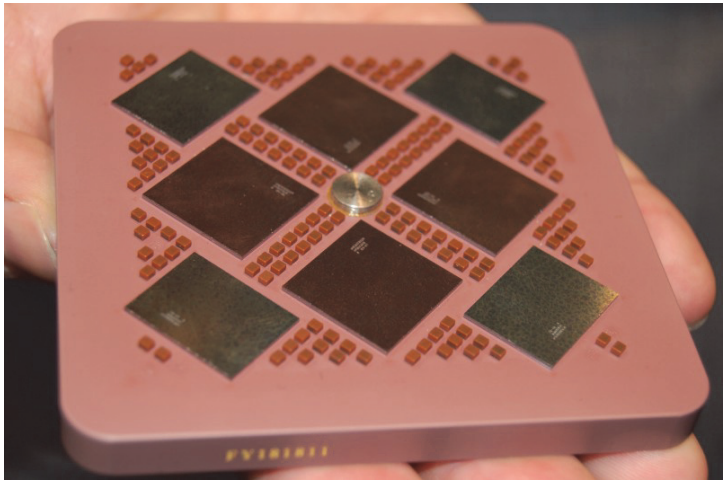
prototyping



hybrid circuit



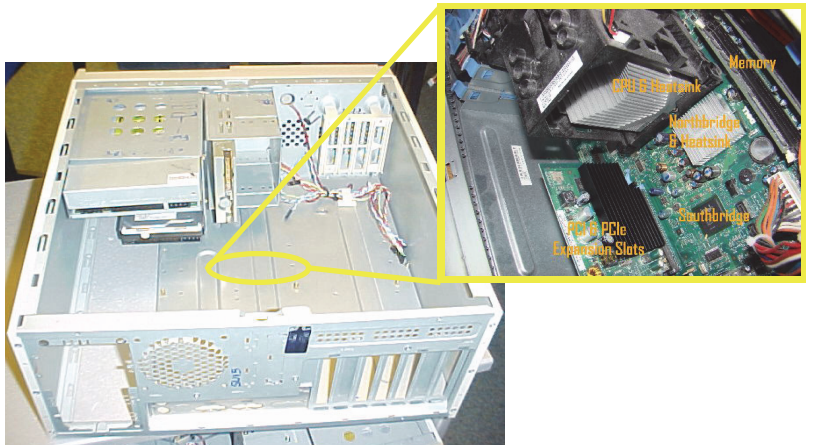
3D assembly

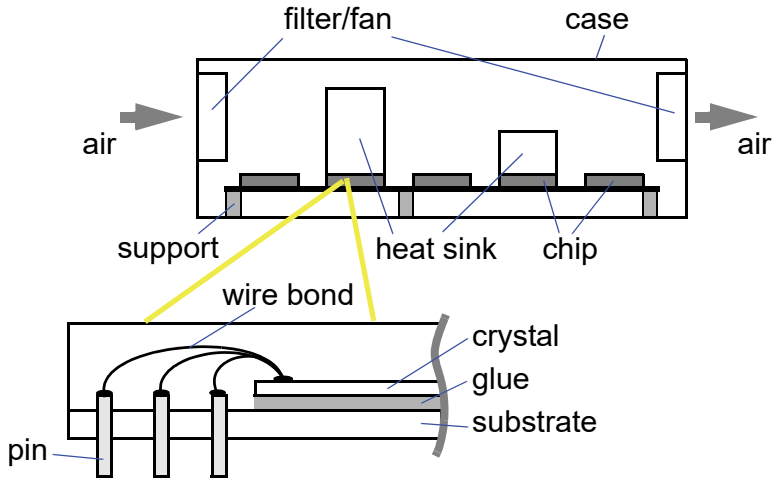


IBM POWER5

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



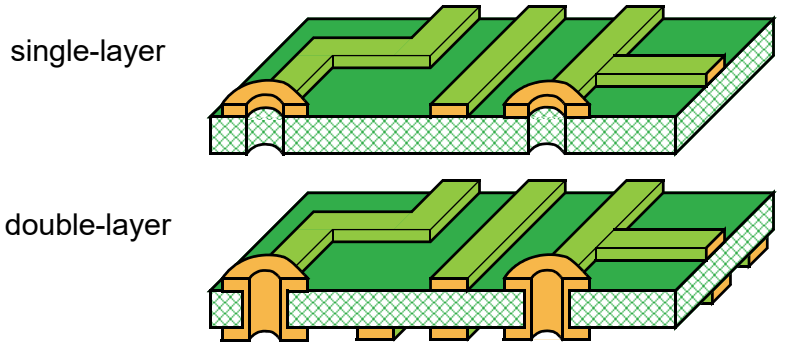


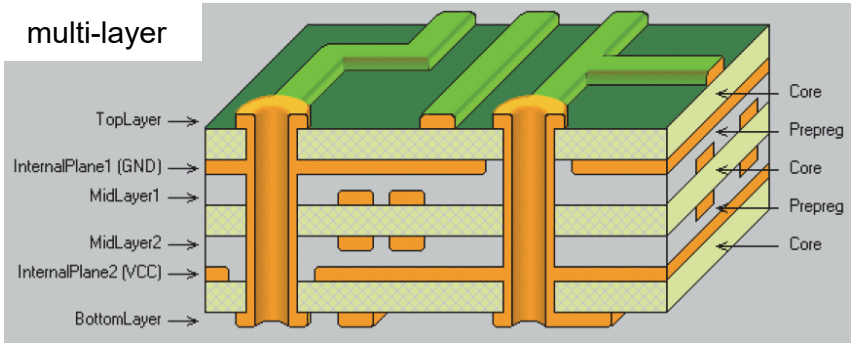
Printed Circuit Board (PCB)

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

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

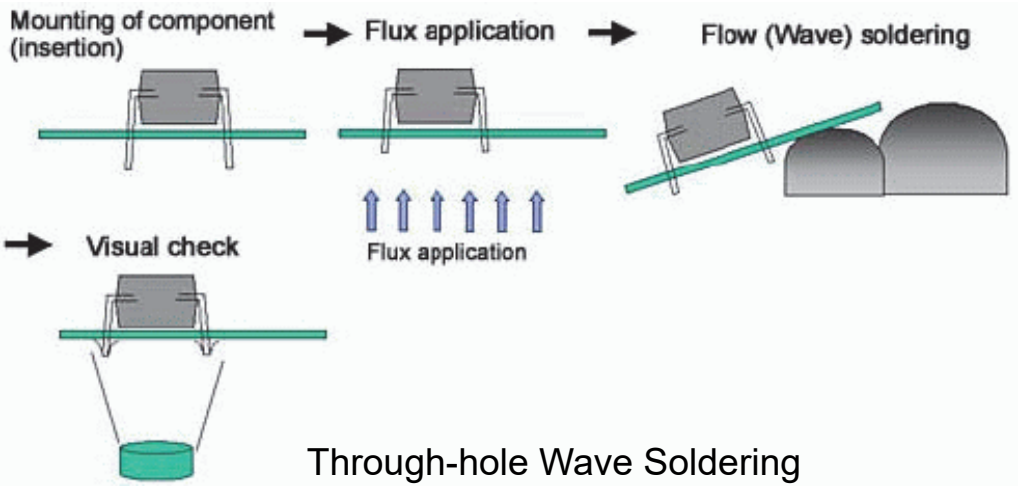
- Single-layer PCB
  - wires (bottom side)
- Double-layer PCB
  - wires + metallized vias
- Multi-layer PCB
  - multiple double-layer PCB-s
  - location of vias!



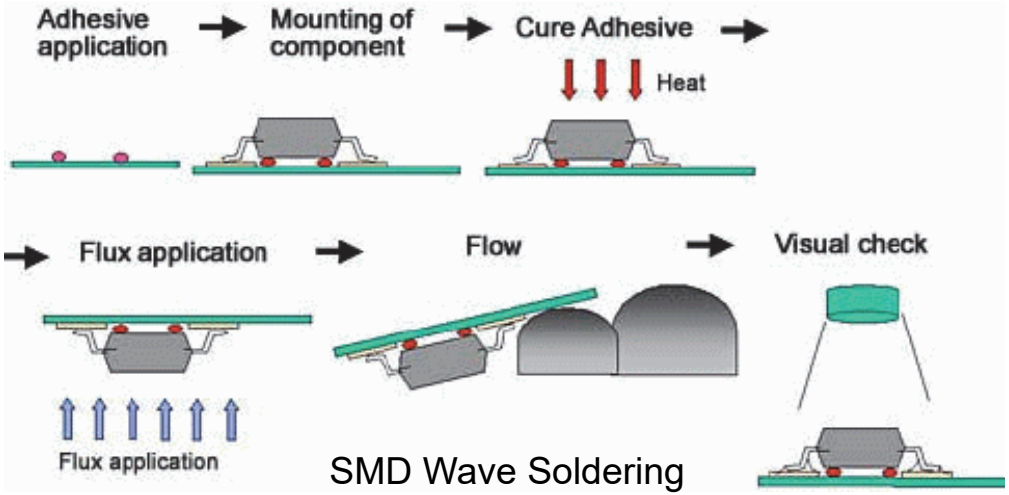


PCB manufacturing

- 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



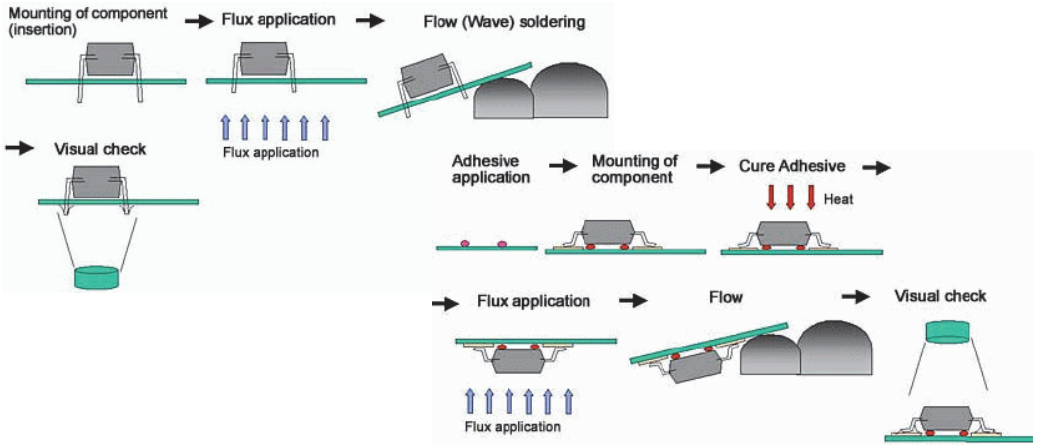
SMD Wave Soldering

PCB manufacturing

Wave Soldering

Electro Soft Inc.
https://www.youtube.com/watch?v=inHzaJIE7-4

Agrowtek Inc.
https://www.youtube.com/watch?v=VWH58QrprVc

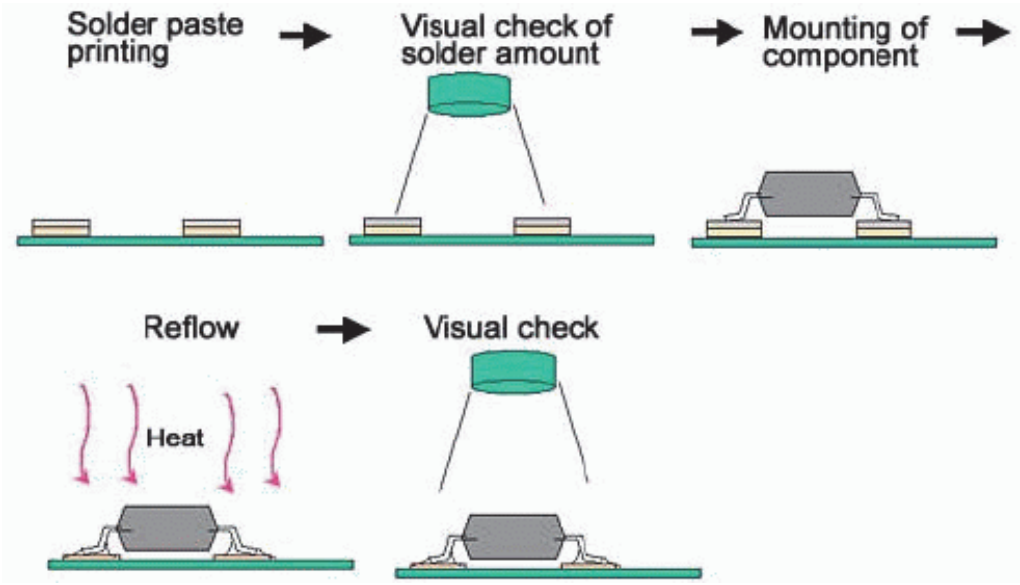


SMD Reflow Soldering

GIGABYTE factory tour
https://www.youtube.com/watch?v=Va3Bfjn4inA

Tutorial
https://www.youtube.com/watch?v=gu0v8lfLcKg

SMD reflow at home
https://www.youtube.com/watch?v=U48Nose31d4



Logic synthesis

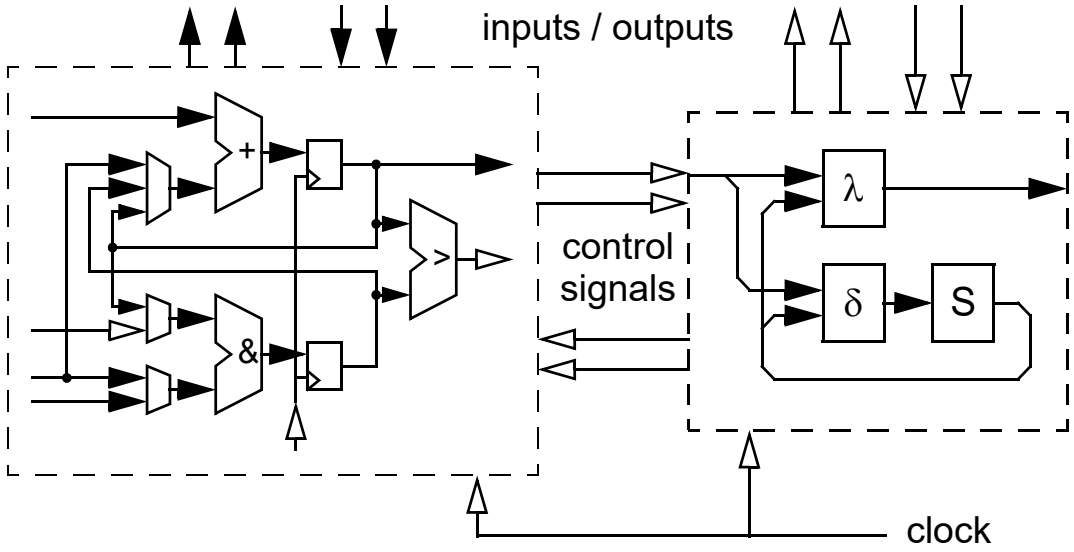
- Transforming logic functions (Boolean functions) into a set of logic gates
  - transformations at logic level from behavioral to structural domain
- Optimizations / Transformations
  - area
  - delay
  - power consumption
- Implementation of Finite State Machines (FSM)
  - state encoding
  - generating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

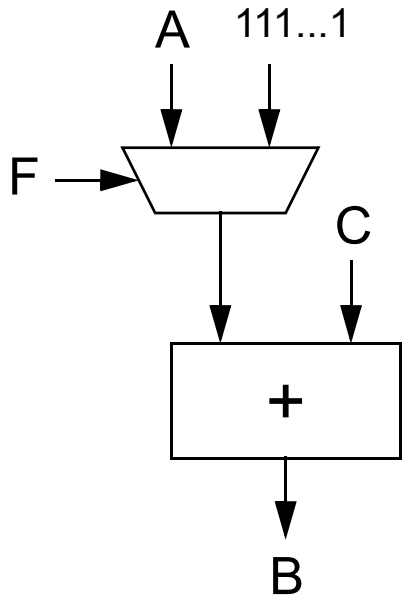
Digital system @RTL = data-path + controller



- *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

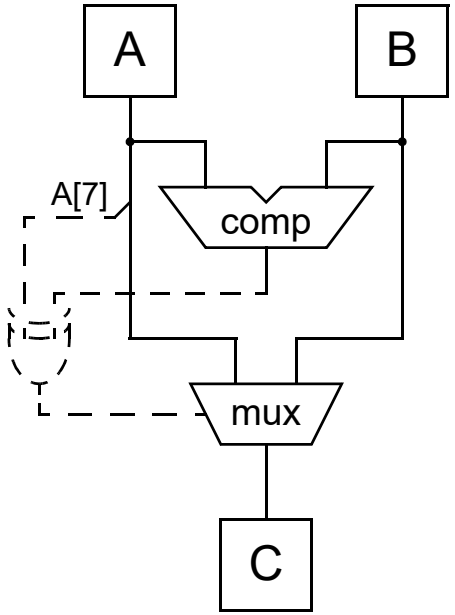
Data-path optimization

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



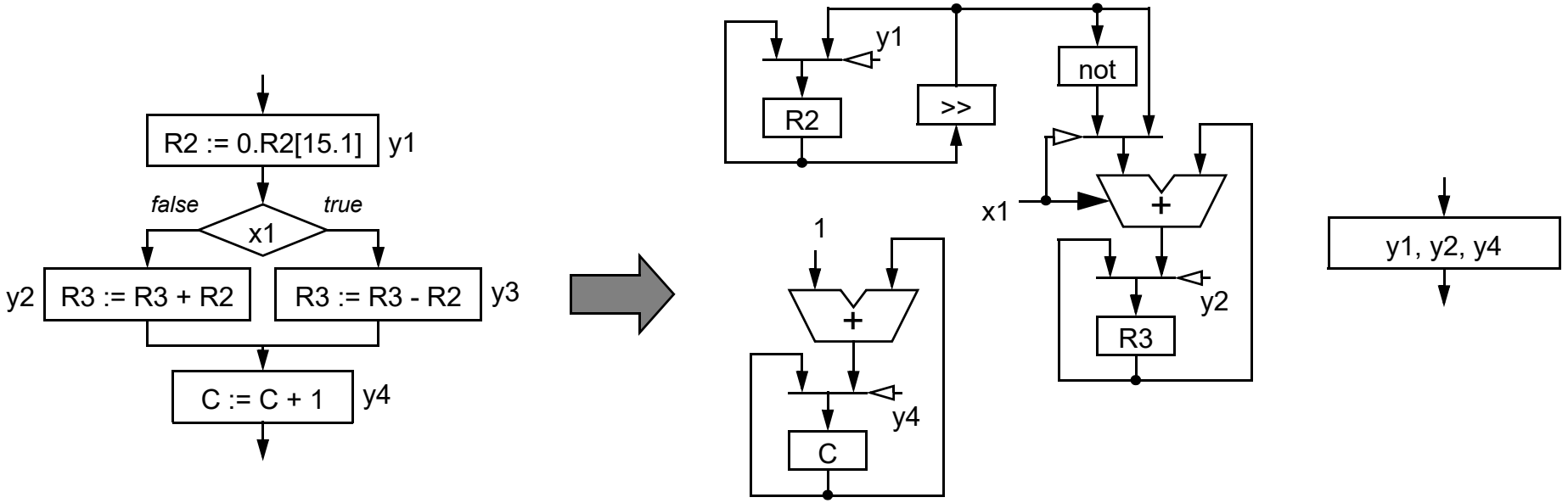

```
read_port(A);
read_port(B);
if A(7)='0' then
  if A>B then
    C := A;
  else
    C := B;
  end if;
else
  if A>B then
    C := B;
  else
    C := A;
  end if;
end if;
```

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

Carry Look Ahead Adder With Timing

Carry Look Ahead Adder

A: 00101110
B: 01100010
bin ● *dec* ○

Total Bits 8 Group Size 4

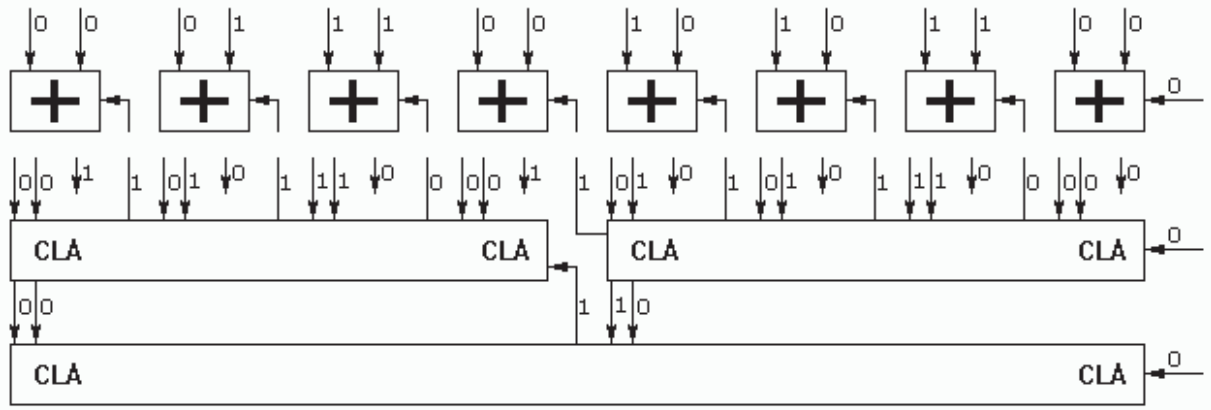
Signal Delays

AND/OR Gate Delay	1.0
XOR Gate Delay	1.6
Maximum Fan-in	4

Compute

Reset

Help



A 00101110 :46
B +01100010 :98

Sum 10010000 :**144**

Time taken to generate all Sum bits - (10.2)units

Delays to calculate Sum in each bit position

BitNumber	7	6	5	4	3	2	1	0
SumDelay	(10.2)units	(10.2)units	(10.2)units	(10.2)units	(7.2)units	(7.2)units	(7.2)units	(7.2)units

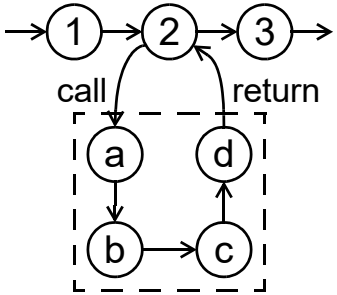
Controller synthesis

- 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.

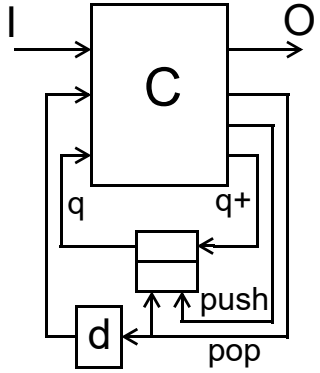
FSM encoding

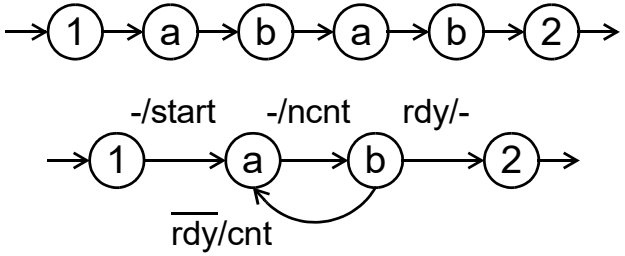
- Input/output encoding – symbols –> binary code
- State encoding – states –> binary code
- *z* – number of symbols/states –> minimum code length is t=ceil($\log_2 z$)
- Two border cases – minimal code length encoding & one-hot-encoding
- General encoding strategy
  - Identification of sets of states (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.

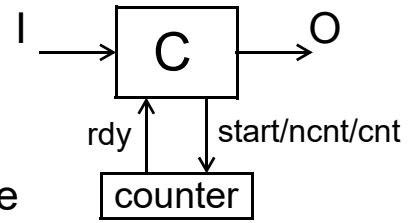
FSM architectures



Stack architecture

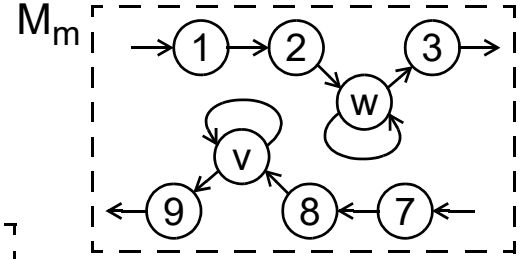


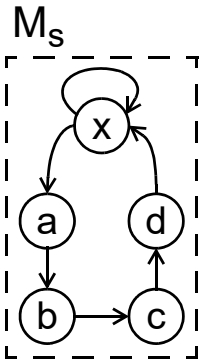


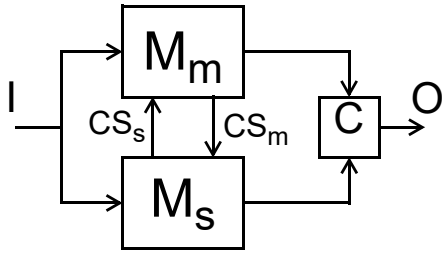


Counter architecture

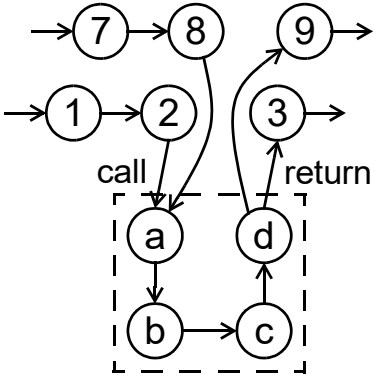
Decomposed architecture

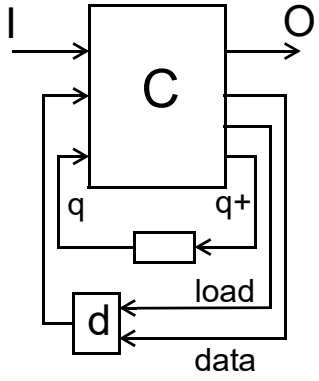






Register architecture





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
- Front-end tasks:
  - Mapping PL/HDL decription into internal graph-based representation
  - Compiler optimizations
- Back-end tasks:
  - Behavioral transformations
  - Essential subtasks – transforming data and control flow into RT level structure
    - scheduling – time or resource constrained
    - resource allocation – functional units, storage elements, interconnects
    - resource assignment (binding) – functional units, storage elements, interconnects
  - Netlist extraction, state machine table generation

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

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
- Resource constrained scheduling (RCS)
  - List scheduling
- Time constrained scheduling (TCS)
  - ASAP – “as soon as possible” / ALAP – “as late as possible”
  - ASAP and ALAP scheduling are used for
  - Force directed scheduling
- Neural net based schedulers
- Path-based / path traversing scheduling (AFAP)

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

Completing the Data Path

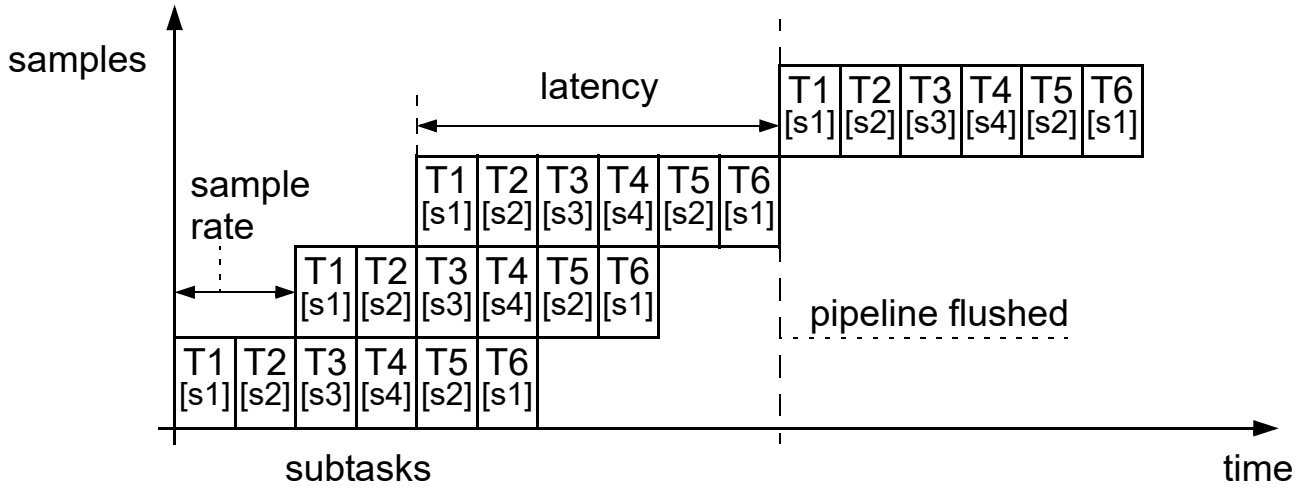
- Subtasks after scheduling
  - Allocation
    - Allocation of FUs (if not allocated before scheduling)
    - Allocation of storage (if not allocated before scheduling)
    - Allocation of busses (if buses are required and not allocated in advance)
  - Binding (assignment)
    - Assignment of operations 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)

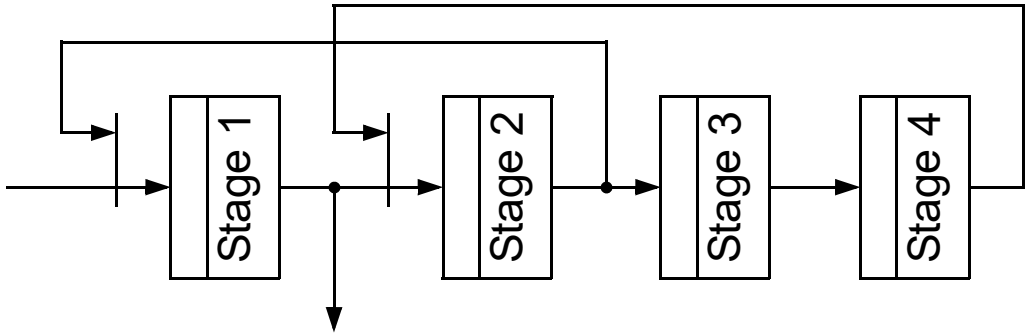
Pipelining

- *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):

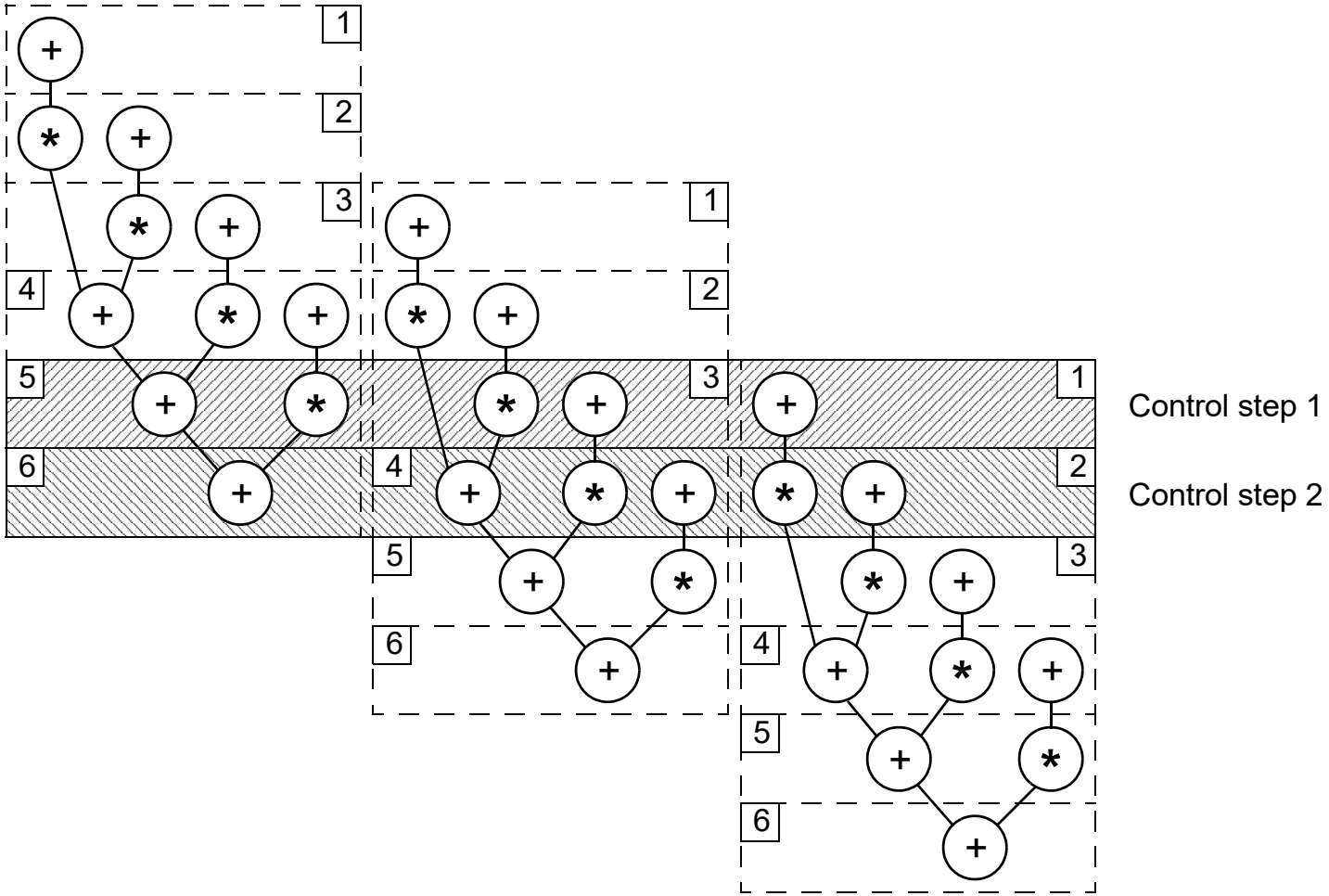
  $R_{init,N\to\infty} = 1 / (R \times t_{stage} + r_{synchro} \times (L-R) \times t_{stage})$

  - 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 $s+n\times L$, for $n\geq 0$, are executed simultaneously and cannot share hardware.

Functional pipelining of 8-point FIR filter



Retiming

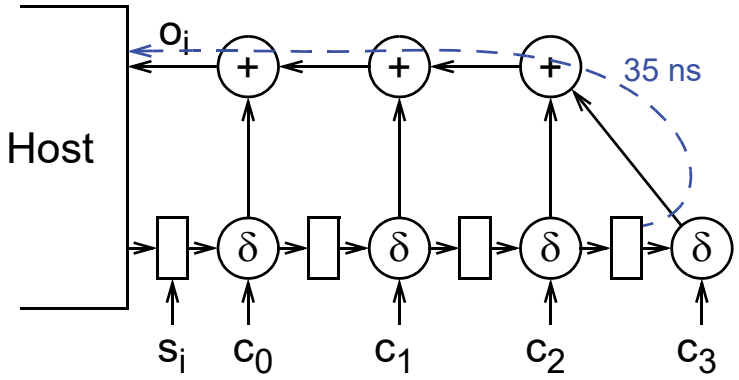
- Minimization of the cycle-time or the area of synchronous circuits by changing the position of the registers
  - cycle-time <- critical path
- The number of registers may increase or decrease
  - area minimization corresponds to minimizing the number of registers
  - combinational circuits are not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

Retiming at higher abstraction levels?

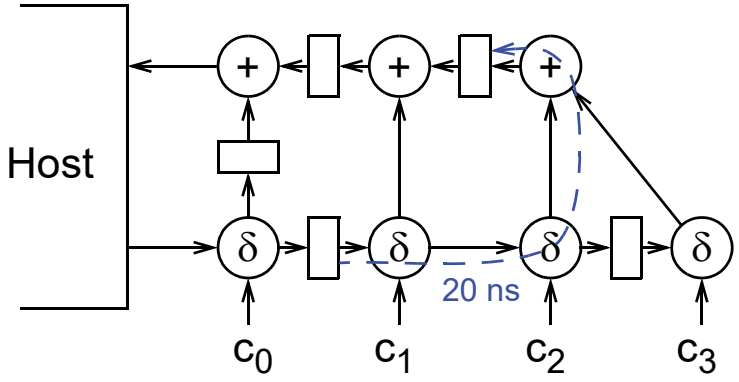
- The same, in principle, as for logic networks
  - operation nodes - functions / delay nodes - e.g. shared resources (memories)
- More possibilities to manipulate the functions - higher complexity of the optimization task
  - partitioning/merging functions
  - reorganizing shared resources

Retiming: example #2

Digital correlator



Before



After

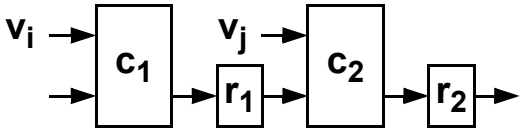
$$o_i = \sum_{j=0}^{3} \delta(s_{i-j}, c_j)$$

+ 10 ns

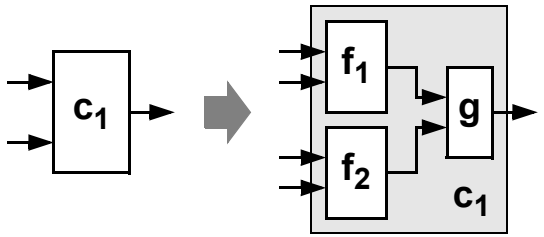
δ 5 ns

Retiming at higher abstraction levels

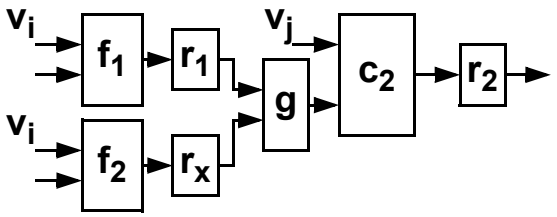
- **Optimization:**
  - control-step #1: $r_1 \leftarrow c_1(v_i,...)$,
  - control-step #2: $r_2 \leftarrow c_2(r_1,v_j,...)$
  - r_1,r_2 - registers; c_1,c_2 - combinational blocks; v_i,v_j - variables



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



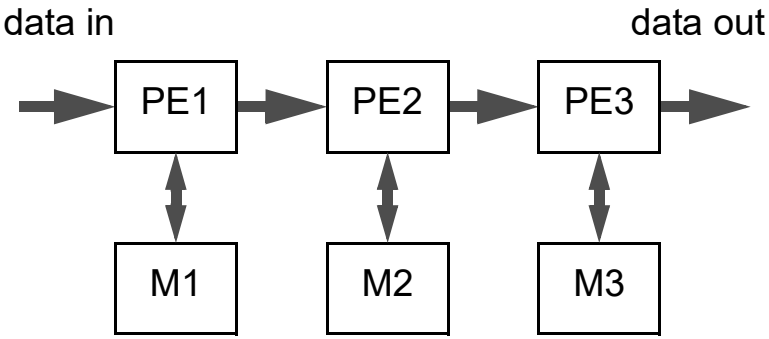
- **After resynthesis, $delay(g)+delay(c_2) < delay(c_1)$:**
  - control-step #1: $r_1 \leftarrow f_1(v_i,...)$; $r_x \leftarrow f_2(v_i,...)$
  - control-step #2: $r_2 \leftarrow c_2(g(r_1,r_x),v_j,...)$



Transformations at System Level

Motivation

- Multimedia applications
  - MPEG 4 – > 2 GOP/s, > 6 GB/s, > 10 MB storage
- Embedded systems, SoC, NoC
  - Memory is a power bottleneck
    - MPEG-2 decodes – logic 27%, i/o 24%, clock 24%, on-chip memory 25%
    - CPU + caches – logic 25%, i/o 3%, clock 26%, on-chip memory 46%
  - Memory is a speed bottleneck [Patterson]
    - Moore's Law – μProc 60%/year
    - DRAM – 7%/year
    - Processor-memory speed gap – 50%/year!
- High Performance Computing
  - CPU constrained vs Memory constrained



What can be done?

- Reduce overheads!
  - do necessary calculations only
    - do not duplicate calculations vs. do not store unnecessary data
    - data must have the right size (bitwidth, array size)
  - select the right data layout
    - bitwidth can be reduced by coding
    - number of switchings on a bus can be reduced by coding
  - exploit memory hierarchy
    - local copies may significantly reduce data transfer
  - distribute load
    - systems are designed to stand the peak performance
- But do not forget trade-offs...

Data layout optimization

- Input/output data range given
  - what about intermediate data?
- Integer vs. fixed point vs. floating point
  - integer
    - + simple operations
    - - no fractions
  - fixed point
    - + simple add/subtract operations
    - - normalization needed for multiplication/division
  - floating point
    - + flexible range
    - - normalization needed

Data layout optimization
Software implementations

- Example: 16-bit vs. 32-bit integers?
  - depends on the CPU, bus & memory architectures
  - depends on the compiler (optimizations)
  - 16-bit
    - + less space in the main memory
    - + faster loading from the main memory into cache
    - - unpacking/packing may be needed (compiler & CPU depending)
    - - relative redundancy when accessing a random word
  - 32-bit
    - + data matches the CPU word ["native" for most of the modern CPUs]
    - - waste of bandwidth (bus load!) for small data ranges
  - The same applies 32 vs. 64-bit integers

Data layout optimization

Hardware implementations

- Example #1: 16-bit integer vs. 16-bit fixed point vs. 16-bit floating point
  - integer – 1+15 bits ~~ -32000 to +32000, precision 1
  - fixed point – 1+5+10 bits ~~ -32 to +32, precision ~0.001 (~0.03%)
    - normalization == shifting
  - floating point – 1+5+10 bits ~~ -64000 to +64000, precision ~0.1%
    - normalization == analysis + shifting

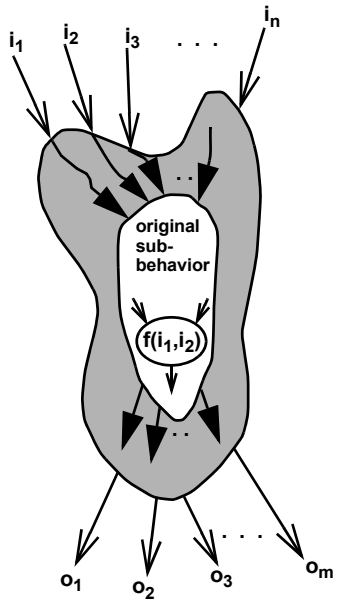
- Example #2: 2's complement vs. integer with sign
  - 2's complement – +1 == 00000001, 0 == 00000000, -1 == 11111111
  - integer + sign – +1 == 00000001, 0 == 00000000/10000000, -1 == 10000001
  - number of switchings when data swings around zero!!!! [e.g., DSP]

Optimizing the number of calculations

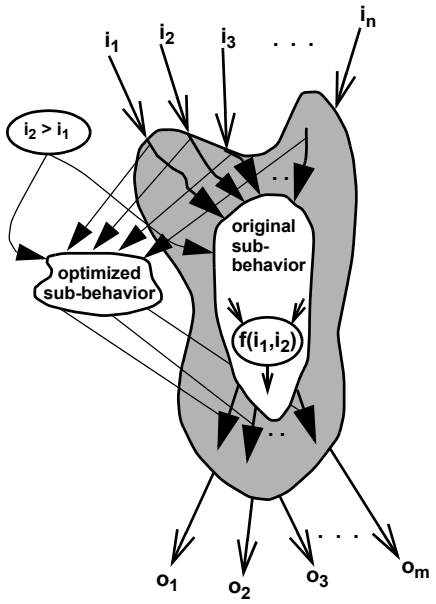
- Input space adaptive design – avoid unnecessary calculations
  - input data is analyzed for boundary cases
  - boundary cases have simplified calculations (vs. the full algorithm)
  - “boundary case” – it may happen in the most of the time
    - ... and the full algorithm is actually calculating nothing
- Trade-offs
  - extra hardware is introduced
    - the algorithm is essentially duplicated (or triplicated or ...)
    - extra units for analysis are needed
  - implementation is faster
    - less operations require less clock steps
    - operations themselves can be faster
  - implementation consumes less power
    - a smaller number of units are involved in calculations
    - the clock frequency can be slowed
    - supply voltage can be lowered

Input space adaptive design – The Case

- **Discrete wavelet transformation**
  - four input words – A, B, C, D
  - in 95.2% cases: A = B = C = D
  - four additions, four subtractions and four shifts are replaced with four <u>assignments</u>
- **Results**
  - area: +17.5%
  - speed: 52.1% (cycles)
  - power: 58.9%
    V_{dd} scaled: 26.6%



original behavior



optimized behavior

Input space adaptive design

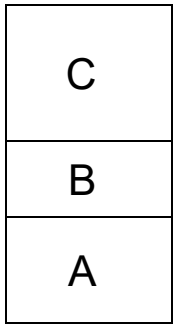
Few examples

- Multiplication with zero
  - result is zero –> nothing to calculate
  - two comparators, few blocking gates, plus some extras to the controller
  - multiplication with one, etc.
  - it must be cost efficient – the extra HW consumes power too!
- Multiplier size
  - $A_{32}*B_{32} = (A_{H16}*2^{16}+A_{L16})*(B_{H16}*2^{16}+B_{L16}) = ...$
  - $... = A_{H16}*B_{H16}*2^{32}+A_{H16}*B_{L16}*2^{16}+A_{L16}*B_{H16}*2^{16}+A_{L16}*B_{L16}$
  - $A < 2^{16}$ & $B < 2^{16}$ –> $A_{H16}=0$ & $B_{H16}=0$
- Multiplication with a constant –> shift-adds
  - 6*x = 4*x+2*x = (x<<2) + (x<<1)
  - - 1.75*x = 0.25*x - 2*x = (x>>2) - (x<<1)

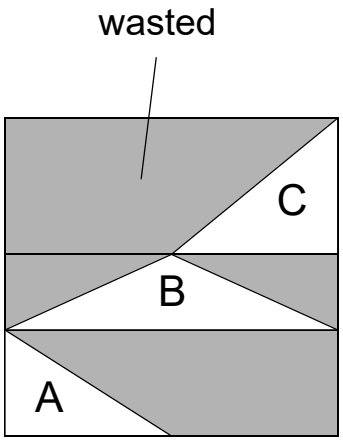
Memory architecture optimization

Life time of array elements

B = f(A);
C = g(B);



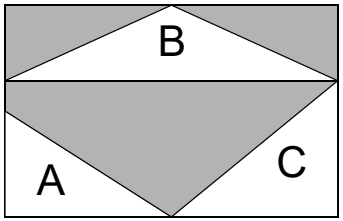
typical allocation



life times

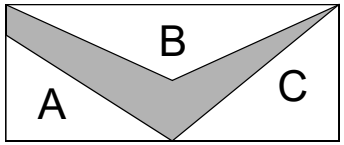
a[i]==mem[i]
b[i]==mem[base_b+i]
c[i]==mem[base_c+i]

Improved allocations



a smarter allocation

a[i]==mem[i]
b[i]==mem[base_b+i]
c[i]==mem[i]



the smallest memory waste

a[i]==mem[i]
b[i]==mem[size-i]
c[i]==mem[i]

Memory access – code transformations

- Code transformations to reduce/balance data transfers
- Loop transformations
  - improve regularity of accesses
  - improve locality: production –> consumption
  - expected influence – reduce temporary and background storage

```
for (j=1; j<=M; j++)
  for (i=1; i<=N; i++)
    A[i]=foo(A[i],...);

for (i=1; i<=N; i++)
  out[i]=A[i];
```

storage size N

```
for (i=1; i<=N; i++) {
  for (j=1; j<=M; j++) {
    A[i]=foo(A[i],...);
  }
  out[i]=A[i];
}
```

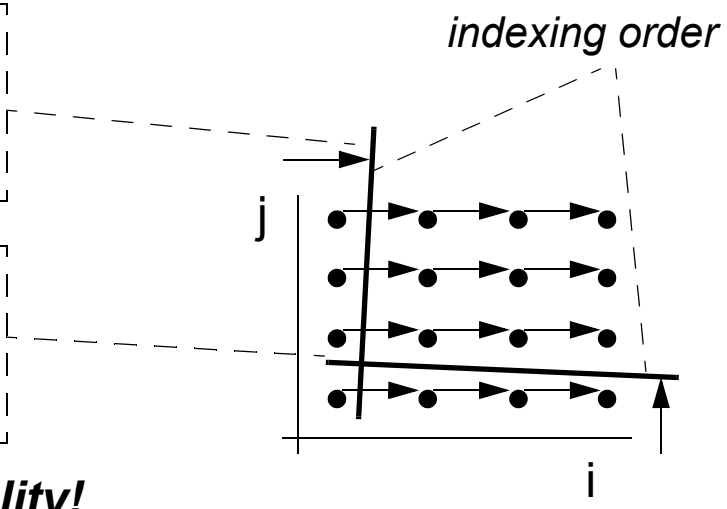
storage size 1

Polyhedral model - polytope placement

Life time analysis of array elements

```
for (i=1; i<=N; i++)
  for (j=N; j>=1; j--)
    A[i][j]=foo(A[i-1][j],...);
```

```
for (j=1; j<=N; j++)
  for (i=1; i<=N; i++)
    A[i][j]=foo(A[i-1][j],...);
```



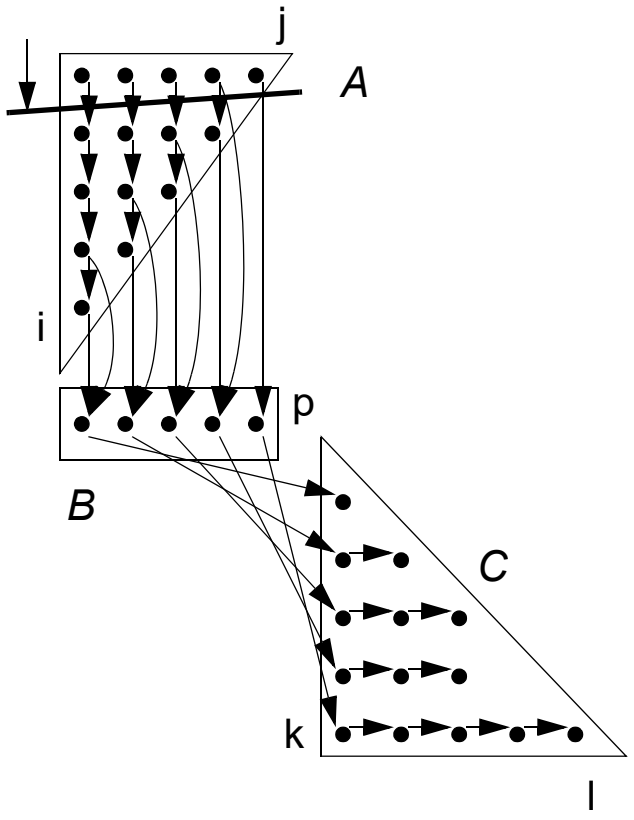
The indexing order must preserve causality!

Storage requirements – # of dependency lines

- **select the right indexing order**
- **merge the loops**
- **can be used for "loop pipelining" – possible index range expansion**

Polytope placement – initial code

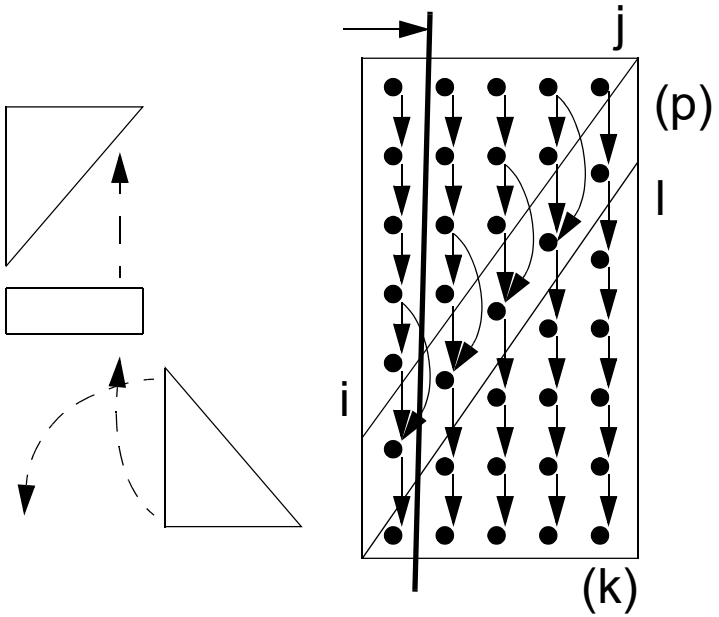
```
/* A */
for (i=1; i<=N; i++)
  for (j=1; j>=N-i+1; j++)
    a[i][j]=in[i][j]+a[i-1][j];
/* B */
for (p=1; p<=N; p++)
  b[p][1]=f(a[N-p+1][p],a[N-p][p]);
/* C */
for (k=1; i<=N; i++)
  for (l=1; l>=k; l++)
    b[k][l+1]=g(b[k][l]);
```



- **storage size – 2*N-1**

Polytope placement – reordering + loop merging

```
for (j=1; j<=N; j++) {
  for (i=1; i>=N-j+1; i++)
    a[i][j]=in[i][j]+a[i-1][j];
  b[j][1]=f(a[N-j+1][j],a[N-j][j]);
  for (l=1; l>=j; l++)
    b[j][l+1]=g(b[j][l]);
}
```



- **storage size – 2**

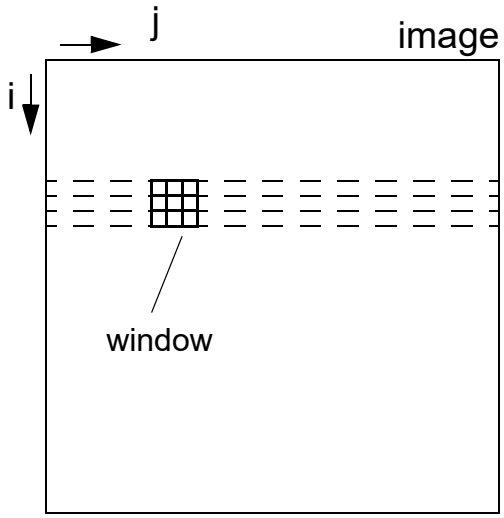
Exploiting memory hierarchy

- Inter-array in-place mapping reduces the size of combined memories
  - source array is consumed while processing and can be replaced with the result
- Introducing local copies reduces accesses to the (main) memory
  - cost of the local memory should be smaller than the cost of reduced accesses
- Proper base addresses reduce cache penalties
  - direct mapped caches –
    the same block may be used by two different arrays accessed in the same loop!
- All optimizations
  - ~60% of reduce in memory accesses
  - ~80% of reduce in total memory size
  - ~75% of reduce in cycle count

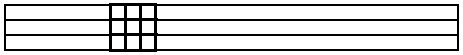
Introducing local copies

- **Data copies to exploit memory hierarchy**

Image processing
blurring
edge detection
etc.



no buffering –
9 accesses per pixel



3 line buffers
(in cache) –
3 accesses per pixel
in the buffer memory

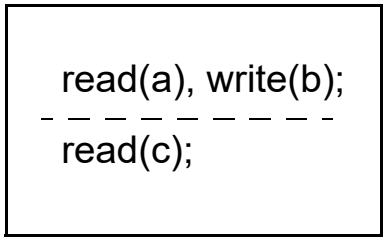
Extra copies should be analyzed for efficiency – time & power

9 pixels in register file –
1 access per pixel
in the main memory

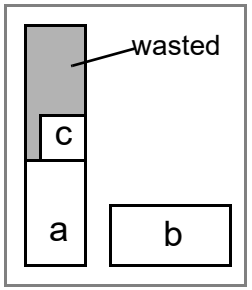
Memory architecture optimization

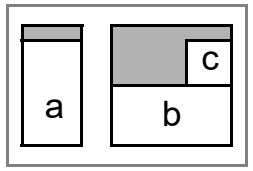
Packing data into memories (#1)

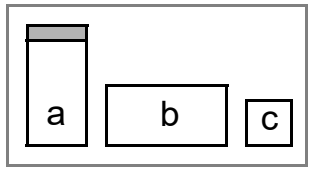
- Packing arrays into memories
- Memory allocation and binding task
  - memory access distribution in time
    - simultaneous accesses
  - mapping data arrays into physical memories
    - single-port vs. multi-port memories
    - unused memory area



access sequence







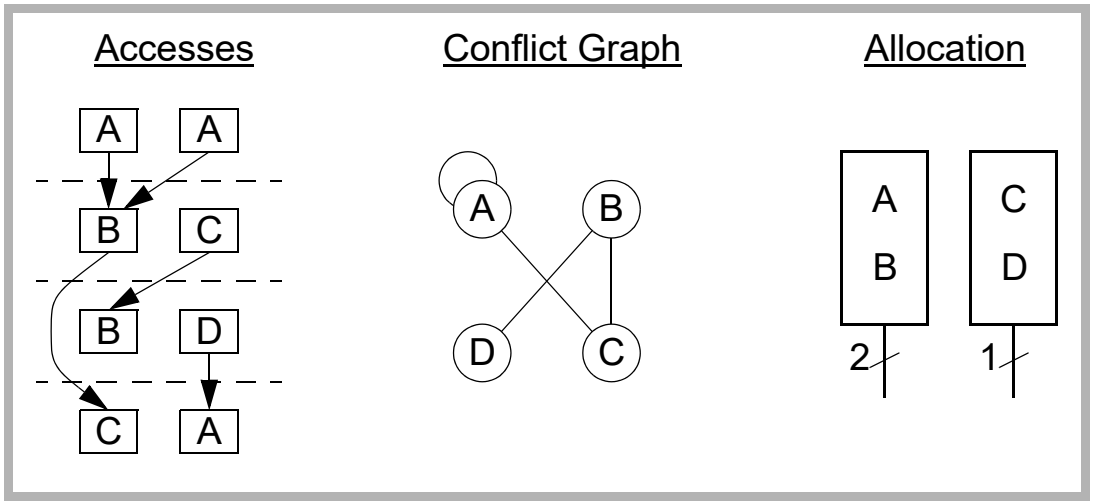
legal mappings

Packing data into memories (#2)

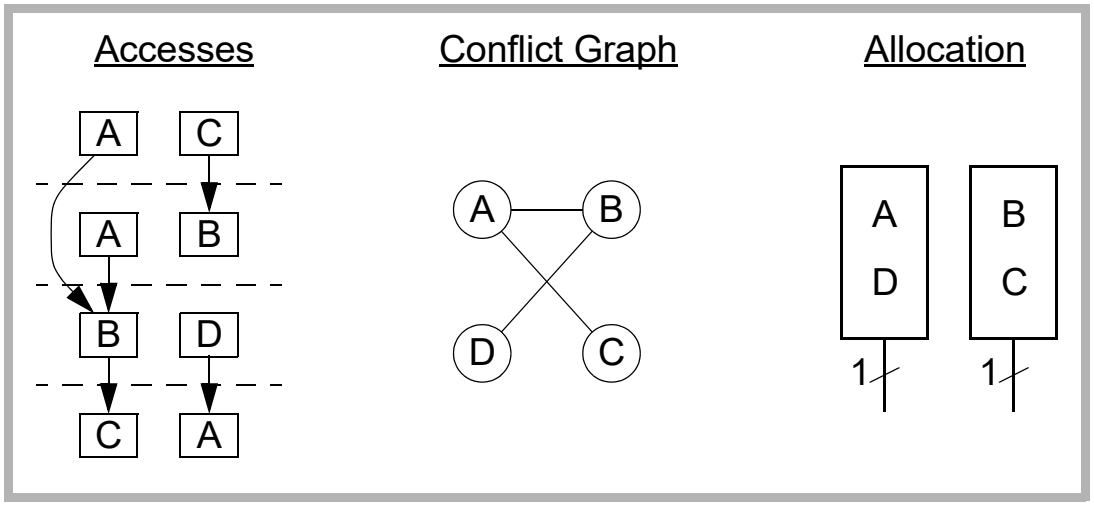
Storage operation scheduling

- 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
  - distributes access more uniformly – reduces required bus bandwidth
  - reduces the number of possible combinations

Storage operation scheduling



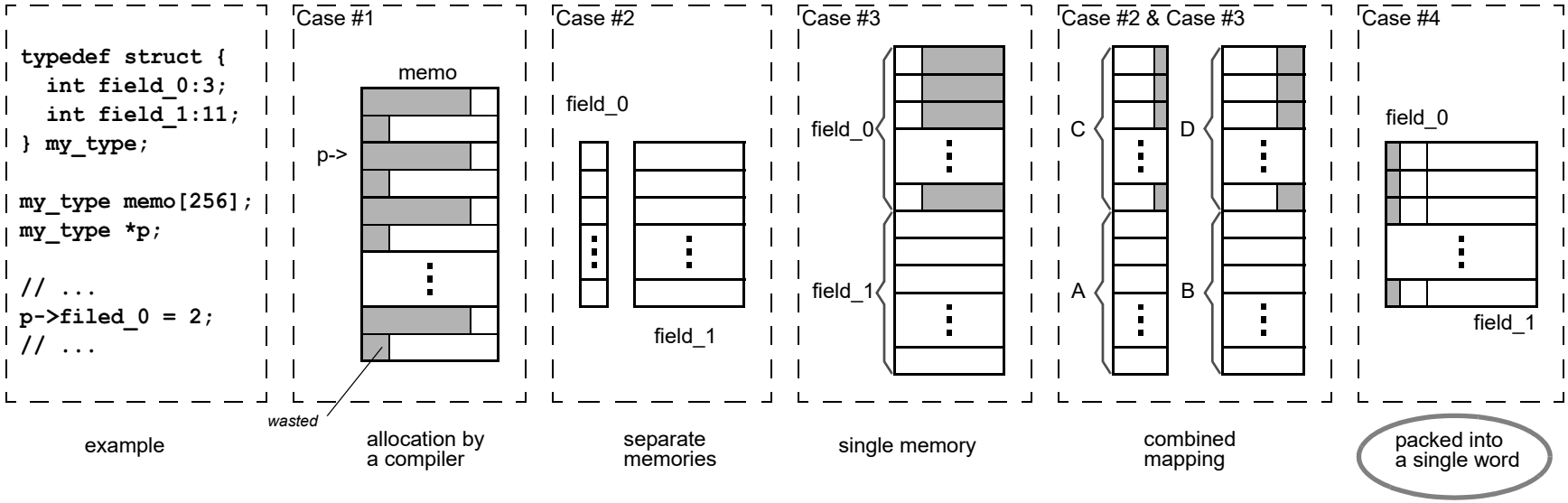
Initial



After scheduling of memory accesses

Packing data into memories (#3)

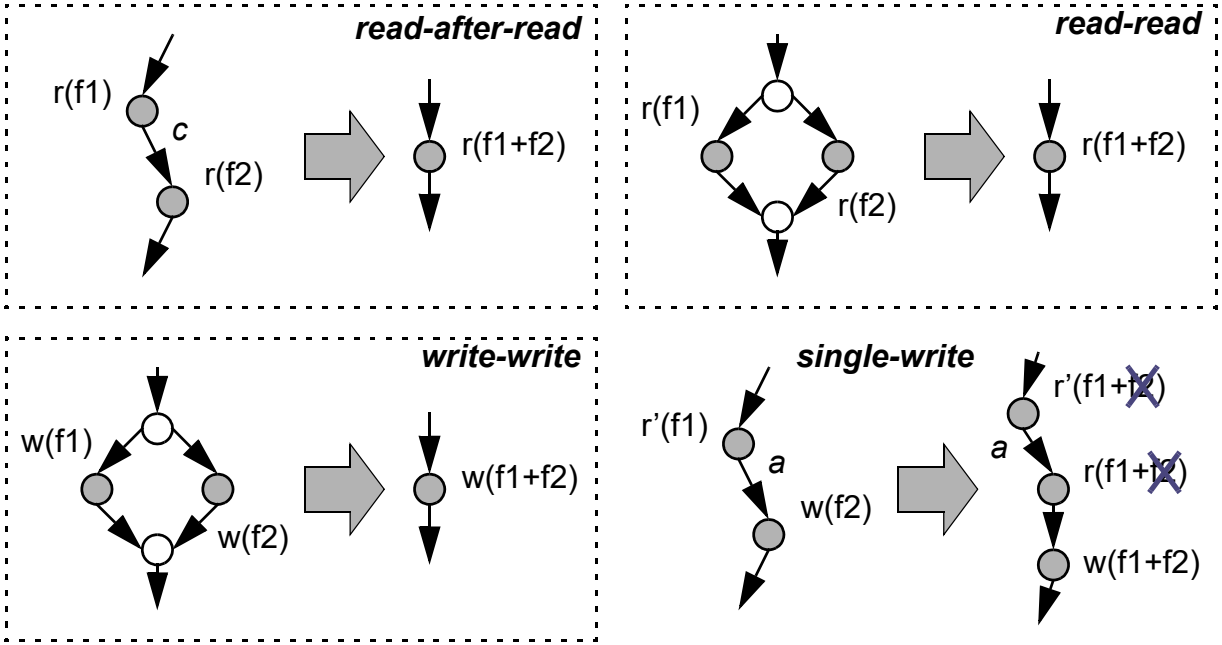
Packing data words (data fields) into memory words



- **How to select candidates for packing?**

Packing data fields into memory words

- Data access duplications may occur
  - a word (field) is read but discarded
  - extra reads may be needed to avoid data corruption when writing



Address code optimization – loop-invariant code motion

- Array access – base address, dimensions, repeated access patterns
- Address calculations that are repeated at every iteration
- Not handled by compilers - it's out of basic block optimization

```
for (k=-4; k<3; k++)
  for (i=-4; i<3; i++)
    Ad[32*(8+k)+16+i]
```

(3+, 1*) x 49 = 245

```
272+32*k+i
```

```
for (k=-4; k<3; k++) {
  tmp_k = 272 + 32*k;
  for (i=-4; i<3; i++)
    Ad[tmp_k+i]
}
```

(1+, 1*) x 7 +

+ (1+) x 49 = 70

Modulo operations

```
for (i=0; i<=20; i++)
  addr = i % 3;
```

```
ptr = -1;
for (i=0; i<=20; i++) {
  if (ptr>=2) ptr-=2;
  else ptr++;
  addr = ptr;
}
```

Counter base address generation units

- Customized hardware
  - special addressing units are cheaper
  - data-path itself gets simpler
  - essentially distributed controllers

Scheduling of high performance applications

- Critical path – sequence of operations with the longest finishing time
  - Critical path in clock steps vs critical path inside clock period (in time units)
    - Topological paths vs. false paths
  - Latency ~ clock_period * critical_path_in_clock_steps
- Loop folding and pipelining
  - Folding – fixed number of cycles
    - no need to calculate conditions in loop header, iterations of the loop body can overlap
  - Pipelining – executing some iterations of the loop body in parallel (no dependencies!)
- Speculative execution / Out of order execution
  - Execution of operations in a branch can be started before the branch itself starts
  - Extra resources required
  - Speeding up the algorithm – guaranteed speed-up vs overall speed-up (statistical)
  - Scheduling outside basic blocks – “flattening” hierarchy
    - increase in the optimization complexity

Conclusion

- 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!